EPS-MoE: Expert Pipeline Scheduler for Cost-Efficient MoE Inference

Yulei Qian, Fengcun Li, Xiangyang Ji, Xiaoyu Zhao, Jianchao Tan, Kefeng Zhang, Xunliang Cai
Meituan Inc.
{qianyulei02, lifengcun, jixiangyang, zhaoxiaoyu17, tanjianchao02, zhangkefeng, caixunliang}@meituan.com

***Abstract*—Large Language Model (LLM) has revolutionized the field of artificial intelligence, with their capabilities expanding rapidly due to advances in deep learning and increased computational resources. The mixture-of-experts (MoE) model has emerged as a prominent architecture in the field of LLM, better balancing the model performance and computational efficiency. MoE architecture allows for effective scaling and efficient parallel processing, but the GEMM (General Matrix Multiply) of MoE and the large parameters introduce challenges in terms of computation efficiency and communication overhead, which becomes the throughput bottleneck during inference. Applying a single parallelism strategy like EP, DP, PP, etc. to MoE architecture usually achieves sub-optimal inference throughput, the straightforward combinations of existing different parallelisms on MoE can not obtain optimal inference throughput yet. This paper introduces EPS-MoE, a novel expert pipeline scheduler for MoE that goes beyond the existing inference parallelism schemes. Our approach focuses on optimizing the computation of MoE FFN (FeedForward Network) modules by dynamically selecting the best kernel implementation of *GroupGemm* and *DenseGemm* [1] for different loads and adaptively overlapping these computations with *all2all* communication, leading to a substantial increase in throughput. Our experimental results demonstrate an average 21% improvement in prefill throughput over existing parallel inference methods. Specifically, we validated our method on DeepSeekV2, a highly optimized model claimed to achieve a prefill throughput of 100K tokens per second. By applying EPS-MoE, we further accelerated it to at least 120K tokens per second.**

***Index Terms*—MoE, LLM Inference, Expert Pipeline Scheduler**

I. Introduction

The remarkable capabilities of the Large Language Model have attracted various organizations to devote resources to optimize their architectures for better performance and efficiency, leading to the development of advanced Mixture-of-Experts (MoE) architectures like Mixtral [1], DeepSeekV2 [2], Grok [3], Gemini 1.5 [4], Snowflake [5] [6] and others [7] [8] [31] [32] [33] [34] [35] [36] [37]. These architectures offer significant advantages by enabling the dynamic selection of specialized experts, thus optimizing performance and computational efficiency. MoE models, such as Snowflake and Mixtral, have demonstrated the ability to scale up model parameters significantly for improved performance while maintaining a manageable computational footprint.

[1]For convenience, we use *GroupGemm* to refer to grouped GEMM from *cutlass* and *DenseGemm* to refer to *cublas* GEMM for dense matrix multiplication.

Typically, MoE architectures incorporate a gating mechanism that directs the output of the attention mechanism to a subset of experts, thereby activating only a fraction of the model's parameters. This approach can expand model capacity with far fewer activated parameters to achieve performance comparable to larger dense models. For instance, DeepSeekV2, with only 21 billion activated parameters, rivals the performance of Llama3's 70 billion parameters. [2]

However, MoE architectures encounter significant challenges when scaling to accommodate large sequence lengths and batch sizes. For example, Mixtral 8x7B requires only 12.6 billion activated parameters per token but can demand up to 46 billion parameters for large batch sizes. Due to the large total number of parameters, MoE models often require multi-GPU parallel inference, which also leads to an increase in communication time. Moreover, the router's top-k gating mechanism, while beneficial for selecting relevant experts, intensifies the communication challenge when k is large. In such scenarios, the communication requirement can be magnified k-fold, as information must be exchanged with k different experts simultaneously. This can result in bottlenecks in the inference pipeline, as the computation for each expert's output cannot commence until the communication of inputs is complete. In addition to communication challenges, MoE models also face issues with low computation density, particularly when the distribution of tokens leads to fragmented workloads across the experts. The resulting imbalance can leave some computational resources underutilized, further impacting the overall efficiency of the inference process.

Due to the rapid scaling of the model parameters, distributed serving architectures have become indispensable for serving Mixture-of-Experts (MoE) models at scale. Common strategies [18] [19] include Data Parallelism (DP), Tensor Parallelism (TP), Pipeline Parallelism (PP), and Expert Parallelism (EP), as shown in Fig. 1. Each method targets different aspects of model serving, such as reducing communication overhead and enhancing computational efficiency. However, a single strategy or a straightforward combination of them cannot obtain optimal inference throughput. Data Parallelism (DP) is a common parallel strategy in inference system scenarios. It accelerates inference by providing request-level or token-level parallel inference. DP can lead to significant increases in memory requirements for large MoE models, limiting its applicability in environments with constrained resources. Tensor Parallelism (TP) necessitates frequent synchronization of computation

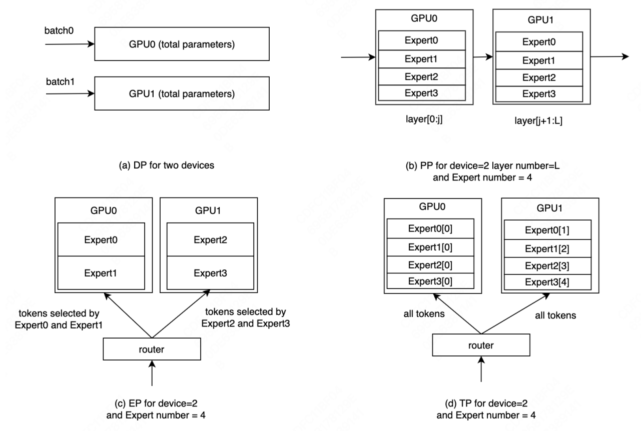


Fig. 1: Illustration of DP, TP, EP, and PP

results across devices, potentially increasing communication overhead, especially for large-scale MoE models. Pipeline Parallelism (PP) may introduce latency, as each stage's output is contingent on the completion of the preceding stage. Expert Parallelism (EP) excels at reducing unnecessary computations but introduces the need for inter-device communication, particularly when the number of experts is substantial.

To address these inference challenges from MoE architectures and go beyond these suboptimal solutions, we propose an EPS-MoE framework, a novel expert pipeline scheduler for efficiently serving MoE architectures. The framework consists of three main highlights. *1) DP+EP Parallel* With a theoretical analysis, we choose to apply DP on the Attention part since the Attention parameters usually account for a relatively small proportion of the total parameters. For the MoE part, we use EP for parallel computation, which can significantly save the I/O overhead of activation values, better than TP. We provide a detailed analysis in the following section. *2) Expert Pipeline Scheduler* The traditional *GroupGemm* mode for computing MoE results will submit all experts to a single kernel for matrix multiplication at once. We propose the expert pipeline scheduler to submit experts sequentially to a kernel for computation, which takes full advantage of *DenseGemm*'s throughput over *GroupGemm*. We overlap this sequential computation with a pipeline parallel to address the overhead. *3) Computation and Communication Overlapping* We also split the input by rows and only transmit the tokens required by the experts of the current stage, so that we can pipeline the computation and communication in parallel at the kernel level. We summarize the core contributions as follows:

- We introduce a novel expert pipeline parallel scheduler for efficient MoE model inference, which involves a fine-grained overlapping between computation and communication at the kernel level.
- Switching from *GroupGemm* to *DenseGemm* for expert computations improves computational efficiency.
- Extensive experiments on the benchmark demonstrate an average 21% throughput improvement over existing serving methods.

II. Related Work

A. MoE architectures

The groundbreaking work by Shazeer et al. [29] introduced the Sparsely-Gated Mixture-of-Experts (MoE) layer, which laid the foundation for scaling neural networks by utilizing a sparse activation pattern. This approach allows for efficient training of large models with a mixture of experts, where each expert is only activated for specific inputs. However, the original MoE layer suffers from challenges in balancing the load among experts and may lead to the underutilization of some experts. Fedus et al. [27] proposed GShard, a method to scale giant models by employing conditional computation and automatic sharding. GShard addresses some of the limitations of the original MoE by enabling dynamic routing and sharding of experts across different devices, thus improving scalability. Nonetheless, GShard may face difficulties in maintaining model coherence across shards and requires sophisticated infrastructure to manage the distributed computation. Switch Transformers by Fedus et al. [30] takes sparsity to the next level by introducing a simple and efficient sparsity pattern that allows scaling to trillion-parameter models. The Switch Transformers utilize a gating mechanism to activate experts based on the input data, which can significantly reduce computational overhead. However, the gating mechanism adds complexity to the model, and the benefits of sparsity may diminish as the model size increases.

Recently, several organizations have released their advanced MoE foundational models, such as Mixtral [1], DeepSeekV2 [2], Grok [3], Gemini 1.5 [4], achieving promising performance and inference efficiency. Beyond the traditional design with a sparse mixture of experts, there are some novel parallel designs on MoE architectures. ScMoE [20] introduces a novel shortcut-connected MoE architecture that adds the shortcut from dense MLP to the expert MLP, achieving a 70% to 100% overlap between communication and computation. Snowflake Arctic [5] [6] proposed to add a long shortcut for parallelism between the multi-experts branch and the whole dense branch (attention plus dense MLP). These designs significantly enhance both training and inference speeds. Nevertheless, the reliance on these specific network topologies for shortcuts might cause to achieve sub-optimal inference efficiency in some scenarios.

B. Disaggregated PD Serving

Disaggregated PD serving technology is a popular architecture recently, such as DistServe [10], PDServe [14], SplitWise [12], MoonCake [13], and others [11]. The tasks in the prefill and decode stages of LLM inference are different: prefill is a compute-bound task constrained by the TTFT [2] SLO, while decode is a memory-bound task constrained by the TPOT [3] SLO. If these two stages are disaggregated to run on different serving instances, each can be optimized based on its respective SLO, thereby fully utilizing machine resources.

[2]TTFT: Time To First Token
[3]TPOT: Time Per Output Token

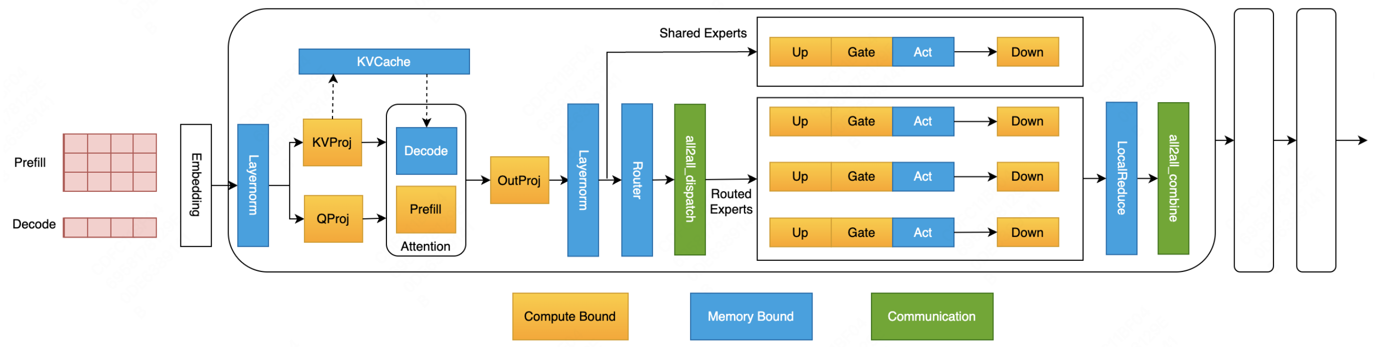


Fig. 2: MoE architecture. The operations in the yellow boxes are compute-bound, mostly *GEMMs*. The light blue box operations are memory-bound, such as *layernorm* and *activation*. The operations in the green boxes are communication operations.

For MoE models, since the per-token FLOPs are relatively low, the computational FLOPs required for the prefill stage are less; however, due to the large total number of parameters in MoE models, the memory-bound nature of the decode stage is more severe. The disaggregated architecture can address the load optimization issues of different stages in MoE models.

However, the disaggregated architecture cannot solve the issue of kernels' execution efficiency in MoE inference, nor can it address the problem of high communication overhead. As shown in Fig. 2, whether in the prefill or decode stage, there are many memory-bound and communicating kernels in their respective execution processes. Regardless of the increment in the computational density of a single stage, these memory-bound kernels cannot be completely eliminated, and therefore, the computational resource utilization of that stage cannot be fully improved. Additionally, both the prefill and decode stages face the overhead brought by communication. The more layers the model has or the more GPUs the model uses, the higher the proportion of communication time during inference.

C. *Operator-level Parallelism*

Operator-level parallelism is a solution that takes advantage of the load characteristics of operators to achieve parallelism. Nanoflow [15] implements pipeline parallelism between operators by splitting the input into finer-grained batches. Nanoflow correctly points out that the computational efficiency of operators and the number of SMs are not always linearly related; as computational efficiency increases, adding more SMs does not significantly improve computational efficiency. ScMoE [20] and Snowflake Arctic [5] [6] propose parallel solutions based on the model structure. Additionally, ScMoE implements kernel-level overlapping strategies for MoE to reduce communication-related overhead. However, these optimization methods do not design the optimal pipeline scheduling strategy based on the GEMM (General Matrix Multiply) characteristics of MoE, nor do they design parallel solutions that leverage the model characteristics of MoE. As we will see in the subsequent analysis, if operator-level parallelism does not fully utilize the characteristics of MoE, it can lead to repeated memory I/O of parameters. Furthermore, MoE inference is highly sensitive to load; ignoring load characteristics and implementing fixed scheduling strategies often fails to achieve the best overall results.

D. *Our Work*

We conducted an in-depth analysis of the performance of GEMM and communicating kernels during MoE inference. We proposed a pipeline scheduler where communication and computation can be overlapped. Based on the fact that the MoE model performs FFN calculations by splitting experts, we designed an expert-based data partitioning method to ensure no redundant parameter loading.

In the remaining chapters, Chapter 3 will introduce the issues of MoE model inference based on modern GPU architectures from three dimensions: computation, memory access, and communication. In Chapter 4, we will discuss the key points of EPS design, including the parallelization scheme of Attention DP + MoE EP, the input tensor partitioning mode, the expert pipeline scheduling strategy, and the overlapping design of communication and computation. In Chapter 5, we will analyze the core benefits of the proposed strategies through ablation experiments and validate the benefits of the proposed solution in multiple open-source model scenarios. In Chapter 6, we will summarize the entire paper.

III. Analysis

The inference latency of Large Language Models is influenced by various factors, and numerous studies have attempted to model this latency [24] [25] [26]. If we categorize the kernels in the inference process into three sets: S_{Comp}, S_{Mem}, and S_{Comm}, representing compute-bound kernels, memory-bound kernels, and communication kernels respectively, then the inference latency of LLM can be expressed as the cumulative sum of the latencies of each type of kernels, as listed below:

$$T_{LLM} = \sum_{k \in S_{Comp}} T_k + \sum_{k \in S_{Mem}} T_k + \sum_{k \in S_{Comm}} T_k \quad (1)$$

where T means time cost. Different types of workloads can also lead to changes in the sets of kernels associated with the prefill and decode stages. The traditional optimization approach maximizes the computational efficiency of each kernel within its respective types. For instance, if a kernel belongs to the S_{Comp} under a certain workload, ideally, it should fully utilize the hardware's computational FLOPs. Similarly, if a kernel belongs to the S_{Mem} under a certain workload, it should fully utilize the memory bandwidth. Achieving such an ideal scenario is extremely challenging. As we will see in the subsequent analysis, even the disaggregated architectures [10] [11] [12] [13] [14] that separate workloads into prefill and decode stages cannot completely resolve the issue of balanced efficiency at the kernel level. This is because both the prefill and decode stages are composed of kernels from multiple types of workloads, and this compositional relationship does not change with the disaggregation of prefill and decode.

As shown in Fig. 2, even if the prefill stage overall exhibits compute-intensive tasks and the decode stage overall exhibits memory-intensive tasks, each stage is still composed of kernels constrained by computation, memory I/O, and communication bandwidth.

Next, we will analyze the inference latency issues of MoE models from the perspectives of three types of kernels: computation, memory access, and communication.

A. Modern GPU Architecture

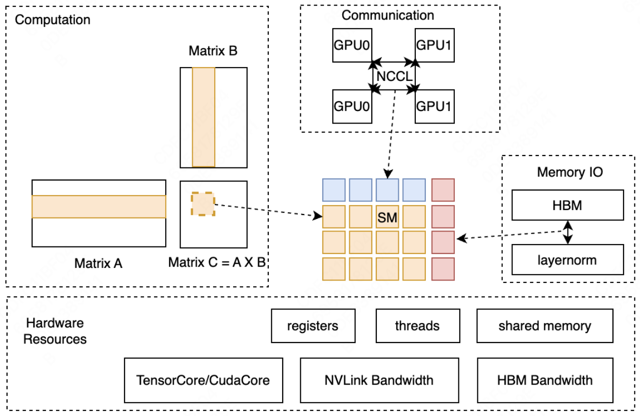


Fig. 3: Resources view of Nvidia GPU Architecture

We use NVIDIA A100-80GB SXM and H800-80GB SXM GPUs to do our tests. The matrix computation libraries used are mainly **cutlass** [22] and **cublas** [23]. NVIDIA's matrix computations are primarily conducted using a tiling strategy. The input matrix and weight matrix are divided into different tiles along rows or columns. Each block processes one or more tiles, and each block runs on a single SM (Streaming Multiprocessor) as shown in Fig. 3. The SM forms the basic computational unit. All hardware resources like Tensor Cores or CUDA Cores, memory bandwidth, and communication bandwidth are accessed through *SMs*.

B. Computation

The inference performance of the MoE model is significantly affected by the load scenario. Let E be the number of

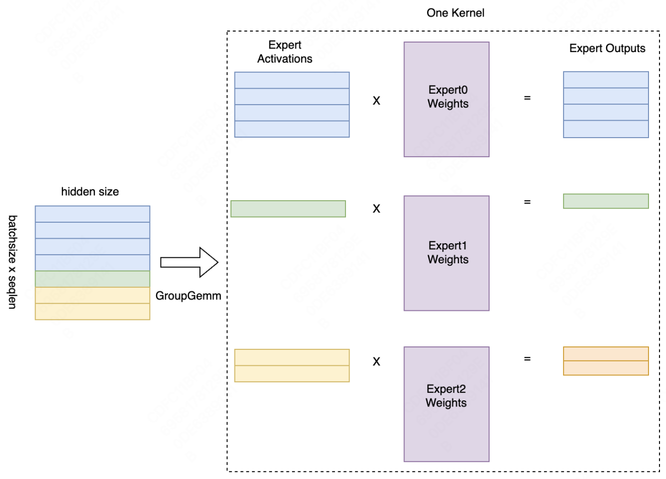


Fig. 4: *GroupGemm* demonstration. Adpoted from [17]. Each matrix represents an expert in the MoE model. All matrix multiplication operations are performed through a single kernel launch.

experts in the MoE model, and k be the number of experts selected by each token in a single forward pass. When the number of tokens in a single forward pass is m, the number of activated experts in the MoE model satisfies the following relationship:

$$ActivatedExperts = (1 - (1 - \frac{k}{E})^m) * E$$

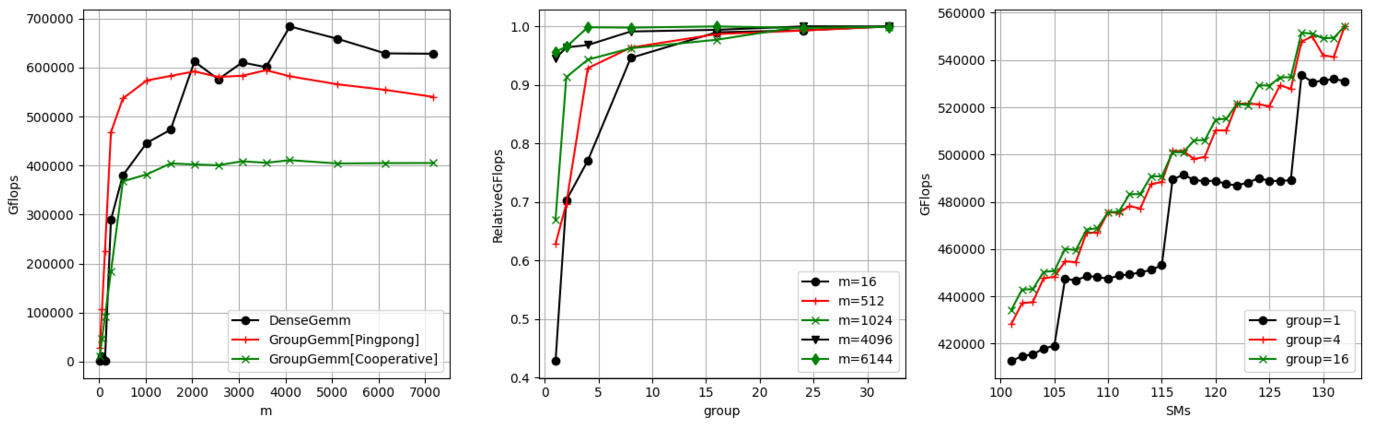
When m is relatively large, such as in the prefill stage, all experts in the MoE are activated. At this time, the MoE model inference is compute-bound. It can fully take advantage of having relatively small per-token FLOPs, thereby accelerating inference in the prefill stage.

When m is relatively small, such as in the decode stage, the MoE inference becomes memory-bound. As m increases within a certain range, the number of activated experts in the model also increases, leading to a higher amount of weight parameters that need to be loaded. Consequently, the computation time for decoding will significantly increase.

Therefore, the inference of a MoE model needs to consider the load conditions of its application scenarios. The inference optimization of the MoE should also take into account different load conditions to perform adaptive optimization.

MoE typically uses *GroupGemm* for computation, as shown in Fig 4 [17]. Generally speaking, *GroupGemm* is a relatively efficient kernel. However, we have found that as the computation load increases, the performance of the *GroupGemm* kernel does not remain constant. We conducted some tests on *GroupGemm* and *DenseGemm* based on real model settings, as shown in Fig 5, and derived several important conclusions.

Conclusion 1: As the input problem size changes, the computational efficiency of *GroupGemm* and *DenseGemm* also varies, and their relative advantages differ across different ranges.



(a) Gemm throughput changes with problem size m for different gemm implementations

(b) Gemm throughput changes with groups for different m

(c) Gemm throughput changes with SM count for different groups

Fig. 5: GEMM profiling data. (a) We tested different GEMMs' throughput with different input problem sizes. In this test, we only calculated the *Gate* and *Up* matrices from MoE blocks. For *GroupGemm*, the number of Experts or Groups is set to 16, and the dimensions of the *Gate* and *Up* matrices are set to $[1536, 5120]$. m represents the number of per expert's input tokens for the *GroupGemm* matrix. This test strictly ensures that the total computation of *GroupGemm* and *DenseGemm* is consistent. *GroupGemm* was implemented by *cutlass*, *DenseGemm* was implemented by *cublas*. (b) In this test, we set GEMM to *GroupGemm [Pingpong]* and tested GEMM's relative throughput with different groups for different given problem sizes. Relative throughput represents the ratio of the computation throughput of the current Group to the maximum computation throughput for the current input problem size. (c) In this test, we set GEMM to *GroupGemm [Pingpong]* and set input problem size $m = 6144$, and tested GEMM's throughput with different SM counts for different given groups.

As shown in Fig. 5(a), let m be the input problem size of a given expert, the computation efficiency of *GroupGemm* and *DenseGemm* gradually improves as the input m increases. After reaching saturation, it no longer changes. Furthermore, when $m > 3584$, the computation efficiency of *GroupGemm* will inevitably be lower than that of *DenseGemm* with the same computation load. When $m \in [2048, 3584]$, the efficiency of *GroupGemm* is roughly equivalent to that of *DenseGemm*; when $m < 2048$, the efficiency of *GroupGemm* is higher than that of *DenseGemm*. This indicates that **the optimal kernel implementation changes with the input scale**. In the inference process of MoE models, variations in input load are common. For example, during the prefill stage, the input often falls within the range where *DenseGemm* is faster; whereas, during the decode stage, the input usually falls within the range where *GroupGemm* is faster.

We also found that different *cutlass* scheduling strategies have a significant impact on kernel throughput. As shown in Fig. 5(a), the *Pingpong* scheduling strategy [22] is superior to the *Cooperative* scheduling strategy. This is because the *Pingpong* scheduling better overlaps memory I/O and computation.

Conclusion 2: For *GroupGemm*, once the input size reaches a certain number, having more groups does not lead to higher throughput.

Increasing the number of groups in *GroupGemm* has always been a method to improve computational throughput, especially in the fine-grained MoE model. This approach always works when the input size is relatively small. However, when the input size is relatively large, having more groups does not further increase computational throughput, as shown in Fig. 5(b). Therefore, when the input size is relatively large, splitting a larger *GroupGemm* into smaller *GroupGemms* does not lead to a decrease in throughput.

Conclusion 3: For *GroupGemm*, once the input size reaches a certain number, using more SMs does not lead to higher throughput.

Nanoflow [15] has pointed out that for *DenseGemm*, the number of SMs it occupies can be reduced within a certain range without affecting GEMM's computational throughput. For *GroupGemm*, we can observe a similar trend from Fig. 5(c). Additionally, the tested results also show that for *GroupGemms* with different numbers of groups, the range of SMs at maximum computational throughput is essentially consistent. This means that **splitting a larger *GroupGemm* into multiple smaller *GroupGemms* for serial execution does not lead to a decrease in total computational throughput**. Furthermore, reducing the number of SMs occupied by GEMM within a certain range also does not affect GEMM's execution efficiency.

C. Memory Access

During MoE inference, there are many memory-bound operators, such as *layernorm*, *residual*, *activation*, *topKGating*, etc. For these memory-bound operators, traditional optimization methods include:

- Kernel fusion: Increasing the computational density of a single operator by fusing operators, such as fusing the computation of bias and residual.

- Vectorized memory access: Improving bandwidth and reducing the total number of instructions by vectorizing memory access [16].

In principle, memory-bound kernels also require more SMs to achieve optimal performance. However, practical tests on the *silu_activation* operator show that while the execution time decreases with an increase in SMs, the performance gains become negligible for MoE inference once the SM count exceeds a certain threshold. As illustrated in Fig. 6, this threshold is closely related to the operator's computational load. When the number of tokens is 64, 40 SMs are sufficient to achieve good kernel performance. When the number of tokens is 128, 256, or 2048, 60 SMs perform adequately.

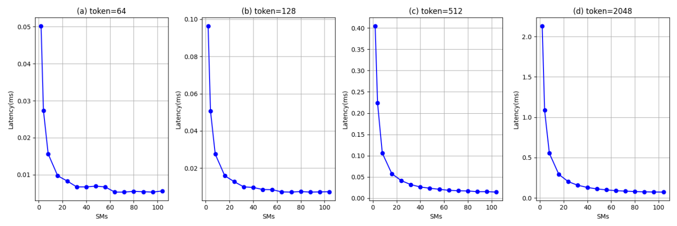


Fig. 6: The latency with different load and SMs of a *silu_activation* kernel, which is memory-bound. (a) Tensor [64,8192]. (b) Tensor [128, 8192]. (c) Tensor [512, 8192]. (d) Tensor [2048, 8192]. For (a), When the number of SMs exceeds 40, the absolute value of performance degradation becomes negligible for the overall LLM inference. For (b),(c), and (d) when the number of SMs exceeds 60, the absolute value of performance degradation becomes negligible for the overall LLM inference.

D. Communication

Communication is a common issue in large model inference because the model size is often too large to be deployed by a single GPU. On the other hand, multi-GPU collaborative inference actually increases the computational resources and memory bandwidth required for single-token generating. If two GPUs are used, the computational and memory resources utilized during each decode process are doubled. However, multi-GPU parallel computation also introduces additional communication overhead between the GPUs.

From Table I, it can be seen that whether in prefill or decode, the proportion of time spent on communication increases with more GPUs. To reduce communication time, we can use FP8 quant to decrease the communication volume, striking a balance between precision and latency. However, this does not completely solve the problem; as batch size increases, the communication volume will further increase. Moreover, as the model size grows, multi-machine parallel strategies will be introduced, where the bandwidth between machines will be much lower than that between GPUs in a single machine. At this point, the proportion of time spent on communication will become even more significant.

Communication consumes SM resources, but we have observed that within a certain range, adjusting the number of SMs allocated to communication kernels has almost no impact on communication throughput, as shown in Fig. 7. Even in scenarios with high communication volumes such as in prefill, when the communication volume reaches 32MB per card, the communication time for 10 SMs is 223.6 ms, while for 30 SMs it is 199.2 ms. The number of SMs is doubled, while the communication time is reduced only by 11%.

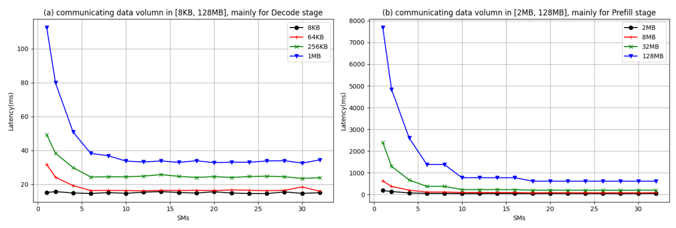


Fig. 7: The latency of *all2all* kernel with different load and SMs. For decoding, using 10 SMs achieves the expected performance under almost all workloads. For prefill, 10 to 20 SMs are sufficient, increasing the number of SMs further has virtually no effect on latency. Tested on NVIDIA A100-80GB SXM with NVLink.

IV. Design

In this section, we take the DeepSeekV2 model as an example to discuss the system design of EPS-MoE. The DeepSeekV2 model is a fine-grained MoE model with outstanding performance in both effectiveness and inference. We use this as a typical case to explore inference optimization methods for MoE models. As [21] has pointed out, the more granularity an MoE model has, the better loss it will achieve. And our method will take a lot of advantage of the granularity of an MoE model.

A. Overview of EPS-MoE

Fig. 9 shows how EPS-MoE works. It demonstrates the expert-level pipeline with communication and computation overlapping. This strategy is based on Attention DP + MoE EP. As a result, EPS-MoE consists of three core modules:

a) DP+EP: DP is deployed for the Attention part. For the DeepSeekV2 model structure, its Attention part introduces MLA to reduce KVCache overhead significantly, and we hope to keep this feature. Using TP for the MLA structure would introduce an extra communication overhead while using DP would result in redundant parameter storage. However, for the latter, since the parameters in the Attention part account for a small proportion, approximately 0.05% of the total parameters, this can be considered negligible. For the MoE part, we use EP for parallel computation. Compared to TP, the advantage of EP is that it can significantly save the I/O overhead of activation values which we will discuss later.

b) Expert Pipeline Scheduler: The traditional *GroupGemm* for computing MoE will submit all experts to a single kernel for matrix multiplication at once. The Expert Pipeline Strategy submits experts sequentially to a kernel for computation. The advantage of this approach is that it

TABLE I: The time consumption proportions of communication, computation, and other kernels in different models and at different concurrency granularities.

in=1024, out=128	prefill			decode		
H800-80GB SXM	all2all	MoE layer	Attn layer	all2all	MoE layer	Attn layer
DeepSeekV2 (bs=256, EP=8)	33.34%	22.22%	44.44%	6.31%	58.56%	35.13%
Mixtral7x8 (bs=128, EP=2)	5.62%	69.11%	25.27%	4.3%	61.5%	34.2%
Mixtral7x8 (bs=128, EP=4)	11.72%	64.43%	23.85%	8.3%	51.2%	37.5%
Mixtral7x8 (bs=256, EP=4)	11.3%	65.0%	23.7%	6.5%	47.8%	45.7%
Mixtral7x8 (bs=512, EP=8)	15.8%	60.8%	23.4%	9.1%	29.3%	61.6%

takes full advantage of *DenseGemm* or *GroupGemm* with smaller groups. However, this will make traditional parallel computation become serial computation. Therefore, we need some pipeline parallel strategies to address the overhead of serial computation.

c) Computation and Communication Overlapping: We split the input by rows and only transmit the tokens required by the current experts to the corresponding device each time. By this method, we pipeline the computation and communication in parallel. There are various ways to overlap computation and communication. Fine-grained parallelism at the kernel level is a more general optimization strategy in MoE models.

B. Parallel Strategy

The MoE model mainly consists of two parts: Attention and MoE. From this point, the inference time of an MoE model can be described as below:

$$T_{LLM} = (T_{Attention} + T_{MoE} + T_{Comm}) * L$$

Attention can use TP (Tensor Parallelism) or DP (Data Parallelism), while MoE can adopt TP or EP (Expert Parallelism). Therefore, we have a lot of parallel strategies for MoE inference. We will discuss Attention TP + MoE TP and Attention DP + MoE EP in detail. And other parallel strategies can be derived from these discussions.

a) TP + TP: The Attention part uses the TP mode, and the MoE part also uses the TP mode. The time cost of Attention and MoE depends on the maximum memory access time and computation time.

$$T_{Attn} = max\{\frac{F(Attn)}{D*F}, \frac{W(Attn)}{D*B}\}$$

$$T_{MoE} = max\{\frac{F(MoE)}{D*F}, \frac{W(MoE)}{D*B}\}$$

$$W(Attn) = M_{in} + M_{out} + \frac{[M_{KVCache}]}{D} + \frac{M_{weight}}{D}$$

$$W(MoE) = M_{in} * k + M_{out} * k + \frac{M_{weight}}{D}$$

The function $F(\cdot)$ represents the computation FLOPs of a certain block, and the function $W(\cdot)$ represents the memory I/O data volume of a certain block, including parameters, activations, and KVCache. M_{in} and M_{out} represent the data volume of the input and output activations on each device respectively. $M_{KVCache}$ represents the total volume of the KVCache of all devices, which is 0 during the prefill stage, and M_{weight} represents the data volume of all the parameters. k represents the number of experts selected by each token in the MoE model. D represents the device number of tensor parallel. F and B refer to the computational FLOPS and memory bandwidth of the hardware, respectively.

As we can see, if the top-k of MoE is large, the impact caused by the activation value I/O is significant. Examine the communication overhead in the TP+TP mode. Since all tokens are present on each device and *ncclAllreduce* communication is used, the communication volume of TP+TP is:

$$V = 2P(D-1)$$

where P is the size of the activation value on a single device.

b) DP + EP: The problem with using TP for the Attention part is that when Attention applies the MLA structure, extra communication needs to be introduced. Therefore, we use DP to accelerate the Attention part. At this point, the time cost of Attention and MoE can be expressed as follows.

$$T_{Attn} = max\{\frac{F(Attn)}{D*F}, \frac{W(Attn)}{D*B}\}$$

$$T_{MoE} = max\{\frac{F(MoE)}{D*F}, \frac{W(MoE)}{D*B}\}$$

$$W(Attn) = \frac{M_{in} + M_{out} + [M_{KVCache}]}{D} + M_{weight}$$

$$W(MoE) = \frac{M_{in} * k}{D} + \frac{M_{out} * k}{D} + \frac{M_{weight}}{D}$$

D represents the device number of expert parallel. It can be seen that DP+EP has little difference compared to TP+TP in terms of computation for Attention and MoE. However, there are pros and cons in terms of memory I/O.

1. TP+TP mode has an advantage in Attention weights memory I/O data volume: DP has the issue of redundant I/O for the parameters in the Attention part, leading to a larger I/O data volume for the weights part compared to TP. For most MoE models, since the parameters in the Attention part do not account for a high proportion, this part of the time consumption can be ignored.

2. DP+EP mode has an advantage in MoE memory I/O data volume: Since tokens are split along the row dimension,

the I/O data volume of the output values of the matrix multiplication in EP mode will be much smaller than in TP mode. This is especially significant when the k value of MoE is large.

For DP+EP, the time consumption of the communication part mainly depends on how the experts are deployed. If all experts selected by a token are on the same device, the communication volume is optimal. However, if all experts selected by a token are distributed across different devices, there will inevitably be an amplification phenomenon in communication. So, DeepSeekV2 has designed a device-limited routing mechanism to restrict MoE-related communication costs [2]. Let g be the device number one token will be routed to, g is bounded by $\min(k, D)$, i.e. $g \leq \min(k, D)$. Therefore, the total data volume of communication for DP+TP is listed below:

$$\frac{2P}{D}(D-1) \leq V \leq g\frac{2P}{D}(D-1)$$

In the worst-case for a normal MoE model where $g = D$, when the experts selected by a token are distributed across all GPU devices, the communication volume of the MoE model $V(DP+EP) = 2P(D-1)$, which equals the communication volume of TP + TP. Therefore, it can be concluded that the communication volume of DP + EP is less than that of TP + TP.

$$V(DP+EP) \leq V(TP+TP)$$

The communication time of DP + EP is greatly affected by the balance of Experts. This can be resolved by methods such as balance loss or token drop [2] [27] [28].

C. Expert Pipeline Scheduler

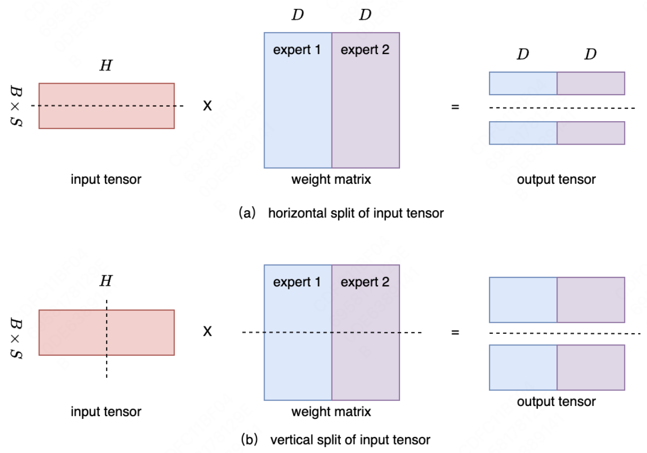


Fig. 8: Two different ways to split input tensor, horizontal split and vertical split. (a) Horizontal split. Splitting rows of input tensor. Weights split by experts. (b) Vertical split. Splitting cols of input tensor. Weights split by cols. B for batch size, S for sequence length, H for hidden size of input tensor, D for output dim.

There are two data partitioning methods that can achieve pipeline parallelism for computation and communication in MoE models. As shown in Fig. 8, horizontal split organizes the input data by row partitioning like applying DP on the input tensor, while vertical split organizes the input data by column partitioning like applying TP on the input tensor.

The two pipeline parallelism methods result in different amounts of I/O. Let the number of samples be m, the size of a single input activation be P_0, the size of a single output activation be P_1, the size of each Expert parameter be W, the total number of experts be E, and the number of pipelines be N. The total I/O data volume V is given by:

$$V(a) = m \cdot P_0 + E \cdot W + m \cdot P_1$$

$$V(b) = m \cdot P_0 + E \cdot W + N \cdot m \cdot P_1$$

Method (a) results in much less I/O data volume compared to method (b).

Let the computational workload of a single expert be C, and the computational throughput of *GroupGemm* with group size G be $R(\text{FLOPS}|G)$. The computational time difference between the two methods is:

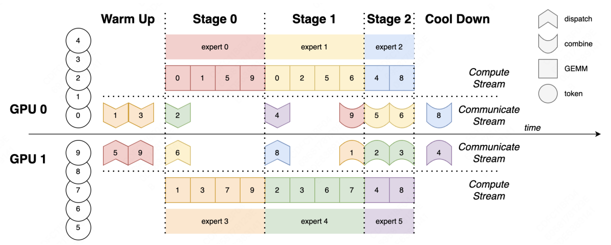
$$T(a) = \frac{C \cdot E/N}{R(\text{FLOPS}|E/N)} \cdot N = \frac{E \cdot C}{R(\text{FLOPS}|E/N)}$$

$$T(b) = \frac{C \cdot E/N}{R(\text{FLOPS}|E)} \cdot N = \frac{E \cdot C}{R(\text{FLOPS}|E)}$$

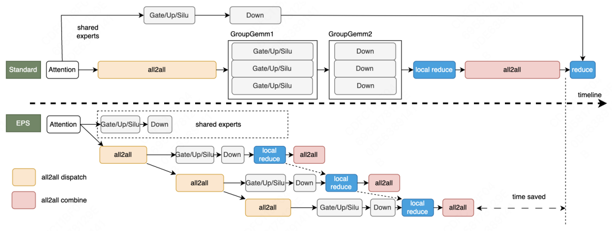
The final difference is reflected in the computational efficiency of *GroupGemm* with different group sizes. When $E = N$, *GroupGemm* can be replaced by *DenseGemm* to achieve better performance for larger input m. We can see that the core advantages of using a horizontal split are a more efficient GEMM implementation and a lower memory I/O data volume. Therefore, the Expert Pipeline Scheduler implements pipeline scheduling by horizontally splitting the input tensor. Meanwhile, the weights are partitioned according to the experts, and each time only the tokens required by a specific group of experts are transmitted.

From the previous analysis data, we can see that the efficiency difference between *GroupGemm* and *DenseGemm* is completely opposite under different computational loads. For example, in the case of the DeepSeekV2 model, *GroupGemm* has an efficiency advantage when the number of tokens is small, while *DenseGemm* has an efficiency advantage when the number of tokens is large. Based on this, we designed a load-aware adaptive scheduling strategy to dynamically select different efficient implementations based on the type of load, as shown in Algorithm 1.

EPS-MoE (Expert Pipeline Scheduler) is based on a parallel strategy where the Attention part uses DP (Data Parallelism) and the MoE part uses EP (Expert Parallelism). DP on the Attention part will bring much more advantage from the horizontal split. Through this parallel strategy, we can relatively easily establish parallelism of token-level transmission and



(a) Expert Pipeline Scheduler. Expert number=6, topk=2, DP=2, EP=2. The outputs of Attention Blocks on GPU0 and GPU1 are token [0,1,2,3,4] and token [5,6,7,8,9] respectively. Expert0 takes token [0,1,5,9], Expert1 takes token [0,2,5,6], and so on.



(b) Overlapping of communication and computation. *LocalReduce* is memory-bound operation. When the computation for all experts of a token is finished, the next round of all2all communication for this token can be initiated.

Fig. 9: Illustration of Expert Pipeline Scheduling.

TABLE II: Performance Comparison of *GroupGemm* and *DenseGemm* with Different Pipeline Numbers and Overlapping Strategies

ID	PN	GEMM	Overlapping	FP8	SM	m=3072		m=1024		m=256	
						5120,1536	5120, 15360	5120,1536	5120, 15360	5120, 1536	5120, 15360
0	1	GroupGemm	N	N	132	5.698	32.469	2.120	8.996	1.032	2.349
1	1	GroupGemm	N	Y	132	4.330	30.980	1.512	8.516	1.513	**2.203**
2	1	DenseGemm	N	N	-	5.620	21.247	3.889	7.497	0.868	2.708
3	5	GroupGemm	Y	N	132	4.837	25.979	1.790	8.333	0.683	2.452
4	5	GroupGemm	Y	N	116	4.248	26.270	1.568	8.808	0.636	2.461
5	5	GroupGemm	Y	Y	132	3.728	24.833	1.418	7.996	0.607	2.358
6	5	GroupGemm	Y	Y	116	3.277	25.928	**1.299**	8.697	**0.597**	2.398
7	5	DenseGemm	Y	N	-	4.299	20.203	1.714	7.128	0.798	2.598
8	5	DenseGemm	Y	Y	-	**3.111**	**19.795**	1.438	6.895	0.727	2.549
9	20	DenseGemm	Y	N	-	4.855	20.599	2.133	7.043	1.016	2.655
10	20	DenseGemm	Y	Y	-	3.465	20.012	1.715	**6.893**	0.977	2.619

[1] PN: Pipeline Number.
[2] FP8 means whether the communication datatype is FP8 or not.
[3] SM means the SM count of GEMM. We do not control the SM count of DenseGemm.
[4] 5120, 1536 means the input dim of GEMM is 5120 and the output dim of GEMM is 1536.
[5] All data units are in milliseconds (ms).
[6] Tested on 4xH800-80GB SXM. We only tested one layer. The computation consists of Gate, Up, and Down GEMM.

computation, as shown in Fig. 9a. In the first communication pipeline, tokens 1 and 3 selected by expert3 on device0 are transmitted to device1, and tokens 5 and 9 selected by expert0 from device1 to device0.

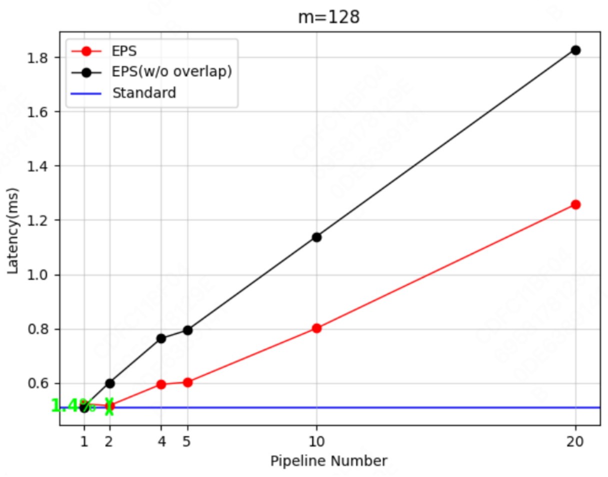
D. Computation and Communication Overlapping

Pipeline parallelism is a common strategy to improve resource utilization when a task requires different types of hardware resources. [9] Generally, when multiple hardware resources work together, pipeline parallelism can be used to enhance overall throughput. In the context of LLM inference, some kernels are memory-bound, some are compute-bound, and some are communication-bound. However, the hardware resource usage for each type of kernel requires the support of SM (Streaming Multiprocessor) resources. Since some kernel's execution may require both memory access and computation, while others may require both memory access and communication, no single type of kernel can utilize all types of hardware resources to their theoretical maximum. From the previous analysis, we can infer that by controlling the number of SMs occupied by some kernels, we can leave a portion of SMs for other kernels while keeping the kernel's execution time unchanged. This is a fundamental prerequisite for implementing pipeline parallelism.

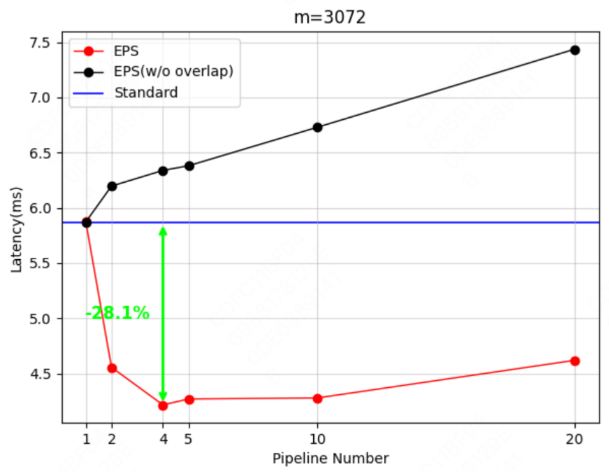
In EPS (Expert Pipeline Scheduler), we found that both *GroupGemm* and *DenseGemm* can achieve the same latency by reducing the SM number a little. Therefore, we parallelized the computation of MoE (Mixture of Experts) and the *all2all* communication to improve hardware resource utilization. As shown in Fig. 9b, the shared experts are used to overlap the first *all2all* communication, which is labeled as *all2all_dispatch*. The *LocalReduce* is used to overlap the second phase *all2all* communication, which is labeled as *all2all_combine*. In the subsequent tests, we will see that pipeline parallelism can achieve benefits even without controlling the number of SMs occupied by the kernel. This indicates that the current number of SMs has some redundancy even for the optimal kernel implementation. If we control the number of SMs occupied by computation and communication, the pipeline can achieve even greater benefits.

The pipeline number is crucial for the acceleration of EPS, as shown in Fig. 10. Different pipeline numbers can lead to completely different results. From Fig. 10, we found that:

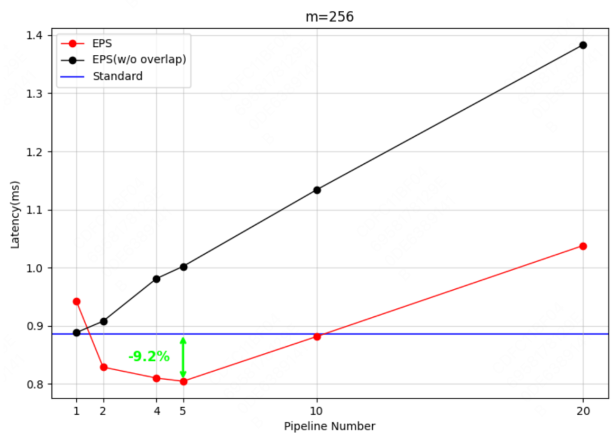
- Given a certain inference task load and model computa-



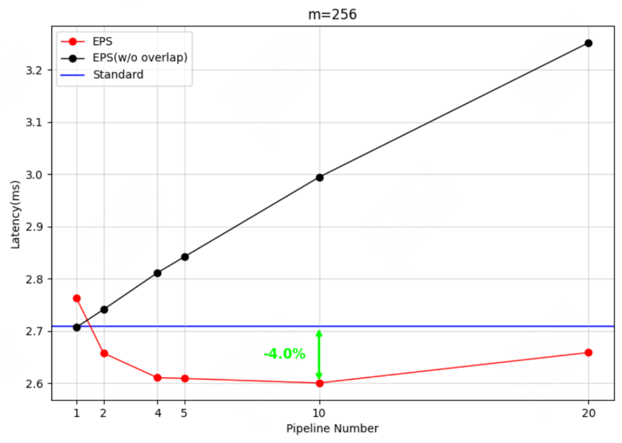
(a) GEMM size is [5120, 1536]



(b) GEMM size is [5120, 1536]



(c) GEMM size is [5120, 1536]



(d) GEMM size is [5120, 15360]

Fig. 10: The latency of one MoE kernel with different pipeline numbers. EPS (w/o overlap) is a strategy that will split communication and computation into chunks and process communication and computation serially. This will show us the overhead of communication. Tested on 4xH800 SXM nvlink gpus. The MoE layer tested in this figure consisted of Gate, Up, and Down GEMM. For (a),(b), and (c), the input dim of Gate and Up is 5120, and the output dim is 1536, which is the same as DeepSeekV2. For (d), the input dim is 5120, and the output dim is 15360, which is 3x of the input dim in the Llama model. In these tests, the overlapping strategy was only applied to *all2all_dispatch* and GEMM for simplicity.

tion cost, there exists an optimal Pipeline Number that maximizes the benefits of the pipeline parallel.

- When the number of tokens is relatively small, it is unlikely to achieve benefits regardless of how the pipeline is divided.

Different levels of granularity in chunks of input tensor will lead to changes in the ratio of communication and computation time. At the same time, due to some fixed overhead in communication, when the number of tokens is small, the benefits of the pipeline are decreased by the communication overhead. When the communication overhead is rarely small as in Fig. 10b, the benefit of EPS will be much more significant. Otherwise, EPS works inconspicuously as the overhead is too big. The pipeline number is influenced by a combination of factors such as hardware, kernel implementation, model parameters, and inference task load. During the optimization process for the DeepSeekV2 model, we first identified the optimal pipeline number through test data analysis and then applied it to our scenario.

We can also establish a method through analysis to adaptively solve for the optimal pipeline number. We analyze the optimal pipeline number setting based on the time consumption of a single layer. Let θ represent the model parameters, N represent the number of pipeline splits, and E represent the number of experts on a single device, such that $1 \leq N \leq E$. Let function $L(\theta; N)$ represent the theoretical maximum benefit after overlapping computation and communication, and $R(N)$ represent the degradation caused by the fixed communication overhead as the number of pipelines increases. We determined from our test results that the fixed communication overhead is linearly related to the pipeline number, which means $R(N) \approx kN + b$. Let T_{comm} denote the total communication time before splitting the pipeline, and T_{comp} denote the total computation time of the MoE layer before splitting. Finding the optimal pipeline number can be formulated by solving the following equation:

$$\underset{1 \leq N \leq E}{argmax} \quad L(\theta; N) - R(N)$$

where,

$$L(\theta; N) = \min \left\{ \frac{T_{\text{comm}}}{N}, \frac{T_{\text{comp}}}{N} \right\} \cdot (N-1).$$

Let $C = \min\{T_{\text{comm}}, T_{\text{comp}}\}$, and G for total latency gain for overlapping. Note that we assume both communication and computation can be linearly partitioned to chunk here, which is a very ideal assumption. The actual test results may differ from this because of the overhead of splitting the communication and computation. However, we aim to establish the analytical framework based on this ideal assumption first and study the overhead in future work.

$$\begin{aligned} G &= C - b - \left(\frac{C}{N} + kN \right) \\ &\leq C - b - 2\sqrt{k \cdot C} \end{aligned}$$

The maximum latency benefit is achieved if and only if $N = \sqrt{\frac{C}{k}}$. The existence of fixed communication overhead implies that there is an optimal pipeline number to achieve the maximum effect of the overlapping strategy. Otherwise, the more pipelines there are, the better the overlapping effect should be.

V. Experiments

TABLE III: Prefill Benefit for Different Sequence Lengths and Pipeline Numbers

Prefill Benefit \seq_len	1K	2K	4K	8K
DeepSeekV2 [PN=20]	+21.27%	+18.05%	+10.00%	+11.60%
DeepSeekV2 [PN=5]	+21.81%	+20.53%	+16.89%	+12.75%
DeepSeekV2 [PN=1]	+11.83%	+11.02%	+6.77%	+5.13%

[1] PN: Pipeline Number.
[2] Tested on 8xH800-80GB SXM. All tests are without FP8 communication.

A. Experiments on DeepSeekV2

We tested the effect of EPS on the DeepSeekV2 model on 8xH800-80GB SXM. In Table III, we present the prefill throughput benefits of the EPS strategy on the DeepSeekV2 model for Pipeline Numbers (PN) of 1, 5, and 20 respectively. It can be seen from the table that the throughput benefit can reach up to 21%. Given that DeepSeekV2 has already provided a relatively high baseline (with a reported prefill throughput of 100,000 tokens/s), this improvement can be considered a significant result.

B. Ablations

1. Contributions of Overlapping and GEMM switching. Switching GEMM and overlapping both yield certain benefits, especially when the number of input tokens is large. From Table II, it can be seen that when the input size is relatively large, such as when $m = 3072$ and GEMM size is [5120, 1536], if switching GEMM from *GroupGemm* to *DenseGemm* can bring a 12% benefit (from 4.837 ms in data [ID=3] to 4.299 ms in data [ID=7]), while the benefit of overlapping is 16% (from 5.698 ms in data [ID=0] to 4.837 ms in data [ID=3]). When the input size is small, the benefit from switching GEMM is smaller and may even be negative, with the main benefits coming from the pipeline strategy. It is important to note that when the input size is small and GEMM is memory-bound, EPS (w/o overlapping) has a slight advantage over the overlapping setting. This is because the communication overhead is much heavier as discussed before.

2. Does FP8 communication reduce the benefits of EPS? From Table II, it can be seen that FP8 communication and the EPS strategy are not in conflict, and in most cases, using both FP8 communication and the EPS strategy together can yield further benefits. For example, considering $m = 3072$ and GEMM size is [5120, 1536], comparing the data [ID=2] and data [ID=8], both of whose communication data type is set to be FP8, the latter still shows a 28.2% improvement over the former.

3. The necessity of controlling the number of SM.

TABLE IV: Time cost under different SM

$m \backslash SM$	132	124	116	108	100	92
3072	4.597	4.403	**3.894**	3.967	4.031	4.081
1536	2.385	2.336	**2.079**	2.130	2.162	2.054
768	1.306	1.268	**1.170**	1.180	1.196	1.167
384	0.894	0.921	**0.787**	**0.780**	0.818	0.797
192	0.591	0.596	**0.540**	0.541	0.591	0.797

[1] SM means the SM count of GEMM. m represents the input count of GEMM per expert.
[2] All data units are in milliseconds (ms).
[3] Tested on 4xH800-80GB SXM. We only tested one layer. The computation consists of Gate, Up, and Down GEMM. EPS is applied to this test.
[4] We have set the SM numbers of the communication kernel to be 16, so the best SM numbers for computation in these tests are all 116.

It is important to control the number of SM used by computation kernels when we apply EPS, as we can see from both Table II and Table IV. Controlling SM is mainly about managing the number of SMs occupied by computation when communication and computation overlap. The goal is to allocate more SMs to communication operators without affecting computation efficiency. The number of SMs allocated is crucial for the time consumption of both computation and communication. When the GEMM operator occupies all SMs, the communication operator cannot be scheduled, slowing down its execution. Additionally, the scheduling of the communication operator impacts the GEMM computation, increasing its execution time as well. In such cases, the benefits of overlapping are reduced. However, by controlling the number of SMs occupied by the GEMM operator, communication and computation operators can be executed in parallel without interfering with each other, achieving perfect overlap.

VI. Conclusion

The MoE (Mixture of Experts) model has already demonstrated its powerful capabilities in large language models. More and more outstanding MoE models are being released or are in the process of training. MoE has significant computational advantages, and increasingly more MoE models are opting for higher sparsity architectures. The inference cost issue due to the enormous number of parameters is not easy to ignore. We conducted an in-depth analysis of kernel performance during the inference process of MoE models and found that the resources consumed by kernels in MoE model inference include computation resources, memory bandwidth, and communication bandwidth. Each type of resource is controlled based on the Streaming Multiprocessors (SMs). Further analysis revealed a marginal diminishing effect between the number of SMs used and the performance of the kernel, meaning that once the kernel performance reaches a certain level, adding more SMs does not result in a perceptible performance improvement. However, if these SMs are used for other parallelizable operations, a significant end-to-end performance improvement can be achieved. Based on this understanding, this paper studies the high-performance inference architecture of MoE models and designs a parallel scheme of Attention Data Parallelism (DP) + MoE Expert Parallelism (EP). To achieve overlapping pipelines of MoE model communication and computation, we designed the Expert Pipeline Scheduler (EPS) mechanism and developed a load-aware adaptive scheduling algorithm to dynamically choose the best pipeline setting based on different load scenarios. The architecture designed in this paper achieved an additional 21% prefill throughput improvement in the DeepSeekV2 scenario, even on a relatively high baseline. Experimental results demonstrate the effectiveness of the proposed solution and provide a new approach for further optimization of MoE inference.

Acknowledgment

The system designed in this paper was tested in various scenarios with the help of many colleagues. We could not include all the names we should acknowledge but only a few: Xianglong Lu from Nvidia and Shuai Du from Meituan.

Algorithm 1: Load-aware Adaptive Expert Pipeline Scheduler Algorithm

```
Data: input: total input tensor
  $m$: token numbers for MoE input
  $k, n$: MoE input dim and output dim
  $ep$: expert parallel number
  $topk$: selected expert numbers of a token
  $e$: expert number
  $PN$: pipeline number
  tensor[0:PN-1]: activation results for each pipeline
  selected_experts[0:PN-1]: selected experts for each pipeline
  stream[0:PN-1]: streams
1  Function ComputeMoE(tensor, experts, stream):
2      GateUpGemm(tensor, experts, stream);
3      SiluAct(tensor, experts, stream);
4      DownGemm(tensor, experts, stream);
5      LocalReduce(tensor,stream);
6  Function LoadAwareAdaptiveScheduler(input):
       /* split tokens by expert */
7      $index \leftarrow$ Router(input);
8      $m \leftarrow$ count(input);
9      tensor[] $\leftarrow$ split(input, $index$);
       /* communication for the first pipeline */
10     $p \leftarrow 0$;
11     All2All(tensor[p], stream[p]);
12     for $p \leftarrow 1$ to $PN$ do
13         if $p \leq PN - 1$ then
14             All2All(tensor[p], stream[p]);
15         ComputeMoE(tensor[p-1], selected_experts[p-1], stream[p-1]);
16         if $p - 2 \geq 0$ then
17             All2All(tensor[p-2], stream[p-2]);
18     $p \leftarrow PN - 1$;
19     All2All(tensor[p], stream[p]);
```

References

[1] Mixtral: https://huggingface.co/mistralai/Mixtral-8x7B-v0.1
[2] Shao, Zhihong et al. "DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model." ArXiv abs/2405.04434 (2024): n. pag.
[3] Grok: https://x.ai/blog/grok-2
[4] Gemini1.5 tech report: https://storage.googleapis.com/deepmind-media/gemini/gemini_v1_5_report.pdf
[5] Snowflake Arctic: The Best LLM for Enterprise AI — Efficiently Intelligent, Truly Open. https://www.snowflake.com/en/blog/arctic-open-efficient-foundation-language-models-snowflake/
[6] Merrick, Luke, et al. "Arctic-Embed: Scalable, Efficient, and Accurate Text Embedding Models." arXiv preprint arXiv:2405.05374 (2024).
[7] https://huggingface.co/xverse/XVERSE-MoE-A36B
[8] Wang, An, et al. "HMoE: Heterogeneous Mixture of Experts for Language Modeling." arXiv preprint arXiv:2408.10681 (2024).
[9] Zhao, Weijie et al. "Distributed Hierarchical GPU Parameter Server for Massive Scale Deep Learning Ads Systems." ArXiv abs/2003.05622 (2020): n. pag.
[10] Yinmin Zhong, et al. Distserve: Disaggregating prefill and decoding for goodputoptimized large language model serving. In Proceedings of USENIX Symposium on Operating Systems Design and Implementation, OSDI, 2024.
[11] Hu Cunchen, Huang Heyang, Xu Liangliang, et al. Inference without interference: Disaggregate llm inference for mixed downstream workloads. arXiv preprint arXiv:2401.11181, 2024.
[12] Pratyush Patel, Esha Choukse, Chaojie Zhang, et al. Splitwise: Efficient generative llm inference using phase splitting. arXiv preprint

arXiv:2311.18677, 2023.
[13] Qin Ruoyu, Li Zheming, He Weiran, et al. Mooncake: A kvcache-centric disaggregated architecture for llm serving. arXiv preprint arXiv:2407.00079, 2024.
[14] Jin, Yibo et al. “P/D-Serve: Serving Disaggregated Large Language Model at Scale.” (2024).
[15] Zhu, Kan, et al. ”NanoFlow: Towards Optimal Large Language Model Serving Throughput.” arXiv preprint arXiv:2408.12757 (2024).
[16] CUDA Pro Tip: Increase Performance with Vectorized Memory Access. https://developer.nvidia.com/blog/cuda-pro-tip-increase-performance-with-vectorized-memory-access/
[17] Kim, Young Jin, et al. ”Who Says Elephants Can’t Run: Bringing Large Scale MoE Models into Cloud Scale Production.” arXiv preprint arXiv:2211.10017 (2022).
[18] Shoeybi, Mohammad, et al. ”Megatron-lm: Training multi-billion parameter language models using model parallelism.” arXiv preprint arXiv:1909.08053 (2019).
[19] https://docs.nvidia.com/nemo-framework/user-guide/latest/nemotoolkit/features/parallelisms.html
[20] Cai, Weilin, et al. ”Shortcut-connected Expert Parallelism for Accelerating Mixture-of-Experts.” arXiv preprint arXiv:2404.05019 (2024).
[21] Krajewski, Jakub, et al. ”Scaling laws for fine-grained mixture of experts.” arXiv preprint arXiv:2402.07871 (2024).
[22] cutlass, https://github.com/NVIDIA/cutlass/tree/main
[23] cublas, https://docs.nvidia.com/cuda/cublas/
[24] Agrawal, Amey, et al. ”Taming throughput-latency tradeoff in llm inference with sarathi-serve.” arXiv preprint arXiv:2403.02310 (2024).
[25] Yuan, Zhihang, et al. ”Llm inference unveiled: Survey and roofline model insights.” arXiv preprint arXiv:2402.16363 (2024).
[26] Agrawal, Amey, et al. ”Vidur: A Large-Scale Simulation Framework For LLM Inference.” Proceedings of Machine Learning and Systems 6 (2024): 351-366.
[27] Lepikhin, Dmitry, et al. ”Gshard: Scaling giant models with conditional computation and automatic sharding.” arXiv preprint arXiv:2006.16668 (2020).
[28] Yang, An, et al. ”M6-t: Exploring sparse expert models and beyond.” arXiv preprint arXiv:2105.15082 (2021).
[29] Shazeer, Noam, et al. ”Outrageously large neural networks: The sparsely-gated mixture-of-experts layer.” arXiv preprint arXiv:1701.06538 (2017).
[30] Fedus, William, Barret Zoph, and Noam Shazeer. ”Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity.” Journal of Machine Learning Research 23.120 (2022): 1-39.
[31] Zoph, Barret, et al. ”St-moe: Designing stable and transferable sparse expert models.” arXiv preprint arXiv:2202.08906 (2022).
[32] He, Jiaao, et al. ”Fastmoe: A fast mixture-of-expert training system.” arXiv preprint arXiv:2103.13262 (2021).
[33] Nie, Xiaonan, et al. ”Flexmoe: Scaling large-scale sparse pre-trained model training via dynamic device placement.” Proceedings of the ACM on Management of Data 1.1 (2023): 1-19.
[34] Singh, Siddharth, et al. ”A hybrid tensor-expert-data parallelism approach to optimize mixture-of-experts training.” Proceedings of the 37th International Conference on Supercomputing. 2023.
[35] Tan, Shawn, et al. ”Scattered Mixture-of-Experts Implementation.” arXiv preprint arXiv:2403.08245 (2024).
[36] Rajbhandari, Samyam, et al. ”Deepspeed-moe: Advancing mixture-of-experts inference and training to power next-generation ai scale.” International conference on machine learning. PMLR, 2022.
[37] Nie, Xiaonan, et al. ”HetuMoE: An efficient trillion-scale mixture-of-expert distributed training system.” arXiv preprint arXiv:2203.14685 (2022).