OS INSPIRED COMPLETE KERNEL FUSION

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Science

by

Osayamen Jonathan Aimuyo

August 2025

ABSTRACT

Distributed Machine Learning (DML) is increasingly recognized as a communication-bound workload, with most existing work aiming to alleviate this bottleneck through communication–computation overlap. However, current approaches—largely reliant on CPU-managed operator scheduling and synchronous communication collectives—leave significant performance on the table.

This thesis identifies, analyzes and addresses the inefficiencies arising from the interplay between CPU-driven collective communication and highly parallel GPU computation. We demonstrate that the standard bulk-synchronous communication model underutilizes GPU interconnect bandwidth and is especially vulnerable to straggler-induced performance degradation. Focusing on dynamic workloads such as Mixture-of-Experts (MoE), we highlight two critical bottlenecks. First, we observe *payload inefficiency* at the application layer, where existing collective primitives force unnecessary padding of GPU network buffers. Second, we expose how CPU-driven execution limits the exploitation of *task locality* and introduces artificial synchronization barriers across distributed GPU tasks.

To overcome these limitations, we propose a model of *complete GPU residency*, where inter-GPU communication is integrated directly into GPU kernels. We realize this vision in *FlashDMoE*: a persistent, in-kernel, actor-style operating system with packet switching that enables complete operator fusion for Distributed MoE (DMoE) into a *single kernel*, the first of its kind. *FlashDMoE* features a modular, message-driven architecture that supports lockless execution across tens of thousands of GPU threads and across distributed GPUs as well. We demonstrate how *FlashDMoE* addresses the all-to-all communication bottleneck in expert parallelism and enables high-throughput, GPU-initiated communication. Evaluated against state-of-the-art distributed MoE frameworks, *FlashDMoE* achieves up to **9**× higher GPU utilization, **6**× lower latency, **5.7**× higher throughput, and **4**× better overlap efficiency compared to state-of-the-art baselines—despite using FP32 while baselines use FP16.

BIOGRAPHICAL SKETCH

Osayamen Jonathan Aimuyo is a computer science researcher interested in distributed and parallel computing systems and algorithms. His work focuses on low-level optimizations for computational and communication bottlenecks in large-scale machine learning execution on accelerators. Jonathan earned a BS in computer engineering, *summa cum laude*, with Tau Beta Pi and Phi Kappa Phi Honors from the University of Texas at Dallas (class of 2023), where he was a Presidential Achievement Scholar and a semifinalist for the national Jack Kent Cooke Transfer Scholarship (2019). He is currently a second-year CS MS student at Cornell University, advised by Dr. Rachee Singh. In industry, he has contributed to large-scale distributed systems through internships at Microsoft ('22 - '24), Chime Financial ('22), and JPMorgan Chase ('20 - '21). After graduating from Cornell, he will intern at NVIDIA with the CUDA Math Libraries team, before beginning his PhD in Computer Science at Stanford University in Fall 2025, where his research would be supported by the **NSF GRFP** fellowship.

To those who stubbornly refuse to give up despite circumstances suggesting otherwise.

ACKNOWLEDGEMENTS

First and foremost, I thank God, the Author of all things good, for sustaining me through every moment of this journey. He remained faithful in seasons of clarity and in shadows of doubt. Whatever merit this work holds is but a fragment of the grace I have received.

To my advisor, **Dr. Rachee Singh**, thank you for the trust, the freedom, and the unwavering support! You gave me the space to think creatively, and your mentorship has been a true gift. I am especially grateful for your generosity, both professionally and personally. Your unique willingness to support your students holistically has helped me blossom as a well-rounded researcher. I also appreciate the lab resources (especially the GPUs) you made readily accessible. Much of the preliminary exploration and hypothesis testing for this research took place on them, and they contributed immense value to this thesis.

To my minor field member, **Dr. Giulia Guidi**, thank you for being an excellent mentor and for your generosity with time and resources. Your provision of high-performance computing infrastructure made a tangible difference in this work, and I recognize the immense privilege it is to have had access to such powerful systems.

To my second minor field member, **Dr. Chris De Sa**, I am deeply grateful for the opportunity to have learned Machine Learning Systems (MLSys) from you. Your passion and enthusiasm for the field is infectious, and your teaching is exceptional. If I could take your MLSys class again, I would in a heartbeat!

To the **Christ Chapel Choir ('24/25)**, thank you for being my second family away from home. I joined you not knowing how to read sheet music, yet you welcomed me warmly and gave me room to grow rapidly. That kindness means more to me than I can express. One of my fondest memories was singing Handel's *Hallelujah Chorus* together; it was both exhilarating and quite fun! I am especially thankful to Dr. Art Ostrander (Choir Director) for his technical precision and leadership; Carrie Ostrander (Soprano); Dr. Deborah Martin (Alto and pianist); Jyying Juliana Kan (Soprano); Margaret Brodhead (Pianist); Jim and Cindy Van Duren (Bass and Soprano); Amy Blumenthal (Alto); and Debbie Axtell (Alto). You have all helped shape the slightly more competent musician and vocalist I am today.

To my colleagues at Cornell who made this journey richer: the CS MS classes of '24 and '25, Dr. Singh's research group, Julian Bellavita, and the broader systems community, thank you for the fun events, collaboration, and conversation.

Finally, to my parents—your quiet strength and unwavering support have been immeasurably valuable to all my academic endeavors. I would not be here without you. Thank you.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

State-of-the-art large language models (LLMs), including DeepSeek-v3 [13], LLama4 [4], DBRX [46] and Snowflake Arctic [47], have adopted the Mixture-of-Experts (MoE) [48] architecture for its computational efficiency [45] and reliable performance across language modeling tasks [13, 4, 24].

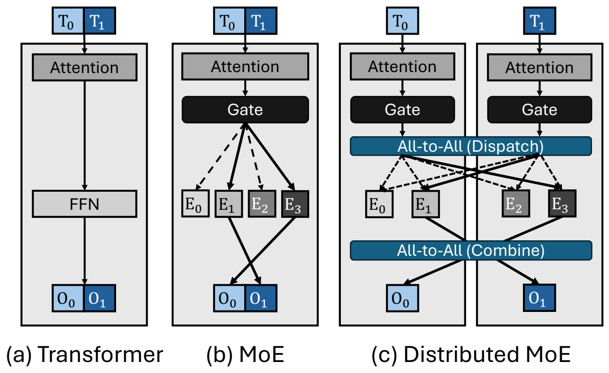


Figure 1.1: Transformer blocks (a) without MoE, (b) with MoE, and (c) with distributed MoE and expert parallelism. `T`, `E`, and `O` represent input tokens, experts, and output activations, respectively.

Depicted in Figure 1.1(a), the conventional Transformer block consists of a self-attention module followed by a feed-forward network (FFN) [53]. In contrast, MoE architectures replace this single FFN with identically sized FFNs, otherwise known as experts, (Figure 1.1(b)). A trainable neural network, known as a gate function, sparsely activates these experts by dynamically routing input tokens to selected experts at runtime. This increase in model parameters (more FFNs) improves model quality without a *corresponding increase in computational cost*.

1.1 Computational Cost Equivalence

The preceding claim seems counterintuitive because *shouldn't the increase in the number of experts yield a proportional increase in the model's computational operations?* The answer is no, due to how tokens are *distributed* across experts in comparison to the singular FFN. For example, consider a token matrix T as defined below where S is the sequence length and H the embedding dimension.

$$T \in \mathbb{R}^{S \times H}$$

The typical FFN operator, defined below,

$$\text{FFN}(x) = W_2 \cdot \phi(xW_1 + b_1) + b_2 \tag{1.1}$$

comprises two linear transformations on learnable weight matrices $W_1 \in \mathbb{R}^{H \times P}$, $W_2 \in \mathbb{R}^{P \times H}$ each followed by additions with bias terms $b_1 \in \mathbb{R}^{1 \times P}$, $b_2 \in \mathbb{R}^{1 \times H}$ and separated by a nonlinear activation ϕ (e.g., GELU [19] or ReLU [33]). Here dimension P is an intermediate projection for the FFN, typically $P = 4 \cdot H$ [30]. If we define $\mathcal{F}_{FFN}$ as the **F**loating **P**oint **OP**erations (FLOPs) needed to compute a forward pass of the FFN, then using Equation 1.1 we have the resulting expression.

$$\mathcal{F}_{FFN} = \mathcal{F}_{L_0} + \mathcal{F}_{L_1} \tag{1.2}$$

where $\mathcal{F}_{L_i}$ is the FLOPs cost for computing linear transformation i. These linear transformations are **GE**neral **M**atrix **M**ultiplications (GEMMs). We know that multiplying two matrices of sizes (M, K) and (K, N) demands $2MNK$ FLOPs, therefore we can expand Equation 1.2 as

$$\mathcal{F}_{FFN} = 2SHP + 2SHP = 4SHP \tag{1.3}$$

An MoE model differs from the dense transformer by *restricting* the number of tokens [28, 14] routed to an FFN (interchangeably called expert). Specifically, for a model with N_e experts, each expert has a fixed capacity for tokens S_e defined as follows

$$S_e = \frac{S}{N_e} \tag{1.4}$$

With the above, we can compute $\mathcal{F}_{MoE}$. Intuitively, this quantity would be the aggregate of $\mathcal{F}_{FFN_j}$ where $j \in \{0, \cdots, N_e - 1\}$.

$$\mathcal{F}_{MoE} = \sum_{j=0}^{N_e-1} \mathcal{F}_{FFN_j} \tag{1.5}$$

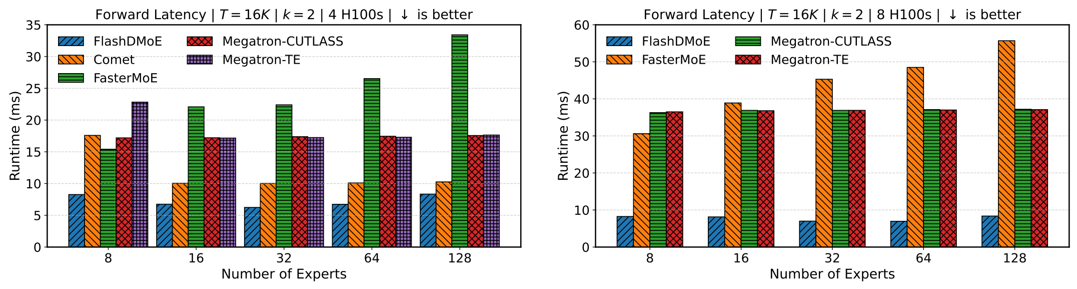
Observe that $\mathcal{F}_{FFN_j}$ is derivable from Equation 1.2 by replacing S with S_e. Applying this observation and evaluating 1.5 gives the below result

$$\mathcal{F}_{MoE} = N_e \cdot 4S_e HP \tag{1.6}$$

Substituting with 1.4, yields the below which proves that the computational cost is equivalent between the MoE and dense transformer models!

$$\mathcal{F}_{MoE} = 4SHP = \mathcal{F}_{FFN} \tag{1.7}$$

This relationship presents empirically as uniform latency independent of changes in N_e ($8 \rightarrow 128$), as *FlashDMoE*, and Megatron-{CUTLASS, TE} exhibit in Figure 1.2.



(a) 4 H100s (b) 8 H100s

Figure 1.2: Correlative trend between forward runtime and number of experts across different MoE operators.

1.2 Communication Overheads in Distributed MoE

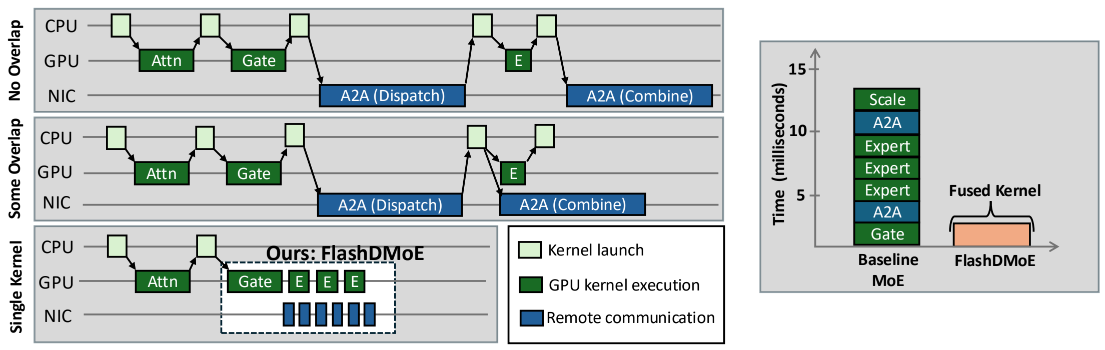


Figure 1.3: Comparing *FlashDMoE* with state-of-the-art techniques that either do not overlap communication and computation (left, top) or do some overlap (left, middle). *FlashDMoE* is a persistent kernel that fuses all computation and communication of the MoE operator (left, bottom). *FlashDMoE* implements device-initiated computation (gate, expert FFN, scale) and communication tasks (right).

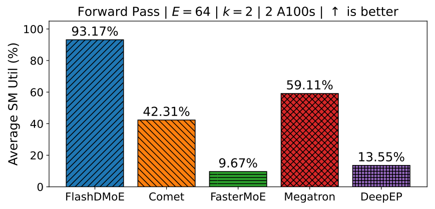
As MoE model sizes grow, GPU memory constraints prevent hosting all experts on a single device. The standard practice is to distribute experts across multiple GPUs using expert parallelism (EP), which requires token routing via many-to-many communication in *AllToAll* or *AllGather* [13, 47, 46, 50]. Another round of said many-to-many communication is also necessary for restoring the permuted tokens processed by experts to their original order within the sequence. Existing work has observed these communication operations taking 68% of total runtime [29, 25], during which the GPU is completely idle, unless the implementation explicitly overlaps with computation. This form of pipelining is challenging to achieve efficiently because it requires *asynchronous GPU-driven communication* and *kernel fusion* to maximize the overlap efficiency. Typically, inter-GPU communication APIs available in frameworks like PyTorch are not of this kind but instead are *CPU-driven* [1].

1.3 Kernel Launch Overheads in Distributed MoE

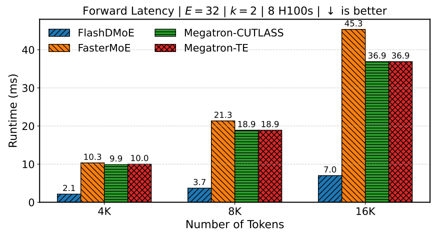
Works	**Launched GPU Ops**
FlashDMoE	1
COMET [59]	33
Megatron-LM CUTLASS [40, 35]	85
Megatron-LM TE [40, 35]	261
Megatron-LM + DeepEP [13]	432
DeepSpeedMoE [45]	550

Table 1.1: **Kernel Fusion Comparison.** Our method is the first to fully fuse the DMoE layer into a single GPU kernel. We report GPU operations from detailed profiling with Nsight Systems.

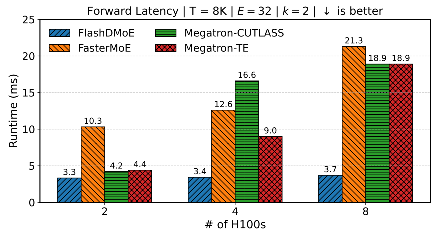
To mitigate these communication bottlenecks, recent work pipelines computation with communication tasks (Figure 1.3, middle left). However, The efficacy of communication overlap is further limited by the overhead of launching many kernels from the CPU. Specifically, existing implementations [45, 59, 40, 17] require launching a large number of kernels per a single layer pass (see Table 1.1). Frequent kernel launches negatively affect performance by: (1) creating non-deterministic kernel start times across GPUs, exacerbating straggler issues; (2) introducing unnecessary synchronization points, causing GPUs to wait on peers or the CPU before proceeding; and (3) incurring repeated global memory round trips at kernel boundaries. Although CUDA graphs [57] can partially mitigate the first issue in static workloads, they are incompatible with MoE's dynamically shaped tensors. Addressing the remaining issues requires novel solutions, which we provide in this work through complete kernel fusion and asynchronous device-initiated communication.



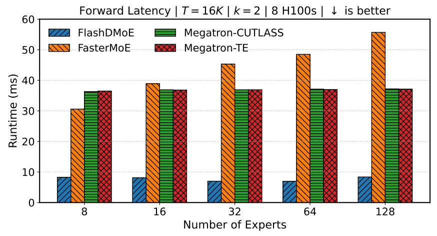
(a) GPU SM Utilization



(b) Scaling Tokens



(c) Weak Scaling across GPUs



(d) Expert Scalability

Figure 1.4: *FlashDMoE* demonstrating (a) improved SM Utilization (b) lower latency for longer token sequences (c) approximately ideal weak scaling and (c) ideal expert scalability.

1.4 This work's Contributions: Fast DMoE in a single kernel

To overcome these fundamental inefficiencies in state-of-the-art MoE models, we develop *FlashDMoE*, where we integrate all DMoE computation and communication tasks into a single persistent GPU kernel, namely a kernel that remains active for the entirety of the MoE operator (Figure 1.3 bottom left). Instead of multiple kernel launches coordinated by the CPU, *FlashDMoE* requires launching only one kernel, significantly reducing the involvement of the CPU. Within the fused kernel, *FlashDMoE* implements a reactive programming model to achieve fine-grained parallelism and loosely coupled, non-blocking execution among tens of thousands of GPU threads. This design enables *FlashDMoE* to efficiently utilize GPU resources by reducing idle time.

1.4.1 Warp Specialization and Tile Parallelism.

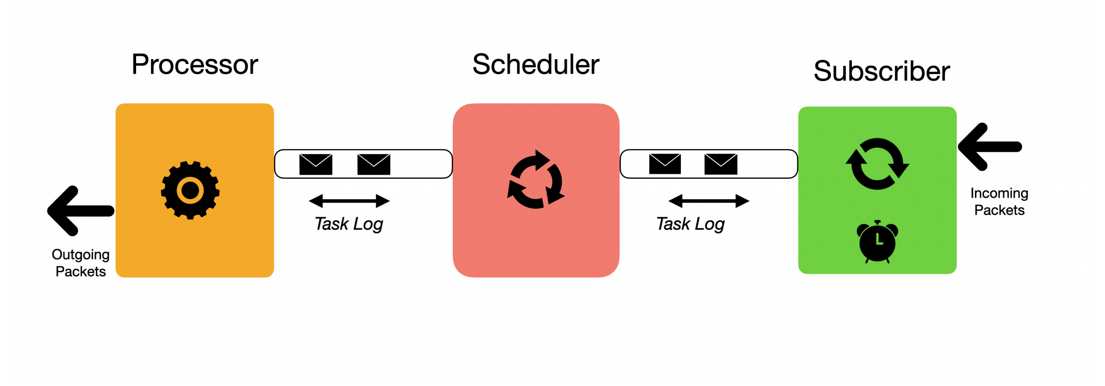


Figure 1.5: *FlashDMoE* Actors. The Subscriber observes received remote *packets* (blob of tiles), decodes them into *tasks* and notifies the scheduler of the enqueued tasks. We assign three CUDA warps to this role. The Scheduler maintains a *ready queue* of Processors from which it schedules tasks. The Processors performs the operation encoded in its task descriptor and eagerly communicates its output, if necessary.

FlashDMoE implements *tile-level parallelism*, meaning it partitions input token matrices into statically sized, *independent* tensors, called *tiles*, which are processed concurrently by a set of GPU thread warps comprising a **C**ooperative **T**hread **A**rray (CTA) or thread block. We specialize every thread block, except one, as *processors* to perform computation and remote data transfers. In addition, we designate a dedicated Operating System (OS) block (4 warps) to perform administrative tasks of (1) scheduling computational work to processors (*scheduler*), and (2) decoding computational tasks from messages received from other GPUs (*subscriber*). This design allows *FlashDMoE* to dynamically assign tasks to GPU blocks based on *readiness*, ensuring that no GPU SM remains idle throughout the lifetime of the DMoE operator. *FlashDMoE* selects tile dimensions to maximize GPU arithmetic intensity while still benefitting from a high-degree of parallelism.

1.4.2 Asynchronous and payload-efficient communication.

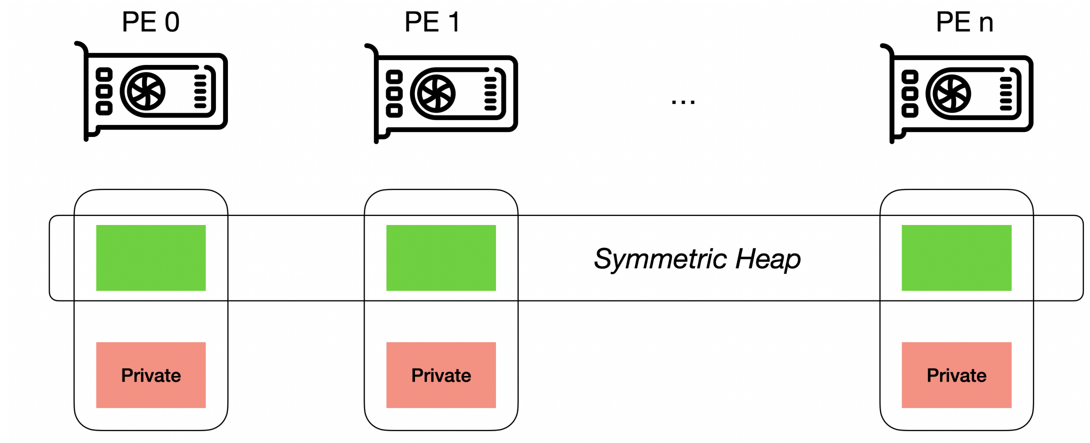


Figure 1.6: Depiction of a Partitioned Global Adress Space (PGAS) [58] across a *team* of **P**rocessing **E**lements (PEs). The symmetric heap is an unifromly sized memroy region resident on each PE and accessible by every other PE via one-sided **D**irect **M**emory **A**ccess (DMA) or **R**emote DMA (RDMA) operations.

By redesigning the MoE operator from the ground up, *FlashDMoE* resolves fundamental inefficiencies inherent in the conventional MoE execution pipeline. One notable inefficiency is token padding during communication. To simplify programming complexity and due to symmetry constraints of collective communication APIs, existing implementations have to zero-pad token payloads to match predefined buffer sizes. This occurs when tokens are asymmetrically routed to experts, resulting in GPUs receiving much less than the expected capacity. However, these null payloads waste communication bandwidth, bloat data transfer latency and may lead to unnecessary computations on null matrices in some implementations. *FlashDMoE* introduces *payload-efficient* communication by sending non-padded tokens only to GPUs with actively selected experts, conserving both communication and computational resources.

1.4.3 Technical Challenges

Realizing the single-kernel design of *FlashDMoE* required solving several technical challenges to achieve high performance: (1) lightweight computational dependency management; (2) navigating optimal SM occupancy configurations; (3) implementing in-device BLAS operations; (4) minimizing inter- and intra-device synchronization overheads; (5) implementing transfer-awareness by leveraging DMA over Unified Virtual Addressing (UVA) when available. In addressing these challenges, *FlashDMoE*'s design presents a radical departure from traditional synchronous *AllToAll* collectives, where GPUs exhibit significant idle time during layer execution. For device-initiated communication, *FlashDMoE* uses NVSHMEM [41] to establish a global address space across all GPUs to achieve the aforementioned Direct Memory Access (DMA) or Remote DMA (RDMA) communication. For in-device BLAS, *FlashDMoE* develops custom high-performance GEMM operations via CUTLASS [52].

1.4.4 Research Papers

This thesis comprises a first-author publication at ACM SIGMETRICS'24 [5] and another first-author submission in review at NeurIPS '25 [6].

CHAPTER 2

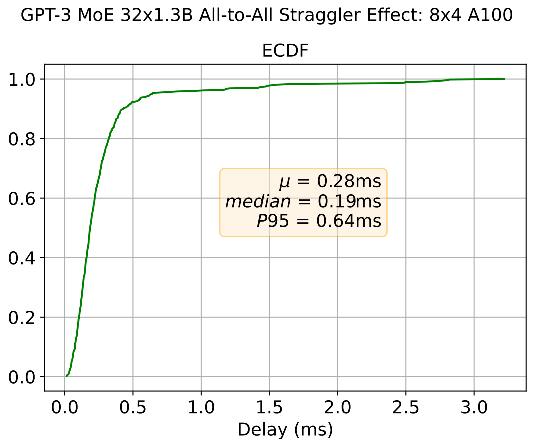
RELATED WORK

Computation-Communication Overlap. To reduce the communication overheads of synchronization in distributed DNN training, many research efforts have been focused on increasing the overlap of computation and communication. For generic Transformer-based models without MoE layers, many works [23, 56, 10, 43, 27, 51, 32, 54, 44] have provided insights and techniques to partition and schedule computation and communication operations, a imed at finer-grained overlapping. To address the challenges posed by *AllToAll* communication and expert parallelism in MoE training, Tutel [22] and FasterMoE [17] overlap *AllToAll* with expert computation. Lancet [26] additionally enables both non-MoE computation in forward pass and weight gradient computation in backward pass to be overlapped with *AllToAll*. Despite overlapping, the performance of these approaches is limited in practice due to blocking synchronous collective communication with barriers. In contrast, *FlashDMoE* fundamentally eliminates the inefficiencies with asynchronous, device-initiated data transfers overlapped with tiled computation all *within a single kernel*, further differentiating itself from SOTA works [60, 59, 13] who also use this form of kernel-initiated communication but at a coarse-grained granularity and without complete kernel fusion.

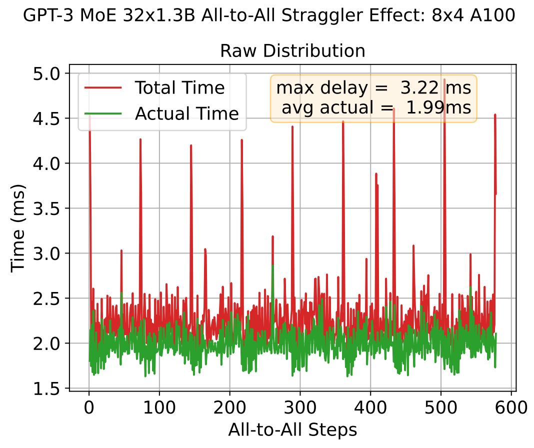
CHAPTER 3

MOTIVATION

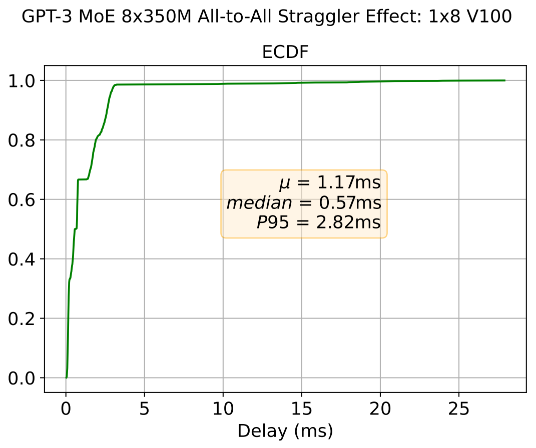
3.1 Straggler Effect



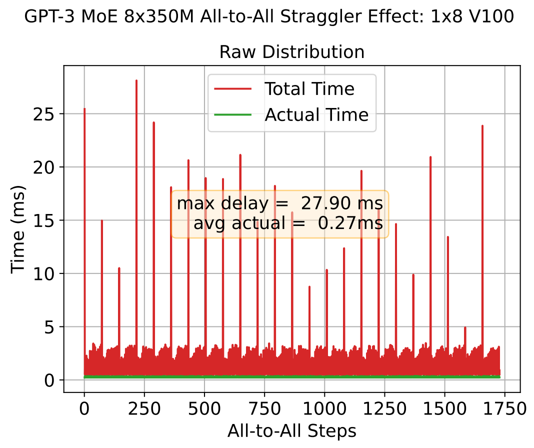
(a) ECDF



(b) Raw Distribution



(c) ECDF



(d) Raw Distribution

Figure 3.1: Straggler effect of synchronous *AllToAll*. $M \times N$ A100 or V100 denotes N GPUs within a node across M nodes. Every GPU communicates with every other GPU per step. **Actual Time** t_a is the fastest kernel execution time across all GPUs, and **Total Time** t is the maximum recorded step time. *Delay* is the maximum difference between t and t_a and is *idle time*. For the 1x8 V100, we profile 1750 steps and 600 steps for the 8x4 A100.

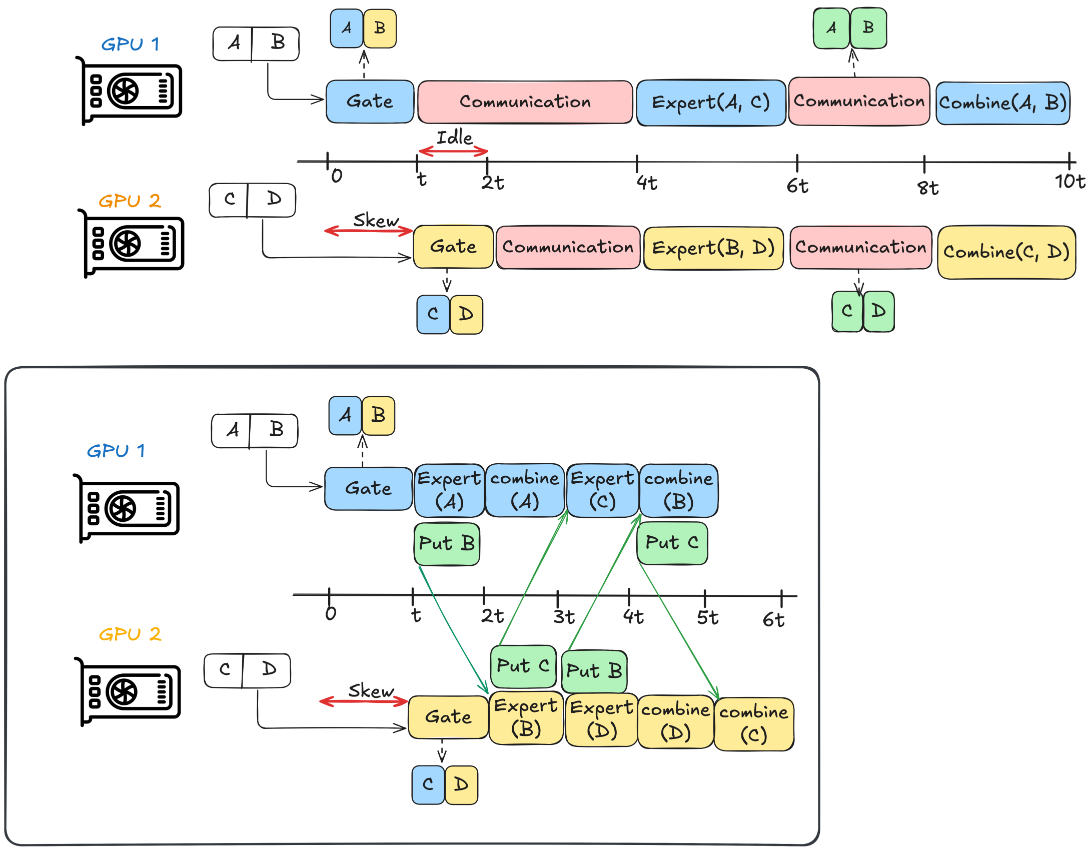


Figure 3.2: Overlapped Schedule (bottom) showing how idle time from the sequential schedule (top) is repurposed for computation. *FlashDMoE* implements the overlapped schedule.

AllToAll or *AllGather* communication as currently used in MoE frameworks is a synchronous collective operation, whose completion requires the participation of all involved GPUs. Here, disparities in processing speeds or kernel scheduling among GPUs induce a straggler effect detrimental (Figure 3.2) to (1) the collective operation's performance and (2) E2E performance, as stalled GPUs cannot proceed to downstream dependent or independent tasks until the collective terminates. Specifically, as shown in Figure 3.1, for distributed training of a 1.3B GPT-3 MoE model across 32 A100 GPUs, we see P95 communication performance degradation of **1.32X** when compared to the mean actual kernel time from Figure 3.1b. This performance reduction is rather tame as the underlying hardware is a supercomputer well-tuned against "software jitter" [36]. However, we observe a more severe p95 performance loss of **11X** in a single-node Virtual Machine (VM). In line with prior HPC works [8, 11], we argue that obviating the inherent barrier in this synchronous collective communication would allow GPUs to repurpose this observed idle time for downstream computation as depicted in Figure 3.2.

3.2 Kernel Launch Overheads

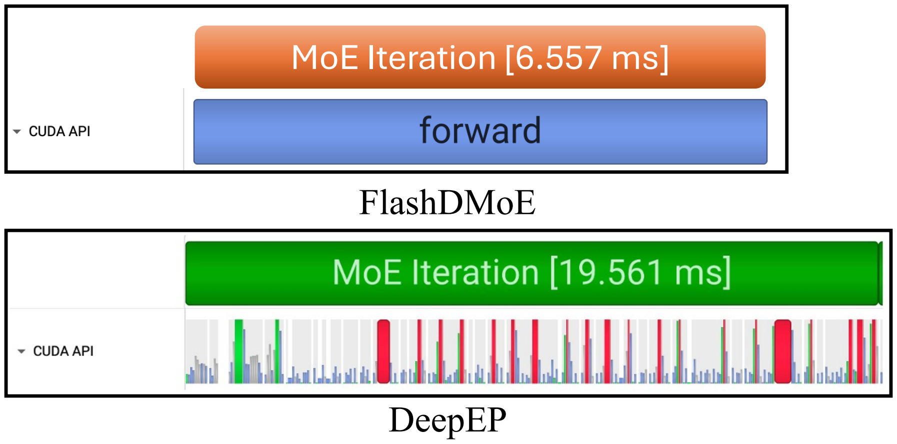


Figure 3.3: Kernel Launch overhead (CUDA API row) juxtaposed with runtime latency. Compared to DeepEP that launches 432 kernels, *FlashDMoE* launches a single one.

Table 1.1 shows the number of kernel launches during a single forward pass: *FlashDMoE* launches exactly one persistent kernel, while the baselines require up to 550 short-lived kernels. Figure 3.3 visually compares *FlashDMoE* and SOTA baseline DeepEP [13] using CUDA API traces captured by NSight Systems. DeepEP exhibits numerous small CUDA API calls, while *FlashDMoE* maintains high GPU utilization by avoiding launch overhead and synchronization gaps—achieving 93.17% GPU utilization (§6) compared to 14% for DeepEP.

CHAPTER 4

METHOD

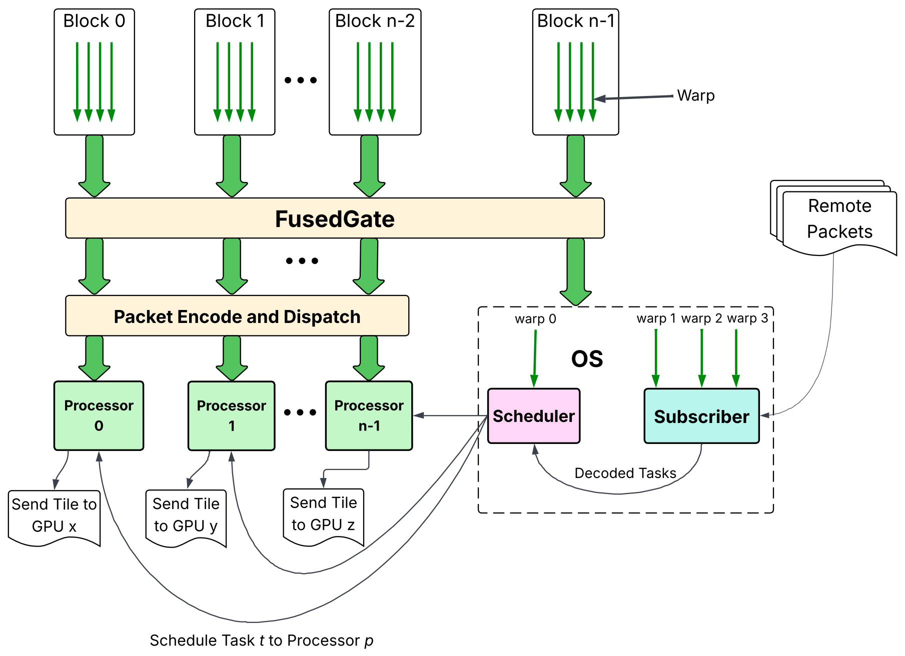


Figure 4.1: FlashDMoE *Fused Kernel*. Green arrows demonstrate block or warp specialization.

Modern distributed MoE systems suffer from two limitations: (1) frequent many-to-many (*AlltoAll or AllGather*) collectives on the critical path, and (2) significant overhead from repeated kernel launches. We address these in *FlashDMoE*, a fully fused MoE operator implemented as a single persistent GPU kernel. Unlike previous approaches [59, 13, 45, 40, 21, 25, 18, 37, 49, 55, 12], *FlashDMoE* is the first solution to implement a *completely fused Distributed MoE kernel*, eliminating kernel launch overhead entirely by requiring only a single kernel launch (see Table 1.1).

Algorithm 1: FlashDMoE *Distributed MoE Fused Kernel*

```
Input: $A, O \in \mathbb{R}^{S \times H},\ X \in \mathbb{R}^{E \times H \times D},\ N$
begin
    $T_\phi, G_\phi \leftarrow$ FusedGate($A$)
    if $blockId + 1 < N$ then
        Dispatch($T_\phi, A$)
        processor::start()
    else
        if $warpID == 0$ then
            scheduler::start()
        else
            subscriber::start($T_\phi$, $G_\phi$, $O$, $X$)
        end if
    end if
end
```

$$\begin{tikzcd}
D^j \arrow[r, "\text{Dispatch packets}"] & S_b^i \arrow[r, "\text{Notify Tasks}"] & S_h^i \arrow[r, "\text{Schedule Task}", "GEMM_0"'] & P^i \arrow[r, "\text{Notify Tasks}"] & S_h^i \arrow[r, "\text{Schedule Task}", "GEMM_1"'] & P^i \arrow[r, "\text{Send Tile}"] & S_b^j \arrow[r, "\text{Notify Tasks}"] & S_h^j \arrow[r, "\text{Schedule Task}", "\text{Combine}"'] & P^j
\end{tikzcd}$$

Figure 4.2: *DMoE Functional Dependencies Expressed as a Chain of Actor Interactions*. We denote S_b, S_h, and P as the Subscriber, Scheduler and Processor actors, respectively. For any actor $a \in \{S_b,\ S_b,\ P\}$, a^i identifies an actor on GPU i. We define D_i^j as the operator, where GPU j dispatches packets of tiles to GPU i, This diagram expresses task dependencies at the granularity of a tile, namely $GEMM_0$, $GEMM_1$, combine and communication produce an output tile. Notifications occur as signals propagated through shared memory (subscriber ↔ scheduler) or global memory (scheduler ↔ processor or inter-GPU communication). Note one-sided inter-GPU transfers (packet or single tile) are *coupled* with a signal to notify S_b^j on the receiving GPU j of the message's delivery.

4.1 Actor Model

The design of *FlashDMoE* is based on the actor model of concurrent computation [3, 20, 15]. We implement this model by specializing GPU thread blocks and warps into three distinct actor roles: (1) **Processor** (§4), (2) **Subscriber** (§2), and (3) **Scheduler**(§3). The Processor performs compute (GEMMs and element-wise operations) and tile communication. We use CUTLASS [52] as the underlying infrastructure for high-performance BLAS routines and NVSHMEM for kernel-initiated communication [41]. The Subscriber and Scheduler perform administrative functions. Specifically, the Scheduler assigns computational tasks to available thread blocks. Our key innovation is making the Scheduler both *multithreaded*, enabling high scheduling throughput, and *work-conserving*, ensuring consistently high GPU SM utilization. On the other hand, the Subscriber decodes *tile packets* from peer GPUs to task descriptors (§5.1). Of the N thread blocks on a GPU, we specialize $N - 1$ to adopt the **Processor** role. We specialize the last block as the Operating System (OS). Within this block, we specialize three warps for the **Subscriber** role and one warp for the **Scheduler** role. This split of thread blocks across actors is intentional: our goal is to use few resources for administrative tasks while reserving bulk of the resources for performing MoE computation tasks. Figure 4.1 summarizes the *FlashDMoE* architecture and its constituent actors, while Algorithm 1 gives a very close translation of the system in code. Note that $A \in \mathbb{R}^{S\times H}$ is the input token matrix; $O \in \mathbb{R}^{S\times H}$ the output matrix; and $X \in \mathbb{R}^{E\times H\times D}$ is a 3-D tensor of expert weights, where E denotes the number of local experts for the executing GPU, H is the embedding dimension, D is the FFN intermediate dimension and S is the sequence length. $T_\phi \in \left(\mathbb{R}^2\right)^{E\times C}$ is a routing table data structure, where $T_\phi(e, c) = (i, w)$ indicates that token i at slot c dispatches to expert e. w is the combine weight (Equation 5.1) and C is expert capacity. The tuple structure of T_ϕ is an implementation detail. $G_\phi \in \mathbb{R}^{S\times E}$ captures the affinity scores produced by the gate (Equation 5.2).

Global Memory	Shared Memory	Registers
80 GB	32 KB / Block	0.99 KB / thread

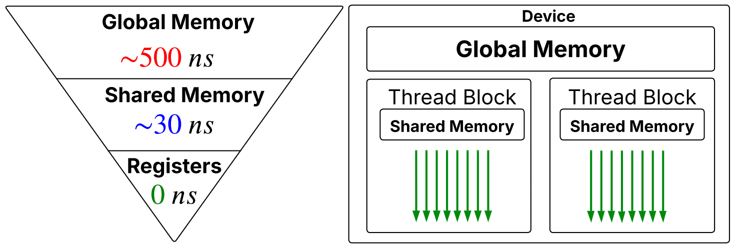


Figure 4.3: *GPU Memory Hierarchy*. The inverted pyramid (left) shows the load/store access latency [31, 2, 42]. Table above outlines the capacity for different memory tiers (for A100 GPUs). The shared memory and register capacity are static configurations for *FlashDMoE*. The right figure shows accessibility scopes: on-chip **registers** are scoped to a thread; on-chip **shared memory** is visible to all threads in a block; and off-chip **global memory** is accessible by all threads on device.

4.2 Inter-Actor Interactions

FlashDMoE decomposes MoE computation and communication at the granularity of a tile, a statically sized partition of a tensor, to achieve parallel execution and efficient overlap of tasks. Each tile maps to a discrete unit of work encapsulated by a *task descriptor*. The **Subscriber** decodes these task descriptors from the remote tile packets it receives. Concurrently, the **Scheduler** receives notifications about available tasks and dispatches them for execution to **Processor** actors that perform computations defined by these tasks, namely the feed-forward network (FFN) and expert-combine operations. Figure 4.2 show the chain of actor interactions, demonstrating how *FlashDMoE* enforces DMoE functional dependencies.

4.3 Tiling

Selecting appropriate tile dimensions in *FlashDMoE* is crucial to ensure efficient GPU utilization. An undersized tile underutilizes the GPU, while excessively large tiles create register pressure, causing performance-degrading register spills to local memory. After careful parameter sweeps, we choose tile dimensions of (128, 64). Our key insights are: increasing tile width significantly raises the register usage per thread, potentially triggering costly spills; increasing tile height without adjusting thread count increases workload per thread, harming performance. Raising the thread count per block beyond our fixed value of 128 threads reduces the number of concurrent blocks, negatively affecting SM occupancy. Larger thread-block sizes also increase overhead from intra-block synchronization (*__syncthreads()* barriers), further degrading performance. Thus, our chosen tile dimensions balance register usage, shared-memory constraints, and GPU occupancy to deliver optimal performance.

Algorithm 2: *Subscriber Actor*: executed by three warps

```
Input: $T_\phi \in \left(\mathbb{R}^2\right)^{E\times C}, G_\phi \in \mathbb{R}^{S\times E}\ O \in \mathbb{R}^{S\times H}, X \in \mathbb{R}^{E\times H\times D}$
begin
    $interrupt \leftarrow$ GetSharedInterrupt()
    $flags \leftarrow$ GetSymmetricFlags()
    $tQ \leftarrow$ GetTQ()
    // Predefined upper bound on the number of tasks.
    // Modulated to the actual task count computed
    // from dispatch signals received from peer GPUs
    $taskBound \leftarrow$ GetTaskBound()
    while AtomicLoad($interrupt$) == False do
        // dispatch flags
        do in parallel
            Visit dispatch flags
            Atomically retrieve signal
            if Signal is set and flag is not visited then
                Mark visited
                SelfCorrectTaskBound($taskBound, Signal$)
                Enforce memory consistency before consuming packet
                Decode packet into a set of $GEMM_0$ task descriptors
                Write task descriptors to $tQ$
                Notify Scheduler of decoded tasks
            end if
        end
        Advance flags by number of dispatch flags length
        Atomically retrieve signal
        // combine signals
        do in parallel
            Visit combine flags: one per tile
            if Signal is set and flag is not visited then
                Mark visited
                Enforce memory consistency before consuming packet
                Decode packet into a set of combine task descriptors
                Write task descriptors to $tQ$
                Notify Scheduler of decoded tasks
            end if
        end
    end while
end
```

Algorithm 3: *Scheduler Actor*: executed by one warp

```
begin
    $scheduled \leftarrow 0$
    $tTB \leftarrow 0$
    $tqS \leftarrow \{\}$
    $pTDB \leftarrow$ GetProcessorDoorbell()
    $sTDB \leftarrow$ GetSubscriberDoorbell()
    $taskBound \leftarrow$ GetTaskBound()
    $tTB \leftarrow$ AtomicLoad($taskBound$)
    // circular buffer ready queue
    $rQ \leftarrow \{\}$
    // Populate ready queue with Processor ids
    PopulateRQ($rQ$)
    while $scheduled < tTB$ do
        $lt \leftarrow 0$
        do in parallel
            Sweep doorbells and populate task counts into $tqS$
            Aggregate locally observed task counts into $lt$
        end
        $qS, taskTally \leftarrow 0$
        // qS is the inclusive output
        WarpInclusiveSum($lt, qS, tasktally$)
        while $tasktally > 0$ do
            Repopulate $rQ$ with ready processor ids
            do in parallel
                Starting at $rQ[qS]$ signal processors about tasks from $tqS$
            end
        end while
        if $threadId == 0$ then
            $tTB \leftarrow$ AtomicLoad($taskBound$)
        end if
        $tTB \leftarrow$ WarpBroadcast($tTB$)
    end while
    InterruptSubscribers()
    InterruptProcessors()
end
```

Algorithm 4: *Processor Actor*: executed by a block

```
begin
    $tQ \leftarrow$ **GetTQ**()
    $signal \leftarrow 0$
    // shared memory variables
    $task \leftarrow \{\}$
    $interrupt \leftarrow$ **False**
    $complete \leftarrow$ **False**
    **while** $interrupt ==$ **False do**
        **if** $warpId == 0$ **then**
            **if** $threadId == 0$ **then**
                **awaitTaskFromScheduler**($interrupt$, $signal$)
                **FencedNotifyRQ**($ready$)
            **end if**
            **syncwarp**()
            **warpReadTQ**($tQ$, $signal$, $task$)
        **end if**
        **syncthreads**()
        **if** $interrupt ==$ **False then**
            **switch** $task.Type$ **do**
                **case** $GEMM_0$ **do**
                    // fused GEMM, epilogue and async tile
                       staging
                    **fGET**($GEMM_0$, $task$)
                    **if** $threadId == 0$ **then**
                        $complete \leftarrow$ **NotifyTileCompletion**()
                    **end if**
                    **syncthreads**()
                    **if** $complete ==$ **True then**
                        **NotifySchedulerNextGEMM**($tQ$)
                    **end if**
                **end case**
                **case** $GEMM_1$ **do**
                    // fused GEMM, epilogue and async tile
                       transfer
                    **fGET**($GEMM_1$, $task$)
                **end case**
                **case** $Combine$ **do**
                    **combine**($task$)
                **end case**
            **end switch**
        **end if**
    **end while**
end
```

CHAPTER 5
PROGRAMMING ABSTRACTIONS

5.1 Task

We describe the FFN in §1.1, so here we explicate the *combine* operation. The expert-combine operation, used in architectures like GShard [28] and DeepSeek [13], merges outputs from multiple experts by computing a weighted combination based on their affinity scores:

$$C_i = \sum_{j=1}^{k} g_{i,e} \tag{5.1}$$

$$\mathbf{h}_i = \sum_{j=1}^{k} \frac{g_{i,e}}{C_i} \cdot \mathbf{h}_i^k \tag{5.2}$$

Above, $i \in 0, S-1$ represents an input token index, $e = E_{i,k}$ identifies the k-th expert selected for token i, and $g_{i,e}$ is the affinity score indicating how relevant expert e is for token i.

5.2 Unified Abstraction

We unify the FFN and combine operations under a common abstraction called a *task*. Tasks provide a uniform interface for communicating tile-level work among Subscribers, Schedulers, and Processors. Formally, a task descriptor $t \in \mathcal{T}$ is defined as a tuple:

$$t = (\mathcal{M}, \star, \phi)$$

where $\mathcal{M}$ is a set of metadata (such as device ID, tile index), $\star$ is a binary tensor operation (specifically, matrix multiplication $\cdot$ or Hadamard product $\odot$), and ϕ

is an element-wise activation function (e.g., ReLU or identity). We define a task t operating on input tensors A, B, D, producing output tensor C, as follows:

$$\mathcal{F}_t(A, B, C, D) \coloneqq C \leftarrow \phi\,(A \star_t B + D) \tag{5.3}$$

The operator $\star_t$ (instantiated from $\star$) may behave differently depending on the task metadata $\mathcal{M}$, and the result of $A \star_t B$ is accumulated into D. We provide an example of task metadata in Figure 5.1.

```
#define GEMMs 2
struct __align__(16) Task {
    const byte* aData;
    array<const byte*, GEMMs> bData;
    array<byte*, GEMMs> cData;
    array<const byte*, GEMMs> dData;
    byte* rcData;
    uint64_t* flags;
    uint M;
    uint syncIdx;
    uint tileIdx;
    uint batchIdx;
    uint peerIdx;
    uint expertIdx;
    uint isPeerRemote;
    TaskType taskType;
    uint16_t tileSize;
    // Pad till 128-byte cache line
    uint padding[6] = {};
}
```

Figure 5.1: *Task Struct*. TaskType $\in \{GEMM_0,\ GEMM_1,\ Combine\}$

In practice, we implement each task defined by Equation 5.3 as a *single fused* `__device__` decorated function which the **Processor** (Algorithm 4) invokes at runtime. Fusion for t entails applying ϕ and the succeeding addition operation to registers storing the results of the binary operator $\star_t$. To illustrate its flexibility, we show how the FFN and expert-combine operations can be expressed using this task framework. Note that we omit the matrix multiplication symbol $(\cdot)$ for simplicity. Also, ϕ_1 can be any activation function, while ϕ_2 is the identity function. The FFN is expressed as:

$$t_1 = (\mathcal{M}, \cdot, \phi_1), \quad t_2 = (\mathcal{M}, \cdot, \phi_2),$$
$$\mathcal{F}_{t_1}(A, B_1, C_1, D_1) \coloneqq C_1 \leftarrow \phi_1 (AB_1 + D_1),$$
$$\mathcal{F}_{t_2}(C_1, B_2, C_2, D_2) \coloneqq C_2 \leftarrow \phi_2 (C_1 B_2 + D_2).$$

Whereas, the expert-combine operation is formalized as:

$$t_3 = (\mathcal{M}, \odot, \phi_2),$$
$$\mathcal{F}_{t_3}(A, S, C, C) \coloneqq C \leftarrow \phi_2 (A \odot S + C).$$

5.3 Symmetric Tensor Layout for Inter-GPU Communication

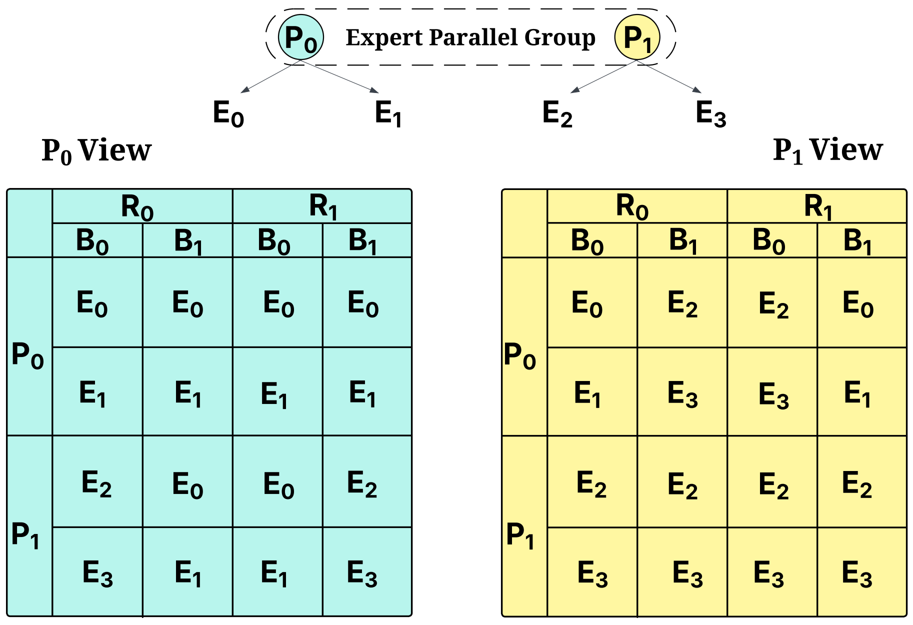


Figure 5.2: *Symmetric Tensor Layout across 2 Expert-parallel Processes.*

Within a single GPU device, the actors in *FlashDMoE* communicate through the GPU's memory subsystem (see Figure 4.3). Specifically, the Scheduler and Subscriber actors exchange data via fast shared memory, while other actor pairs communicate through global memory. For communication across multiple devices, *FlashDMoE* uses *device-initiated communication*, leveraging the one-sided PGAS (Partitioned Global Address Space) programming model [58]. However, achieving scalable and correct one-sided memory accesses in PGAS without costly synchronization is a known challenge [13, 60]. We address this challenge with a provably correct and scalable solution: a symmetric tensor layout L, supporting fully non-blocking memory accesses. We define L as:

$$L \in \mathbb{R}^{P \times R \times B \times E \times C \times H}$$

where: P is the expert parallel world size, R identifies communication rounds (two rounds, one for token dispatch and one for combine), B is number of staging buffers, E is the number of local experts, C is the upscaled expert capacity (§5.4) and H is the embedding dimension. Our core insight to enable non-blocking communication is *temporal buffering*. Specifically, we overprovision memory for the underlying token matrix by at least $2 \cdot r$ times, where r is the number of communication rounds in the dependency graph, and the factor of 2 accounts for separate buffers for incoming and outgoing data within each communication round. For MoE models, we have $2 \cdot r = 4$. This modest increase in memory usage eliminates the need for synchronization during one-sided data transfers. Figure 5.3 illustrates how cells within this symmetric tensor layout are indexed and used for Direct Memory Access (DMA) and Remote DMA (RDMA) operations. As Theorem 5.3.1 reinforces, this indexing scheme over L is the underlying mechanism that allows for fully non-blocking

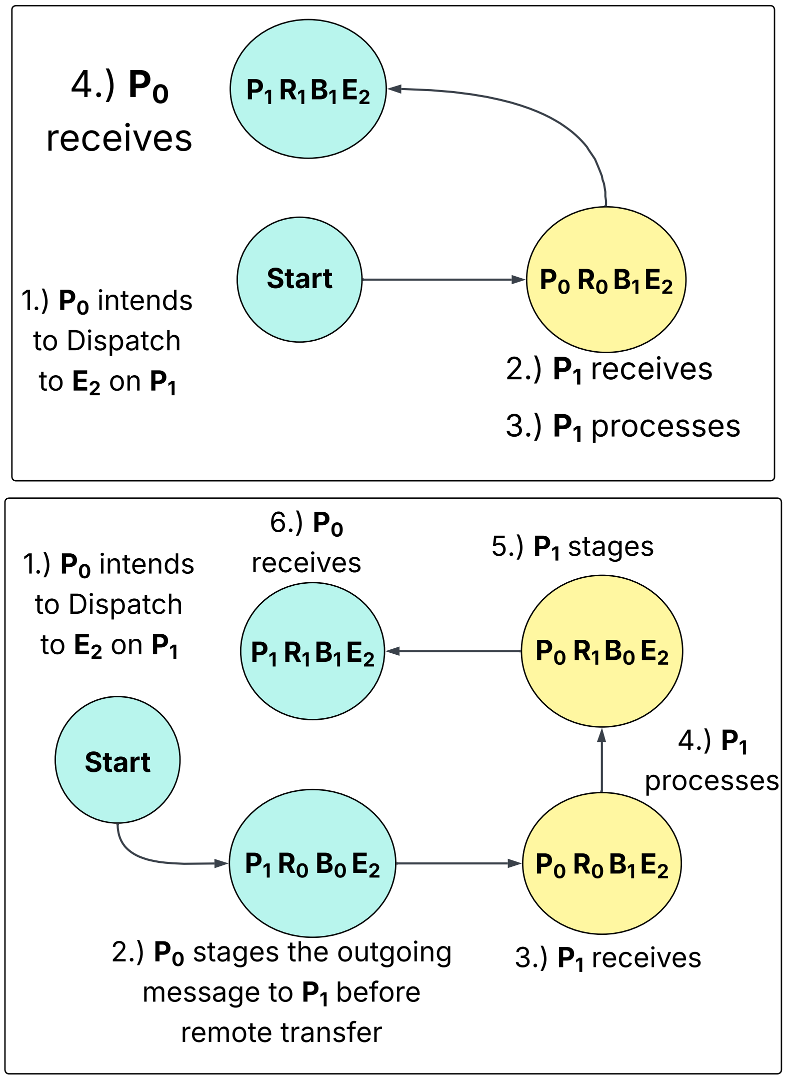


Figure 5.3: *State machine for DMA (top) and RDMA (bottom) communication.*

accesses eliding synchronization because all accesses are write *conflict-free*.

Definition 5.3.1. *Define a write as $w(p_s, p_t, i)$, where p_s is the source process and i is an ordered tuple indicating the index coordinates for L residing on the target process p_t. A write-write conflict occurs when there exist at least two distinct, un-synchronized, concurrent writes $w_1(p_{s_1}, p_{t_1}, i_1)$ and $w_2(p_{s_2}, p_{t_2}, i_2)$, such that $p_{t_1} = p_{t_2}$ and index coordinates $i_1 = i_2$ but $p_{s_1} \neq p_{s_2}$*

Definition 5.3.2. *For any source process p_s, a valid index coordinate $i = (p*, r, b, e, c)$ satisfies the following:*

1. *For inter-device writes, it must hold that $p* = p_s$ and $b = 1$. Note this also applies to self-looping writes $w(p_t, p_t, i)$.*
2. *For any write $w(p_s, p_t, i)$, if $b = 0$, then $p_s = p_t$. This rule describes intra-device staging writes.*

Theorem 5.3.1. *The symmetric tensor layout L is write-write conflict-free.*

Proof. As is the case for typical physical implementations, assume that each index coordinate i maps to a distinct memory segment in L. Next, we show by contradiction that no write-write conflicts can exist when accessing L using *valid i*. For simplicity, we only include the index coordinates when describing a write. Assume that there exist at least two writes $w_1(p_{s_1}, p_{t_1}, i_1)$, $w_2(p_{s_2}, p_{t_2}, i_2)$ with $p_{t_1} = p_{t_2}$ and valid destination coordinates i_1, i_2, where $i_1 = i_2$ lexicographically and both are unpacked below.

$$i_1 = (p_1, r_1, b_1, e_1, c_1),\ i_1 = (p_2, r_2, b_2, e_2, c_2)$$

Note that for the message staging state, even though $i_1 = i_2$ the resultant memory segments reside in different physical buffers resident in p_{s_1} and p_{s_2} respectively. Therefore, for this state, there are no conflicts as intra-process writes always have distinct c_j coordinates, where $j \in \{0, C-1\}$. For inter-process transfers, we have two cases.

Case 1: $p_{s_1} = p_{s_2}$

Here, w_1 and w_2 are identical operations. This contradicts the definition of a conflict, which requires that $p_{s_1} \neq p_{s_2}$. In practice, such repeat writes never even occur.

Case 2: $p_{s_1} \neq p_{s_2}$

To ensure validity for i_1 and i_2, it is the case that $p_1 = p_{s_1}$ and $p_2 = p_{s_2}$. However, this implies that $i_1 \neq i_2$ yielding a contradiction as desired. □

To construct L, we start from the original token buffer $T \in \mathbb{R}^{S \times H}$, where S is the sequence length and H is the hidden dimension. We first reorganize the sequence dimension S into three sub-dimensions representing the expert capacity (C), local expert slots (E), and the expert parallel world size (W), st:

$$C \cdot E \cdot W = C \cdot E' = S', \quad \text{where} \quad S' \geq S \text{ and } E' \geq E_W$$

In the typical case of uniform expert distribution (illustrated in Figure 5.2), we have $S' = S$ and $E' = E_W$, where E_W is the total number of experts in the model. Thus, the size of the token buffer is $Size(T) = S' \cdot H$. In Figure 5.2, each cell labeled E_i (with $i \in \{0, \ldots, 3\}$) is a matrix of size (C, H). Extending prior work [28, 59], we introduce additional temporal dimensions R (communication rounds) and B (staging buffers). Each communication round has two fixed staging slots: one for outgoing tokens and another for incoming tokens. Each slot, indexed by dimension P, forms a tensor of shape (S', H). Therefore, the tensor size $Size(L)$ is generally at least four times the original token buffer size, becoming exactly four times larger in the case of uniform expert distribution. Empirically (§6.8), we find:

$$Size(L) \approx 4 \cdot Size(T)$$

5.4 In-place Padding for Payload Efficiency

Due to the dynamic and uneven distribution of tokens in MoE dispatch [7], GPUs commonly receive fewer tokens than their predefined expert capacity.

Current MoE frameworks [45] typically pad these buffers with null tokens before computation, unnecessarily increasing communication payloads and degrading performance. In contrast, we propose *in-place padding*, performing padding directly within the local symmetric tensor buffers and thus eliminating excess network communication. As we show in Figure 5.2 as a reference, each cell E_i is sized according to the expert capacity C. We further align this capacity to ensure divisibility by the tile block size $bM = 128$, guaranteeing safe and aligned memory reads by Processor threads consuming remote tokens. This in-place padding strategy slightly increases the memory footprint of L, as described below:

$$Size(L) \approx \begin{cases} 4 \cdot Size(T), & \frac{S}{E} \geq bM \\ 4 \cdot \frac{bM \cdot E}{S} \cdot Size(T), & \text{otherwise} \end{cases}$$

CHAPTER 6

EVALUATION

Table 6.1: Implementation metrics of *FlashDMoE* using fully inlined NVSHMEM.

Metric	Value
Total lines of code (CUDA/C++)	6820
Kernel stack frame size	0 B
Spill stores (per thread)	0
Spill loads (per thread)	0
Shared memory usage (per block)	46 KB
Registers per thread	255
Max active blocks per SM	2
Compilation time	53 seconds
Binary size	29 MB

We implement (§6.1) and evaluate *FlashDMoE* and evaluate across five metrics: **Forward Latency** (§ 6.2), **GPU Utilization** (§ 6.3), **Overlap Efficiency** (§ 6.4), **Throughput** (§ 6.5), and **Expert Scalability** (§ 6.6).

6.1 Setup

We run experiments on a server with 8 NVIDIA H100 80G GPUs interconnected via NVLink, 125 GB of RAM, and 20 vCPUs. We used PyTorch 2.6.0, CUDA 12.8, and Ubuntu 22.04. All experiments use MoE transformer models configured with 16 attention heads, an embedding dimension of 2048, and an FFN intermediate size of 2048. We apply Distributed Data Parallelism (DDP) and Expert Parallelism for all experiments. We execute only the forward pass over a single MoE layer and measure the average runtime of 32 passes after 32 warmup passes. We use top-2 routing with a capacity factor of 1.0. We compare *FlashDMoE* against several state-of-the-art MoE systems: (1) **Comet** [59], (2) **Faster-**

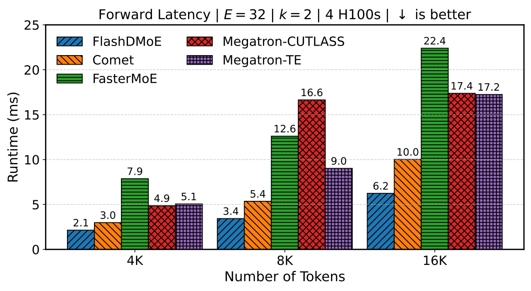
MoE [17], (3) **Megatron-CUTLASS** [34], and (4) **Megatron-TE**: Megatron-LM with Transformer Engine [39]. Comet relies on `cudaMemcpyPeerAsync` [9], while FasterMoE and Megatron-LM use NCCL exclusively for communication. We also evaluate *FlashDMoE* on a multi-node environment and discuss our findings in §6.7.

6.1.1 Desiderata

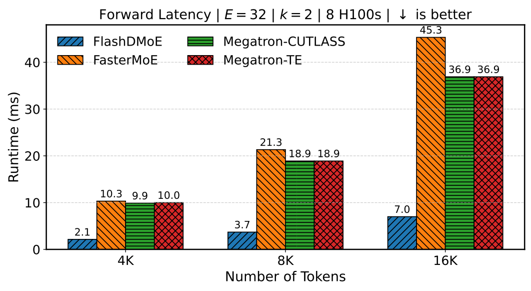
In our experiments, we observe Comet exhibiting anomalously bad performance values at 8 GPUs, so we exclude their results from evaluations at 8 GPUs and only include for results at ≤ 4 GPUs. **Note** we evaluate *FlashDMoE* using FP32 precision whereas all baselines use FP16. We do so because (1) no baseline supports FP32 and (2) time constraints prevent us from tuning our system to peak performance at FP16. Most importantly, this precision discrepancy disadvantages *FlashDMoE* by doubling the communication and computation precision, *making our results a conservative lower bound*. Yet, as we show in the succeeding sections, *FlashDMoE* outperforms all baselines.

6.2 Forward Latency

We first measure the forward latency of FlashDMoE across different sequence lengths on both 4 and 8 GPU setups (Figure 6.1). FlashDMoE consistently outperforms all baselines, with especially notable improvements at longer sequence lengths. On 4 GPUs, it achieves up to **4.6**x speedup over Megatron-TE at 16K tokens, and **2.6**x over FasterMoE. The gains are even more pronounced at



(a) 4 H100s



(b) 8 H100s

Figure 6.1: Forward Latency as the *Number of Tokens* per GPU increases.

8 GPUs where FlashDMoE maintains low latency, exhibiting up to **6.4**x speedup over baselines that degrade steeply due to increasing communication costs as token buffers increase proportionally. These results highlight FlashDMoE's ability to scale token throughput without suffering from the communication penalties that plague other implementations.

6.3 GPU SM Utilization

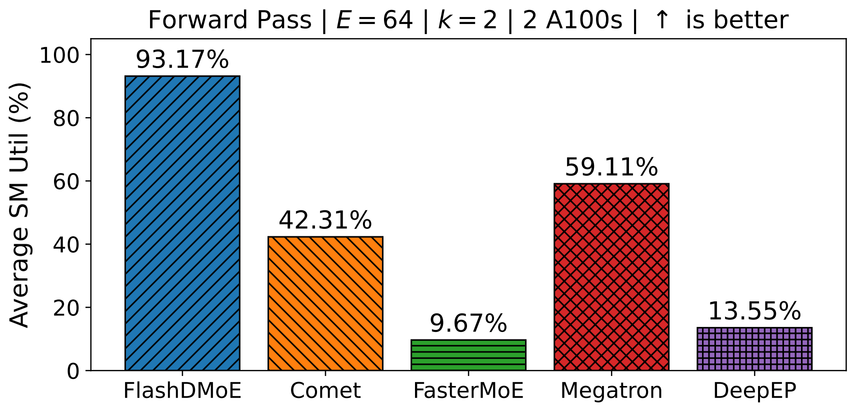
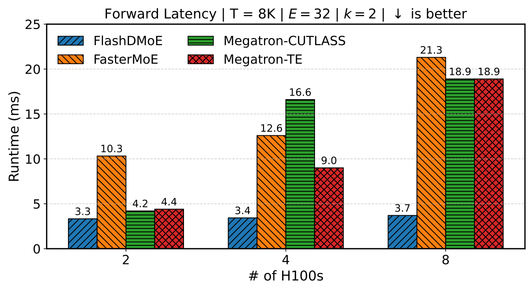


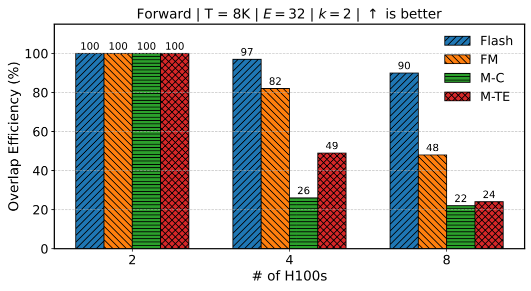
Figure 6.2: Comparison of SM utilization, defined as the ratio of cycles in which SMs have at least one warp in flight to the total number of cycles [38]. Values represent the average SM utilization over 100 iterations. All experiments use T = 8K and E = 64 on two A100s

To quantify GPU efficiency, we measure Streaming Multiprocessor (SM) utilization during the forward pass (Figure 6.2). FlashDMoE achieves 93.17% average SM utilization, over **9**x higher than FasterMoE (9.67%), **6.8**x higher than DeepEP+Megatron-LM (13.55%) **4**x higher than Megatron-TE (59.11%), and **2.2**x higher than Comet (42.31%). This improvement stems from our fully fused kernel architecture and fine-grained pipelining of compute and communication tasks. By eliminating idle gaps due to kernel launches and enabling in-kernel task scheduling, FlashDMoE ensures SMs remain busy with productive work throughout execution.

6.4 Overlap Efficiency



(a) Latency as Number of GPUs increases.



(b) Weak scaling efficiency.

Figure 6.3: Forward Latency as the *Number of Tokens* per GPU increases. We define Overlap Efficiency O_e to be $O_e = T(2)/T(N_G)$, where $T(N_G)$ is the latency at N_G GPUs and $T(2)$ is the latency at 2 GPUs.

We evaluate the extent to which *FlashDMoE* overlaps communication and computation by measuring weak scaling efficiency as the number of GPUs increases (Figure 6.3b). We note that most baselines fail to execute at a single GPU, hence why we use 2 GPUs as the reference point. We observe that Megatron-CUTLASS and Megatron-TE degrade significantly, with overlap efficiency dropping below 0.5 at $\geq$ 4 GPUs. *FlashDMoE* gives up to **3.88**x and **4**x higher efficiency at 4 and 8 GPUs, respectively. Figure 6.3a further illuminates this efficiency, as *FlashDMoE* shows stable forward latency growth, whereas baselines Megatron-CUTLASS and Megatron-TE experience approximately linear latency amplification while FasterMoE exhibits sublinear scaling. We attribute this suboptimal performance to straggler effects and exposed communication. In contrast, *FlashDMoE* demonstrates uniform latency as expected since the workload per GPU is fixed in this weak scaling experiment. These results further corroborate that *FlashDMoE*'s actor-based design and asynchronous data movement achieve near-ideal overlap, even at scale.

6.5 Throughput

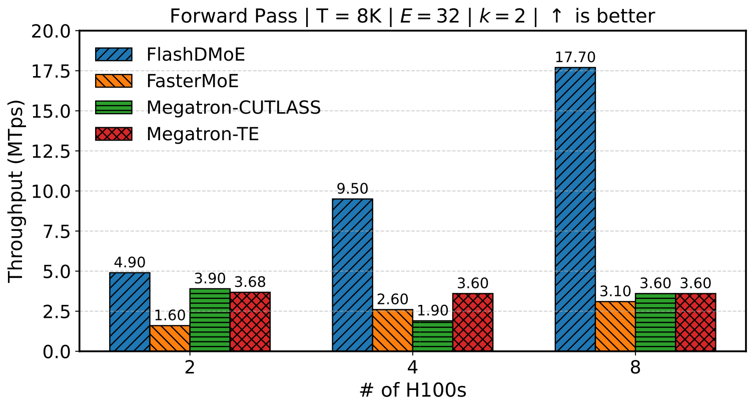
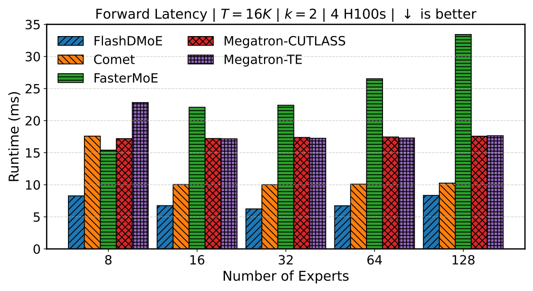


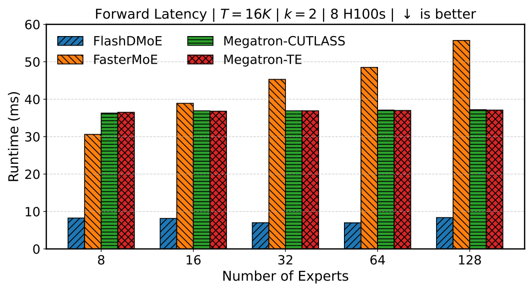
Figure 6.4: Throughput as the amount of GPUs increases. We compute throughput as $\frac{T * N_G}{latency}$, where N_G is the number of GPUs.

Throughput, measured in tokens per second (MTokens/s), reflects end-to-end system efficiency. As shown in Figure 6.4, *FlashDMoE* scales linearly with GPU count, reaching 17.7 MTokens/s at 8 GPUs. This is over **5.7**x higher than FasterMoE and **4.9**x higher than Megatron-TE and Megatron-CUTLASS. Notably, these results are achieved despite *FlashDMoE operating entirely in FP32,* *while baselines use FP16*. This indicates that FlashDMoE's design eliminates throughput bottlenecks not by exploiting lower precision, but by maximizing hardware utilization and eliminating host-driven inefficiencies.

6.6 Expert Scalability



(a) 4 H100s



(b) 8 H100s

Figure 6.5: Forward Latency as the *Number of experts* increases.

We analyze how *FlashDMoE* scales with increasing number of experts at fixed sequence length (T = 16K). Note that for the discussed plots, the number of experts on the x-axis is the *total number across all GPUs*. Each GPU gets 1/8th of this value. As seen in Figure6.5, *FlashDMoE* maintains *low, uniform* latency, as desired, even as the number of experts grows from 8 to 128. In contrast, baselines exhibit superlinear latency increases due to increased kernel launch overheads. *FlashDMoE* outperforms these baselines by up to **4**X at 4 H100s and **6.6**X at 8 H100s, both at 128 experts. *FlashDMoE* 's payload-efficient communication and scheduler-driven in-kernel dispatching allow it to sustain expert parallelism without incurring the communication and orchestration penalties seen in other systems. These results reinforce FlashDMoE's scalability for ultra-sparse MoE configurations.

6.7 Multi-Node Evaluation

In this experiment, we seek to evaluate *FlashDMoE* in the multi-node setting, where GPU interconnects span both InfiniBand (internode) and NVLink connections (intranode).

6.7.1 Setup

We use 4 nodes, where each node comprises 4 A100 GPUs fully interconnected via NVLink. Across nodes, each GPU uses a single NIC providing 25 GB/s of bandwidth. We set the number of experts to be 16 and assign each GPU to host only one, so the number of local experts is 1. Note that we define MIV formally as follows:

$$MIV = \frac{Tokens}{Experts} * local\_experts * precision * hidden\_size * 2 * n_{rg}$$

where n_{rg} is the number of remote peers and the multiplicative factor of 2 accounts for communication rounds (dispatch and combine). $n_{rg} = 12$ for this experiment.

6.7.2 Results

We observe a sublinear increase in latency as we scale the number of tokens. However, we observe at $Tokens > 2048$, that the application fails to terminate due to failure to receive expectant messages. We hypothesize this failure to be due to buffer overflow at the networking hardware layer as is common for applications that generate many and large messages [36] like our system. We

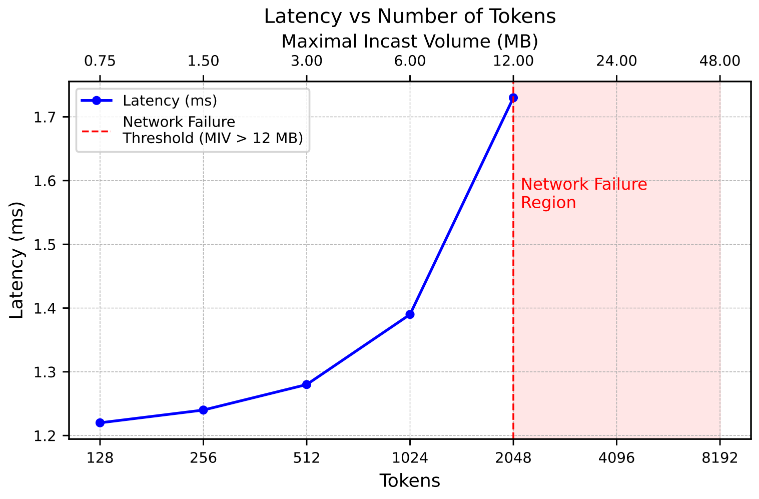


Figure 6.6: Multi-node Latency evaluation. Embbeding dimension is 1024 and FFN intermediate size is 4096. We define Maximal Incast Volume (MIV) as the worst case upper bound for data volume that a NIC receives in a single incast occurence.

note that this failure is addressable by tuning hardware configurations [16] but we consider this exploration as an exercise orthogonal to this work.

6.8 Memory Overhead

We measure the GPU memory required for the symmetric tensor L and runtime bookkeeping state of *FlashDMoE*. Memory overhead depends primarily on the tile size, expert capacity (EC), and the number of experts (E). Table 6.2 summarizes memory overhead under various configurations, confirming that *FlashDMoE* maintains a modest and predictable memory footprint.

Table 6.2: Memory overhead with (tile size $bM = 128$), $Size(T)$ = Tokens $* 4KB$).

Tokens	**Experts**	**EC**	**max(bM, EC)**	**Bookkeeping (MB)**	$Size(L)$ **(MB)**	**Total (MB)**
4K	16	256	256	64.57	64.00	128.57
4K	32	128	128	64.55	64.00	128.55
4K	64	64	128	128.90	128.01	256.91
4K	128	32	128	257.96	256.02	513.98
8K	16	512	512	128.95	128.01	256.95
8K	32	256	256	128.90	128.01	256.91
8K	64	128	128	128.90	128.01	256.91
8K	128	64	128	258.15	256.02	514.17
16K	16	1024	1024	257.89	256.02	513.90
16K	32	512	512	257.79	256.02	513.81
16K	64	256	256	257.80	256.02	513.81
16K	128	128	128	258.53	256.02	514.54

CHAPTER 7

LIMITATIONS AND FUTURE WORK

Despite the performance gains and architectural innovations of *FlashDMoE*, there are several limitations worth acknowledging—both practical and conceptual—that open the door to future research.

- **Programming complexity.** Developing fully fused, persistent kernels is a non-trivial engineering task. While *FlashDMoE* proves the feasibility and benefit of such kernels, their construction demands deep expertise in GPU architectures, synchronization and distributed protocols, and memory hierarchies. This high barrier to entry limits adoption. Future work may consider compiler-level abstractions or DSLs to democratize this technique.
- **FP16 support and shared memory bank conflicts.** Although modern GPUs natively support half-precision computation, adapting FLASHDMOE to FP16 is non-trivial for the Processor's computational operators. Specifically, our manually tuned swizzle shared memory layouts are not the most efficient template parameters for CUTLASS' Collective Mainloop operator which we use to implement our in-device GEMMs. This suboptimal configuration degrades memory throughput as shown in §A. Overcoming this for Ampere GPUs and below would require careful investigation of optimal layouts, but for Hopper GPUs and above, we anticipate using the builder interface that CUTLASS provides in our future improvements.
- **Lack of backward pass and training support.** While this work focuses on inference, enabling training requires fusing backward computation and

gradient communication into the kernel. Supporting this entails non-trivial changes to both memory bookkeeping and task descriptor definitions. Nevertheless, it remains an exciting direction for extending this system to fully support end-to-end training.

CHAPTER 8

CONCLUSION

This work introduces *FlashDMoE*, the first system to fuse the entire Mixture-of-Experts (MoE) operator into a single, persistent GPU kernel. We show that prevailing MoE implementations suffer from two critical inefficiencies: (1) CPU-managed synchronous communication that leads to underutilized interconnects and (2) fragmented execution via multiple GPU kernels, introducing overhead and synchronization delays.

In contrast, *FlashDMoE* embraces a model of GPU autonomy by embedding computation, communication, and scheduling within a unified kernel. It leverages actor-style concurrency, warp specialization, and asynchronous (R)DMA to achieve fine-grained communication–computation overlap.

Our evaluation demonstrates up to **6× speedup** over state-of-the-art systems, up to **9**× improved GPU utilization, and **5.7**× increased throughput for Distributed MoE. *FlashDMoE* challenges the dominant execution paradigms in distributed deep learning and presents a compelling template for building future GPU-native systems.

While several limitations remain, programming complexity and lack of FP16 support, this work lays the groundwork for a new era of *in-kernel distributed computation*. Future systems may build upon this foundation to enable kernel fusion for entire training pipelines, ushering in a design shift from CPU orchestration to fully autonomous GPU execution.

BIBLIOGRAPHY

[1] NVIDIA Collective Communications Library (NCCL). `https://developer.nvidia.com/nccl`.

[2] Hamdy Abdelkhalik, Yehia Arafa, Nandakishore Santhi, and Abdel-Hameed Badawy. Demystifying the nvidia ampere architecture through microbenchmarking and instruction-level analysis, 2022.

[3] Gul A. Agha. Actors: A model of concurrent computation in distributed systems. Technical report, 1985. MIT Artificial Intelligence Laboratory Technical Reports.

[4] Meta AI. The llama 4 herd: The beginning of a new era of natively multimodal ai innovation, 2025.

[5] Osayamen J Aimuyo. Aristos: Pipelining one-sided communication in distributed mixture of experts. *SIGMETRICS Perform. Eval. Rev.*, 52(4):3–5, March 2025.

[6] Osayamen Jonathan Aimuyo, Byungsoo Oh, and Rachee Singh. Flashdmoe: Fast distributed moe in a single kernel, 2025.

[7] Quentin Anthony, Yury Tokpanov, Paolo Glorioso, and Beren Millidge. Blackmamba: Mixture of experts for state-space models, 2024.

[8] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick. Optimizing bandwidth limited problems using one-sided communication and overlap. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, pages 10 pp.–, 2006.

[9] Bytedance. Flux's overlap performance is worse than non-overlap [4xrtx 4090], 2025.

[10] Chang Chen, Xiuhong Li, Qianchao Zhu, Jiangfei Duan, Peng Sun, Xingcheng Zhang, and Chao Yang. Centauri: Enabling efficient scheduling for communication-computation overlap in large model training via communication partitioning. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 24)*, pages 178–191, 2024.

[11] Yuxin Chen, Benjamin Brock, Serban Porumbescu, Aydin Buluc, Katherine Yelick, and John Owens. Atos: A task-parallel gpu scheduler for graph analytics. In *Proceedings of the 51st International Conference on Parallel Processing*, ICPP '22, New York, NY, USA, 2023. Association for Computing Machinery.

[12] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. In *Advances in Neural Information Processing Systems*, volume 35, pages 16344–16359. Curran Associates, Inc., 2022.

[13] DeepSeek-AI. Deepseek-v3 technical report, 2025.

[14] Trevor Gale, Deepak Narayanan, Cliff Young, and Matei Zaharia. Megablocks: Efficient sparse training with mixture-of-experts. In D. Song, M. Carbin, and T. Chen, editors, *Proceedings of Machine Learning and Systems*, volume 5, pages 288–304. Curan, 2023.

[15] Irene Greif. *SEMANTICS OF COMMUNICATING PARALLEL PROCESSES*. PhD thesis, Massachusetts Institute of Technology, 1975.

[16] OpenFabrics Interfaces Working Group. fi_cxi. `https://ofiwg.github.io/libfabric/v1.21.0/man/fi_cxi.7.html`, 2025. [Accessed 23-05-2025].

[17] Jiaao He, Jidong Zhai, Tiago Antunes, Haojie Wang, Fuwen Luo, Shangfeng Shi, and Qin Li. Fastermoe: modeling and optimizing training of large-scale dynamic pre-trained models. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 22)*, pages 120–134, 2022.

[18] Jiaao He, Jidong Zhai, Tiago Antunes, Haojie Wang, Fuwen Luo, Shangfeng Shi, and Qin Li. Fastermoe: modeling and optimizing training of large-scale dynamic pre-trained models. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, page 120–134, New York, NY, USA, 2022. Association for Computing Machinery.

[19] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus), 2023.

[20] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor

formalism for artificial intelligence. IJCAI'73, page 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[21] Changho Hwang, Wei Cui, Yifan Xiong, Ziyue Yang, Ze Liu, Han Hu, Zilong Wang, Rafael Salas, Jithin Jose, Prabhat Ram, HoYuen Chau, Peng Cheng, Fan Yang, Mao Yang, and Yongqiang Xiong. Tutel: Adaptive mixture-of-experts at scale. In D. Song, M. Carbin, and T. Chen, editors, *Proceedings of Machine Learning and Systems*, volume 5, pages 269–287. Curan, 2023.

[22] Changho Hwang, Wei Cui, Yifan Xiong, Ziyue Yang, Ze Liu, Han Hu, Zilong Wang, Rafael Salas, Jithin Jose, Prabhat Ram, et al. Tutel: Adaptive mixture-of-experts at scale. *Proceedings of Machine Learning and Systems (MLSys 23)*, 5:269–287, 2023.

[23] Abhinav Jangda, Jun Huang, Guodong Liu, Amir Hossein Nodehi Sabet, Saeed Maleki, Youshan Miao, Madanlal Musuvathi, Todd Mytkowicz, and Olli Saarikivi. Breaking the computation and communication abstraction barrier in distributed machine learning workloads. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 22)*, pages 402–416, 2022.

[24] Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mixtral of experts, 2024.

[25] Chenyu Jiang, Ye Tian, Zhen Jia, Shuai Zheng, Chuan Wu, and Yida Wang. Lancet: Accelerating mixture-of-experts training via whole graph computation-communication overlapping. In P. Gibbons, G. Pekhimenko, and C. De Sa, editors, *Proceedings of Machine Learning and Systems*, volume 6, pages 74–86, 2024.

[26] Chenyu Jiang, Ye Tian, Zhen Jia, Shuai Zheng, Chuan Wu, and Yida Wang. Lancet: Accelerating mixture-of-experts training via whole graph computation-communication overlapping. In *Proceedings of Machine Learning and Systems (MLSys 24)*, pages 74–86, 2024.

[27] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, et al.

{MegaScale}: Scaling large language model training to more than 10,000 {GPUs}. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 745–760, 2024.

[28] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.

[29] Juncai Liu, Jessie Hui Wang, and Yimin Jiang. Janus: A unified distributed training framework for sparse mixture-of-experts models. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 486–498, New York, NY, USA, 2023. Association for Computing Machinery.

[30] Xin Lu, Yanyan Zhao, Bing Qin, Liangyu Huo, Qing Yang, and Dongliang Xu. How does architecture influence the base capabilities of pre-trained language models? a case study based on ffn-wider and moe transformers. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 87411–87437. Curran Associates, Inc., 2024.

[31] Weile Luo, Ruibo Fan, Zeyu Li, Dayou Du, Qiang Wang, and Xiaowen Chu. Benchmarking and Dissecting the Nvidia Hopper GPU Architecture . In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 656–667, Los Alamitos, CA, USA, May 2024. IEEE Computer Society.

[32] Kshiteej Mahajan, Ching-Hsiang Chu, Srinivas Sridharan, and Aditya Akella. Better together: Jointly optimizing {ML} collective scheduling and execution planning using {SYNDICATE}. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 809–824, 2023.

[33] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, page 807–814, Madison, WI, USA, 2010. Omnipress.

[34] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan 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*, pages 1–15, 2021.

[35] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 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*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[36] NERSC. Network - NERSC Documentation. `https://docs.nersc.gov/performance/network/`, 2025. [Accessed 23-05-2025].

[37] Xiaonan Nie, Xupeng Miao, Zilong Wang, Zichao Yang, Jilong Xue, Lingxiao Ma, Gang Cao, and Bin Cui. Flexmoe: Scaling large-scale sparse pre-trained model training via dynamic device placement. *Proc. ACM Manag. Data*, 1(1), May 2023.

[38] NVIDIA. NVIDIA Nsight Systems Metrics.

[39] NVIDIA. Transformer engine.

[40] NVIDIA. Megatron-lm, 2025. v0.11.0.

[41] NVIDIA. Nvidia openshmem library (nvshmem), 2025. v3.2.5.

[42] NVIDIA. Ptx isa: Version 8.7, 2025.

[43] Suchita Pati, Shaizeen Aga, Mahzabeen Islam, Nuwan Jayasena, and Matthew D Sinclair. T3: Transparent tracking & triggering for fine-grained overlap of compute & collectives. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 24)*, pages 1146–1164, 2024.

[44] Kishore Punniyamurthy, Khaled Hamidouche, and Bradford M Beckmann. Optimizing distributed ml communication with fused computation-collective operations. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–17, 2024.

[45] Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. DeepSpeed-MoE: Advancing mixture-of-experts inference and training to power next-generation AI scale. In *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 18332–18346. PMLR, 17–23 Jul 2022.

[46] Mosaic Research. Introducing dbrx: A new state-of-the-art open llm, 2024.

[47] Snowflake AI Research. Snowflake arctic: The best llm for enterprise ai — efficiently intelligent, truly open, 2024.

[48] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc V. Le, Geoffrey E. Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[49] Shaohuai Shi, Xinglin Pan, Qiang Wang, Chengjian Liu, Xiaozhe Ren, Zhongzhe Hu, Yu Yang, Bo Li, and Xiaowen Chu. Schemoe: An extensible mixture-of-experts distributed training system with tasks scheduling. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 236–249, New York, NY, USA, 2024. Association for Computing Machinery.

[50] Siddharth Singh, Olatunji Ruwase, Ammar Ahmad Awan, Samyam Rajbhandari, Yuxiong He, and Abhinav Bhatele. A hybrid tensor-expert-data parallelism approach to optimize mixture-of-experts training. In *Proceedings of the 37th ACM International Conference on Supercomputing*, ICS '23, page 203–214, New York, NY, USA, 2023. Association for Computing Machinery.

[51] Weigao Sun, Zhen Qin, Weixuan Sun, Shidi Li, Dong Li, Xuyang Shen, Yu Qiao, and Yiran Zhong. CO2: Efficient distributed training with full communication-computation overlap. In *The Twelfth International Conference on Learning Representations (ICLR 24)*, 2024.

[52] Vijay Thakkar, Pradeep Ramani, Cris Cecka, Aniket Shivam, Honghao Lu, Ethan Yan, Jack Kosaian, Mark Hoemmen, Haicheng Wu, Andrew Kerr, Matt Nicely, Duane Merrill, Dustyn Blasig, Fengqi Qiao, Piotr Majcher, Paul Springer, Markus Hohnerbach, Jin Wang, and Manish Gupta. CUTLASS, 2025.

[53] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[54] Hulin Wang, Yaqi Xia, Donglin Yang, Xiaobo Zhou, and Dazhao Cheng. Harnessing inter-gpu shared memory for seamless moe communication-computation fusion. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 170–182, 2025.

[55] Hulin Wang, Yaqi Xia, Donglin Yang, Xiaobo Zhou, and Dazhao Cheng. Harnessing inter-gpu shared memory for seamless moe communication-computation fusion. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '25, page 170–182, New York, NY, USA, 2025. Association for Computing Machinery.

[56] Shibo Wang, Jinliang Wei, Amit Sabne, Andy Davis, Berkin Ilbeyi, Blake Hechtman, Dehao Chen, Karthik Srinivasa Murthy, Marcello Maggioni, Qiao Zhang, et al. Overlap communication with dependent computation via decomposition in large deep learning models. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 22)*, pages 93–106, 2022.

[57] Michael Wendt and Joshua Wyatt. Getting started with CUDA graphs. `https://developer.nvidia.com/blog/cuda-graphs/`, 2019. Accessed: 2024-05-15.

[58] Katherine Yelick, Dan Bonachea, Wei-Yu Chen, Phillip Colella, Kaushik Datta, Jason Duell, Susan L. Graham, Paul Hargrove, Paul Hilfinger, Parry Husbands, Costin Iancu, Amir Kamil, Rajesh Nishtala, Jimmy Su, Michael Welcome, and Tong Wen. Productivity and performance using partitioned global address space languages. In *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*, PASCO '07, page 24–32, New York, NY, USA, 2007. Association for Computing Machinery.

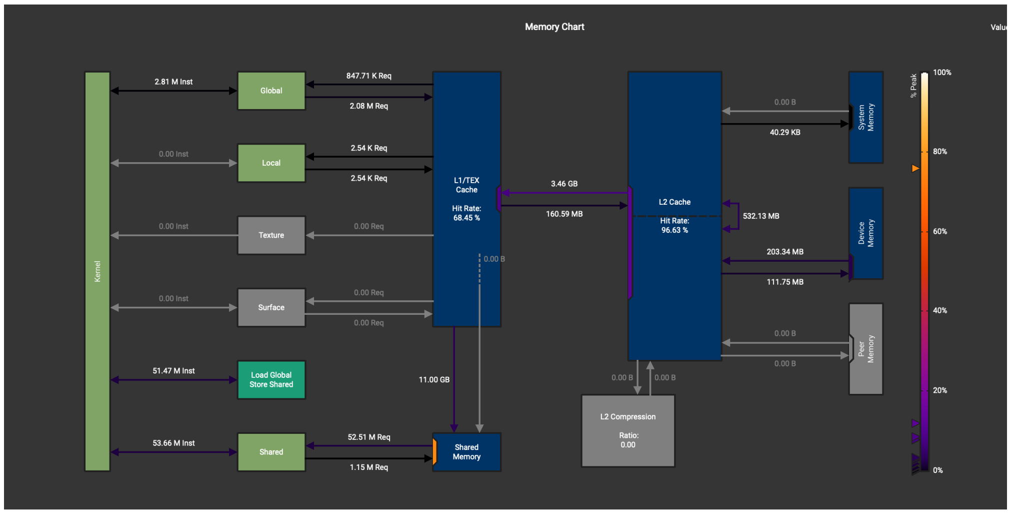
[59] Shulai Zhang, Ningxin Zheng, Haibin Lin, Ziheng Jiang, Wenlei Bao, Chengquan Jiang, Qi Hou, Weihao Cui, Size Zheng, Li-Wen Chang, Quan Chen, and Xin Liu. Comet: Fine-grained computation-communication overlapping for mixture-of-experts. In *MLSys '25*.

[60] Size Zheng, Wenlei Bao, Qi Hou, Xuegui Zheng, Jin Fang, Chenhui Huang,

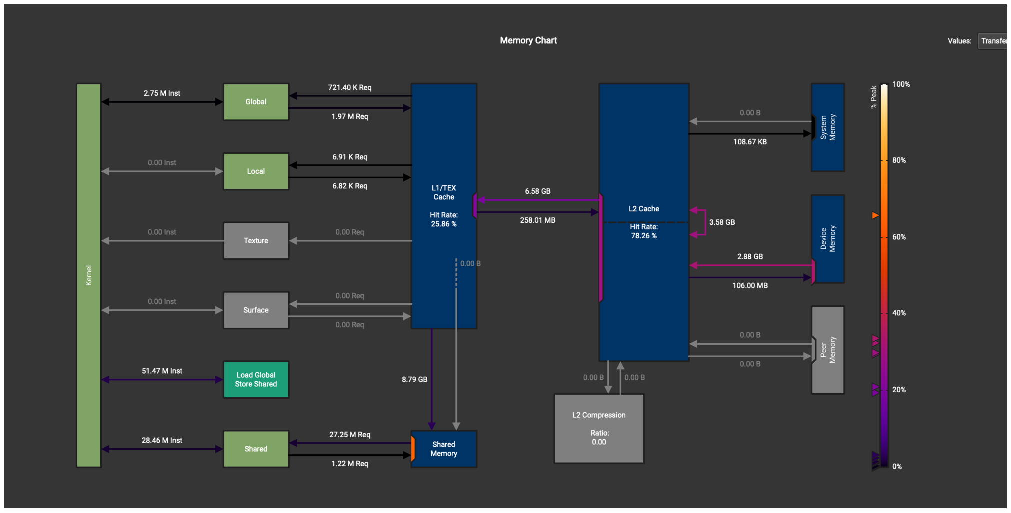
Tianqi Li, Haojie Duanmu, Renze Chen, Ruifan Xu, Yifan Guo, Ningxin Zheng, Ziheng Jiang, Xinyi Di, Dongyang Wang, Jianxi Ye, Haibin Lin, Li-Wen Chang, Liqiang Lu, Yun Liang, Jidong Zhai, and Xin Liu. Triton-distributed: Programming overlapping kernels on distributed ai systems with the triton compiler, 2025.

APPENDIX A

FP16 MEMORY THROUGHPUT



(a) Memory subsystem throughput for FP16



(b) Memory subsystem throughput for FP32

Figure A.1: Here, we report the total GPU memory throughput for both FP16 (top) and FP32 (bottom) variants of *FlashDMoE*. Notably, the FP16 implementation issues approximately 2× more shared memory instructions compared to its FP32 counterpart under identical workloads. We attribute this inefficiency to suboptimal shared memory layouts in *FlashDMoE* when operating on half-precision data. While this bottleneck is addressable through improved layout strategies, we leave its resolution to future work due to time constraints.