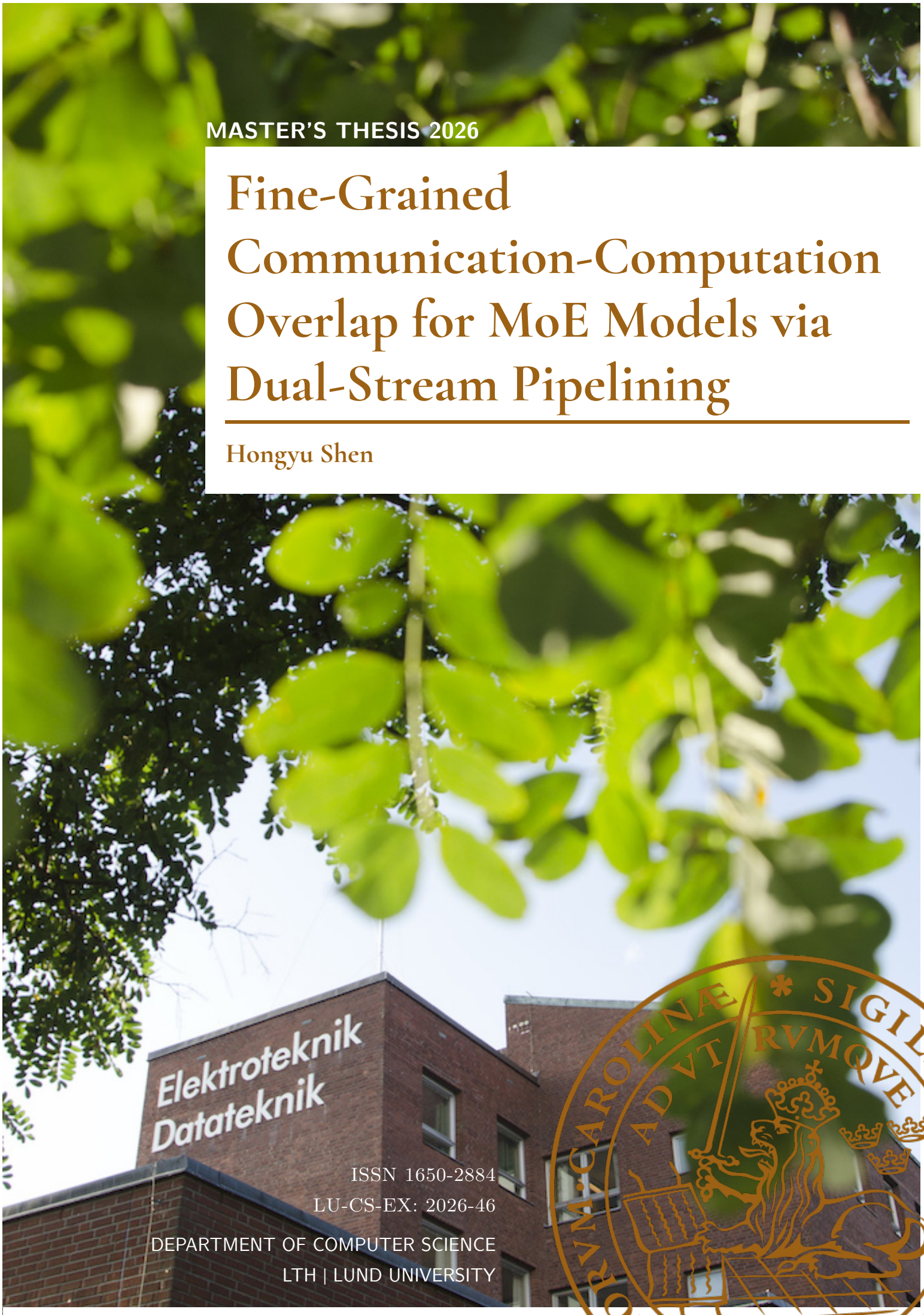
MASTER'S THESIS 2026

# Fine-Grained Communication-Computation Overlap for MoE Models via Dual-Stream Pipelining

Hongyu Shen




ISSN 1650-2884
LU-CS-EX: 2026-46

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2026-46

# Fine-Grained Communication-Computation Overlap for MoE Models via Dual-Stream Pipelining

Finkornig överlappning av kommunikation och beräkning för MoE-modeller genom dubbelströms-pipelining

**Hongyu Shen**

# Fine-Grained Communication-Computation Overlap for MoE Models via Dual-Stream Pipelining

Hongyu Shen
ho5075sh-s@student.lu.se

June 30, 2026

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisor: Arseni Ivanov, arseni.ivanov@cs.lth.se
Minyu Cui, minyu@chalmers.se

Examiner: Michael Doggett, michael.doggett@cs.lth.se

## Abstract

Mixture-of-Experts (MoE) architectures have emerged as an important approach for scaling large language models while maintaining manageable computational costs. However, under Expert Parallelism, MoE systems introduce significant communication overhead and fragmented expert computation.

This thesis proposes a fine-grained communication–computation overlap framework for distributed MoE system. The core idea is to overlap token communication and expert computation through chunk-level pipelining and asynchronous dual-stream scheduling. To support efficient overlap, the proposed framework contains fine-grained token dispatch and combine operations, a grouped GEMM implementation for fragmented expert workloads, and a dual-stream execution pipeline that coordinates communication and computation through CUDA event synchronization.

Experimental evaluations on a multi-GPU NVIDIA A100 platform demonstrate that the proposed parallel execution framework achieves lower end-to-end latency than a conventional sequential implementation by overlapping communication and computation. The performance benefits are particularly significant under communication-intensive workloads, high Top-K routing configurations, and imbalanced routing distributions. The results show that fine-grained communication–computation overlap can effectively reduce end-to-end execution time for distributed MoE systems.



**Keywords**: MoE, distributed system, communication-computation overlap

# Acknowledgements

I would like to thank my supervisors, Minyu Cui and Arseni Ivanov, for their insightful discussions on the technical roadmap, methodological guidance, and provision of computational resources throughout my project.

# Contents

# Chapter 1
# Introduction

## 1.1 Background

In recent years, the Mixture-of-Experts (MoE) architecture [1] has been widely adopted in Large Language Models (LLMs) due to its superior parameter scalability[2, 3, 4]. Unlike conventional dense Transformer models, where all parameters in the Feed-Forward Network (FFN) are activated for every token, MoE employs a routing mechanism that dynamically selects only a small subset of expert networks for computation. This sparse activation strategy enables substantial increases in model capacity while maintaining relatively constant computation cost per token. Representative models such as Switch Transformer [5], Mixtral [3], and DeepSeek-V3[4] have further demonstrated the effectiveness of MoE architectures for LLMs.

As model sizes and the number of experts continue to grow, it becomes increasingly impractical to place all expert parameters on a single GPU. Consequently, modern large-scale MoE models commonly adopt Expert Parallelism [6, 7], where experts are distributed across multiple GPUs. Under this execution model, tokens are dynamically transferred to the devices hosting their assigned experts according to routing decisions, processed by the corresponding experts, and then returned to their original devices. Therefore, the performance of MoE models depends not only on expert computation efficiency but also on the efficiency of inter-GPU communication.

However, the distributed execution of MoE models introduces several system-level challenges. First, frequent token transfers between GPUs generate substantial communication overhead, which affects the performance as system scale increases. Second, due to the dynamic routing mechanism, different experts may receive significantly different numbers of tokens, leading to irregular computational workloads and reduced GPU utilization. Together, these factors limit the overall execution efficiency of distributed MoE systems.

To address these challenges, a variety of system-level optimization techniques have been proposed in recent years. For example, COMET [8] achieves efficient pipelined execution

of MoE layers through fine-grained communication-computation overlap, shared tensor dependency analysis, and adaptive task scheduling. SonicMoE [9] adopts operator fusion techniques and combines fine-grained scheduling with asynchronous execution to improve the efficiency of MoE execution, achieving significant performance gains in fine-grained sparse MoE workloads. These approaches typically rely on substantial modifications to low-level operator implementations and deep optimization tailored to specific hardware characteristics.

This thesis investigates a communication-computation co-execution strategy centered on scheduling-level optimization. Unlike approaches that depend on heavily customized communication libraries or computation kernels, the proposed method primarily achieves communication-computation overlap through execution scheduling. Specifically, a dual-stream execution mechanism is designed to overlap Token Dispatch/Combine communication with Expert Computation, thereby reducing GPU idle time and improving the overall execution efficiency of distributed MoE systems.

## 1.2 Motivation

Existing research on distributed MoE systems primarily focuses on three directions: communication optimization, computation optimization, and communication–computation overlap and fusion.

Communication optimization aims to reduce the overhead of token exchange between GPUs. Early MoE systems such as GShard [6] and Switch Transformer [5] relied on conventional collective communication libraries to perform expert routing. More recently, specialized communication frameworks have been proposed to better support the sparse and dynamic communication patterns of MoE workloads. DeepEP [10], for example, introduces communication mechanisms specifically designed for large-scale expert parallelism.

Computation optimization focuses on improving the efficiency of expert execution. Traditional GEMM libraries such as cuBLAS are primarily optimized for large and regular dense matrix multiplication workloads. However, expert workloads in MoE systems are often highly imbalanced, resulting in many small matrix operations that fail to fully utilize GPU resources. To address this issue, several studies have explored grouped GEMM and kernel fusion techniques for irregular workloads.

A third research direction aims to overlap or fuse communication and computation. Recent systems such as COMET [8], FlashMoE [11], and FLUX [12] demonstrate that fine-grained scheduling, kernel fusion, and asynchronous execution can significantly improve GPU utilization and reduce end-to-end latency in distributed AI workloads.

Although these approaches have achieved substantial performance improvements, many existing solutions rely on deeply customized low-level operators, fused kernels, or tightly coupled communication–computation implementations. While such designs can achieve high efficiency on specific hardware platforms, they often reduce modularity and make integration with alternative communication backends or computation kernels more difficult.

In contrast, this thesis investigates whether communication–computation overlap can be achieved through a more modular execution framework. Rather than introducing a fully fused implementation, the proposed approach combines existing communication and computation components through chunk-level scheduling and dual-stream execution, enabling

communication latency hiding while preserving flexibility and portability across different software stacks and hardware platforms.

## 1.3 Purpose

The objective of this thesis is to improve the execution efficiency of distributed MoE systems by designing a communication-computation co-execution mechanism. Unlike existing approaches that focus on communication library redesign, operator fusion, or hardware-specific optimizations, this work primarily investigates scheduling-level optimization and explores how communication and computation can be effectively overlapped within existing distributed MoE execution frameworks.

Specifically, this thesis proposes a dual-stream execution framework in which communication and expert computation are executed concurrently whenever possible. By overlapping token communication with expert computation, the proposed approach aims to reduce GPU idle time caused by communication stalls and improve overall hardware utilization.

To facilitate practical deployment, the proposed framework is designed to reuse existing communication and computation components rather than requiring substantial modifications to low-level communication libraries or computation kernels. As a result, communication-computation overlap can be achieved with relatively low integration cost while maintaining flexibility for different communication and computation backends.

This thesis investigates the design and implementation of the proposed execution framework and evaluates its effectiveness in improving the performance of distributed MoE workloads.

## 1.4 Aim and Scope

The primary aim of this thesis is to improve the execution efficiency of distributed Mixture-of-Experts (MoE) systems through communication-computation overlap. Specifically, this work investigates whether communication latency can be effectively hidden behind expert computation through scheduling-level optimization, thereby reducing GPU idle time and improving overall throughput.

A secondary aim is to study how different computation workload sizes and routing imbalance patterns affect the effectiveness of the proposed overlap strategy. Particular attention is given to the impact of workload fragmentation and load imbalance, which are common characteristics of large-scale distributed MoE systems.

The scope of this thesis is limited to the execution optimization of distributed MoE layers. The work focuses on the interaction between token communication and expert computation, and investigates how these two processes can be coordinated more efficiently within a unified execution framework. In addition, a Triton-based grouped GEMM implementation is developed and evaluated as a computation backend for MoE workloads.

# Chapter 2
# Theory

## 2.1 Transformer and MoE

Since its introduction, the Transformer architecture has become the foundation of modern Large Language Models (LLMs) [13]. A standard Transformer layer mainly consists of two components: Attention Layers and FFN Layers. The attention layer is responsible for modeling relationships among different tokens, while the FFN layer performs nonlinear feature transformations on each token independently.

In conventional dense Transformer models, all input tokens are processed by the same FFN, meaning that every token activates the entire set of FFN parameters during computation. As model sizes continue to increase, FFNs gradually become one of the largest contributors to the overall parameter count. Consequently, scaling model capacity is typically accompanied by proportional growth in both computational cost and memory consumption, making further model scaling increasingly expensive.

To improve parameter efficiency, various sparse architectures have been proposed, among which the MoE architecture has emerged as one of the most successful and widely adopted approaches [1, 6]. The MoE model replaces the single FFN in a FFN layer with multiple expert networks and introduces a routing mechanism that dynamically selects only a small subset of experts for each input token, and the layer is called MoE layer. Unlike dense models, where all parameters participate in computation, only a few experts are activated for each token in an MoE model, while the remaining experts remain inactive.

### 2.1.1 Expert Parallel of MoE

As the number of experts continues to grow, a single GPU is often unable to accommodate all expert parameters. Therefore, modern MoE models commonly adopt Expert Parallelism, where different experts are distributed across multiple GPUs for execution [5, 6].

In an MoE layer, routing decisions are first generated by a router. The router is typically

implemented as a lightweight Multi-Layer Perceptron (MLP), whose output is normalized using a Softmax function to produce routing probabilities for all experts. Based on these probabilities, the Top-K experts with the highest scores are selected as the target experts for each token.

Under expert parallelism, each expert is assigned to a specific GPU. Therefore, for each token, the router determines not only the target expert but also the corresponding target GPU. According to the routing results, tokens must be transferred to the GPUs hosting their assigned experts for further computation.

This process is commonly referred to as Token Dispatch. In large-scale distributed MoE systems, Token Dispatch is typically implemented using All-to-All communication. All-to-All is a communication operation in which every GPU simultaneously sends data to and receives data from all other GPUs. After the dispatch stage, each GPU receives the tokens assigned to its local experts.

During the Expert Computation stage, the received tokens are first grouped according to their expert assignments. Tokens belonging to the same expert are aggregated into a larger input matrix and multiplied by the corresponding expert weights. Since each GPU may host multiple experts, the FFN computations of different experts are executed independently to generate expert outputs.

After expert computation is completed, the output tokens must be returned to their original devices for subsequent network layers. Therefore, the outputs are reorganized according to the source GPU information of each token and transferred back through the Token Combine stage. Similar to Token Dispatch, Token Combine is typically implemented using All-to-All communication, allowing expert outputs to be sent back to their originating devices and reordered into the original token sequence.

Consequently, the execution of an MoE layer under expert parallelism can generally be divided into four stages: Router, Token Dispatch, Expert Computation, and Token Combine. Among these stages, Token Dispatch and Token Combine are communication-intensive operations, while Expert Computation is computation-intensive. Efficiently coordinating communication and computation across these stages has become a key challenge in distributed MoE systems.

### 2.1.2 System Challenges of MoE

Despite its scalability advantages, the sparse activation mechanism of MoE introduces several system-level challenges.

First, different tokens may be routed to different experts, resulting in dynamic and irregular computation patterns. Unlike dense models with a fixed computation graph, the execution path of an MoE layer varies according to the routing decisions for each input. Second, the number of tokens assigned to different experts can vary significantly. Such imbalance reduces GPU utilization and negatively affects overall execution efficiency.

Furthermore, experts are typically distributed across multiple GPUs in practical deployments. When a token is routed to an expert located on a different device, the system must transfer token data between GPUs, introducing additional communication overhead. As both model size and system scale continue to grow, communication overhead can become a major performance bottleneck in distributed MoE systems, with recent studies reporting that inter-device communication may account for a substantial fraction of MoE execution

time [8, 14, 11].

## 2.2 Communication-Computation Overlap

As discussed previously, apart from the router, the execution of an MoE layer generally consists of three stages: Token Dispatch, Expert Computation, and Token Combine. Since the Dispatch stage must first transfer tokens to the devices hosting their assigned experts, and the Combine stage must wait for expert computation to complete before collecting the results, there exist inherent data dependencies between communication and computation.

In conventional implementations, these three stages are typically executed sequentially. The system first performs Token Dispatch communication, followed by expert computation, and finally Token Combine communication. Under this execution model, communication latency is directly exposed on the critical execution path. During communication, computation kernels are forced to wait until the communication operation completes, resulting in underutilized computational resources. Similarly, communication resources may remain idle while waiting for computation results to become available. Consequently, sequential execution often leads to poor overall resource utilization.

To mitigate the performance impact of communication latency, communication-computation overlap has emerged as an effective optimization technique. The key idea is to execute communication and computation concurrently whenever data dependencies allow, thereby hiding part of the communication overhead behind useful computation. Unlike approaches that directly reduce communication time, communication-computation overlap improves overall execution efficiency by increasing resource utilization and shortening the end-to-end execution time.

Communication-computation overlap is particularly important for MoE models. On the one hand, the communication volume associated with Token Dispatch and Token Combine continues to increase as the number of experts and GPUs grows. On the other hand, load imbalance among experts often results in significant variation in computation time, further increasing resource idle periods. These characteristics provide substantial opportunities for hiding communication latency through overlap.

In recent years, numerous optimization techniques have been proposed to improve communication-computation overlap. For example, COMET achieves coordinated communication and computation execution through fine-grained task partitioning and pipelined scheduling, effectively hiding communication latency and improving GPU utilization [8, 12]. Beyond intra-layer communication-computation overlap, recent MoE inference systems have also explored batch-level overlap strategies. By pipelining communication and computation across different batches or micro-batches, these approaches enable communication for one batch to be executed concurrently with computation for another batch, further improving resource utilization and reducing the impact of communication latency [15, 16].

## 2.3 Communication

In expert-parallel environments, the Token Dispatch and Token Combine phases require frequent data exchanges across GPUs. Therefore, efficient communication mechanisms are

critical to the performance of distributed MoE systems. To support collective communication operations in distributed deep learning, a variety of high-performance communication libraries have been developed for GPU clusters.

### 2.3.1 GPU Peer-to-Peer Communication

In distributed GPU systems, GPUs are typically interconnected through PCIe (Peripheral Component Interconnect Express) or NVLink. PCIe is the most widely used high-speed interconnect standard and supports data transfers between CPUs and GPUs as well as between GPUs. NVLink [17], in contrast, is a high-bandwidth interconnect technology developed by NVIDIA specifically for GPU communication, providing significantly higher bandwidth and lower latency than PCIe.

Built upon these hardware interconnects, modern GPUs support Peer-to-Peer (P2P) communication, which allows one GPU to directly access the memory of another GPU without staging data through host memory. Compared with traditional communication approaches, P2P communication significantly reduces data copying overhead and improves inter-GPU data transfer efficiency. As a result, P2P communication has become a fundamental building block for high-performance communication in distributed deep learning systems and MoE models.

### 2.3.2 NCCL and RCCL

Building upon P2P communication, NVIDIA Collective Communications Library (NCCL) [18] and Radeon Collective Communications Library (RCCL) [19] further abstract low-level communication details and provide unified collective communication interfaces for multi-GPU environments. These libraries offer high-performance communication primitives for GPU clusters and are widely used in distributed training and inference systems [18, 19].

NCCL and RCCL libraries share similar communication models and programming interfaces and as the experimental platform used in this thesis is based on NVIDIA A100 GPUs, NCCL is introduced as the representative communication backend in this section.

NCCL is a collective communication library designed for multi-GPU systems. Its key advantage is that data exchange among devices is abstracted into collective communication operations, eliminating the need for developers to explicitly manage P2P communication between individual GPUs. NCCL provides a variety of collective communication primitives, including Broadcast, Reduce, AllReduce, AllGather, and All-to-All. Furthermore, it can automatically select communication paths based on the underlying hardware topology and fully utilize high-speed interconnects such as NVLink, PCIe, and InfiniBand to achieve high-bandwidth and low-latency communication.

In MoE systems, Token Dispatch and Token Combine are typically implemented through All-to-All communication. NCCL's All-to-All is a representative collective communication operation in which each GPU simultaneously sends data to and receives data from all other participating GPUs. By leveraging NCCL's optimized collective communication primitives, distributed MoE systems can efficiently perform cross-GPU token exchanges and support expert-parallel execution.

However, NCCL was originally designed for conventional dense deep learning workloads, where communication volumes across devices are often relatively balanced. In MoE

systems, dynamic routing introduces significant communication irregularity and imbalance across GPUs. Consequently, general-purpose collective communication libraries may not fully exploit the communication characteristics of MoE workloads, potentially limiting overall system performance.

### 2.3.3 DeepEP

To improve communication efficiency in expert-parallel environments, a recent work has proposed DeepEP [10], a communication framework specifically designed for MoE systems. Unlike general-purpose communication libraries such as NCCL/RCCL, DeepEP introduces specialized optimizations for the Token Dispatch and Token Combine phases to better accommodate the dynamic routing and sparse communication patterns inherent in MoE models.

DeepEP adopts a token-centric communication design and incorporates efficient data exchange mechanisms tailored for expert-parallel execution. Compared with conventional collective communication approaches, it is more capable of handling highly imbalanced token distributions across experts while reducing communication overhead.

The fine-grained communication mechanism employed by DeepEP provides a favorable foundation for communication-computation overlap.

## 2.4 Computation

### 2.4.1 GEMM

In Transformer and MoE models, expert networks are typically composed of multiple linear layers and nonlinear activation functions. Among these components, the linear layers are primarily implemented using matrix multiplication operations. Consequently, General Matrix Multiplication (GEMM) constitutes the dominant computational workload during the expert computation stage.

For a linear layer, the computation can be expressed as

$$Y = XW \tag{2.1}$$

where $X$ denotes the input feature matrix, $W$ represents the weight matrix, and $Y$ is the output feature matrix. In practical MoE models, each expert maintains an independent set of FFN parameters. Therefore, expert computation can be viewed as a collection of GEMM operations performed across different experts. Since modern GPUs are highly optimized for matrix multiplication workloads, GEMM has become one of the most important computational kernels in deep learning systems.

### 2.4.2 Batched GEMM

In many deep learning applications, multiple independent matrix multiplication operations must be executed simultaneously. Launching a separate GPU kernel for each GEMM opera-

tion introduces considerable kernel launch overhead and often leads to inefficient utilization of GPU resources.

To address this issue, Batched GEMM allows multiple matrix multiplication tasks with identical matrix dimensions to be executed within a single kernel launch, thereby reducing launch overhead and improving computational efficiency.

Consider a set of matrix multiplication tasks:

$$Y_i = X_i W_i, \quad i = 1, \ldots, N \tag{2.2}$$

Rather than executing each GEMM independently, Batched GEMM schedules all operations together and processes them within a unified execution framework. This approach improves GPU utilization by exposing more parallelism to the hardware and reducing the cost associated with launching multiple kernels. However, Batched GEMM generally assumes that all matrix multiplication tasks share the same or similar dimensions. When matrix sizes vary significantly, its effectiveness becomes limited.

### 2.4.3 Grouped GEMM

In MoE systems, different experts often receive different numbers of tokens due to the dynamic routing mechanism. Since the number of input tokens determines the dimensions of the input matrix for each expert, the resulting GEMM operations frequently have different matrix sizes.

For example, during a single forward pass, one expert may receive hundreds of tokens while another expert receives only a few dozen. Such load imbalance generates a large number of small and irregular GEMM operations, making conventional Batched GEMM difficult to apply effectively.

To improve performance under irregular workloads, Grouped GEMM has been proposed as an extension of Batched GEMM. Unlike Batched GEMM, which requires all matrix multiplication tasks to have identical dimensions, Grouped GEMM allows multiple GEMM operations with different matrix sizes to be executed within a single kernel invocation. By jointly scheduling heterogeneous GEMM tasks, Grouped GEMM reduces kernel launch overhead and improves GPU resource utilization.

To further exploit GPU parallelism, large matrix multiplication tasks are typically partitioned into smaller computational blocks, commonly referred to as tiles. Each tile corresponds to a subregion of the output matrix and can be processed independently by a GPU execution unit. Tile-based decomposition enables the GPU to execute many computational tasks concurrently while improving memory access efficiency.

In Grouped GEMM, GEMM tasks from different experts are decomposed into tiles and executed within a unified kernel. This tile-level scheduling improves GPU utilization, reduces kernel launch overhead, and is well suited to the irregular workloads generated by MoE routing.

# Chapter 3
# Method

This chapter presents the proposed fine-grained communication–computation overlap method for distributed Mixture-of-Experts execution. The method is built on an expert-parallel setup in which expert parameters are distributed across multiple GPUs. It consists of four components: expert-parallel execution setup, fine-grained token chunking, Triton-based grouped expert GEMM, and a dual-stream pipeline that overlaps DeepEP communication with local expert computation. These components are described in Sections 3.1–3.4, respectively.

## 3.1 Expert-Parallel Execution Setup

This work adopts the standard expert-parallel execution model for distributed MoE layers. The expert parameters are statically partitioned across the available GPU ranks, such that each GPU stores only a subset of the experts. Let $E$ denote the total number of experts and $R$ the number of GPU ranks. Under an even placement, each rank stores approximately $E/R$ local experts.

An expert-to-rank mapping records the GPU rank on which each expert is located. After the router selects the target experts for each token, this mapping is used to determine the destination rank of every token–expert assignment. Tokens whose selected experts are located on remote GPUs must therefore be transferred across devices before expert computation can begin.

The inter-GPU communication required by expert parallelism is implemented using DeepEP. DeepEP performs Token Dispatch by sending token features to the GPU ranks that host the selected experts. After the local expert computation is completed, Token Combine returns the expert outputs to their originating ranks and restores the required token order.

Expert Parallelism is used in this thesis as the underlying distributed execution strategy rather than as a newly developed parallel library. The main implementation contributions of this work are the fine-grained token chunking strategy, the Triton-based grouped expert GEMM, and the dual-stream pipeline that overlaps DeepEP communication with local expert

computation.

## 3.2 Fine-Grained Token Chunking

Let

$$X \in \mathbb{R}^{T \times H} \tag{3.1}$$

denote the input token matrix, where $T$ represents the number of tokens and $H$ denotes the hidden dimension.

In this work, DeepEP is adopted as the communication backend for token dispatch and combine operations. DeepEP provides efficient communication primitives for expert-parallel MoE execution. If communication is typically performed at batch granularity, communication and expert computation are executed on large batches of tokens, which limits opportunities for communication–computation overlap.

To increase scheduling flexibility, the input token matrix is partitioned into multiple fixed-size chunks:

$$X = \{X_0, X_1, \ldots, X_{C-1}\}, \tag{3.2}$$

where $C$ denotes the number of chunks. Each chunk contains at most $S$ tokens, where $S$ is the chunk size. The number of chunks is therefore determined by

$$C = \left\lceil \frac{T}{S} \right\rceil. \tag{3.3}$$

Each chunk is treated as an independent scheduling unit throughout execution. Instead of dispatching and combining the entire token batch at once, communication are performed on a chunk-by-chunk basis. This finer granularity enables different chunks to progress through different execution stages independently.

The chunk size introduces a trade-off between scheduling flexibility and communication overhead. Smaller chunks create more opportunities for communication–computation overlap by increasing pipeline depth. However, they also increase communication launch overhead because a larger number of communication operations must be issued. Conversely, larger chunks reduce communication overhead but decrease overlap opportunities, causing execution behavior to gradually approach the serialized case.

Therefore, selecting an appropriate chunk size is important for balancing communication overhead and overlap efficiency. After comparing experiments, 2048 is adopted as chunk size in the remaining experiments unless otherwise specified.

## 3.3 Grouped Expert GEMM

After the Token Dispatch stage, each GPU processes the tokens assigned to its local experts. Although highly optimized GEMM libraries such as cuBLAS provide excellent performance for large and regular matrix multiplications, MoE workloads commonly contain many small and irregular GEMM operations. This irregularity is caused by the routing process, since

different experts may receive different numbers of tokens. In addition, a GPU generally hosts multiple experts under expert parallelism. Executing one GEMM kernel for each expert therefore introduces repeated kernel-launch overhead and may result in low GPU utilization, especially when the expert-level matrices are small.

To address this problem, this work implements a grouped expert GEMM kernel using Triton. Instead of launching an independent GEMM operation for every expert, the output matrices of all local experts are decomposed into tiles and processed within one kernel launch. This section describes the input layout, work-queue construction, tile computation, and scheduling strategy used in the implementation.

**Expert-contiguous input layout.** Let $E_r$ denote the number of experts hosted by GPU rank $r$, and let $m_e$ denote the number of tokens received by local expert $e$. The input matrix of expert $e$ is represented as

$$X_{r,e} \in \mathbb{R}^{m_e \times K}, \qquad e = 0, 1, \ldots, E_r - 1, \tag{3.4}$$

where $K$ is the input feature dimension of the linear layer. The dispatched tokens are stored contiguously in expert order to form a single concatenated matrix

$$A_{\text{cat}} = \begin{bmatrix} X_{r,0} \\ X_{r,1} \\ \vdots \\ X_{r,E_r-1} \end{bmatrix} \in \mathbb{R}^{M \times K}, \qquad M = \sum_{e=0}^{E_r-1} m_e. \tag{3.5}$$

An offset array is used to describe the row interval associated with each expert:

$$o_0 = 0, \qquad o_{e+1} = o_e + m_e. \tag{3.6}$$

Consequently, the tokens assigned to expert $e$ occupy the row range

$$A_{\text{cat}}[o_e : o_{e+1}, :]. \tag{3.7}$$

In the implementation, the offset array is generated directly from the per-expert token counts returned by the communication stage using a prefix sum. This layout avoids maintaining a separate input tensor for every expert and allows all expert computations to share the same input and output buffers.

For one expert linear layer, the expert weights are stored in a three- dimensional tensor

$$B \in \mathbb{R}^{E_r \times K \times N}, \tag{3.8}$$

where $B_e = B[e, :, :]$ is the weight matrix of expert $e$, and $N$ denotes the output feature dimension. The grouped GEMM computes

$$C_{\text{cat}}[o_e : o_{e+1}, :] = A_{\text{cat}}[o_e : o_{e+1}, :]B_e, \qquad e = 0, 1, \ldots, E_r - 1. \tag{3.9}$$

For the two expert FFN linear layers used in this work, Equation (3.9) is applied as

$$Z_{r,e} = \phi\left(X_{r,e} W_1^{(e)}\right), \tag{3.10}$$

$$Y_{r,e} = Z_{r,e} W_2^{(e)}, \tag{3.11}$$

where $W_1^{(e)}$ and $W_2^{(e)}$ are the two expert-specific weight matrices, and $\phi(\cdot)$ denotes the nonlinear activation function.

**Tile-level work-queue construction.** Each expert GEMM has dimensions

$$(m_e \times K)(K \times N) \to (m_e \times N). \tag{3.12}$$

The output matrix is partitioned into tiles of size $B_M \times B_N$, while the reduction dimension is processed in blocks of size $B_K$. The number of output tiles associated with expert $e$ is

$$T_e = \left\lceil \frac{m_e}{B_M} \right\rceil \left\lceil \frac{N}{B_N} \right\rceil. \tag{3.13}$$

The total number of work items is therefore

$$T_{\text{work}} = \sum_{e=0}^{E_r - 1} T_e. \tag{3.14}$$

Each work item is represented by a tuple

$$q_j = (e_j, t_j^{(m)}, t_j^{(n)}), \qquad j = 0, 1, \ldots, T_{\text{work}} - 1, \tag{3.15}$$

where $e_j$ identifies the expert, and $t_j^{(m)}$ and $t_j^{(n)}$ identify the tile coordinates along the output $M$ and $N$ dimensions.

**Work scheduling.** All output tiles from the local experts are organized into a global work queue. Each work item records the expert identifier and the two-dimensional tile coordinates required to locate the corresponding output tile. The grouped GEMM kernel is launched with a fixed number of Triton blocks, which is configured according to the number of Streaming Multiprocessors (SMs) allocated to computation on the NVIDIA GPU.

Let $T$ denote the total number of tile tasks in the work queue and let $S$ denote the number of launched Triton blocks. Each block first processes the tile whose index is equal to its block identifier. After completing the current tile, the block advances through the work queue with a stride of $S$. Therefore, block $i$ processes the sequence of tile indices

$$i,\ i + S,\ i + 2S,\ \ldots \tag{3.16}$$

until the tile index reaches $T$. For each selected work item, the block loads the associated expert identifier, row-tile index, and column-tile index, and then performs the corresponding tile-level GEMM computation.

This striped scheduling strategy allows a fixed number of blocks to process an arbitrary number of tile tasks within a single kernel launch. By allowing each block to compute multiple tiles, the implementation reduces kernel-launch overhead and jointly schedules fragmented GEMM workloads from different experts.

**Tile computation.** For a queue entry $q_j = (e, t_m, t_n)$, the corresponding output tile begins at

$$m_0 = t_m B_M, \qquad n_0 = t_n B_N. \tag{3.17}$$

The Triton program loads a block of $A_{\text{cat}}$ from the row range of expert $e$ and a block of the corresponding expert weight matrix $B_e$. The output tile is accumulated over the reduction dimension as

$$C_e[m_0 : m_0{+}B_M,\ n_0 : n_0{+}B_N] = \sum_{k_0=0,B_K,2B_K,\ldots} A_e[m_0 : m_0{+}B_M,\ k_0 : k_0{+}B_K]B_e[k_0 : k_0{+}B_K,\ n_0 : n_0{+}B_N]. \tag{3.18}$$

The output tile is computed by iterating over the $K$ dimension in blocks of size $B_K$. In each iteration, a $B_M \times B_K$ tile from the input matrix is multiplied by a $B_K \times B_N$ tile from the corresponding expert weight matrix, and the resulting partial product is accumulated into the current output tile.

Since $m_e$, $N$, and $K$ are not necessarily integer multiples of the tile dimensions, masked memory operations are used for boundary tiles. For example, a loaded element of the input tile is valid only if

$$m_0 + i < m_e \qquad \text{and} \qquad k_0 + k < K, \tag{3.19}$$

while a weight element is valid only if

$$k_0 + k < K \qquad \text{and} \qquad n_0 + j < N. \tag{3.20}$$

Out-of-range elements are replaced with zero, and output stores are masked when the tile extends beyond the valid expert rows or output columns. Therefore, the implementation does not require the token count of each expert to be padded to a fixed size.

The kernel uses tensor strides supplied by PyTorch rather than assuming a specific contiguous memory layout. In addition, address calculations involving the expert index and expert offsets use 64-bit integer arithmetic. This is necessary for large expert-weight tensors, where the product between the expert index and the per-expert weight stride may exceed the range of 32-bit indexing.

**Integration with token dispatch.** The grouped GEMM implementation assumes that the output of Token Dispatch is arranged contiguously by local expert. Given the number of received tokens for each expert, the prefix-sum offset array in Equation (3.6) identifies the valid row range of every expert. If a communication backend produces a different token layout, the received tokens must first be reordered into this expert-contiguous form.

The communication–computation overlap is coordinated outside the grouped GEMM kernel. Once the dispatched data of a chunk becomes valid, its grouped GEMM can be launched on the computation stream, while communication for other chunks proceeds on the communication stream. Thus, the grouped GEMM kernel acts as an independent computation backend and does not embed communication operations or stream synchronization internally.

**Tile configurations.** The expert network evaluated in this work contains two linear layers with dimensions $7168 \rightarrow 2048$ and $2048 \rightarrow 7168$. Because these two layers have different matrix shapes, using a single tile configuration for both layers does not provide the best performance.

After empirical tuning, the first grouped GEMM, $7168 \rightarrow 2048$, uses

$$(B_M, B_N, B_K) = (128, 64, 64), \tag{3.21}$$

whereas the second grouped GEMM, $2048 \rightarrow 7168$, uses

$$(B_M, B_N, B_K) = (64, 128, 64). \tag{3.22}$$

The three values represent the tile dimensions along the $M$, $N$, and $K$ dimensions, respectively. These configurations are used throughout the remaining experiments.

## 3.4 Communication–Computation Overlap Pipeline

Missing figure file: `Figures/figure1.1.pdf`

**Figure 3.1:** Single-stream serialized pipeline (a) and proposed dual-stream parallel pipeline (b).

This work designs a communication–computation overlap pipeline by assigning communication operations and expert computation to different CUDA streams, thereby creating opportunities for concurrent execution.

Figure 3.1 illustrates the proposed dual-stream pipeline execution framework. A serialized implementation typically uses a single-stream execution mode, where one chunk must complete the Dispatch, Expert Computation, and Combine stages before the next chunk can start. In this execution mode, communication and computation remain on the same execution path.

To improve resource utilization, this work adopts two independent CUDA streams:

- Communication Stream: responsible for Dispatch and Combine communication;
- Compute Stream: responsible for Grouped GEMM expert computation.

The Dispatch and Combine operations are executed by DeepEP, while Expert Computation is performed by the Grouped GEMM implementation described in the previous section.

After chunk partitioning, each chunk is treated as an independent scheduling unit. Since different chunks do not have direct data dependencies, their execution can be organized in a pipelined manner. Once the pipeline reaches the steady state, different chunks can occupy different execution stages simultaneously:

$$\text{Combine}(i-1) \quad \| \quad \text{Compute}(i) \quad \| \quad \text{Dispatch}(i+1), \tag{3.23}$$

where $\|$ denotes concurrent execution.

In other words, when chunk $i$ is performing expert computation, chunk $i-1$ can simultaneously perform Combine communication, while chunk $i+1$ can start Dispatch communication. In this way, communication and computation are no longer executed strictly sequentially, but can be overlapped across different chunks.

To guarantee execution correctness, synchronization dependencies are established only between different stages of the same chunk. Specifically, expert computation must wait for the Dispatch of the corresponding chunk to complete, while Combine must wait for the expert computation of the same chunk to finish. These dependencies are implemented using CUDA events, where the Communication Stream and the Compute Stream synchronize only at necessary dependency points, avoiding additional overhead from global synchronization.

This work controls the resource allocation between communication kernels and computation kernels. The experimental platform uses NVIDIA A100 GPUs, where each GPU contains 108 Streaming Multiprocessors (SMs). To increase the opportunity for communication and computation to execute concurrently, this work limits the number of thread blocks used by the communication and computation kernels.

Specifically, the number of thread blocks for the communication kernel is set to 16, while the number of thread blocks for the Grouped GEMM computation kernel is set to 92, resulting in a total of 108 blocks. This configuration is intended to reduce resource contention between communication and computation kernels, making it easier for the two types of kernels to reside on the GPU concurrently.

This configuration is used because expert computation usually accounts for most of the MoE layer execution time, while the computation demand of communication operations is relatively smaller. Therefore, most SM resources are allocated to the Grouped GEMM computation kernel, while a smaller portion is reserved for Dispatch and Combine communication. Based on empirical tuning, the 16:92 resource split provides good overlap behavior on the target hardware platform and is kept fixed in the remaining experiments.

It should be noted that this method does not reduce the amount of communication itself. Instead, it creates opportunities for communication–computation overlap through chunk-level pipeline scheduling, dual-stream concurrent execution, and resource partitioning. In practice, the actual degree of kernel concurrency is still affected by factors such as register usage, shared memory usage, and GPU hardware scheduling policies. Nevertheless, reducing resource contention between the two types of kernels can increase the likelihood of concurrent execution and reduce the impact of communication waiting time on overall execution.

# Chapter 4
# Results

This chapter evaluates the proposed communication–computation overlap pipeline for distributed MoE systems. The experiments focus on three aspects: communication backend efficiency, expert computation efficiency, and end-to-end overlap performance. Specifically, the communication benchmark compares DeepEP-based fine-grained dispatch communication with NCCL All-to-All communication. The expert computation benchmark evaluates the proposed grouped GEMM implementation under irregular MoE workloads. Finally, the end-to-end benchmark evaluates the throughput improvement achieved by the proposed dual-stream overlap pipeline.

## 4.1 Experimental Setup

### 4.1.1 Hardware and Workload Configuration

All experiments were conducted on a single server equipped with four NVIDIA A100 GPUs interconnected through NVLink. The experimental configuration is summarized in Table 4.1.

### 4.1.2 Routing Imbalance Generation

To evaluate system performance under different levels of routing imbalance, three routing modes are considered: balanced, mild skew, and severe skew.

For each experiment, synthetic token features are generated as router inputs. Specifically, each token is represented by a 7168-dimensional feature vector whose elements are independently sampled from a Gaussian distribution with zero mean and unit variance. The generated token features are then passed through a linear router layer with the default PyTorch weight initialization to produce the original router logits.

Routing imbalance is subsequently introduced by modifying the router logits before Top-K selection. Specifically, the router logits are adjusted according to

**Table 4.1:** Experimental configuration.

| Parameter | Value |
|---|---|
| Number of nodes | 1 |
| Number of GPUs | 4 |
| GPU model | NVIDIA A100 80GB |
| GPU interconnect | NVLink |
| Global tokens | 32768 |
| Tokens per rank | 8192 |
| Hidden dimension | 7168 |
| Intermediate dimension | 2048 |
| Data type | BF16 |
| Chunk size | 2048 tokens |
| Communication blocks | 16 |
| Computation blocks | 92 |

$$\mathbf{z}' = \frac{\mathbf{z}}{\tau} + \mathbf{b}, \tag{4.1}$$

where $\mathbf{z}$ denotes the original router logits produced by the router layer, $\tau$ is the routing temperature, and $\mathbf{b}$ is an expert-specific bias vector.

For the balanced configuration, no routing bias is applied and the routing temperature is set to 1.0. Under this setting, routing decisions are primarily determined by the randomly generated token features and the randomly initialized router weights, resulting in an approximately uniform routing distribution across experts.

For skewed configurations, the routing temperature is first reduced, causing all router logits to be scaled by a factor of $1/\tau$. Since $\tau < 1$, the differences among router logits are amplified, making routing decisions more selective. Subsequently, approximately 12.5% of experts are randomly selected as hot experts. And an additional positive bias is added to the logits corresponding to these experts for every token. Consequently, the hot experts become more likely to appear in the Top-K routing results, causing a larger proportion of token assignments to concentrate on a small subset of experts and thereby creating routing imbalance.

The routing configurations used in this work are summarized in Table 4.2.

**Table 4.2:** Routing imbalance configurations.

| Mode | Temperature | Bias | Hot Expert Fraction |
|---|---|---|---|
| Balanced | 1.0 | 0.0 | 0.0 |
| Mild Skew | 0.8 | 1.0 | 0.125 |
| Severe Skew | 0.5 | 2.5 | 0.125 |

### 4.1.3 Imbalance Metrics

Routing imbalance is quantified using the coefficient of variation (CV) of token assignments. The CV is defined as the ratio between the standard deviation and the mean:

$$CV = \frac{\sigma}{\mu}. \tag{4.2}$$

For expert-level imbalance, let $n_e$ denote the number of token assignments received by expert $e$, and let $N_E$ be the total number of experts. The mean number of token assignments per expert is

$$\mu_E = \frac{1}{N_E} \sum_{e=1}^{N_E} n_e, \tag{4.3}$$

and the corresponding standard deviation is

$$\sigma_E = \sqrt{\frac{1}{N_E} \sum_{e=1}^{N_E} (n_e - \mu_E)^2}. \tag{4.4}$$

The expert-level coefficient of variation is then defined as

$$CV_{\text{expert}} = \frac{\sigma_E}{\mu_E}. \tag{4.5}$$

Similarly, for rank-level imbalance, let $m_r$ denote the total number of token assignments hosted by GPU rank $r$, and let $N_R$ be the number of GPU ranks. The mean number of token assignments per rank is

$$\mu_R = \frac{1}{N_R} \sum_{r=1}^{N_R} m_r, \tag{4.6}$$

and the corresponding standard deviation is

$$\sigma_R = \sqrt{\frac{1}{N_R} \sum_{r=1}^{N_R} (m_r - \mu_R)^2}. \tag{4.7}$$

The rank-level coefficient of variation is then defined as

$$CV_{\text{rank}} = \frac{\sigma_R}{\mu_R}. \tag{4.8}$$

A smaller CV indicates a more balanced routing distribution, while a larger CV indicates stronger workload skew.

## 4.2 Communication Benchmark

This subsection evaluates the communication efficiency of PyTorch NCCL all-to-all communication and the DeepEP-based fine-grained dispatch communication backend.

**Table 4.3:** Communication latency under severe skew routing.

| Experts | Top-K | PyTorch A2A (ms) | DeepEP (ms) | Speedup |
|---|---|---|---|---|
| 32 | 4 | 11.414 | 3.955 | 2.89× |
| 128 | 4 | 12.075 | 4.216 | 2.86× |
| 128 | 8 | 21.890 | 4.952 | 4.42× |

### 4.2.1 Communication Performance Comparison

To evaluate the communication performance under different MoE configurations, experiments are conducted with varying numbers of experts and different Top-K routing settings. Table 4.3 summarizes the communication latency under severe skew routing patterns.

Under the severe skew setting with 32 experts and Top-K equal to 4, PyTorch all-to-all communication requires 11.414 ms, while DeepEP only requires 3.955 ms, achieving approximately 2.89× speedup.

When the number of experts increases from 32 to 128 while keeping Top-K fixed at 4, the communication latency of PyTorch all-to-all increases slightly to 12.075 ms and DeepEP slightly increases to 4.216 ms, achieving approximately 2.86× speedup, which is similar to the case of 32 experts. This result suggests that the total number of experts has only a limited impact on communication latency under the evaluated configuration. Since increasing the number of experts does not significantly change the number of tokens exchanged across GPUs, the communication overhead will not change largely.

More significant differences appear when Top-K increases. Under the configuration of 128 experts and Top-K equal to 8, PyTorch all-to-all latency increases dramatically to 21.890 ms, while DeepEP only increases to 4.952 ms, achieving approximately 4.42× speedup.

Compared with the Top-K equal to 4 setting, the communication latency of PyTorch all-to-all nearly doubles when Top-K equal to 8, while DeepEP experiences only minor latency growth. This result demonstrates that traditional NCCL all-to-all communication is considerably more sensitive to assignment expansion caused by increasing Top-K values, whereas DeepEP exhibits significantly better scalability under high fan-out routing scenarios.

### 4.2.2 Impact of Routing Imbalance

This subsection further evaluates the impact of routing imbalance on communication latency, the total number of experts was fixed at 128 and Top-K was fixed at 4.

Table 4.4 presents DeepEP communication latency under different routing skew levels.

**Table 4.4:** Fine-grained DeepEP communication latency under different routing imbalance levels.

| Routing Pattern | Latency (ms) | Rank CV | Expert CV |
|---|---|---|---|
| Balanced | 2.003 | 0.000 | 0.000 |
| Mild Skew | 4.117 | 0.156 | 0.643 |
| Severe Skew | 4.216 | 0.335 | 1.375 |

As routing imbalance increases, communication latency also increases. Specifically, DeepEP communication latency rises from 2.003 ms in the balanced setting to 4.117 ms under mild

skew and further increases to 4.216 ms under severe skew. This result indicates that DeepEP achieves its lowest communication latency under balanced routing. When routing imbalance is introduced, the communication latency has a more than twofold increase. This suggests that uneven token distributions across experts introduce additional communication overhead and reduce communication efficiency.

However, although routing imbalance becomes significantly more severe from mild skew to severe skew, DeepEP communication latency only increases slightly from 4.117 ms to 4.216 ms. This result suggests that DeepEP exhibits relatively strong robustness against increasingly skewed expert-level routing distributions.

For comparison, Table 4.5 shows PyTorch/NCCL all-to-all communication latency.

**Table 4.5:** PyTorch/NCCL all-to-all latency under different routing imbalance levels.

| Routing Pattern | Latency (ms) | Rank CV | Expert CV |
|---|---|---|---|
| Balanced | 9.476 | 0.000 | 0.000 |
| Mild Skew | 10.649 | 0.156 | 0.643 |
| Severe Skew | 12.075 | 0.335 | 1.375 |

Unlike DeepEP, PyTorch all-to-all communication exhibits continuously increasing latency as routing imbalance becomes more severe. This result suggests that Pytorch all-to-all communication is more sensitive to communication imbalance.

## 4.3 Expert Computation Benchmark

This section evaluates the performance of the grouped GEMM kernel for MoE expert computation and compares it with a traditional PyTorch GEMM implementation. To simulate practical MoE Expert Parallel workloads, experiments were conducted under two expert configurations. Using four GPUs, the total number of experts was set to either 64 or 128, corresponding to 16 and 32 local experts per GPU, respectively.

The assignment count is defined as:

$$\text{assignments} = \text{tokens} \times \text{Top-}\boldsymbol{K},$$

which corresponds to the total number of token-expert assignments.

The PyTorch baseline adopts per-expert execution. Tokens belonging to each expert are first gathered according to routing results, followed by independent cuBLAS GEMM execution for each expert FFN. In contrast, Grouped GEMM aggregates GEMM operations from multiple experts into a single kernel execution, rather than launching separate GEMM kernels for each expert.

### 4.3.1 Computation under 32 Local Experts

Table 4.6 shows the performance comparison between PyTorch per-expert GEMM and the grouped GEMM under 32 local experts.

**Table 4.6:** Expert computation benchmark with 32 local experts

| Assignments | Load Mode | PyTorch GEMM (ms) | Grouped GEMM (ms) | Speedup |
|---|---|---|---|---|
| 4096 | Balanced | 3.745 | 2.344 | 1.60× |
| 4096 | Mild Skew | 4.000 | 2.791 | 1.43× |
| 4096 | Severe Skew | 4.087 | 3.024 | 1.35× |
| 8192 | Balanced | 5.057 | 3.742 | 1.35× |
| 8192 | Mild Skew | 5.245 | 4.392 | 1.19× |
| 8192 | Severe Skew | 5.100 | 4.351 | 1.17× |
| 16384 | Balanced | 8.383 | 6.753 | 1.24× |
| 16384 | Mild Skew | 7.716 | 7.246 | 1.06× |
| 16384 | Severe Skew | 7.360 | 7.307 | 1.01× |

The results show that grouped GEMM consistently outperforms the traditional per-expert Pytorch GEMM when the number of assignments is small or moderate. For 4096 assignments, the grouped GEMM achieves a computation latency of 2.344 ms, compared with 3.745 ms for the PyTorch GEMM implementation, corresponding to an approximately 1.60× speedup under balanced routing. Under mild skew and severe skew, grouped GEMM achieves **1.43×** and **1.35×** speedup respectively.

Although Pytorch GEMM is highly optimized for large matrix multiplication workloads, launching large numbers of small GEMMs introduces considerable kernel launch overhead and low GPU occupancy. By grouping multiple expert computations into a unified kernel execution, grouped GEMM significantly improves GPU utilization in the small-to-medium assignment regime.

As the assignment count increases, the speedup gradually decreases. For 16384 assignments, the speedup becomes nearly neutral under severe skew. This trend is expected because larger per-expert matrices allow cuBLAS to better utilize Tensor Cores and amortize kernel launch overhead more effectively.

The effect of routing imbalance also differs between the two implementations. For grouped GEMM, skewed routing generally increases computation latency because irregular token distributions introduce additional padding and under-utilized tiles. In contrast, the PyTorch baseline may occasionally become slightly faster under severe skew at large assignment counts because skewed routing creates a few very large expert GEMMs, which are favorable for cuBLAS execution.

### 4.3.2 Computation under 16 Local Experts

Table 4.7 further shows the benchmark results with 16 local experts.

Compared with the 32-expert configuration, reducing the number of local experts significantly increases the average number of tokens processed by each expert under the same assignment count. Consequently, the matrix size of each expert GEMM becomes substantially larger.

The experimental results show that the advantage of grouped GEMM becomes much smaller in this regime. At 8192 and 16384 assignments, especially under skewed routing distributions, the PyTorch baseline begins to outperform Triton grouped GEMM.

**Table 4.7:** Expert computation benchmark with 16 local experts

| Assignments | Load Mode | PyTorch GEMM (ms) | Grouped GEMM (ms) | Speedup |
|---|---|---|---|---|
| 4096 | Balanced | 2.559 | 2.203 | 1.16× |
| 4096 | Mild Skew | 2.676 | 2.539 | 1.05× |
| 4096 | Severe Skew | 2.553 | 2.511 | 1.02× |
| 8192 | Balanced | 4.204 | 3.684 | 1.14× |
| 8192 | Mild Skew | 3.840 | 3.996 | 0.96× |
| 8192 | Severe Skew | 3.710 | 3.950 | 0.94× |
| 16384 | Balanced | 6.306 | 6.694 | 0.94× |
| 16384 | Mild Skew | 6.143 | 6.906 | 0.89× |
| 16384 | Severe Skew | 5.930 | 6.823 | 0.87× |

This observation further demonstrates the different applicability regimes of grouped GEMM and per-expert GEMM.

When the number of local experts is small, each expert GEMM becomes significantly larger, allowing cuBLAS to fully exploit its highly optimized large-matrix execution pipeline, including higher Tensor Core utilization, better memory throughput, and lower relative kernel launch overhead.

On the other hand, grouped GEMM is most beneficial when the number of experts is large and each expert receives relatively few tokens. These conditions closely match practical large-scale MoE workloads, especially under sparse expert activation and fine-grained chunked execution.

### 4.3.3 Overall Analysis

Overall, the experimental results demonstrate that the grouped GEMM is most effective for cases with large expert counts, sparse expert activation and highly fragmented MoE workloads.

Compared with traditional PyTorch/cuBLAS per-expert GEMM execution, grouped GEMM significantly reduces kernel launch overhead. However, when per-expert GEMM matrices become sufficiently large, the highly optimized large-matrix execution capability of cuBLAS gradually dominates, causing the advantage of grouped GEMM to diminish or disappear.

Therefore, the grouped GEMM is particularly suitable for fine-grained chunked MoE execution and naturally complements the proposed communication–computation overlap pipeline.

## 4.4 End-to-End MoE Benchmark

This section evaluates the proposed communication–computation overlap pipeline under complete MoE layer execution. Unlike the previous sections that separately evaluate communication and expert computation kernels, this benchmark includes the entire MoE forward execution pipeline, including token dispatch, expert computation and token combine. Therefore, the results more accurately reflect practical large-scale distributed MoE execution performance.

Two end-to-end execution paths are compared:

1. Baseline pipeline: PyTorch all-to-all + PyTorch/cuBLAS per-expert GEMM.
2. Optimized pipeline: DeepEP communication + grouped GEMM;

The optimized pipeline adopts the proposed Dual-stream overlap pipeline. In contrast, the baseline pipeline executes communication and computation sequentially.

### 4.4.1 High Workload Setting: Top-$K = 32$

We first evaluate the high-workload setting with Top-$K = 32$. Under this configuration, each token generates a large number of token-expert assignments, significantly increasing both communication and expert computation workloads.

The benchmark results are shown in Table 4.8.

**Table 4.8:** End-to-end MoE Prefill benchmark with Top-$K = 32$

| Routing Mode | Backend | Avg Step Time (ms) | Speedup |
|---|---|---|---|
| balanced | Torch A2A + Torch GEMM | 373.50 | 1.00× |
| balanced | DeepEP + Grouped GEMM | 201.98 | 1.85× |
| mild skew | Torch A2A + Torch GEMM | 447.66 | 1.00× |
| mild skew | DeepEP + Grouped GEMM | 213.04 | 2.10× |
| severe skew | Torch A2A + Torch GEMM | 449.63 | 1.00× |
| severe skew | DeepEP + Grouped GEMM | 218.13 | 2.06× |

The results show that the proposed optimized pipeline significantly outperforms the baseline under high Top-$K$ workloads.

Under balanced routing, the average step latency is reduced from 373.50 ms to 201.98 ms, achieving approximately 1.85× speedup. Under mild skew and severe skew, the speedup further increases to approximately 2.10× and 2.06×, respectively.

Consequently, the optimized pipeline achieves substantially higher GPU utilization under communication-intensive workloads.

### 4.4.2 Moderate Workload Setting: Top-$K = 8$

We further evaluate a lower-workload setting with Top-$K = 8$.

Compared with Top-$K = 32$, the assignment count decreases by approximately 4×, significantly reducing communication pressure and expert computation workload.

The benchmark results are shown in Table 4.9.

Although the optimized pipeline still consistently outperforms the baseline, the speedup becomes noticeably smaller, remaining around 1.28× ∼ 1.29×.

This trend indicates that as assignment workloads decrease, both communication overhead and fragmented expert computation overhead become less dominant, reducing the benefit of grouped GEMM and overlap execution.

Therefore, the proposed optimization pipeline is particularly effective for communication-intensive workloads, highly fragmented expert workloads and high Top-$K$ routing settings.

**Table 4.9:** End-to-end MoE Prefill benchmark with Top-$K = 8$

| Routing Mode | Backend | Avg Step Time (ms) | Speedup |
|---|---|---|---|
| balanced | Torch A2A + Torch GEMM | 82.84 | 1.00× |
| balanced | DeepEP + Grouped GEMM | 64.62 | 1.28× |
| mild skew | Torch A2A + Torch GEMM | 89.79 | 1.00× |
| mild skew | DeepEP + Grouped GEMM | 69.97 | 1.28× |
| severe skew | Torch A2A + Torch GEMM | 94.97 | 1.00× |
| severe skew | DeepEP + Grouped GEMM | 73.69 | 1.29× |

### 4.4.3 Effect of Routing Imbalance

The above experiments further show that routing imbalance affects the two execution pipelines differently.

In Table 4.8, for the proposed DeepEP + Grouped GEMM pipeline, the end-to-end latency increases from 201.98 ms under balanced routing to 218.13 ms under severe skew, corresponding to an increase of approximately 8%. In comparison, the latency of the Torch baseline increases from 373.50 ms to 449.63 ms, corresponding to an increase of approximately 20%.

Similarly, in Table 4.9. When Top-K is 8, the end-to-end latency of the proposed DeepEP + Grouped GEMM pipeline increases by approximately 14.0% from balanced routing to severe skew. In comparison, the latency of the Torch A2A + Torch GEMM baseline increases by approximately 14.6%.

These results suggest that the proposed overlap pipeline is less sensitive to routing imbalance than the Torch baseline. One possible reason is that communication and computation are partially overlapped in the proposed design. As routing imbalance increases, both communication and expert computation become more uneven across experts. In the Torch baseline, these overheads accumulate sequentially and are directly reflected in the end-to-end latency. In contrast, the proposed pipeline can partially hide communication overhead behind expert computation, reducing the performance impact of routing imbalance. Furthermore, both DeepEP and Grouped GEMM are designed to handle irregular MoE workloads more efficiently, which further improves robustness under skewed routing distributions.

### 4.4.4 Top-$K$ Sensitivity Analysis

Table 4.10 compares the scaling behavior of the two pipelines under different Top-$K$ settings.

**Table 4.10:** Top-$K$ sensitivity analysis

| Backend | Top-$K = 8$ (ms) | Top-$K = 32$ (ms) | Scaling |
|---|---|---|---|
| Torch A2A + Torch (balanced) | 82.84 | 373.50 | 4.51× |
| Torch A2A + Torch (severe) | 94.97 | 449.63 | 4.73× |
| DeepEP + Triton (balanced) | 64.62 | 201.98 | 3.12× |
| DeepEP + Triton (severe) | 73.69 | 218.13 | 2.96× |

The Torch baseline scales almost linearly with assignment count, indicating that NCCL all-to-all communication and per-expert GEMM strongly depend on the number of token-

expert assignments. In contrast, the proposed DeepEP + Grouped GEMM overlap pipeline exhibits substantially slower scaling growth.

This result demonstrates that chunked dispatch, grouped GEMM and communication–computation overlap effectively mitigate the workload explosion introduced by high Top-$K$ routing.

### 4.4.5 Overall Analysis

The experimental results verify that the proposed fine-grained dual-stream overlap pipeline can effectively hide a portion of the communication overhead and improve end-to-end execution efficiency for distributed MoE workloads. Across different workload configurations and routing imbalance levels, the proposed approach consistently achieves lower latency than the sequential execution baseline.

Furthermore, the proposed framework adopts a highly modular design. Rather than relying on deeply customized low-level operators, it is built upon existing communication and computation kernels and coordinates them through asynchronous scheduling. As a result, both the communication backend and computation kernels can be replaced with alternative implementations when necessary. This modularity improves the portability of the system and facilitates deployment across different hardware platforms and software environments.

# Chapter 5
# Conclusion

## 5.1 Research Objectives

The objective of this thesis was to investigate whether communication–computation overlap can improve the execution efficiency of distributed MoE systems. To achieve this goal, a fine-grained dual-stream overlap pipeline was designed and implemented.

The experimental results show that, compared with sequential execution model, the proposed dual-stream overlap strategy achieves lower end-to-end latency across different workload configurations and routing imbalance levels. These results indicate that, by carefully scheduling the Dispatch, Expert Computation, and Combine stages, a portion of the communication overhead can be hidden behind computation, leading to improved overall resource utilization and execution efficiency.

At the same time, the experimental results suggest that the effectiveness of the proposed method depends on workload characteristics. When Top-K is reduced, the total number of token-expert assignments decreases substantially, resulting in lower communication and computation overhead. Consequently, the amount of communication latency that can be hidden through overlap is also reduced. Under such conditions, the performance advantage of the proposed method becomes smaller. This observation indicates that fine-grained dual-stream communication–computation overlap is particularly beneficial for communication-intensive workloads, while its benefits may be less pronounced when communication overhead is relatively limited.

## 5.2 Practical Implications

Fine-grained communication–computation overlap is a practical optimization direction for distributed MoE systems. Instead of optimizing communication or computation in isolation, this thesis shows that coordinating both through execution scheduling can improve end-to-

end efficiency.

A practical advantage of the proposed framework is its modular design. The method does not rely on deeply customized low-level operators or a tightly coupled communication-computation kernel. Instead, it coordinates existing communication and computation components through chunk-level scheduling and CUDA stream synchronization.

This modularity makes the framework easier to adapt and extend. In principle, the communication backend could be replaced by another sparse communication implementation, and the computation backend could be replaced by another grouped GEMM or expert computation kernel. This makes the method more convenient for engineering deployment and future experimentation across different hardware platforms and software stacks.

## 5.3 Future Research

Several directions remain for future research. First, the current evaluation is limited to a single-node platform with four NVIDIA A100 GPUs connected by NVLink. Future work should evaluate the proposed overlap pipeline on larger multi-node clusters, where inter-node communication may introduce different latency and bandwidth characteristics.

Second, the current implementation uses manually selected scheduling parameters, including the chunk size and the communication/computation thread-block allocation. In this work, these parameters were selected empirically based on experimental tuning. Future research could explore adaptive scheduling mechanisms that automatically adjust chunk size and GPU resource allocation according to workload characteristics.

Third, this thesis focuses on the MoE Prefill stage. However, MoE inference also includes Decode-stage execution, where batch sizes, token counts, and communication patterns can differ significantly from Prefill. Extending the proposed overlap pipeline to Decode-stage workloads would provide a more complete evaluation of its practical applicability.

# References


[1] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. Le, G. Hinton, and J. Dean, "Outrageously large neural networks: The sparsely-gated mixture-of-experts layer," *arXiv preprint arXiv:1701.06538*, 2017.

[2] A. Yang, A. Li, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Gao, C. Huang, C. Lv *et al.*, "Qwen3 technical report," *arXiv preprint arXiv:2505.09388*, 2025.

[3] A. Q. Jiang, A. Sablayrolles, A. Roux, A. Mensch, B. Savary, C. Bamford, D. S. Chaplot, D. d. l. Casas, E. B. Hanna, F. Bressand *et al.*, "Mixtral of experts," *arXiv preprint arXiv:2401.04088*, 2024.

[4] A. Liu, B. Feng, B. Xue, B. Wang, B. Wu, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan *et al.*, "Deepseek-v3 technical report," *arXiv preprint arXiv:2412.19437*, 2024.

[5] W. Fedus, B. Zoph, and N. Shazeer, "Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity," *Journal of Machine Learning Research*, vol. 23, no. 120, pp. 1–39, 2022.

[6] D. Lepikhin, H. Lee, Y. Xu, D. Chen, O. Firat, Y. Huang, M. Krikun, N. Shazeer, and Z. Chen, "Gshard: Scaling giant models with conditional computation and automatic sharding," *arXiv preprint arXiv:2006.16668*, 2020.

[7] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro *et al.*, "Efficient large-scale language model training on gpu clusters using megatron-lm," in *Proceedings of the international conference for high performance computing, networking, storage and analysis*, 2021, pp. 1–15.

[8] S. Zhang, N. Zheng, H. Lin, Z. Jiang, W. Bao, C. Jiang, Q. Hou, W. Cui, S. Zheng, L.-W. Chang *et al.*, "Comet: Fine-grained computation-communication overlapping for mixture-of-experts," *Proceedings of Machine Learning and Systems*, vol. 7, 2025.

[9] W. Guo, M. Mishra, X. Cheng, I. Stoica, and T. Dao, "Sonicmoe: Accelerating moe with io and tile-aware optimizations," *arXiv preprint arXiv:2512.14080*, 2025.

[10] C. Zhao, S. Zhou, L. Zhang, C. Deng, Z. Xu, Y. Liu, K. Yu, J. Li, and L. Zhao, "Deepep: an efficient expert-parallel communication library," https://github.com/deepseek-ai/DeepEP, 2025.

[11] O. Aimuyo, B. Oh, and R. Singh, "Flashmoe: Fast distributed moe in a single kernel," *Advances in Neural Information Processing Systems*, vol. 38, pp. 100 676–100 699, 2026.

[12] L.-W. Chang, W. Bao, Q. Hou, C. Jiang, N. Zheng, Y. Zhong, X. Zhang, Z. Song, C. Yao, Z. Jiang *et al.*, "Flux: Fast software-based communication overlap on gpus through kernel fusion," *arXiv preprint arXiv:2406.06858*, 2024.

[13] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[14] S. Luo, P. Li, J. Peng, H. Wang, Y. Cheng, T. Chen *et al.*, "Occult: Optimizing collaborative communication across experts for accelerated parallel moe training and inference," *arXiv preprint arXiv:2505.13345*, 2025.

[15] L. Zheng, L. Yin, Z. Xie, C. Sun, J. Huang, C. H. Yu, S. Cao, C. Kozyrakis, I. Stoica, J. E. Gonzalez *et al.*, "Sglang: Efficient execution of structured language model programs," *Advances in neural information processing systems*, vol. 37, pp. 62 557–62 583, 2024.

[16] C. Jiang, Y. Tian, Z. Jia, S. Zheng, C. Wu, and Y. Wang, "Lancet: Accelerating mixture-of-experts training via whole graph computation-communication overlapping," *Proceedings of Machine Learning and Systems*, vol. 6, pp. 74–86, 2024.

[17] NVIDIA Corporation, "Nvidia nvlink high-speed interconnect," 2024, accessed: June 2026. [Online]. Available: https://www.nvidia.com/en-us/data-center/nvlink/

[18] ——, "NVIDIA Collective Communications Library (NCCL)," https://docs.nvidia.com/deeplearning/nccl/, 2026, accessed: 2026-06-04.

[19] Advanced Micro Devices, Inc., "Radeon Collective Communications Library (RCCL)," https://rocm.docs.amd.com/projects/rccl/en/latest/, 2026, accessed: 2026-06-04.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2026-06-30

**EXAMENSARBETE** Fine-Grained Communication-Computation Overlap for MoE Models via Dual-Stream Pipelining
**STUDENT** Hongyu Shen
**HANDLEDARE** Arseni Ivanov (LTH), Minyu Cui (CTH)
**EXAMINATOR** Michael Doggett (LTH)

# Hur kan stora AI-modeller vänta mindre?

POPULÄRVETENSKAPLIG SAMMANFATTNING **Hongyu Shen**

Moderna AI-system använder ofta många GPU:er som arbetar tillsammans. Samtidigt kan mycket tid gå åt till att vänta på att data flyttas mellan olika enheter. I detta arbete undersöks hur dataöverföring och beräkning kan ske samtidigt, så att väntetiden minskar och systemet används mer effektivt.

Många av dagens mest avancerade AI-modeller använder en teknik som kallas Mixture-of-Experts, eller MoE. I stället för att använda hela modellen för varje indata aktiveras bara några få specialiserade delar, så kallade experter. Det gör det möjligt att bygga mycket större modeller utan att beräkningskostnaden ökar lika mycket för varje ord eller token som behandlas.

Problemet är att experterna ofta ligger utspridda över flera GPU:er. Innan en expert kan göra sin beräkning måste data därför skickas till rätt GPU. När beräkningen är klar måste resultatet skickas tillbaka. För små system kan denna väntetid vara relativt begränsad, men när modellerna och antalet GPU:er växer blir kommunikationen mellan enheterna en viktig flaskhals.

I detta examensarbete utvecklades en schemaläggningsmetod för att minska sådan väntetid. I en enkel implementation görs först all kommunikation, sedan all beräkning och därefter skickas resultatet tillbaka. Det innebär att vissa delar av systemet ofta står stilla medan andra arbetar.

Den föreslagna metoden delar i stället upp arbetet i mindre bitar. Dessa bitar kan röra sig genom systemet ungefär som i ett löpande band: medan en bit data skickas mellan GPU:er kan en annan bit redan beräknas av en expert. På så sätt kan kommunikation och beräkning överlappa varandra.

För att genomföra detta används två separata CUDA-strömmar, en för kommunikation och en för beräkning. Kommunikationen hanterar utskick och insamling av token-data, medan beräkningsströmmen kör expertberäkningarna. Synkronisering mellan strömmarna ser till att varje del behandlas i rätt ordning utan att hela systemet behöver vänta i onödan.

Experiment på ett system med flera NVIDIA A100-GPU:er visade att metoden minskade den totala körningstiden jämfört med en traditionell sekventiell lösning. Förbättringen var särskilt tydlig när kommunikationen utgjorde en stor del av arbetet, till exempel vid högre Top-K-routing eller ojämn fördelning av data mellan experter.

En viktig fördel är att lösningen inte kräver att alla underliggande komponenter byggs om från grunden. Eftersom optimeringen främst sker på schemaläggningsnivå kan olika kommunikationsbibliotek eller beräkningskärnor bytas ut vid behov. Det gör metoden mer flexibel och lättare att anpassa till framtida AI-system.

Resultaten visar att överlappning mellan kommunikation och beräkning är ett praktiskt sätt att göra distribuerade AI-modeller snabbare och mer resurseffektiva.