

Optimizing Distributed Large Model Inference: A Comparative Analysis of Interleaved Parallelism, Memory Virtualization, and IO-Aware Kernels

Team 10

Names: 吳亞宸, 林席葦, 鍾鎮嶸

Student IDs: R14944010, R14922092, R13944064

December 28, 2025

1 Introduction and Background

The precipitous rise of Large Language Models (LLMs) has fundamentally altered the landscape of high-performance computing and distributed systems. Models such as GPT-3, OPT-175B, and GLM-130B, characterized by hundreds of billions of parameters, have unlocked transformative capabilities in natural language understanding and generation.

However, the deployment of these models in production environments faces a daunting operational reality: the computational and memory demands of inference far exceed the capabilities of any single hardware accelerator. Consequently, distributed inference—spanning multiple Graphics Processing Units (GPUs) and often multiple compute nodes—has transitioned from a niche research topic to a critical infrastructure requirement.

This report presents a comprehensive research analysis centered on **Liger**, a novel runtime system that introduces **Interleaved Parallelism** to resolve the fundamental conflict between latency and throughput in distributed inference.

To provide a rigorous and holistic evaluation, this analysis integrates findings from three other seminal systems that address orthogonal bottlenecks in the inference stack: **vLLM**, which revolutionizes memory management via PagedAttention; **FlashAttention**, which resolves kernel-level memory bandwidth limitations; and **AlpaServe**, which addresses cluster-level statistical multiplexing.

1.1 The Inference Crisis: The Memory and Bandwidth Walls

The primary challenge in serving LLMs stems from their autoregressive nature. Unlike discriminative models (e.g., ResNet) where input sizes are static, generative models produce output tokens sequentially, with each step depending on the entire history of the sequence. This characteristic creates unique pressures on the hardware infrastructure, widely categorized as the "Memory Wall" and the "Bandwidth Wall."

The Memory Wall refers to the capacity limitations of accelerator memory (HBM). A 175-billion parameter model requires approximately 350GB of memory merely to store its weights in FP16 precision, necessitating a minimum of five NVIDIA A100 (80GB) GPUs just to load the model. Beyond weights, the transient state generated during inference—specifically the Key-Value (KV) cache—grows linearly with sequence length and batch size, consuming gigabytes of memory per request. As detailed in the vLLM research, existing systems suffer from catastrophic memory fragmentation, wasting 60% to 80% of available memory, thereby artificially limiting batch sizes and throughput [1].

The Bandwidth Wall refers to the disparity between compute speed (FLOPS) and memory transfer speeds. Modern GPUs have seen compute capabilities grow significantly faster than memory bandwidth. In standard attention mechanisms, the necessity to repeatedly read and write large attention matrices ($N \times N$) to High Bandwidth Memory (HBM) creates a bottleneck where powerful Tensor Cores sit idle waiting for data. This memory-bound nature makes standard attention implementations quadratic in time and memory complexity with respect to sequence length, severely penalizing long-context applications [1].

1.2 The Parallelism Dilemma

To span multiple devices, system architects rely on model parallelism. However, the Liger research identifies a critical "dilemma" in current parallelization strategies regarding the trade-off between cost (throughput) and effect (latency) [3]:

- **Intra-Operator Parallelism (Tensor Parallelism):** This strategy partitions individual weight tensors (e.g., splitting a matrix multiplication) across devices. While it effectively reduces the latency of a single inference step by aggregating compute power, it requires intensive collective communication (All-Reduce) after every operation. In environments with limited interconnect bandwidth (e.g., PCIe), this communication overhead dominates execution time, leaving compute units idle and reducing overall throughput.
- **Inter-Operator Parallelism (Pipeline Parallelism):** This strategy partitions the model by layers, assigning stages to different devices. While efficient in communication (point-to-point only), it processes requests sequentially through the pipeline. This introduces "pipeline bubbles" (idle times) and fails to reduce the latency for any single request; in fact, latency often increases due to pipeline overheads.

1.3 The Statistical Multiplexing Challenge

Compounding these device-level inefficiencies is the nature of production traffic. Workloads are rarely uniform; they are "bursty," with request rates spiking up to 50 times the average. Traditional serving approaches that allocate dedicated hardware to specific models suffer from low utilization. **AlpaServe** demonstrates that static provisioning for peak load is economically unsustainable. It proposes leveraging model parallelism not just for memory capacity, but for statistical multiplexing—partitioning models across shared resources to dynamically absorb bursts [1].

This report synthesizes these advancements to evaluate how Liger’s Interleaved Parallelism, supported by vLLM’s memory virtualization, FlashAttention’s IO-awareness, and AlpaServe’s placement strategies, defines the next generation of high-performance inference engines.

2 Survey Strategy and Methodology

Our survey methodology adopts a "full-stack" architectural analysis, examining the inference problem from the kernel level up to the cluster scheduler level. We utilize **Liger** as the central pivot point, analyzing how its scheduling innovations rely on and enhance the capabilities of the other three systems.

2.1 The Analytical Framework

We deconstruct the research material across four interconnected layers of the inference stack:

1. **The Kernel Layer (FlashAttention):** Analyzing how micro-optimizations in memory access patterns within the GPU (SRAM vs. HBM) alter the fundamental execution time of compute kernels.
2. **The Memory Layer (vLLM):** Analyzing how virtual memory management for tensors eliminates fragmentation, enabling the large batch sizes required for high throughput.
3. **The Execution Layer (Liger):** Analyzing how concurrent scheduling of compute and communication streams on multi-GPU nodes hides latency and improves resource utilization.
4. **The Orchestration Layer (AlpaServe):** Analyzing how model placement and cluster-wide scheduling manage bursty traffic and define the workload distribution that the lower layers must execute.

2.2 Taxonomy of Parallelism Strategies

To ensure precision in our comparative analysis, we adhere to the taxonomy defined across the Liger and AlpaServe literature:

Intra-Operator Parallelism (Tensor Parallelism): Splitting computation of a single operator across devices (e.g., Megatron-LM style).

Focus: Latency reduction. **Cost:** High communication bandwidth.

Inter-Operator Parallelism (Pipeline Parallelism): Splitting the computation graph into stages across devices.

Focus: Throughput/Memory. **Cost:** High latency, pipeline bubbles.

Interleaved Parallelism (Liger): A novel hybrid approach that interleaves the *computation* kernels of one request batch with the *communication* kernels of another batch on the same device.

Focus: Simultaneous latency and throughput optimization.

2.3 Evaluation Metrics

Our evaluation synthesizes data based on the following key metrics derived from the research materials:

- **SLO Attainment:** The percentage of requests completed within a strict latency deadline (critical for online services).
- **Throughput:** Tokens generated per second or requests processed per second.
- **Latency:** End-to-end response time for a single request (Time to First Token + Decode Time).
- **Memory Fragmentation:** The percentage of allocated HBM that does not store useful data (Internal + External fragmentation).
- **Burst Tolerance:** The ratio of peak traffic to average traffic a system can handle without SLO violation.

3 Liger: The Architecture of Interleaved Parallelism

Liger stands as the focal point of this report because it directly addresses the runtime scheduling inefficiencies that persist even after kernel and memory optimizations are applied. While Intra-Operator parallelism is the de facto standard for latency-sensitive inference, it is plagued by the "communication wall."

In a typical Transformer layer execution using Tensor Parallelism, the GPU alternates between computing matrix multiplications (GEMMs) and performing All-Reduce synchronization. During All-Reduce, the powerful Tensor Cores often sit idle, waiting for data to traverse the interconnect (NVLink or PCIe).

Liger proposes **Interleaved Parallelism**, a technique that reclaims these idle cycles. By treating the GPU not as a monolithic processor but as a collection of distinct sub-resources (Compute Units and Copy Engines/Interconnects), Liger schedules the *computation* phase of a secondary batch of requests to run concurrently with the *communication* phase of the primary batch.

3.1 The Function Assembler

The entry point to Liger’s runtime is the **Function Assembler**. In standard inference engines (e.g., PyTorch, FasterTransformer), the execution graph is relatively static. Liger’s Function Assembler dynamically intercepts incoming batches and generates a "Function Vector"—a serialized list of kernel launch wrappers. Crucially, the Function Assembler enriches these wrappers with metadata that is typically absent in standard execution:

- **Kernel Type:** Explicit classification of kernels as COMPUTATION (e.g., GEMM, Soft-max) or COMMUNICATION (e.g., NCCL All-Reduce).
- **Duration Estimation:** Predicted execution time based on input dimensions (Batch Size, Sequence Length).
- **Resource Requirements:** Estimates of Shared Memory usage and Register pressure.

This metadata allows the downstream scheduler to identify "matching pairs"—a communication kernel from the primary batch and a computation kernel from the secondary batch that have compatible durations [3]. The Function Assembler also manages the memory pointers for intermediate results, ensuring that the interleaved execution does not corrupt the state of either batch. This component effectively "linearizes" the complexity of the model graph into a schedulable stream of operations.

3.2 Multi-GPU Multi-Stream Scheduler

The heart of Liger is its scheduler. Standard GPU execution models (like CUDA streams) support concurrency, but they are non-deterministic. If a large GEMM kernel is launched on Stream 1, the GPU’s hardware scheduler (e.g., NVIDIA’s Gigathread) might allocate *all* Streaming Multiprocessors (SMs) to it. If a communication kernel is subsequently launched on Stream 2, it may be blocked until the GEMM finishes, even if it primarily requires link bandwidth, not SMs. This "left-over" scheduling policy is disastrous for latency-critical inference, as it can delay the communication of the primary batch.

Liger implements a deterministic **Multi-Stream Scheduler** that enforces precise orchestration. It prioritizes the **Primary Batch** (the latency-sensitive batch) while opportunistically inserting the **Secondary Batch** (the throughput-filling batch).

Principle of Operation:

- **Switch Point Detection:** The scheduler scans the Function Vector of the primary batch to identify the exact moment the model transitions from local computation to collective communication (the "Switch Point").
- **Duration Matching:** It accumulates the estimated duration of the upcoming communication kernels. It then scans the secondary batch's vector to find computation kernels that fit within this time window.
- **Stream Assignment:** It assigns the primary batch to a High-Priority Stream and the secondary batch to a Lower-Priority stream (though Liger's custom synchronization makes the hardware priority less critical) [3].

3.3 Hybrid Synchronization Mechanism

A major innovation in Liger is its **Hybrid Synchronization** approach. Controlling execution order typically involves the CPU: the CPU launches kernel A, waits for completion, then launches kernel B. However, the latency of CPU-GPU interaction (kernel launch overhead) can exceed 20 microseconds [3]. In the context of rapid inference steps, this overhead is prohibitive and negates the benefits of interleaving.

Liger bypasses this via a hybrid approach:

- **CPU-GPU Synchronization (Pre-Launch):** The CPU "pre-launches" distinct queues of kernels into separate CUDA streams. It does not wait for individual kernels to finish. This hides the driver overhead.
- **Inter-Stream Synchronization (GPU-Side):** To ensure the secondary compute kernel doesn't start too early (contending with primary compute) or run too long (delaying primary compute), Liger inserts `cudaEventRecord` and `cudaStreamWaitEvent` calls directly into the streams.

This creates a dependency graph *inside the GPU*. The secondary stream's compute kernel waits on an event recorded by the primary stream's compute completion. This ensures the overlap happens exactly during the communication phase, with zero CPU intervention during the critical path [3]. This tight loop control is what allows Liger to achieve precise interleaving without the jitter associated with OS-level scheduling.

3.4 Contention Anticipation and Mitigation

A naive view of GPU hardware assumes that Computation (Tensor Cores) and Communication (NVLink/PCIe) are orthogonal resources. Liger's analysis reveals this is false. They share critical resources: DRAM Bandwidth, SM Utilization, and Power/Thermal Limits. Running them blindly together causes "Resource Contention," slowing down the primary batch and violating SLOs.

Liger addresses this via **Contention Factors**. Before deployment, Liger runs an offline profiling phase where it deliberately runs compute and communication kernels concurrently to measure the "slowdown ratio." For example, on an NVIDIA A100 node with PCIe interconnects, the contention factor might be 1.15x (meaning kernels run 15% slower when overlapped) [3]. The scheduler uses these factors to "inflate" the estimated duration of kernels. It will only schedule an overlap if the *inflated* duration of the combined operation is still within the allowable latency budget.

Furthermore, Liger actively *mitigates* contention by tuning the communication library. It sets environment variables (e.g., `NCCL_NTHREADS`, `NCCL_MAX_NCHANNELS`) to limit the number of SMs consumed by communication kernels, reserving the bulk of compute power for the secondary batch's GEMMs [3].

3.5 Runtime Kernel Decomposition

The final piece of Liger’s architecture is **Runtime Kernel Decomposition**. Inference workloads are heterogeneous. A secondary batch might present a massive GEMM kernel that takes 10ms, while the primary batch’s communication window is only 5ms. Scheduling this GEMM would block the primary batch for 5ms, destroying latency performance.

Liger solves this by dynamically splitting large kernels into smaller ”micro-kernels” (tiles) at runtime.

- **Vertical Decomposition:** Liger favors splitting matrices along the batch or sequence dimension (vertical) rather than the hidden dimension (horizontal). Horizontal splitting would create skinny matrix multiplications that suffer from poor data locality and memory access efficiency [3].
- **Granularity Control:** By decomposing a large kernel into, for example, 8 smaller sub-kernels, the scheduler gains fine-grained control. It can insert 4 of these sub-kernels into the 5ms communication window and defer the remaining 4, perfectly filling the gap without overrunning it.

4 Complementary Technologies: The Foundation of High-Performance Inference

Liger’s scheduling logic assumes the existence of manageable memory footprints and efficient kernels. The feasibility of Liger in a real-world production environment is heavily dependent on the innovations provided by vLLM, FlashAttention, and AlpaServe.

4.1 vLLM and PagedAttention: Solving the Memory Fragmentation Crisis

Inference throughput is fundamentally bound by memory capacity. The larger the available memory, the larger the batch size the system can process. Traditional systems allocate memory for the KV cache in contiguous chunks based on the *maximum possible* sequence length (e.g., 2048 tokens), even if the actual request is short.

The Problem of Fragmentation: The vLLM paper provides a stark analysis of this inefficiency. In systems like Orca, memory waste is categorized into: Internal Fragmentation, External Fragmentation, and Reservation Waste. The data reveals that existing systems waste 60% to 80% of GPU memory. Specifically, Orca (Max) wastes 41.6% to internal fragmentation and 57.3% to reservation [1].

PagedAttention: vLLM introduces PagedAttention, borrowing the concept of virtual memory paging from operating systems. It divides the KV cache into fixed-size blocks (e.g., 16 tokens) that do not need to be contiguous in physical memory. A ”Block Table” maps logical sequences to physical blocks [1].

Impact on Liger: This is a critical enabler for Liger. Liger requires running a ”secondary batch” to fill idle time. If memory is fragmented, there may not be enough continuous space to load this secondary batch, causing the system to be memory-bound before it becomes compute-bound. vLLM reduces memory waste to under 4% [1]. This massive reclamation of memory allows Liger to fit the necessary concurrent batches into GPU VRAM, making Interleaved Parallelism physically possible.

4.2 FlashAttention: The IO-Aware Kernel Revolution

While vLLM optimizes memory *capacity*, FlashAttention optimizes memory *bandwidth*. Standard attention mechanisms scale quadratically $O(N^2)$ and are memory-bound—execution time is dominated by moving the $N \times N$ attention matrix between HBM and on-chip SRAM.

IO-Awareness and Tiling: FlashAttention introduces the concept of IO-Awareness. It restructures the attention computation into tiles that fit entirely within the fast on-chip SRAM (192KB per SM on A100).

- **Tiling:** It computes attention scores, softmax, and value aggregation in a single fused kernel pass, without writing the huge intermediate $N \times N$ matrix to HBM.
- **Recomputation:** In training (and applicable to the logic of memory savings), it recomputes attention on the fly during backward passes rather than storing it, trading cheap compute (FLOPS) to save expensive memory bandwidth [?].

IO Complexity Analysis: FlashAttention reduces HBM accesses from quadratic $\Omega(Nd + N^2)$ to linear $O(N^2d^2M^{-1})$, where M is SRAM size. This results in up to 9x fewer HBM accesses [?].

Impact on Liger: FlashAttention fundamentally changes the "shape" of the kernels Liger schedules.

- **Reduced Contention:** By drastically reducing HBM access, FlashAttention lowers the pressure on DRAM bandwidth. This directly reduces the **Contention Factors** Liger must account for. When compute kernels are less bandwidth-hungry, they can coexist more peacefully with communication kernels, allowing Liger to schedule overlaps more aggressively.
- **Faster Compute:** FlashAttention speeds up attention computation by up to 7.6x [?]. While this is generally good, it shortens the "compute" phase of the primary batch. Paradoxically, this might make Liger *more* relevant in bandwidth-constrained environments (like PCIe nodes), where communication time (which FlashAttention doesn't speed up) becomes an even larger percentage of total runtime, creating larger "bubbles" that Liger must fill with interleaved work.

4.3 AlpaServe: Statistical Multiplexing at the Cluster Scale

While Liger optimizes the execution within a node, **AlpaServe** optimizes the placement of models across the entire cluster. It addresses the macro-level inefficiency caused by **bursty workloads** and strict Service Level Objectives (SLOs).

4.3.1 The Statistical Multiplexing Opportunity

Production traces analyzed in AlpaServe reveal that request rates can spike up to 50x the average [4]. A traditional serving strategy of allocating dedicated GPUs per model (Static Replication) leads to severe underutilization, as expensive hardware sits idle between bursts.

AlpaServe introduces a paradigm shift by leveraging **Model Parallelism (MP)** for statistical multiplexing, even for models that fit within a single GPU.

- **The Trade-off Space:** AlpaServe identifies a fundamental trade-off between the *overhead* of model parallelism (communication and fragmentation) and the *opportunity* for multiplexing.
- **Mechanism:** By partitioning multiple models across a shared group of GPUs (Model Co-location), AlpaServe allows a burst of requests for *any* single model to utilize the aggregate compute power of the entire group. This "fan-out" capability significantly reduces tail latency during traffic spikes compared to restricting a model to a single device.

4.3.2 Automatic Parallelization for Inference

To enable efficient multiplexing, AlpaServe extends the auto-parallelization compiler from *Alpa* (originally for training) to optimize specifically for inference workloads:

- **Inter-Operator Parallelism (Pipeline):** Unlike training, inference requires only the forward pass. AlpaServe reformulates the dynamic programming algorithm to minimize the maximal stage latency without considering backward propagation or weight synchronization, resulting in balanced pipeline stages tailored for low latency.
- **Intra-Operator Parallelism:** The system filters out data-parallel configurations (which are handled by replication strategies) and focuses on tensor partitioning to reduce the latency of individual operators.

4.3.3 Placement Algorithm

Determining the optimal placement of models and their parallelization strategies is a complex combinatorial problem. AlpaServe employs a two-level hierarchical algorithm:

1. **Group Partitioning (Outer Loop):** The algorithm partitions the cluster into disjoint groups of devices and determines the optimal parallel configuration (e.g., 4-way pipeline parallelism vs. 2-way tensor parallelism) for each group.
2. **Simulator-Guided Model Selection (Inner Loop):** Given a group configuration, AlpaServe uses a trace-driven simulator to greedily select which models to place in which group. This simulator accurately predicts SLO attainment by modeling the arrival processes and execution latencies, ensuring that the chosen placement maximizes the number of requests served within the deadline.

Impact on Liger: AlpaServe acts as the high-level orchestrator. It determines *where* models run and *how* they are grouped. When AlpaServe co-locates a latency-sensitive model with a throughput-oriented background model on the same GPU group, it relies on a node-level runtime—like **Liger**—to efficiently interleave their execution kernels, ensuring that the statistical multiplexing decision translates to actual hardware efficiency without resource contention.

4.4 Robustness to Burstiness (AlpaServe Evaluation)

The most significant advantage of integrating model parallelism into the serving stack is the system’s resilience to unpredictable traffic spikes. AlpaServe’s evaluation on a 64-GPU cluster using production Azure Functions traces demonstrates this capability against state-of-the-art baselines like Clockwork++ [4].

Table 1: Cluster-Level Performance Comparison (AlpaServe)

Metric	Baselines	AlpaServe	Improvement
Throughput	Base Rate	10x Higher Rate	Massive capacity gain
Burst Tolerance	Base Burstiness	6x More Bursty	Superior stability
SLO Stringency	99% at 50ms	99% at 20ms	2.5x tighter deadlines
Resource Usage	100% GPUs	43% GPUs	Saves >50% hardware

Analysis:

- **Superiority over Reactive Scheduling:** Clockwork++ relies on reactively swapping models in and out of GPU memory to handle traffic. However, AlpaServe’s static model-parallel placement outperforms it because “fanning out” a burst of requests across multiple

GPUs (statistical multiplexing) provides immediate compute capacity without the latency penalty of memory swapping [4].

- **Handling Extreme Burstiness:** Experiments varying the Coefficient of Variation (CV) of the arrival process show that as traffic becomes more bursty (higher CV), the gap between AlpaServe and replication-based methods widens. AlpaServe can tolerate traffic that is **6x more bursty** while maintaining SLOs, effectively absorbing spikes that would overwhelm a single replica [4].
- **Robustness to Unknown Patterns:** Even when the actual traffic pattern deviates from the historical traces used for planning, AlpaServe maintains high performance. This confirms that statistical multiplexing via model parallelism is fundamentally more robust than precise but brittle scheduling algorithms [4].

5 Comparative Evaluation and Findings

This section synthesizes the quantitative results from the papers to demonstrate the efficacy of the combined approach.

5.1 Latency vs. Throughput Performance

The core metric for Liger is the ability to break the latency-throughput trade-off. The evaluation on an NVIDIA A100 node (4 devices) yields critical data points [3].

Table 2: Latency vs. Throughput Performance Analysis

Parallelism Strategy	Latency Impact	Throughput Impact	Mechanism of Action
Inter-Operator (Pipeline)	High (Base)	High (Base)	Sequential stage processing
Intra-Operator (Tensor)	-36.0% vs Inter-Op	Low (Comm. bound)	Parallel compute reduces latency
Liger (Interleaved)	-36.0% vs Inter-Op	1.34x vs Intra-Op	Matches Intra-Op latency

Analysis: Liger successfully matches the low latency of Intra-Operator parallelism. This confirms that the scheduler correctly prioritizes the primary batch and that the "Contention Anticipation" effectively prevents the secondary batch from interfering. The **1.34x throughput gain** confirms that the "secondary batch" is doing meaningful work. The gap between 1.0x and 1.34x represents the "wasted" time in standard Tensor Parallelism that Liger reclaimed. The results are sensitive to interconnect bandwidth. On V100 nodes with NVLink (higher bandwidth), the gain is smaller because the communication windows are shorter. On A100 nodes with PCIe (lower bandwidth), the communication windows are wider, giving Liger more opportunity to interleave, thus yielding higher gains [3].

5.2 Memory Efficiency and Batch Size

vLLM’s contribution is quantified by its reduction in memory waste, which directly translates to batch size capacity [1].

Table 3: Memory Efficiency Comparison

Metric	Existing Systems (Orca)	vLLM (PagedAttention)	Improvement
Memory Waste	60% - 80%	< 4%	Near-optimal utilization
Throughput	1x (Baseline)	2x - 4x	Enabled by larger batches

Analysis: This data validates vLLM as a foundational layer. The 2-4x throughput gain stems solely from being able to fit more requests into memory. For Liger, this implies that the "secondary batch" can be substantial, not just a few tokens. Without PagedAttention, Liger would likely be constrained to very small secondary batches, limiting the throughput gain from interleaving.

5.3 Speed and Scalability via IO-Awareness

FlashAttention’s impact is measured in wall-clock speedup and scalability to long sequences [?].

Table 4: FlashAttention Performance

Workload	Metric	FlashAttention Result
GPT-2 Training	Speedup	3x faster than baseline
BERT-Large	Speedup	15% faster than MLPerf record
Long Context	Max Sequence	Scales to 64k (Path-256 task)
HBM Access	Complexity	Linear $O(N)$ vs Quadratic $O(N^2)$

Analysis: FlashAttention’s ability to scale to 64k sequences with linear memory growth complements vLLM’s memory management. While vLLM manages the storage of the KV cache, FlashAttention optimizes the computation over it. For Liger, the 3x speedup in attention computation means the "compute" slots in the schedule are shorter. This requires the Liger scheduler to be even more precise (microsecond-level timing) and makes the Runtime Kernel Decomposition even more vital to fit work into these tighter windows.

5.4 Robustness to Burstiness

AlpaServe’s evaluation highlights the cluster-level benefits of model parallelism [?].

Table 5: Cluster-Level Performance (AlpaServe)

Metric	Traditional Serving	AlpaServe (Model Parallelism)
Burst Tolerance	1x (Baseline)	6x higher burstiness handled
Throughput	1x (Baseline)	10x higher rates handled
SLO Attainment	95% at 50ms	99% at 20ms (2.5x tighter)

Analysis: AlpaServe proves that model parallelism is a superior strategy for bursty traffic compared to simple replication. By allowing a burst of requests to fan out across multiple GPUs, latency is reduced. Liger enhances this by ensuring that when multiple models share a GPU group (as dictated by AlpaServe), their execution is interleaved efficiently rather than serialized, preventing the "noisy neighbor" problem from violating SLOs.

6 Conclusion and Future Outlook

The landscape of distributed large model inference is undergoing a rapid transformation, moving away from static, monolithic execution towards dynamic, fine-grained orchestration. The analysis of **Liger**, **vLLM**, **FlashAttention**, and **AlpaServe** reveals a converging stack of technologies that address the inference crisis from different angles.

Liger resolves the scheduling dichotomy, proving that latency and throughput are not mutually exclusive if one treats computation and communication as separate, schedulable resources. Its **Interleaved Parallelism**, powered by **Hybrid Synchronization** and **Kernel Decomposition**, effectively "hyper-threads" the GPU for deep learning workloads.

However, Liger’s success is contingent upon the ecosystem. **vLLM** breaks the Memory Wall, providing the virtualized memory space necessary to hold concurrent batches. **FlashAttention** breaks the Bandwidth Wall, preventing memory I/O from stalling the compute cores that Liger aims to keep busy. **AlpaServe** provides the high-level intelligence, routing bursty workloads to the appropriate model-parallel groups that Liger manages.

Future Outlook: We anticipate a unification of these layers. Future inference engines will likely integrate the ”OS-like” features of these systems into a single kernel:

- **Unified Schedulers:** A scheduler that combines AlpaServe’s cluster awareness with Liger’s kernel interleaving.
- **Hardware Support:** Future GPUs may include hardware support for ”Paged Tensors” (validating vLLM) and finer-grained async compute queues (validating Liger) to reduce the software overhead of these management techniques.
- **Kernel Evolution:** As architectures evolve (e.g., Hopper’s TMA), FlashAttention’s principles of IO-awareness will likely be baked directly into the compiler, making manual tiling less necessary but reinforcing the importance of memory hierarchy management.

In conclusion, the future of efficient AI serving lies not just in bigger chips, but in the intelligent virtualization and scheduling of the massive resources we already possess.

7 Team Contribution

Table 6: Team contributions and assigned research.

Team Member	Contribution Ratio	Contribution
吳亞宸	33.33%	Liger, FlashAttention, Discussion, Report Writing
林席葦	33.33%	Liger, AlpaServe, Discussion, Report Writing
鍾鎮嶸	33.33%	Liger, vLLM, Discussion, Report Writing

References

- [1] vLLM: Efficient Memory Management for Large Language Model Serving with PagedAttention.
- [2] FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness.
- [3] Liger: Interleaving Intra- and Inter-Operator Parallelism for Distributed Large Model Inference.
- [4] AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving.