Turbo: Efficiently Serving Long-Context Large Language Models with In-Network Aggregation

Ying Wan[1,2*] Yuchen Xu[3*] Chuwen Zhang[4] Yingsheng Huang[1] Yong Feng[5]
Wenquan Xu[5] Jialin Li[6] Mingwei Xu[5] Wenfei Wu[3,7] Congcong Miao[6]

[1]Southeast University [2]State Key Laboratory of Internet Architecture [3]Peking University
[4]Tsinghua Unigroup [5]Tsinghua University [6]National University of Singapore
[7]Beijing Key Laboratory of Software and Hardware Cooperative Artificial Intelligence Systems

Abstract

LLM supporting long contexts faces a critical memory bottleneck due to the linear growth of KV cache. Distributing the storage across multiple GPUs alleviates this burden but introduces significant communication overhead or traffic incast, especially during the decoding phase. We propose Turbo, a first-of-its-kind in-network aggregation system that accelerates long-context inference by offloading query broadcast and attention aggregation to switches. We address three key challenges to map complex attention mechanisms onto restricted switch hardware: (*i*) To bypass the switch's inability to buffer global states or perform complex operations, we devise *online table-based aggregation*, which decomposes global reduction into pairwise operations and approximates nonlinear functions via lookup tables. (*ii*) To circumvent the restriction on retroactive state access in RMT pipelines, we introduce a *rolling forward* scheme that propagates states to enable cross-stage updates. (*iii*) To mitigate aggregation stragglers caused by topology-induced load imbalance, we construct a *load-aware aggregation tree* that optimizes workload distribution. Evaluations on a Tofino2-based testbed show that Turbo reduces end-to-end inference latency by up to 37%. Large-scale simulations on NS-3 demonstrate that Turbo significantly outperforms state-of-the-art baselines in both inference latency and network traffic reduction with negligible accuracy loss.

CCS Concepts

• **Networks** → **Programmable networks**; **In-network processing**; • **Computing methodologies** → **Distributed artificial intelligence**.

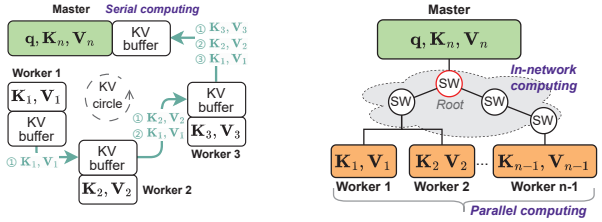
Keywords

Long-Context Inference, Large Language Models, KV Cache, In-Network Aggregation, Programmable Switches

ACM Reference Format:
Ying Wan, Yuchen Xu, Chuwen Zhang, Yingsheng Huang, Yong Feng, Wenquan Xu, Jialin Li, Mingwei Xu, Wenfei Wu, Congcong Miao. 2026. Turbo: Efficiently Serving Long-Context Large Language Models with In-Network Aggregation. In *ACM SIGCOMM 2026 Conference (SIGCOMM '26), August 17–21, 2026, Denver, CO, USA.* ACM, New York, NY, USA, 19 pages.

*Co-first authors.

This work is licensed under a Creative Commons Attribution 4.0 International License.
SIGCOMM '26, Denver, CO, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2467-1/26/08
https://doi.org/10.1145/3789240.3829140



(a) Ring-Attention (b) Turbo (ours)

Figure 1: Comparison of Ring-Attention and Turbo in decoding.

https://doi.org/10.1145/3789240.3829140

1 Introduction

Transformer [1, 11, 12, 74, 76, 84, 85] has become the backbone of Large Language Models (LLMs). To tackle complex tasks like reasoning, LLMs are evolving to support increasingly longer contexts, progressing from 2K tokens[1] in GPT-3 [11] to millions in Gemini Pro [68].

However, serving longer contexts is hindered by Out-Of-Memory (OOM) issues. LLM inference involves a *prefill* phase to process prompts and a *decoding* phase for autoregressive token generation. To avoid redundant computation, the *Key* and *Value* vectors of preceding tokens are cached in GPU as the *KV cache* [67]. Since the KV cache grows linearly with context length, decoding becomes memory-bound. For instance, a 175B GPT-3 model requires ~500 GB KV cache for a 100K-token context, far exceeding the 192 GB HBM capacity of the flagship NVIDIA B200 GPU [62]. Thus, supporting ultra-long contexts on a single GPU is impractical.

To address this, a promising approach is to leverage a distributed system, where the large KV cache is split into multiple KV blocks distributed among multiple GPUs. Earlier tensor parallelism (TP) [58, 77, 78] and pipeline parallelism (PP) [22, 29, 33, 56, 57] demonstrate limited scalability for long contexts, as the size of each partitioned KV cache grows proportionally with the context length. Instead, leveraging sequence parallelism (SP), also known as context parallelism (CP)[2] [15, 32, 41, 42, 47, 48, 60, 90, 93], is more attractive, which splits KV cache along the sequence length dimension and has been adopted in production-oriented systems such as NVIDIA TensorRT-LLM [61] and LoongServe [90].

State-of-the-art SP [47] strategies mainly fall into two categories: *Pass-KV* (a.k.a. *Q-stable*) and *Pass-Q* (a.k.a. *KV-stable*). Ring-Attention [48], typifying Pass-KV (Figure 1(a)), circulates KV blocks.

[1]Basic units of data (e.g., words, subwords, characters).
[2]We use sequence and context interchangeably.

While effective for training and prefilling, it incurs high latency during decoding, during which Pass-Q is widely adopted [82, 90, 93]. The master in Pass-Q broadcasts the query for parallel computation and aggregates partial results. However, as sequence length scales, the centralized aggregation in Pass-Q overwhelms the master, causing significant computational overhead and incast congestion.

Motivated by these inefficiencies, we propose Turbo, a novel Pass-Q based inference system that leverages *In-Network Computing (INC)* to break the scalability wall of long-context serving. As visualized in Figure 1(b), Turbo transforms the network from a passive carrier into an active computer, offloading the bandwidth-intensive query broadcast and the computation-intensive attention aggregation from the master GPU to switches. However, realizing this vision on commodity hardware (e.g., Tofino) entails overcoming significant *system-level* constraints—such as limited stage resources and lack of native floating-point support—and *algorithmic* hurdles regarding aggregation accuracy and load balancing. Turbo addresses these through three key designs (§3).

First, to enable complex attention aggregation (involving exponentiation and division) on resource-limited switches, we devise an *online table-based aggregation* mechanism. Unlike prior offline reduction approaches that buffer all partial attentions, or prior switch-based INC systems that mainly support simple sum/min/max-style reductions [39, 79], Turbo performs pairwise aggregation logic utilizing quantized lookup tables. We abstract the mathematical operations into concise primitives mapped to small hardware tables, ensuring high-throughput processing while maintaining negligible accuracy loss (§4.1).

Second, to reconcile the *strict stage locality* of Reconfigurable Match Tables (RMT) [8] with the stage-spanning operations, we propose a *rolling forward state update* mechanism. Since the hardware enforces single-stage state atomicity and prohibits retroactive updates, Turbo migrates the aggregation state downstream along the pipeline. This design decouples logical computation from physical stage boundaries, enabling aggregation for arbitrary context lengths, and addressing the cross-stage state-update limitation of RMT switches (§4.2).

Third, to resolve the conflict between *rigid topology mapping* and the *finite processing capacity* of individual switches, we design an *inter-switch load balancing* framework. Statically mapping aggregation trees to the physical network creates hotspots at confluences, where overloaded switches suffer from resource exhaustion and excessive packet recirculation. Turbo addresses this by formulating the tree construction as a Maximum Flow problem. This algorithm dynamically distributes the workload across the network fabric, ensuring each switch operates within its recirculation-free capacity to minimize tail latency (§4.3).

Finally, we bridge the gap between theoretical design and practical deployment through a full-stack implementation (§5). We prototype Turbo on Tofino2 switches, mapping the rolling forward logic to limited physical stages via resource-efficient quantization. We further develop a custom transport protocol to ensure reliable aggregation. To demonstrate generality, we also provide an FPGA-based implementation.

Comprehensive experiments on a testbed with a Tofino2 switch and 8 GPUs show that Turbo supports high-precision aggregation of 64-element chunks at line rate. On real-world workloads, Turbo reduces end-to-end decoding latency by up to 37% compared to Ring-Attention. Large-scale simulations on NS-3 demonstrate that Turbo can reduce inference delay by 99% and network traffic by orders of magnitude in massive clusters with negligible accuracy loss (§6).

2 Background and Motivation

In this section, we first introduce the LLM inference with KV cache (§2.1), followed by a discussion on existing SP approaches for long-context prefill and decoding phases (§2.2), and conclude with the motivation and opportunity to leverage in-network processing (§2.3).

2.1 LLM Inference with KV Cache

Conceptually, to process the s-th token at a given layer, the attention module first obtains its three vectors: query q_s, key k_s and value v_s, i.e., the inputs multiplying the respective parameter matrices, where $q_s, k_s, v_s \in \mathbb{R}^{1\times d}$ and d is the hidden size. Then it calculates the causal attention $a_s \in \mathbb{R}^{1\times d}$,

$$a_s = \mathrm{Attn}(q_s, k_{[1:s]}, v_{[1:s]}) = \mathrm{softmax}\left(\frac{q_s k_{[1:s]}^\top}{\sqrt{d}}\right) v_{[1:s]}$$

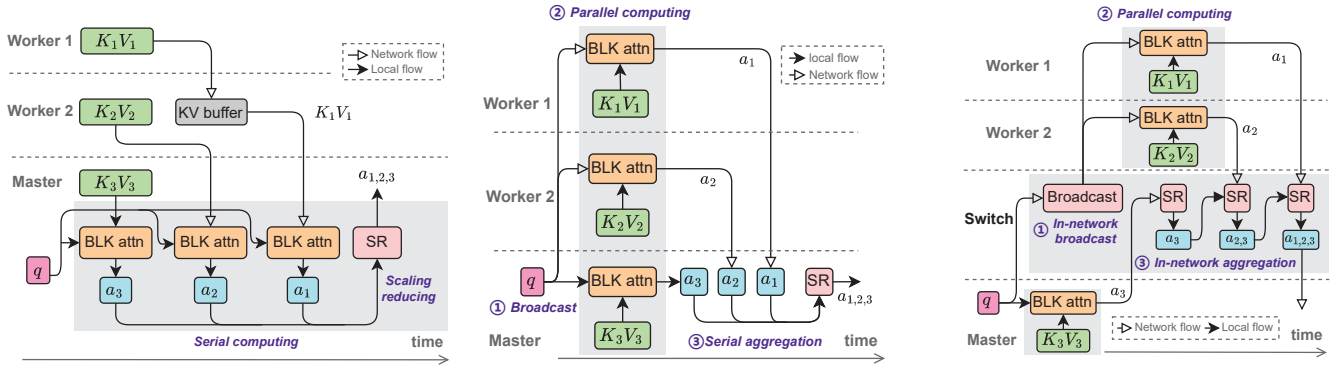
where $k_{[1:s]} = [k_1; k_2; \ldots; k_s]$ and $v_{[1:s]} = [v_1; v_2; \ldots; v_s]$.

In terms of attention processing, prefill and decoding phases face different bottlenecks. Prefill phase is *compute-intensive*, as the attentions for the whole input prompt must be computed before generating the first token. In decoding phase, for a sequence of length s, we only need to compute the attention a_s for the last generated token, which requires the current KV (k_s, v_s) and the preceding KVs $(k_{[1:s-1]}, v_{[1:s-1]})$. To avoid redundant computation of the preceding KVs, they are temporarily stored until the request ends, making the decoding phase *memory-intensive*. As a result, prefill-decoding (PD) disaggregation has been widely adopted [28, 64, 100], which separates prefill and decoding phases across different servers/GPUs and provides a wider optimization space for each phase, e.g., Kimi [70] and DeepSeek-V3 [45]. This paper focuses on optimizing the decoding performance to support ultra-long contexts.

2.2 SP for Long Contexts

As the context length increases for up-to-date models, the KV cache expands dramatically and tends to initiate the OOM issue. One approach is to migrate training techniques such as TP and PP to save the memory occupied by model parameters and leave more space for KV cache, but they are not scalable for extremely long or even infinite contexts[3], where the KV cache can easily exceed the memory capacity of a single GPU. Another promising parallelism is the emerging SP, which splits the KV cache into blocks along the sequence/context length dimension and distributes the KV blocks among GPUs. In this way, the OOM issue is always avoidable by involving more GPUs (i.e., scaling memory) to accommodate the huge KV cache. Let s and c denote the sequence length and block size, respectively, so there are $n = \lceil \frac{s}{c} \rceil$ KV blocks, denoted by $\mathbf{K}_1, \mathbf{K}_2, \ldots, \mathbf{K}_n$ and $\mathbf{V}_1, \mathbf{V}_2, \ldots, \mathbf{V}_n$, where $\mathbf{K}_i = k_{[(i-1)c+1:\min(ic,s)]}$ and $\mathbf{V}_i = v_{[(i-1)c+1:\min(ic,s)]}$.

[3]Length extrapolation (e.g., YaRN [65]) enables unlimited context during inference, regardless of training length.



(a) Pass-KV (ring) (b) Pass-Q (p2p) (c) Turbo (ours)

Figure 2: Decoding process of Pass-KV (ring), Pass-Q (p2p) and Turbo, where the master and two workers are linked to the same switch.

In the following analysis, we define the GPU responsible for generating the current token (handling attention aggregation and FFN) as the *master* node, while GPUs that store distributed KV blocks and compute blockwise attention are *worker* nodes[4]. Note that a master node typically also functions as a worker by holding a partition of the KV cache. The attention computation in SP is also adjusted based on the *blockwise* KVs and attentions. First, For the s-th token, the attention value $a_{s,i}$ of its i-th block is $\text{Attn}(q_s, \mathbf{K}_i, \mathbf{V}_i)$. Then, the master needs to aggregate all the n blockwise attentions by a *scaling-reduce* operation $\text{SR}(\{a_{s,i}\}_{i=1}^{n})$ to recover a complete attention a_s for the s-th token. For simplicity, we omit the subscript s of $a_{s,i}$ and q_s in the rest of this paper. The detailed formal derivation will be expanded in §4.1. Then, the key problem is how to implement the SP mechanism and accomplish the computation paradigm efficiently. Due to the heterogeneity of the prefill and decoding phases, there are mainly two classes of SP strategies, *Pass-KV* and *Pass-Q*, suitable for the two phases, respectively [93].

Pass-KV. The intuitive SP approach is to *pass KV blocks across GPUs*. All worker nodes deliver their respective KV blocks to the master nodes, which are responsible for computing and aggregating the blockwise attentions. Based on the KV transmission manner, Pass-KV can be further categorized into two types: Pass-KV (p2p) and Pass-KV (ring). Each worker in Pass-KV (p2p) simultaneously sends its KV block to the master via peer-to-peer (p2p) communication, resulting in incast congestion at the receiver. Additionally, the master must allocate huge memory to store the received KV blocks, leaving the OOM problem unresolved. Pass-KV (ring) mitigates the traffic burst by orchestrating workers to deliver their KV blocks in a sequential, ring-structured manner. As shown in Figure 1(a), all KV blocks form a ring, and each round of transmission shifts them by an offset of one node. Thus, each block passes through every worker exactly once, and each worker requires only one additional buffer as a relay.

Pass-Q. As shown in Figure 2(b), the Pass-Q strategy has four steps: (*i*) Broadcast: The master broadcasts Q to all workers; (*ii*) Computation: All workers compute the blockwise attention in parallel; (*iii*) Gather: Each worker sends its attention to the master; (*iv*) Aggregation: The master aggregates partial attentions. It can also be categorized into two types based on the Q transmission manner: Pass-Q (p2p) [82] and Pass-Q (ring) [90, 93]. The master in Pass-Q (p2p) interacts with each worker directly to send the Q and receive the partial attention, and all workers compute in parallel, achieving short delay. Each request in Pass-Q (ring) is assigned to a distinct master node. Specifically, each node conducts an all-gather in the ring manner to collect all queries, computes the corresponding blockwise attentions for each query in parallel, and performs an all-to-all operate to redistribute the blockwise attentions back to their respective masters.

Limitations. Tabel 1 theoretically compares the four aforementioned SP schemes for the decoding phase. Pass-KV approaches (both p2p and ring) impose fundamental limitations: they suffer from massive network traffic and long serial computation delays, making them prohibitive for latency-sensitive decoding tasks. Conversely, Pass-Q minimizes data movement by transmitting lightweight queries, making it the preferred strategy for long-context decoding [47]. However, current implementations face critical bottlenecks as the cluster scales: (*i*) *SM overhead*: Communication and aggregation consume significant GPU Streaming Multiprocessors (SMs). This overhead grows with the number of workers due to increased connection management and computation, e.g., DeepSeek-V3 [45] reserves 20 of 132 SMs solely for communication. (*ii*) *Memory overhead*: The aggregation buffer at the master scales linearly with the number of workers. (*iii*) *Redundant traffic*: Unicasting identical queries to each worker wastes outbound bandwidth. (*iv*) *Incast congestion*: The synchronized arrival of partial attentions from all workers triggers severe incast at the master. As for Pass-Q (ring), it is only applicable for multiple concurrent inference requests with similar context lengths to avoid pipeline bubbles. Besides, its inference delay is linear to the number of workers due to the ring-like serial transmission and computation. Moreover, it introduces an all-to-all communication for attention redistribution, which complicates the flow scheduling and requires sophisticated algorithms to mitigate collision [36, 69].

2.3 Motivation for In-Network Processing

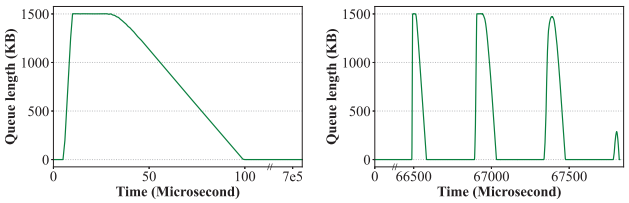
To overcome these inefficiencies (Table 1), we propose leveraging INC to offload communication and aggregation tasks, yielding two critical benefits:

Minimizing Network Traffic. Alternative hardware accelerators

[4]We omit another SP class, namely Ulysses [32], which coverts parallels on the sequence length to the head dimension by an all-to-all operation, due to its limited scalability, with details in Appendix A.

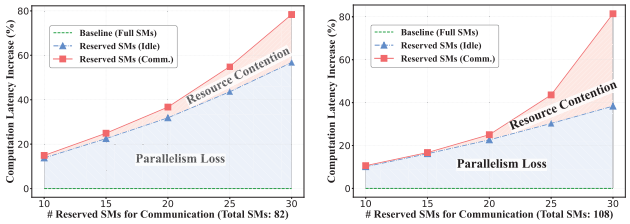
Strategy	Approach	Network traffic	Delay	Computation	Buffer
Pass-KV	Pass-KV (p2p)	$O(sd) = O(ncd)$	$T_{\text{net}} + n \times T_{\text{block_attn}} + T_{\text{reduce}}$	$n \times C_{\text{block_attn}} + C_{\text{reduce}}$	$O(sd)$
	Pass-KV (ring) [48]	$O(sd) = O(ncd)$	$n \times \max\{T_{\text{net}}, T_{\text{block_attn}}\} + T_{\text{reduce}}$	$n \times C_{\text{block_attn}} + C_{\text{reduce}}$	$O(cd)$
Pass-Q	Pass-Q (p2p) [82]	$O(nd)$	$2T_{\text{net}} + T_{\text{block_attn}} + T_{\text{reduce}}$	$C_{\text{block_attn}} + C_{\text{reduce}}$	$O(nd)$
	Pass-Q (ring) [90, 93]	$O(nd)$	$n \times \max\{T_{\text{net}}, T_{\text{block_attn}}\} + T_{\text{reduce}}$	$C_{\text{block_attn}} + C_{\text{reduce}}$	$O(d)$
	Turbo	$O(d)$	$2T_{\text{net}} + T_{\text{block_attn}}$	$C_{\text{block_attn}}$	$O(d)$

Table 1: Theoretical comparison of data loop strategies for the master node during the decoding phase. Here, s is the sequence length, c is the KV block size, n is the number of workers (KV blocks), and d is the hidden size. Turbo achieves optimal scalability by minimizing network traffic, inference delay, computation burden, and memory overhead.



(a) Pass-KV (p2p) (b) Pass-Q (p2p)

Figure 3: Queue depth of Pass-KV (p2p) and Pass-Q (p2p) on the ToR topology for the GPT-3 model with 32 concurrent tasks.



(a) RTX3090 with PCIe (b) A100 with NVLink

Figure 4: Computation latency degradation due to SM occupation and resource contention from communication.

such as Processing-in-Memory (PIM) can accelerate memory-bound operations local to a GPU [23, 35, 101]. However, they do not change the cross-node communication pattern in distributed long-context decoding: workers' partial attentions still need to traverse the network and converge at the master for final aggregation. Therefore, the master-side fan-in traffic, linear aggregation buffer, and incast congestion remain. In contrast, INC transforms the communication pattern. First, *In-Network Broadcast* eliminates redundant query transmission by allowing the master to send a single query, which the switch replicates and forwards to all workers. Second, *In-Network Aggregation* consolidates partial attentions at the switch, delivering only the final result to the master. This reduces incoming traffic, eliminates linear memory overhead, and crucially, mitigates incast congestion. As shown in Figure 3, under 32 concurrent GPT-3 tasks in NS-3 simulation, both Pass-KV (p2p) and Pass-Q (p2p) suffer severe queue bursts (>1000 packets) for over 200 μs even with DCQCN. INC effectively suppresses these bursts, extending tail latency performance by avoiding the sustained congestion that plagues Pass-Q (p2p) for over 10% of the transmission duration.

Releasing GPU Resources. INC liberates scarce GPU resources by offloading communication management and aggregation computation. This benefit is twofold: (*i*) *Maximizing Compute Parallelism*: Reducing SMs reserved for communication directly boosts inference performance. As shown in Figure 4(a), reserving 15/30 SMs for PCIe communication on an RTX3090 increases GEMM latency by 24.9%/78.4%. By offloading these tasks, INC returns these SMs to the inference pool. (*ii*) *Reducing Resource Contention*: Minimizing network I/O alleviates contention for shared resources like HBM bandwidth. On an A100 GPU, using 78 of 108 SMs for compute, performance degradation drops from 81.4% (with NVLink communication) to 38.3% (without communication), highlighting that traffic reduction releases significant non-SM resources. With the slow iteration of GPU and the rapid expansion of model demands, multi-GPU parallelism is inevitable for long contexts, yet it introduces heavy communication overheads. Therefore, leveraging INC to reduce traffic and release GPU resources is a critical path forward. Notable hardware advancements, such as the **NVLink6 Switch** [88], already support offloading complex operations, validating the feasibility of this direction.

3 High-level Architecture

In this section, we first propose Turbo, an in-network-based Pass-Q approach addressing the existing issues (§3.1), and then propose its key challenges (§3.2).

3.1 Working Flow

As shown in Figure 2(c), Turbo offloads the communication-intensive query broadcast and the computation-intensive attention aggregation from master to switches.

Phase 1: In-network broadcast. Instead of unicasting queries to individual workers, the master node generates a single copy of the query packet and transmits it to its ToR switch. Turbo maintains a logical *multicast tree* in the data plane. Upon receiving the query packet, each switch in the tree performs hardware multicast to its downstream ports. Isomorphically mapping the physical topology, this phase ensures that the query reaches all workers with $O(1)$ transmission overhead at the master side, effectively maintaining constant latency regardless of the cluster scale.

Phase 2: Parallel computation. Upon receiving the query, each worker i computes its blockwise attention a_i using its local KV block. The worker encapsulates the partial result into custom Turbo packets and transmits them to the upstream aggregation switch immediately.

Phase 3: In-network aggregation. The switch aggregates the received blockwise attention into its local states and updates a local bitmap to track the completion status. Once all expected partial attentions are aggregated, the switch sends the aggregated result to its parent node in the *aggregation tree*, which is congruent with the multicast tree but with reversed flow direction. As exemplified in Figure 2(c), the switch aggregates the blockwise attentions a_1, a_2, and a_3 from two workers and the master to obtain the final result $a_{1,2,3}$, and finally sends it to the master node.

Note that the aggregation tree and broadcast tree are topologically identical, defining the boundary of INC, e.g., the participating switches and their parent-child configurations. This structural duality allows Turbo to leverage standard multicast protocols like BIER [89] for efficient management.

3.2 Key Challenges

While the proposed workflow theoretically mitigates the computation and communication bottlenecks at the master node, implementing it on RMT-based switches, such as Broadcom Trident [10] and Intel Tofino [31], presents three fundamental hardware-algorithmic challenges.

Challenge 1: Misalignment between Scaling-Reduce Semantics and Switch Primitives. Unlike the reduce operations (e.g., summation and direct averaging) successfully deployed on switches for gradient aggregation in ATP [39], SwitchML [79], and SHARP [25], attention aggregation in SP decoding relies on *scaling-reduce*—a weighted averaging process involving complex floating-point exponentiation and division in Eq. (1), which are natively unsupported by the integer-only Arithmetic and Logic Units(ALUs) of RMT architectures. Moreover, the standard offline scaling-reduce necessitates buffering all partial attentions, which violates the scarce on-chip memory (SRAM) constraint of commodity switches. Thus, it is imperative to redesign the scaling-reduce logic into a memory-efficient online pairwise-aggregation process that operates within the constraints of switch primitives.

Challenge 2: Architectural Restrictions on Cross-Stage State Updates. The online pairwise aggregation necessitates a sequential *Read-Compute-Update* cycle for the global attention state upon each partial attention arrival. In RMT architectures, stateful objects (e.g., registers) are strictly bound to specific Match-Action Units (MAUs). Due to the strict data dependencies, the logical steps—reading the accumulated attention, computing the update, and writing the result—must be physically distributed across consecutive pipeline stages. This layout conflicts with the hardware's **strict stage locality**: a subsequent stage cannot retroactively access or update the stateful memory of a preceding stage. While packet recirculation permits cross-stage state updates via multi-pass processing, it inherently compromises switch throughput. Consequently, a novel state management mechanism is required to orchestrate these updates efficiently.

Challenge 3: Aggregation Stragglers caused by Workload Imbalance. Mapping the logical aggregation tree rigidly to a physical network topology induces significant load imbalance. Switches at topological confluences or higher tiers process disproportionate shares of aggregation workloads. These overloaded nodes become *stragglers*, inflating the tail latency which dictates the performance of SP decoding. Meanwhile, each switch operates under strict resource constraints (e.g., memory capacity for aggregation groups). Consequently, a global load balancing strategy is required to dynamically offload and distribute aggregation tasks across the network fabric, ensuring minimized aggregation latency while respecting the resource limits of switches.

Finally, reliability is a prerequisite for in-network computing. We adopt the mature fault-tolerance protocols from SHARP [25], SwitchML [79] and NetReduce [49] to handle packet loss and switch failures. As these mechanisms are orthogonal to our core aggregation logic, we detail the integration in Appendix C to maintain focus on our primary contributions.

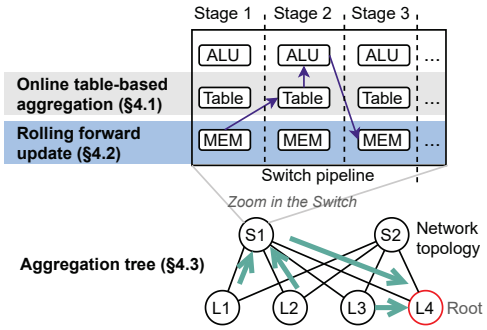


Figure 5: Turbo's framework.

4 Turbo Design

Figure 5 shows the framework of Turbo's design, consisting of (*i*) an online table-based aggregation mechanism for resource-limited switches (§4.1), (*ii*) a rolling forward update to manage cross-stage update during aggregation (§4.2), and (*iii*) a load-aware aggregation tree to balance the workload among switches (§4.3).

4.1 Table-based Online Aggregation

The i-th worker node computes its blockwise attention as follows, i.e., $a_i = \text{Attn}(q, \mathbf{K}_i, \mathbf{V}_i)$. Then, using the scaling-reduce operation to aggregate all blockwise attentions is actually to perform a weighted average. However, such `sum-exp` values may result in the overflow problem. Motivated by Flash-decoding [15], we take the following `log-sum-exp` in the scaling-reduce operation:

$$a = \text{ScalingReduce}\left(\{a_i\}_{i=1}^n\right)$$
$$= \sum_{i=1}^{n}\left(\frac{e^{\frac{q\mathbf{K}_i^\top}{\sqrt{d}}}}{\sum_{j=1}^{n}\left(\sum e^{\frac{q\mathbf{K}_j^\top}{\sqrt{d}}}\right)}\mathbf{V}_i\right) = \sum_{i=1}^{n}\left(\frac{\sum e^{\frac{q\mathbf{K}_i^\top}{\sqrt{d}}}}{\sum_{j=1}^{n}\left(\sum e^{\frac{q\mathbf{K}_j^\top}{\sqrt{d}}}\right)}a_i\right) = \sum_{i=1}^{n}\left(\frac{2^{\gamma_i}}{2^{\gamma}}a_i\right) \quad (1)$$

where γ and γ_i denote the `log-sum-exp` value, with $\gamma_i = \log(\sum e^{\frac{q\mathbf{K}_i^\top}{\sqrt{d}}})$ and $\gamma = \log(\sum_{i=1}^{n} 2^{\gamma_i})$.

Eq. (1) inherently dictates an *offline* batch-processing paradigm, as the normalization factor depends on the global summation of all partial results. This dependency mandates buffering all incoming partial attention blocks prior to aggregation—a requirement feasible for HBM-equipped GPUs but prohibitive for switches constrained by scarce on-chip SRAM. To address this resource bottleneck, we transform the global scaling-reduce into an *online aggregation* mechanism. This approach enables incremental, pairwise updates upon each packet arrival, achieving $O(1)$ memory complexity and sustaining line-rate processing.

Online aggregation. Turbo transforms Eq. (1) into an online fashion: the switch maintains local partially aggregated attention $\bar{a}$ and a scalar $\bar{\gamma}$, and updates them once receiving a new attention packet. As shown in Figure 2(c), $\bar{a}$ is initialized to the first received block attention a_3. Then, after sequentially aggregating a_2 and a_1, $\bar{a}$ becomes $a_{2,3}$ and $a_{1,2,3}$, respectively.

Mathematically, when receiving a new packet which carries i-th block's attention a_i and γ_i, the switch updates the states $\bar{a}$ and $\bar{\gamma}$ as follows:

$$\begin{cases} \bar{a} = \bar{a}\dfrac{2^{\bar{\gamma}}}{2^{\bar{\gamma}}+2^{\gamma_i}} + a_i\dfrac{2^{\gamma_i}}{2^{\bar{\gamma}}+2^{\gamma_i}}, \\ \bar{\gamma} = \log\left(2^{\bar{\gamma}}+2^{\gamma_i}\right). \end{cases} \tag{2}$$

To show the correctness of Eq. (2), we prove that the online scaling-reduce operation is equivalent to the offline aggregation in Eq. (1), regardless of the aggregation order.

Proof. Assume a sequence is arbitrarily split into n disjoint blocks $\{S_1, S_2, \cdots, S_n\}$. For any non-empty block set $S \subseteq \{S_1, \cdots, S_n\}$, let $\mathbf{K}_S$ and $\mathbf{V}_S$ denote the concatenation of the corresponding KV blocks. Given a query q, the attention over the block set S is

$$a_S = \mathrm{Attn}(q, \mathbf{K}_S, \mathbf{V}_S) = \frac{1}{\sum e^{\frac{q\mathbf{K}_S^T}{\sqrt{d}}}} e^{\frac{q\mathbf{K}_S^T}{\sqrt{d}}} \mathbf{V}_S \tag{3}$$

For any two disjoint block sets S_1 and S_2, define the binary aggregation operator $\circ$ as

$$a_{S_1} \circ a_{S_2} = \frac{2^{\gamma_{S_1}}}{2^{\gamma_{S_1}}+2^{\gamma_{S_2}}} a_{S_1} + \frac{2^{\gamma_{S_2}}}{2^{\gamma_{S_1}}+2^{\gamma_{S_2}}} a_{S_2} \tag{4}$$

This is exactly the pairwise update in Eq. (2). We next show that it produces the attention over the union of the two block sets:

$$\begin{aligned} a_{S_1} \circ a_{S_2} &= \frac{1}{\sum e^{\frac{q\mathbf{K}_{S_1}^T}{\sqrt{d}}} + \sum e^{\frac{q\mathbf{K}_{S_2}^T}{\sqrt{d}}}} \left(e^{\frac{q\mathbf{K}_{S_1}^T}{\sqrt{d}}} \mathbf{V}_{S_1} + e^{\frac{q\mathbf{K}_{S_2}^T}{\sqrt{d}}} \mathbf{V}_{S_2} \right) \\ &= \frac{1}{\sum e^{\frac{q\left[\mathbf{K}_{S_1}^T, \mathbf{K}_{S_2}^T\right]}{\sqrt{d}}}} \left(e^{\frac{q\left[\mathbf{K}_{S_1}^T, \mathbf{K}_{S_2}^T\right]}{\sqrt{d}}} \begin{bmatrix} \mathbf{V}_{S_1} \\ \mathbf{V}_{S_2} \end{bmatrix} \right) = a_{S_1 \cup S_2} \end{aligned} \tag{5}$$

Therefore, each pairwise online aggregation over two disjoint block sets is equivalent to the offline attention over their union.

By applying the above identity inductively, aggregating singleton block attentions in any binary tree gives

$$a_{S_1} \circ a_{S_2} \circ \cdots \circ a_{S_n} = a_{\{S_1, S_2, \cdots, S_n\}} = \mathrm{ScalingReduce}\left(\{a_{S_i}\}_{i=1}^n\right) \tag{6}$$

The operator $\circ$ is commutative because Eq. (4) is symmetric in S_1 and S_2. It is associative for disjoint block sets because both $(a_{S_1} \circ a_{S_2}) \circ a_{S_3}$ and $a_{S_1} \circ (a_{S_2} \circ a_{S_3})$ are reduced to $a_{S_1 \cup S_2 \cup S_3}$ by the identity above. Thus, the online scaling-reduce operation is equivalent to the offline scaling-reduce operation for any aggregation order. □

Specialized lookup tables. Commodity switches (e.g., Intel Tofino) fundamentally lack support for Floating-Point (FP) number, which is the native data type for LLM. Prior INC systems such as SwitchML bridge this gap by scaling FP into 32-bit integers (INT32). However, in the context of LLM inference, low-bit quantization (e.g., INT8 or INT4) has emerged as a standard optimization to reduce memory bandwidth and latency [18, 20]. Leveraging this trend, Turbo adopts the INT8 data representation. This choice not only aligns with the quantization paradigms of modern LLM serving engines but also significantly reduces the storage and bandwidth overhead on switch data planes.

Algorithm 1: Online scaling-reduce operation

Input: i-th block's attention a_i and γ_i in packet, local state $\bar{a}$ and $\bar{\gamma}$.

```
/* Step1: Prepare                                          */
Initialize Δγ ← |γ̄ − γ_i|, γ_max = γ̄, f ← True
if γ̄ < γ_i then γ_max ← γ_i, f ← False
if Δγ > 8 then γ̄ ← γ_max, ā ← f ? ā : a_i return
/* Step2: update scalar γ̄                                  */
γ̄ ← γ_max + g(Δγ)
/* Step3: update attention ā                               */
if f then
    for k ∈ 0 → len(ā) do
        ā[k] ← w1(Δγ, a_i[k]) + w2(Δγ, ā[k])
else for k ∈ 0 → len(ā) do
    ā[k] ← w1(Δγ, ā[k]) + w2(Δγ, a_i[k])
```

$$\begin{cases} \bar{a} = \begin{cases} a_i\dfrac{1}{1+2^{\Delta\gamma}} + \bar{a}\left(1-\dfrac{1}{1+2^{\Delta\gamma}}\right) & if\ \ \gamma_i \le \bar{\gamma} \\ \bar{a}\dfrac{1}{1+2^{\Delta\gamma}} + a_i\left(1-\dfrac{1}{1+2^{\Delta\gamma}}\right) & else \end{cases} \\ \bar{\gamma} = \gamma_{\max} + \log(1+2^{-\Delta\gamma}) \end{cases} \tag{7}$$

While simple integer arithmetic is native to switches, the complex non-linear operations (logarithm, exponentiation) required by Eq. (2) are absent. Although prior works [14, 96] approximate such functions via lookup tables, their generic designs incur prohibitive storage costs and pipeline latencies when applied directly. Furthermore, direct evaluation of the exponential term 2^{γ} in Eq. (2) poses the overflow risk. We thus derive a numerically stable reformulation, Eq. (7), to bypass large-value exponentiation. Here $\Delta\gamma = |\bar{\gamma} - \gamma_i|$ and $\gamma_{\max} = \max\{\bar{\gamma}, \gamma_i\}$. The table required to manage $\Delta\gamma$ is much smaller than that for original $\bar{\gamma}$ and γ_i.

The structured form in Eq. (7) motivates us to make an abstraction with three nonlinear functions: $\boldsymbol{g}(x) = \log(1+2^{-x})$, $\boldsymbol{w_1}(x,y) = \frac{1}{1+2^x}y$, and $\boldsymbol{w_2}(x,y) = (1-\frac{1}{1+2^x})y$. Consequently, the state update logic requires only a single lookup table for $g(\cdot)$ (to update $\bar{\gamma}$) and two tables for $w_1(\cdot)/w_2(\cdot)$ (to update $\bar{a}$). Crucially, these functions exhibit rapid convergence. When $\Delta\gamma > 8$, the weight term $\frac{1}{1+2^{\Delta\gamma}}$ diminishes below 0.004, rendering the contribution negligible for INT8 precision. This property allows us to truncate the lookup tables to a compact range of $\Delta\gamma$, significantly compressing the storage footprint without compromising aggregation accuracy. The detailed table lookup workflow is formalized in Algorithm 1.

4.2 Rolling Forward State Update

Standard RMT-based switches support atomic operations strictly following a `MEM_R`→`ALU_EXE`→`MEM_W` cycle within one stage. However, updating $\bar{\gamma}$ and $\bar{a}$ requires calculating $\Delta\gamma$ through lookup tables, extending the dependency chain to `MEM_R`→**`TBL_LKP`**→`ALU_EXE`→`MEM_W`. This insertion forces the write-back logic into subsequent stages, violating the single-stage RMW constraint identified in Challenge 2.

Workflow. To circumvent this limitation, we decompose the monolithic update logic into multiple pipeline stages (Figure 6). Calculating the global scale factor $\bar{\gamma}$ (Eq. (7)) requires three logical steps:

- *Fetch*: read $\bar{\gamma}$ from the register and compare it with γ_i to determine $\gamma_{\max}$ and $\Delta\gamma$;
- *Compute*: look up a table to retrieve $g(\Delta\gamma)$;

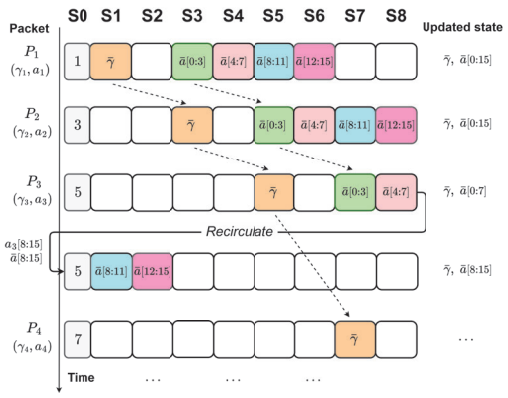


Figure 6: An example of the rolling forward state update.

- *Update*: write $\gamma_{\max}+g(\Delta\gamma)$ into the register as $\bar{\gamma}$.

Similarly, $\bar{a} = w_1(\Delta\gamma, a_i) + w_2(\Delta\gamma, \bar{a})$ can be realized through the following four-stage pipeline after $\Delta\gamma$ is ready.

- *Precompute*: look up a table to retrieve $w_1(\Delta\gamma, a_i)$;
- *Fetch*: read $\bar{a}$ from the local register;
- *Compute*: look up a table to retrieve $w_2(\Delta\gamma, \bar{a})$;
- *Update*: write $w_1(\Delta\gamma, a_i)+w_2(\Delta\gamma, \bar{a})$ into the register as $\bar{a}$.

The initially fetched values of $\bar{\gamma}$ and $\bar{a}$, along with intermediate results, are carried in the packet header vector (PHV) to bridge cross-stage dependencies; the stateful registers are updated only when the fully computed $\bar{\gamma}$ and $\bar{a}$ are ready.

Example. Figure 6 exemplifies the mechanism of the rolling forward state update in an RMT-based switch of nine stages S0∼S8, where $\bar{a}[0{:}15]$ is evenly partitioned into into four chunks and colored blocks indicate register writes. Stage S_0 tracks a dynamic offset of $\bar{\gamma}$ that increments by two per packet, cyclically shifting the resident stage of $\bar{\gamma}$ across S_1∼S_8 Note that pending updates at the pipeline exit (e.g., $\bar{a}[8{:}15]$ for P_3) are recirculated to S_0 for completion.

To maximize the supported attention size while maintaining line-rate forwarding, we implement two-dimensional pipelining. *Horizontally*, the computation for different attention chunks is distributed across sequential stages (e.g., the update of $\bar{a}$ for P_2 spans from S_5 to S_8). *Vertically*, each pipeline stage exploits parallelism to perform distinct operations on different attentions simultaneously. For instance, when P_2 traverses stage S_5, the switch concurrently precomputes $a_1[12{:}15]$, fetches $a_1[8{:}11]$, computes $a_1[4{:}7]$, and updates $\bar{a}_2[0{:}3]$ in S5.

Analysis. While rolling forward allows the number of aggregated attentions n_{attn} to grow theoretically without bound, it incurs periodic recirculation overhead. We quantify this behavior to optimize throughput.

Consider a switch with S stages each aggregating m_s attention elements, a packet carrying m elements requires $S_{\text{int}} = \lceil m/m_s \rceil + 2$ contiguous stages for aggregation. Since the update window shifts forward by two stages per packet (Figure 7), the active states progressively migrate from the initial S_{int} stages (blue) toward the end of the pipeline (green). Consequently, the first $(S - S_{\text{int}})/2 + 1$ packets complete processing within a single pass, constituting a *plateau region* with zero recirculation. Subsequent packets force the state window to wrap around, necessitating recirculation. However, the

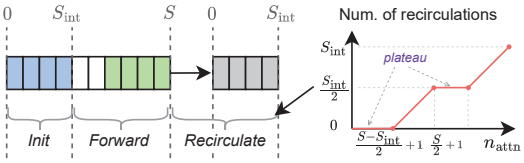


Figure 7: Relationship between the number of packet recirculations and aggregated attentions.

alignment is periodic: after $S/2+1$ packets, the state window resets to the initial position (grey), beginning a new cycle.

To minimize recirculation, n_{attn} should ideally fall within these plateau regions. We can extend the plateau by increasing S or m_s tailored to the switch specifications, thereby accommodating larger n_{attn} at zero overhead. For multi-pipeline switches, *Pipeline Folding* [63] can concatenate independent pipelines into a larger logical pipeline, effectively increasing S (e.g., up to S=80 on a four-pipeline Tofino2) and enlarging the recirculation-free plateau. Note that reducing m is suboptimal, as the increased packet count reduces effective bandwidth due to header overheads. Although recirculation introduces additional bandwidth consumption, this overhead can be compensated for by Turbo's master-side $O(n) \rightarrow O(1)$ traffic shrinking, as shown in Table 1.

4.3 Inter-Switch Load Balancing

While rolling forward enables unbounded online aggregation in theory, relying on a single switch encounters two scalability bottlenecks. First, workloads exceeding the recirculation-free plateau incur mandatory packet looping, which compromises effective throughput. Second, resources (e.g., SRAM and ALUs) on individual devices are inherently finite. In large-scale clusters, traffic skew often exacerbates these issues, forming hotspots where overloaded switches suffer from excessive recirculation and resource exhaustion, while others remain underutilized. To transcend these single-node limitations, we propose an inter-switch load balancing approach, which orchestrates aggregation tasks into a hierarchical tree, distributing workload to ensure efficient resource utilization and minimal recirculation across the network. The load-balancing logic is executed by Turbo's centralized control node (Appendix D), which maintains GPU and switch states, and installs the selected tree rules before decoding. For a request involving n workers, this requires $O(n)$ state maintenance and is off the per-token data path, so it does not affect line-rate packet processing.

Tree structure. Turbo constructs an aggregation tree where each switch aggregates partial attentions from child nodes and forwards the results to its parent. The topology defines three roles: leaf switches (connected to workers), intermediate switches, and a root switch (connected to the master). Unlike ATP and VAT [83] which rely on node-based bitmaps for state tracking, Turbo employs a lightweight *port-based bitmap*. By mapping aggregation states to physical ports rather than node IDs, this design minimizes processing complexity and metadata overhead. Furthermore, it exploits structural duality: the aggregation tree serves as the inverse path for query broadcasting, allowing rule sharing in MAU to conserve resources.

Problem formulation. To guarantee recirculation-free operation, we constrain the aggregation load on each switch u by its capac-

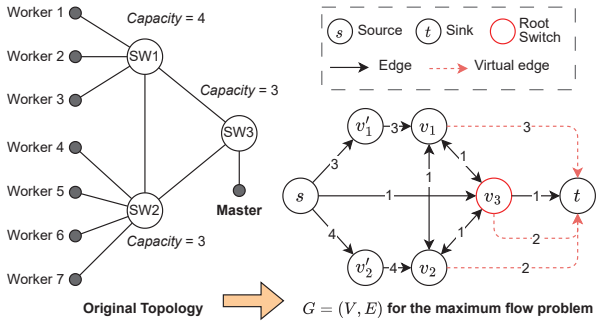


Figure 8: An example of converting a network topology to a maximum flow problem. The vertex s is an abstracted source of all flows, t is the sink which corresponds to the master node, v_1' and v_2' represents the the two worker set under SW1 and SW2, v_1, v_2, and v_3 corresponds to the switches. We add virtual edges for the switch vertices, and set the capacity of each edge properly.

ity $C(u)$, corresponding to the plateau limit derived in §4.2 (i.e., $C(u) = (S - S_{\text{int}})/2 + 1$). We model the network topology as a directed graph $G = (V, E)$, where V comprises the master, workers, and switches, and E represents physical links. For each edge $e \in E$, we define a flow variable $f(e)$ denoting the number of assigned attention streams, constrained by the link capacity $w(e)$. Given Turbo's port-based aggregation mechanism, distinct attention streams must arrive via separate physical ports to maintain state isolation; hence, $w(e)$ corresponds to the number of physical links available for the task.

Let $V_{\text{wk}}, V_{\text{sw}} \subset V$ denote the sets of worker and switch nodes, and let $in(u), out(u)$ be the incoming and outgoing edges for node u. Since each worker generates at most one attention stream, we restrict the outgoing flow for $u \in V_{\text{wk}}$ such that $\sum_{e \in out(u)} f(e) \leq 1$. For switches, the aggregation workload is determined by the number of concurrent attention streams processed. To strictly enforce the recirculation-free capacity $C(u)$, we formulate the optimal tree construction as an Integer Linear Programming (ILP) problem:

$$\begin{aligned} \max \quad & \sum_{\forall u \in V_{\text{wk}}} \sum_{e \in out(u)} f(e) \\ \text{s.t.} \quad & f(e) \leq w(e), \ \forall e \in E \\ & \sum_{e \in out(u)} f(e) \leq 1, \ \forall u \in V_{\text{wk}} \\ & \sum_{e \in in(u)} f(e) - \sum_{e' \in out(u)} f(e') + 1 \leq C(u), \forall u \in V_{\text{sw}}. \end{aligned}$$

Algorithmic Resolution. While ILP is generally NP-hard, we observe that the specific structure of our aggregation problem facilitates a reduction to the classic Maximum Flow problem, enabling an efficient polynomial-time solution. We construct an augmented graph (Figure 8) by introducing a *virtual edge* from each switch u directly to the sink (Master). Physically, a switch aggregates k incoming streams into a single outgoing stream ($f_{out} = 1$). To model this within flow conservation constraints, we direct the $k - 1$ "aggregated" streams through the virtual edge, setting its capacity to $C(u) - 1$. This transformation enforces the switch aggregation capacity limit while balancing ingress and egress flows at each node. We then employ the Ford-Fulkerson algorithm [24] to compute the maximum flow. To ensure valid tree connectivity (i.e., at least one stream traverses the physical physical link to the parent), we enforce a prioritized augmenting path search strategy (detailed in Appendix B).

Fallback Strategies. In cases where network capacity cannot fully accommodate the attention workload or some workers become stragglers, Turbo activates hierarchical fallback mechanisms: (*i*) *Topological Reconfiguration*: Designating an alternative switch as the aggregation root; (*ii*) *Capacity Expansion*: Increasing switch capacity by relaxing the zero-recirculation constraint or reducing packet size; (*iii*) *Master Offloading*: When a switch-side timer detects missing partial attentions, the switch forwards the aggregated result together with its completion bitmap to the master; the master then aggregates late-arriving partial attentions separately, or, if a worker is delayed beyond a configured timeout, loads the corresponding KV block and computes the missing blockwise attention locally as a correctness-preserving fallback; (*iv*) *Request Deferral*: Temporarily postponing long-sequence queries until resources are released.

5 Implementation

We detail the hardware dataplane of Turbo on programmable switches and FPGAs (§5.1), and the corresponding protocol design (§5.2). The source code and artifact suite is available in our repository[5].

5.1 Hardware Dataplane

Switch. We implement Turbo on an Intel Tofino2 switch using 2,352 lines of P4 code. The primary implementation challenge lies in mapping the logical rolling forward pipeline onto the fixed hardware resources (20 physical stages). We allocate 2 stages for γ logic, leaving 18 stages for attention chunks. Due to stage-local ALU constraints, the maximum chunk size m_s is limited to 4 elements. This imposes a trade-off between per-packet payload size and the recirculation-free plateau length (as defined in §4.2). We compare two configurations: (*i*) *64-element segments*: consumes 64/4=16 stages, yielding a plateau of (18−16)/2+1=2 packets; (*ii*) *32-element segments*: consumes 8 stages, yielding a plateau of 6 packets. We adopt the former by default to maximize effective bandwidth by amortizing header overheads.

To efficiently implement the non-linear scaling functions (w_1, w_2) within RMT constraints, we utilize MAU-based lookup tables. Direct mapping of high-precision inputs would exhaust SRAM (2^{16} entries). Instead, we employ a bit-slicing quantization strategy, constructing the lookup key as $[\Delta\gamma \gg (8-m_1), a \gg (8-m_2)]$, where m_1 and m_2 denote selected significant bits. Empirical evaluation (§6.3) indicates that an 11-bit index ($m_1 = 5, m_2 = 6$) offers the optimal Pareto frontier, strictly constraining table size to 2KB while maintaining high aggregation fidelity.

FPGA. We prototype the table-based aggregation on a Xilinx Alveo U280 accelerator to extend Turbo to a run-to-completion (RTC) switch with an FPGA card. Still using the same lookup tables as in Tofino2 switches, we parallel 64 such tables to realize only a two-cycle delay for the attention size of 64.

Hardware Generality. While prototyped on Intel Tofino2, Turbo employs generic RMT primitives, allowing seamless migration to modern programmable switches like Broadcom Trident [10]. Note that deploying Turbo's aggregation logic does not reduce the

[5] https://github.com/xxds-hys/turbo

switch's forwarding throughput, because a compiled RMT pipeline processes packets at the fixed line rate once it satisfies per-stage resource constraints. Furthermore, our FPGA adaptation demonstrates the compatibility of the core aggregation logic with RTC architectures, indicating broad hardware adaptability.

5.2 Protocol Specification

We design a protocol to perform Turbo, including the following specifics on (*i*) packet format, (*ii*) switch workflow for query broadcast and attention aggregation, and (*iii*) endhost network stack with reliability.

Packet format. Turbo employs a SHARP [25]-like packet structure, encapsulated in `Ethernet-IPv4-UDP` header. The custom `ATTN` header consists of a 4-byte metadata tuple (`type`, `seq_id`, `layer_idx`, `seg_idx`) and a 64-byte payload. Request packets carry query segments (q), while response packets transport attention chunks (a) paired with scaling factors (γ). The metadata tuple uniquely identifies the KV cache segment to ensure precise aggregation.

Aggregation Bitmap. Unlike ATP which relies on packet-carried state, Turbo maintains a switch-resident bitmap B to track aggregation completeness. Initialized to zero, B accumulates bits upon packet arrivals; validity is checked against a pre-configured mask B' (e.g., `0b1100` for aggregating the first two ports). Upon completion ($B = B'$), the switch triggers the Packet Replication Engine (PRE) to construct and forward the aggregated result upstream.

Reliable Transport. To ensure robustness over connectionless UDP, the host stack implements a sliding window protocol with size W. Workers stream attention packets within the window, awaiting acknowledgments from the master. Upon receiving an aggregated result, the master immediately replies with an independent ACK for that sequence number. This selective acknowledgment mechanism pinpoints packet losses, enabling workers to perform targeted retransmission and advance the window efficiently. We detail the handling of corner cases such as loss recovery in Appendix C.

Software Integration. We integrate Turbo into the PyTorch [43] distributed ecosystem by developing a custom collective communication backend. Specifically, we extend the `all-reduce` primitive to support scaling-reduce semantics, allowing it to carry scaling factors alongside attention tensors. This backend is seamlessly connected to vLLM [38], enabling the inference engine to offload attention aggregation to the network transparently without modifying the model architecture.

6 Evaluation

We conduct extensive experiments to evaluate Turbo from three aspects: end-to-end LLM serving performance on a real-world testbed (§6.1), performance improvement on large-scale networks through simulation (§6.2), an ablation study on Turbo design (§6.3), and Turbo's system overheads (§6.4).

6.1 Testbed

Setup. We deploy Turbo on a testbed comprising four nodes (each with two NVIDIA RTX 3090 GPUs and a 100 Gbps NIC) connected via a Tofino2 programmable switch. We utilize the LLAMA-2 7B [84] model and the ShareGPT [80] dataset, which contains multi-turn, task-oriented human-machine QA conversations with input prompts

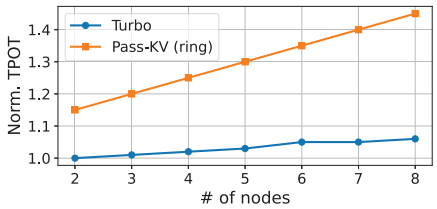


Figure 9: End-to-end TPOT on our testbed.

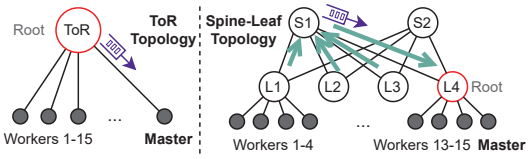


Figure 10: Two typical topologies.

ranging from 4 to 2.3K tokens. To fully exercise the distributed KV cache system under SP, we enforce a fine-grained KV block size of 256 tokens. This setup distributes KV blocks across multiple GPUs, effectively emulating the communication patterns of long-context workloads. We benchmark against Pass-KV (ring), the de facto deployment pattern in widely used LLM serving systems such as vLLM [38], SGLang [99], and Cache-DiT [17], and use *Time Per Output Token* (TPOT) to evaluate decoding latency.

Inference delay. Figure 9 shows the normalized TPOT for groups with different context lengths. As expected, the inference delay of Pass-KV (ring) increases linearly as the context length increases. In contrast, Turbo maintains a nearly constant inference delay by offloading computation to the switch and reducing communication overhead, and can reduce the inference delay by up to 37% compared to Pass-KV (ring) when the context spans 8 GPUs.

Cost-effectiveness discussion. Turbo does not eliminate the need for GPUs or HBM; instead, it reduces the communication and aggregation overhead on the GPU's critical path. Therefore, the relevant comparison is the incremental cost and power required to scale long-context serving. The price difference between a programmable switch and a commodity Ethernet switch with the same capacity is about $4000 [7, 59], only 1/10 of an 80GB-HBM GPU card [13]. The power footprint is also amortized across many GPUs: using the Tofino2 per-port power of 4.2 W, a 64-port switch consumes about $64 \times 4.2 = 268.8$ W for the datapath [2], which is below the 700 W TDP of a single H100 GPU. Since one switch can serve tens of GPUs, the incremental power per attached GPU is small compared with adding GPU/HBM capacity. Thus, Turbo improves long-context serving efficiency by using a shared network-side aggregation resource rather than provisioning additional GPU memory capacity solely to absorb KV-cache and aggregation overheads.

6.2 Large-scale Simulation

Setup. To evaluate Turbo at scale, we develop a packet-level simulation on NS-3 [27], modeling a cluster of 16 nodes (1 master, 15 workers) connected via 200 Gbps links with a 2 μs switch forwarding delay, running RoCEv2 with DCQCN as the congestion-control algorithm. We configure two topologies as shown in Figure 10: (*i*) *ToR*: all nodes connected to a single switch; and (*ii*) *Spine-Leaf*: nodes distributed under leaf switches connected to spines. The switch buffer is set to 32 MB.

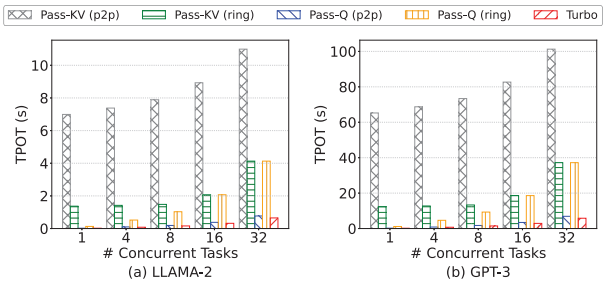


Figure 11: TPOT on the ToR topology.

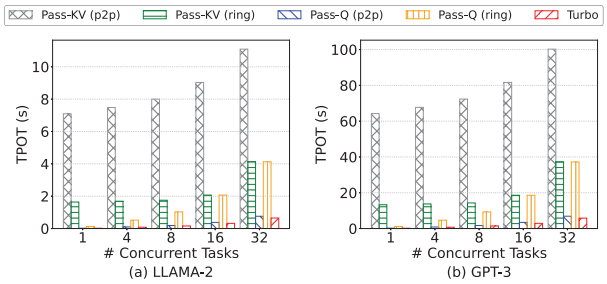


Figure 12: TPOT on the Spine-Leaf topology.

Model	LLAMA-2								GPT-3							
# Tasks	4				16				4				16			
Time(%)	T_{commu}	T_{attn}	T_{reduce}	T_{other}	T_{commu}	T_{attn}	T_{reduce}	T_{other}	T_{commu}	T_{attn}	T_{reduce}	T_{other}	T_{commu}	T_{attn}	T_{reduce}	T_{other}
Pass-KV(p2p)	93.1	6.06	0.17	0.68	77.1	20.1	0.57	2.24	93.1	6.02	0.17	0.67	77.2	19.1	0.56	2.23
Pass-KV(ring)	96.2	—	0.76	3.0	—	87.7	2.48	9.80	95.8	—	0.84	3.32	—	87.7	2.48	9.80
Pass-Q(p2p)	1.83	31.7	13.4	53.1	1.44	31.8	13.5	53.3	1.47	31.8	13.5	53.2	2.18	31.6	13.4	52.9
Pass-Q(ring)	—	26.1	70.9	2.93	—	26.2	70.9	2.92	—	26.2	70.9	2.92	—	26.1	70.9	2.93
Turbo	0.74	37.1	—	62.2	0.27	37.3	—	62.4	0.32	37.3	—	62.4	0.16	37.3	—	62.5

Table 2: Breakdown of TPOT on the Spine-Leaf topology ("—" indicates that the latency is masked by other operations).

Workloads and Baselines. We generate traffic patterns mirroring real-world LLM inference characteristics for LLAMA-2 7B (32 layers) and GPT-3 175B [11] (96 layers). Instead of simulating GPU instruction execution, we drive network events using trace-driven latencies profiled from our RTX 3090 testbed, decomposed into four components: (1) T_{attn}, the attention computation time for a single KV block (The full KV cache is partitioned into 16 blocks); (2) T_{reduce}, the time to aggregate a pair of blockwise attention results; (3) T_{layer}, the per-layer non-attention overhead, including pre-attention operations (QKV Projection, RoPE) and post-attention operations (LayerNorm, FFN, and Residuals); and (4) T_{fixed}, the layer-independent constant latency (Input Embedding and Output Projection). For LLAMA-2, we measure T_{attn}=227μs, T_{reduce}=75.8μs, T_{layer} =395μs (93μs pre-attention and 302μs post-attention overheads), and T_{fixed}=129μs. For simplicity, we unify T_{layer} and T_{fixed} as T_{other}. Each worker stores the KV cache corresponding to 4096 tokens and single packet carries 64 attention elements. We compare Turbo against four state-of-the-art baselines: Pass-KV (p2p), Pass-KV (ring) [48], Pass-Q (p2p), and Pass-Q (ring) [93].

Inference delay. Figures 11 and 12 illustrate the TPOT across varying topologies and models. Since the results on the two topologies exhibit a highly similar trend, we focus our analysis on the Spine-Leaf scenario. Turbo delivers substantially lower latency than all baselines. Taking four concurrent LLAMA-2 tasks on the Spine-Leaf topology as an instance, Turbo achieves a TPOT of 80 ms. This represents a speedup of 93.5×, 21.1×, 1.25×, and 6.5× over Pass-KV (p2p), Pass-KV (ring), Pass-Q (p2p), and Pass-Q (ring), respectively. The massive delay in Pass-KV arises from the transmission of sheer KV block volumes, which saturates network bandwidth and prevents computation overlapping. While Pass-Q reduces traffic volume by four orders of magnitude (transmitting only queries and partial attentions), it remains limited by point-to-point incast or ring latencies. By enforcing a strict one-packet-in, one-packet-out traffic pattern at the master, Turbo eliminates incast congestion and network bottlenecks.

Moreover, Turbo demonstrates exceptional scalability under heavy load. When scaling from 1 to 32 LLAMA-2 tasks, the latency for baselines skyrockets by up to 4 seconds. Conversely, Turbo remains robust, with TPOT increasing by 0.63 seconds, which attests to the effectiveness of in-network aggregation in isolating the master from traffic surges.

	Traffic size		Incast
	LLAMA-2	GPT-3	
Pass-KV (p2p)	30 GB	90 GB	✓
Pass-KV (ring)	30 GB	90 GB	✗
Pass-Q (p2p)	3.75 MB	11.25MB	✓
Pass-Q (ring)	56.25 MB	168.75MB	✗
Turbo	0.256 MB	0.768 MB	✗

Table 3: Master-side inbound traffic volume.

TPOT Breakdown. Table 2 details the latency composition on the Spine-Leaf topology. We observe a critical bottleneck shift: Pass-KV (p2p) is communication-bound (e.g., communication occupies 93.1% of TPOT for GPT-3 with four tasks), whereas Turbo is computation-bound with negligible communication cost (0.74%). This efficiency stems from the massive traffic reduction shown in Table 3: Turbo reduces traffic received by the master by 1∼5 orders of magnitude (e.g., only 256 KB for LLaMA-2) compared to baselines. This minimal footprint, combined with in-network aggregation, effectively eliminates the incast congestion inherent to p2p baselines shown in Figure 3, allowing GPUs to operate with maximum parallelism. Therefore, Table 3 shows that Turbo addresses the two bottlenecks identified in §2.3. Regarding Pass-Q, ring-based approaches suffer from sequential dependencies, and both variants incur significant reduction overhead (T_{reduce}) at the master. In contrast, Turbo offloads reduction to the switch, successfully masking T_{reduce} entirely within the transmission phase.

Conservative Estimation. The NS-3 simulator models transmission and queueing, but it cannot model GPU-side computation–communication resource contention, such as SM reservation for communication kernels and memory-system contention induced by data movement. As shown in Figure 4, this effect is non-negligible: reserving 30 SMs for communication on an RTX3090 increases GEMM latency by 78.4%. Therefore, the simulated speedup of Turbo over Pass-Q (p2p) should be interpreted conservatively. As analyzed in Table 1, Turbo reduces the master-side fan-out/fan-in traffic to $\frac{1}{n}$

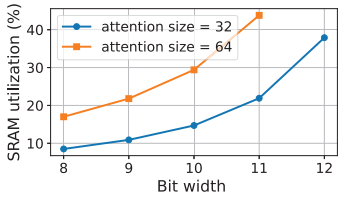


Figure 13: SRAM consumption on Tofino2.

	FP16	Turbo	INT8 Quantization
LLAMA-2	5.5654	5.6966	5.6302
GPT-2	19.137	19.3537	19.4281
Qwen-2.5	6.9435	7.462	7.8577

Table 4: Perplexity comparison across three LLMs.

Context Length	50K	100K	150K	200K	250K
KL divergence	0.007927	0.009656	0.008938	0.009187	0.008893

Table 5: KL divergence under ultra-long contexts.

of Pass-Q (p2p), where n is the number of workers. Moreover, compared with Pass-Q (p2p), Turbo removes GPU-side scaling-reduce operation from the critical path by offloading aggregation to the switch. In deployments where communication competes with inference kernels for GPU resources, this reduction in master-side communication and aggregation work can further increase the end-to-end performance gap beyond what is captured in NS-3.

6.3 Ablation Study

We conduct an ablation study to analyze the trade-off between resource consumption and accuracy.

Resource Consumption on Tofino Switch. Figure 13 plots the SRAM utilization on Tofino2 against the total table bit width ($m_1 + m_2$). Since the table size scales as $2^{m_1+m_2}$ (see §5.1), memory consumption exhibits exponential growth. Specifically, supporting 64-element attention aggregation with 11-bit tables consumes 43.8% of the total SRAM. However, increasing the width to 12 bits exceeds physical resource limits and fails to compile, establishing 11 bits as the maximum feasible precision on the current hardware platform.

End-to-end Generation Accuracy. We evaluate the generation accuracy of Turbo using KL divergence and Perplexity (PPL), benchmarking against the industry-standard INT8 quantization. Note that they apply solely to the variables involved in online aggregation, maintaining FP16 precision for all other components to preserve fidelity.

First, we measure the KL divergence between the output logits of the original FP16 model and the optimized methods to quantify information loss. Figures 14(a) to 14(c) depict the results for open-source models adopted as-is, including GPT-2 (Large) [75], LLaMA-2 7B, and Qwen-2.5 7B [71] across context lengths from 512 to 2048. Even at 2048 tokens, Turbo maintains extremely low divergence (e.g., 0.0058 for Qwen-2.5 and 0.0051 for LLAMA-2), orders of magnitude below the typical 0.05 threshold for perceptual loss. Furthermore, Turbo's accuracy loss is comparable to, and in cases like GPT-2 and Qwen-2.5, significantly lower than that of INT8 quantization. This is because Turbo populates lookup tables using high-precision FP16, avoiding the cumulative errors of full INT8 quantization. This confirms that Turbo achieves inference acceleration while preserving high model fidelity. Second, we assess language modeling quality using PPL on the WikiText-2 dataset [54]. As shown in Table 4, Turbo incurs only a marginal PPL increase compared to the FP16 baseline (e.g., +0.13 for LLAMA-2 and +0.22 for GPT-2). Crucially, this degradation is on par with INT8 quantization (e.g., -0.07 for GPT-2 and -0.40 for Qwen-2.5). Given that INT8 has been accepted in industrial LLM serving (e.g., OptiQuant [30] for DeepSeek), the similar PPL degradation of Turbo demonstrates its viability for production.

	LUT	FF	Block RAM
Turbo	40,768	67,072	128/1,024
All resources	1,303,680	2,607,360	2,016
Ratio (%)	3.13	2.57	6.3/50.29

Table 6: Resource consumption of Turbo on FPGA U280.

In fact, the approximation error introduced by Turbo is governed by the number of online aggregation operations and the LUT resolution, rather than by the context length or model size directly. Under a fixed aggregation topology, longer context increases the KV data stored at workers but not the number of partial attentions aggregated per output token. We valid this on Qwen-3.5 9B [72] with the LongBench [5] benchmark across context lengths from 50K to 250K on 8 nodes. As shown in Table 5, the KL divergence remains below 0.01 for all tested lengths, including 0.008893 at 250K tokens, and does not exhibit monotonic growth with context length.

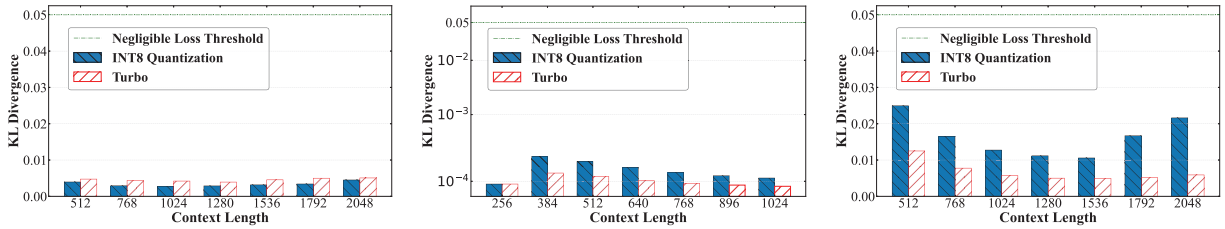
Sensitivity of LUT Configurations. We explore the impact of lookup table design—specifically total bit width and bit allocation—on model fidelity, as shown in Figure 15.

First, increasing the total bit width ($W = m_1 + m_2$) improves approximation granularity at the cost of exponential memory growth (2^W). For example, increasing m_2 (for a) from 6 to 7 while keeping m_1 (for $\Delta\gamma$) at 5 reduces LLAMA-2's KL divergence from 0.007418 to 0.003766 (1.97× improvement), confirming that allocating additional memory directly enhances accuracy. Second, under a fixed memory budget, optimizing bit allocation is crucial. With a total width of 11 bits, the configuration (m_1=5, m_2=6) achieves a significantly lower KL divergence (0.000444) compared to (m_1=6, m_2=5) (0.002403), reducing information loss by 5.4×. Accordingly, we adopt the (m_1=5, m_2=6) configuration for the Tofino2 implementation, which requires only 2048 entries per stage to maximize accuracy while minimizing SRAM consumption. In practice, operators can use this accuracy–SRAM trade-off to select the minimal LUT size that satisfies a target fidelity threshold.

6.4 System Overhead

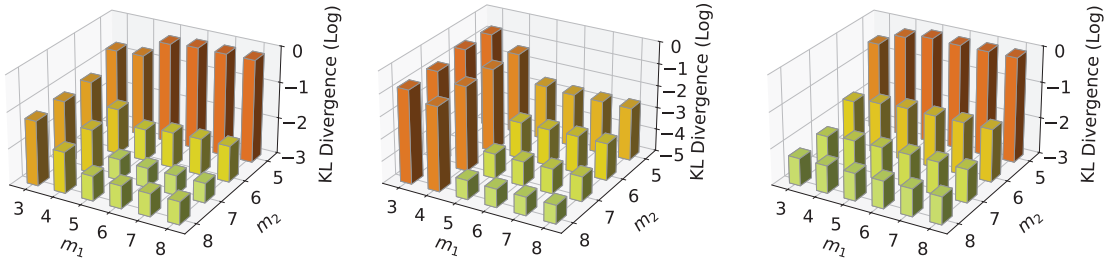
Turbo's setup overhead on switch control plane. We measure the setup overhead of instantiating Turbo on the EdgeCore Wedge 100BF-32X programmable switch [21]. When a new inference request requires installing fresh Exact-Match (EM) lookup table entries, the per-entry insertion latency is below 40 μs and remains constant regardless of the number of existing rules in the table. This low and size-independent insertion cost enables Turbo to complete all required table population during the prefill phase, overlapping setup with the initial prompt processing, thereby achieving minimal impact on the subsequent decoding phase.

Turbo's overhead on FPGA. Table 6 compares the resource consumption of tables using 11 and 14-bit widths, demonstrating that Turbo consumes minimal LUTs and FFs and low BRAM resources. It implies Turbo's practicality in production environment.



(a) LLAMA-2 (b) GPT-2 (c) Qwen-2.5

Figure 14: End-to-end inference distribution fidelity (KL Divergence) across different LLMs.



(a) LLAMA-2 (b) GPT-2 (c) Qwen-2.5

Figure 15: Impact of table configuration on end-to-end LLM inference accuracy.

7 Related Work

KV cache compression. Researchers have tried to compress KV cache from various aspects: (*i*) Low-bit quantization [19, 20, 53, 86, 87, 94], e.g., using FP8 can halve the KV cache size than FP16; (*ii*) KV sharing [3, 81], e.g., *h*-head attention sharing one or several KV vectors to compress the KV cache on the *d* dimension; (*iii*) Sparse attention, which maintains a small KV cache using a sliding window [6, 91, 92] or discarding low-scoring tokens [4, 51, 98], or keeps the full context but only selects the top-k most relevant tokens each time [46, 52, 95]; (*v*) Linear attention [26, 34, 55, 66, 71, 97], e.g., using a fixed-size memory to store the compressed historical KV information; (*vi*) reducing network bandwidth cost [50]; (*vii*) Low-rank transformation [45]. Note that sparse attention without a full context and linear attention are scalable to support an infinite context in theory, but sacrifice accuracy. Turbo is actually orthogonal to them, and can realize high accuracy and scalability through the elaborated in-network aggregation.

Distributed LLM inference. To address the large KV cache issue, a series of work focuses on splitting the large KV cache into multiple blocks and distributing them across GPUs. Prior work has employed TP [58, 77, 78] and PP [22, 29, 33, 56, 57], but are not scalable for the LLM inference with long or even infinite context. A recent approach is the emerging SP (a.k.a. CP) [32, 41, 42, 44, 47, 48, 90]. However, the traditional SP may suffer from high communication overhead and long inference delay. Tree Attention [82] exploits a hierarchical aggregation structure for long-context decoding, but its aggregation is executed on GPUs and therefore still consumes GPU resources. As discussed in this paper, Turbo addresses the above issues by leveraging the in-network computing paradigm. Besides, under the trend of PD Disaggregation [28, 45, 64, 70, 100], deploying Turbo in the decoding cluster is of highly practical utility.

In-netowrk aggregation for machine learning. Previous work focuses on implementing basic *all-reduce* operations (e.g., sum and max) on different hardware platforms: (*i*) ATP [39] and SwithML [79] use P4 switch for high throughput; (*ii*) Flare [16] uses Risc-V cores for flexibility; (*iii*) NetReduce [49] leverages FPGA to realize transport-transparency; (*iv*) SHARP [25] is designed for NVIDIA's switches and GPUs. In contrast, we are the first to use in-network aggregation to address the complex *scaling-reduce* operation for LLM inference, and our Turbo, including the online table-based aggregation algorithm and rolling forward state update approach, addresses key system-level challenges that have not been overcome before.

8 Conclusion

We present Turbo, a first-of-its-kind in-network aggregation system that accelerates long-context inference. Turbo employs (*i*) Online Table-based Aggregation to transform global reduce into pairwise aggregation via lookup tables, (*ii*) Rolling Forward to enable cross-stage state update, and (*iii*) Load-aware Aggregation Tree to balance workloads among switches. The Tofino2-based prototype reduces end-to-end latency by 37% compared to state-of-the-art approaches, while large-scale simulations demonstrate up to 1.25× latency reduction and five orders of magnitude traffic decrease with negligible accuracy loss. Turbo effectively transforms the network from a passive data conduit into an active accelerator for LLM serving.

This work **does not** raise any ethical issues.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd for their insightful comments and suggestions. Ying Wan was supported by the State Key Laboratory of Internet Architecture (HLW2025MS13, HLW2025ZD08) and Tencent Rhino-Bird Joint Research Program (JR2026TEG624). Wenfei Wu is supported by "The Fundamental Research Funds for the Central Universities, Peking University". Congcong Miao and Jialin Li are supported by the Singapore Ministry of Education, under Academic Research Fund Tier 3 grant MOE-MOET32024-0004. Corresponding Authors: Congcong Miao (mccmiao@163.com) and Wenfei Wu (wenfeiwu@pku.edu.cn).

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[2] Anurag Agrawal and Changhoon Kim. 2020. Intel tofino2–a 12.9 tbps p4-programmable ethernet switch. In *2020 IEEE Hot Chips 32 Symposium (HCS)*. IEEE, 1–32.

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

[4] Sotiris Anagnostidis, Dario Pavllo, Luca Biggio, Lorenzo Noci, Aurelien Lucchi, and Thomas Hofmann. 2023. Dynamic context pruning for efficient and interpretable autoregressive transformers. *Advances in Neural Information Processing Systems* 36 (2023), 65202–65223.

[5] Yushi Bai, Shangqing Tu, Jiajie Zhang, Hao Peng, Xiaozhi Wang, Xin Lv, Shulin Cao, Jiazheng Xu, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. 2025. LongBench v2: Towards Deeper Understanding and Reasoning on Realistic Long-context Multitasks. arXiv:2412.15204 [cs.CL] https://arxiv.org/abs/2412.15204

[6] Iz Beltagy, Matthew E Peters, and Arman Cohan. 2020. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150* (2020).

[7] bm switch. 2024. 32x400G Netberg Aurora 810 (Intel Tofino 2 IFP) – SONiC ready. https://bm-switch.com/product/32x400g-netberg-aurora-810-intel-tofino-2-ifp-sonic-ready.

[8] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Mark Horowitz, and Scott Yalagandula. 2013. RMT: A Reconfigurable Match Table Architecture for Software-Defined Networks. In *ACM SIGCOMM Computer Communication Review*, Vol. 43. ACM, 11–22.

[9] William Brandon, Aniruddha Nrusimha, Kevin Qian, Zachary Ankner, Tian Jin, Zhiye Song, and Jonathan Ragan-Kelley. 2023. Striped attention: Faster ring attention for causal transformers. *arXiv preprint arXiv:2311.09431* (2023).

[10] Broadcom. 2025. High-Capacity StrataXGS® Trident 5 Programmable Ethernet Switch Series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm78800.

[11] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[12] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2023. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research* 24, 240 (2023), 1–113.

[13] GMI Cloud. 2026. How Much Does the NVIDIA H100 GPU Cost in 2025? Buy vs. Rent Analysis. https://www.gmicloud.ai/blog/how-much-does-the-nvidia-h100-gpu-cost-in-2025-buy-vs-rent-analysis.

[14] Penglai Cui, Heng Pan, Zhenyu Li, Jiaoren Wu, Shengzhuo Zhang, Xingwu Yang, Hongtao Guan, and Gaogang Xie. 2021. NetFC: Enabling accurate floating-point arithmetic on programmable switches. In *2021 IEEE 29th International Conference on Network Protocols (ICNP)*. IEEE, IEEE, Dallas, TX, USA, 1–11.

[15] Tri Dao, Daniel Haziza, Francisco Massa, and Grigory Sizov. 2023. Flash-Decoding for long-context inference. https://pytorch.org/blog/flash-decoding/. PyTorch.

[16] Daniele De Sensi, Salvatore Di Girolamo, Saleh Ashkboos, Shigang Li, and Torsten Hoefler. 2021. Flare: Flexible in-network allreduce. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Association for Computing Machinery, New York, NY, USA, 1–16.

[17] etc. DefTruth, vipshop.com. 2025. Cache-DiT: A PyTorch-native Inference Engine with Cache, Parallelism and Quantization for Diffusion Transformers. Open-source software available at https://github.com/vipshop/cache-dit.git. https://github.com/vipshop/cache-dit.git

[18] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. 2022. Gpt3.int8 (): 8-bit matrix multiplication for transformers at scale. *Advances in Neural Information Processing Systems* 35 (2022), 30318–30332.

[19] Tim Dettmers, Ruslan Svirschevski, Vage Egiazarian, Denis Kuznedelev, Elias Frantar, Saleh Ashkboos, Alexander Borzunov, Torsten Hoefler, and Dan Alistarh. 2023. Spqr: A sparse-quantized representation for near-lossless llm weight compression. *arXiv preprint arXiv:2306.03078* (2023).

[20] Tim Dettmers and Luke Zettlemoyer. 2023. The case for 4-bit precision: k-bit inference scaling laws. In *International Conference on Machine Learning*. PMLR, PMLR, Honolulu, Hawaii, USA, 7750–7774.

[21] EdgeCore. 2024. Wedge 100BF-32X series switches. https://www.epsglobal.com/products/switches/data-center-switches/100g-data-center-switches/dcs800-32x-100g-qsfp28-data-center-switch/. https://www.edge-core.com/productsEOLInfo.php?cls=1&cls2=5&cls3=181&id=128

[22] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, Lansong Diao, Xiaoyong Liu, and Wei Lin. 2021. DAPPLE: A pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Association for Computing Machinery, New York, NY, USA, 431–445.

[23] Xiang Fei, Jianhui Han, Jianqiang Huang, Weimin Zheng, and Youhui Zhang. 2022. Accelerating Neural Network Training with Processing-in-Memory GPU. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 414–421. doi: 10.1109/CCGrid54584.2022.00051

[24] Lester Randolph Ford and Delbert R Fulkerson. 1956. Maximal flow through a network. *Canadian journal of Mathematics* 8 (1956), 399–404.

[25] Richard L. Graham, Devendar Bureddy, Pak Lui, Hal Rosenstock, Gilad Shainer, Gil Bloch, Dror Goldenerg, Mike Dubman, Sasha Kotchubievsky, Vladimir Koushnir, Lion Levi, Alex Margolin, Tamir Ronen, Alexander Shpiner, Oded Wertheim, and Eitan Zahavi. 2016. Scalable hierarchical aggregation protocol (SHArP): A hardware architecture for efficient data reduction. In *2016 First International Workshop on Communication Optimizations in HPC (COMHPC)*. IEEE, IEEE, Salt Lake City, UT, USA, 1–10.

[26] Albert Gu and Tri Dao. 2024. Mamba: Linear-time sequence modeling with selective state spaces. In *First conference on language modeling*.

[27] Thomas R Henderson, Mathieu Lacage, George F Riley, Craig Dowell, and Joseph Kopena. 2008. Network simulations with the ns-3 simulator. *SIGCOMM demonstration* 14, 14 (2008), 527.

[28] Cunchen Hu, Heyang Huang, Liangliang Xu, Xusheng Chen, Jiang Xu, Shuang Chen, Hao Feng, Chenxi Wang, Sa Wang, Yungang Bao, Ninghui Sun, and Yizhou Shan. 2024. Inference without interference: Disaggregate llm inference for mixed downstream workloads. *arXiv preprint arXiv:2401.11181* (2024).

[29] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and Zhifeng Chen. 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems* 32 (2019).

[30] Huawei Ascend. 2025. Ascend Inference cluster: OptiQuant. https://gitcode.com/ascend-tribe/ascend-inference-cluster/tree/main/OptiQuant. (2025).

[31] Intel. 2024. Intel® Tofino™. https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/tofino.html.

[32] Sam Ade Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Leon Song, Samyam Rajbhandari, and Yuxiong He. 2023. Deepspeed ulysses: System optimizations for enabling training of extreme long sequence transformer models. *arXiv preprint arXiv:2309.14509* (2023).

[33] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond Data and Model Parallelism for Deep Neural Networks. *Proceedings of Machine Learning and Systems* 1 (2019), 1–13.

[34] Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. 2020. Transformers are rnns: Fast autoregressive transformers with linear attention. In *International conference on machine learning*. PMLR, 5156–5165.

[35] W.H. Kautz. 1969. Cellular Logic-in-Memory Arrays. *IEEE Trans. Comput.* C-18, 8 (1969), 719–727. doi: 10.1109/T-C.1969.222754

[36] Kawthar Shafie Khorassani, Ching-Hsiang Chu, Quentin G Anthony, Hari Subramoni, and Dhabaleswar K Panda. 2021. Adaptive and hierarchical large message all-to-all communication algorithms for large-scale dense gpu systems. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, IEEE, Melbourne, Australia, 113–122.

[37] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*. Association for Computing Machinery, New York, NY, USA, 611–626.

[38] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*. Association for Computing Machinery, New York, NY, USA, 611–626.

[39] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. 2021. ATP: In-network aggregation for multi-tenant

learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, N/A, 741–761.

[40] Dacheng Li, Rulin Shao, Anze Xie, Eric P Xing, JE Gonzalez, Ion Stoica, X Ma, and Hao Zhang. 2023. DISTFLASHATTN: Distributed Memory-efficient Attention for Long-context LLMs Training. *arXiv preprint arxiv:2310.03294* (2023).

[41] Shenggui Li, Fuzhao Xue, Chaitanya Baranwal, Yongbin Li, and Yang You. 2021. Sequence parallelism: Long sequence training from system perspective. *arXiv preprint arXiv:2105.13120* N/A, N/A (2021), 2391–2404.

[42] Shenggui Li, Fuzhao Xue, Yongbin Li, and Yang You. 2021. Sequence parallelism: Making 4d parallelism possible. *arXiv preprint arXiv:2105.13120* (2021).

[43] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. 2020. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704* (2020).

[44] Bin Lin, Chen Zhang, Tao Peng, Hanyu Zhao, Wencong Xiao, Minmin Sun, Anmin Liu, Zhipeng Zhang, Lanbo Li, Xiafei Qiu, Shen Li, Zhigang Ji, Tao Xie, Yong Li, and Wei Lin. 2024. Infinite-LLM: Efficient LLM Service for Long Context with Distattention and Distributed KVCache. *arXiv preprint arXiv:2401.02669* (2024).

[45] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437* (2024).

[46] Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, et al. 2025. Deepseek-v3. 2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556* (2025).

[47] Hao Liu and Pieter Abbeel. 2024. Blockwise Parallel Transformers for Large Context Models. *Advances in Neural Information Processing Systems* 36 (2024), 8828–8844.

[48] Hao Liu, Matei Zaharia, and Pieter Abbeel. 2023. Ring attention with blockwise transformers for near-infinite context. *arXiv preprint arXiv:2310.01889* (2023).

[49] Shuo Liu, Qiaoling Wang, Junyi Zhang, Wenfei Wu, Qinliang Lin, Yao Liu, Meng Xu, Marco Canini, Ray CC Cheung, and Jianfei He. 2023. In-network aggregation with transport transparency for distributed training. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. Association for Computing Machinery, New York, NY, USA, 376–391.

[50] Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, Michael Maire, Henry Hoffmann, Ari Holtzman, and Junchen Jiang. 2023. CacheGen: Fast Context Loading for Language Model Applications. *arXiv preprint arXiv:2310.07240* (2023).

[51] Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. 2024. Scissorhands: Exploiting the persistence of importance hypothesis for llm kv cache compression at test time. *Advances in Neural Information Processing Systems* 36 (2024), 52342–52364.

[52] Enzhe Lu, Zhejun Jiang, Jingyuan Liu, Yulun Du, Tao Jiang, Chao Hong, Shaowei Liu, Weiran He, Enming Yuan, Yuzhi Wang, Zhiqi Huang, Huan Yuan, Suting Xu, Xinran Xu, Guokun Lai, Yanru Chen, Huabin Zheng, Junjie Yan, Jianlin Su, Yuxin Wu, Yutao Zhang, Zhilin Yang, Xinyu Zhou, Mingxing Zhang, and Jiezhong Qiu. 2025. Moba: Mixture of block attention for long-context llms. *arXiv preprint arXiv:2502.13189* (2025).

[53] Shuming Ma, Hongyu Wang, Lingxiao Ma, Lei Wang, Wenhui Wang, Shaohan Huang, Li Dong, Ruiping Wang, Jilong Xue, and Furu Wei. 2024. The era of 1-bit llms: All large language models are in 1.58 bits. *arXiv preprint arXiv:2402.17764* (2024).

[54] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2016. Pointer Sentinel Mixture Models. arXiv:1609.07843 [cs.CL]

[55] Tsendsuren Munkhdalai, Manaal Faruqui, and Siddharth Gopal. 2024. Leave no context behind: Efficient infinite context transformers with infini-attention. *arXiv preprint arXiv:2404.07143* (2024).

[56] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM symposium on operating systems principles*. Association for Computing Machinery, New York, NY, USA, 1–15.

[57] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. 2021. Memory-efficient pipeline-parallel dnn training. In *International Conference on Machine Learning*. PMLR, PMLR, N/A, 7937–7947.

[58] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Mostofa Patwary, Raghavendra Puri, Patrick LeGresley, and Bryan Catanzaro. 2021. 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)*. Association for Computing Machinery, New York, NY, USA, 1–15.

[59] E.C.I. NETWORKS. 2023. 32 x 400G QSFP-DD Switch Baremetal Switch with programmable Broadcom Trident 4. https://ecin.ca/32-x-400g-qsfp-dd-switch-baremetal-switch-with-programmable-broadcom-trident-4.

[60] NVIDIA. [n. d.]. Context parallelism overview. https://docs.nvidia.com/megatron-core/developer-guide/latest/api-guide/context_parallel.html.

[61] NVIDIA. 2020. NVIDIA TensorRT. https://developer.nvidia.com/tensorrt. (2020).

[62] NVIDIA. 2024. NVIDIA DGX-B200. https://www.nvidia.com/en-us/data-center/dgx-b200/.

[63] Tian Pan, Nianbing Yu, Chenhao Jia, Jianwen Pi, Liang Xu, Yisong Qiao, Zhiguo Li, Kun Liu, Jie Lu, Jianyuan Lu, Enge Song, Jiao Zhang, Tao Huang, and Shunmin Zhu. 2021. Sailfish: accelerating cloud-scale multi-tenant multi-service gateways with programmable switches. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 194–206. doi: 10.1145/3452296.3472889

[64] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Aashaka Shah, Saeed Maleki, and Ricardo Bianchini. 2023. Splitwise: Efficient generative llm inference using phase splitting. *arXiv preprint arXiv:2311.18677* (2023).

[65] Bowen Peng, Jeffrey Quesnelle, Honglu Fan, and Enrico Shippole. 2023. Yarn: Efficient context window extension of large language models. *arXiv preprint arXiv:2309.00071* (2023).

[66] Bo Peng, Ruichong Zhang, Daniel Goldstein, Eric Alcaide, Xingjian Du, Haowen Hou, Jiaju Lin, Jiaxing Liu, Janna Lu, William Merrill, Guangyu Song, Kaifeng Tan, Saiteja Utpala, Nathan Wilce, Johan S. Wind, Tianyi Wu, Daniel Wuttke, and Christian Zhou-Zheng. 2025. Rwkv-7" goose" with expressive dynamic state evolution. *arXiv preprint arXiv:2503.14456* (2025).

[67] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2023. Efficiently scaling transformer inference. *Proceedings of Machine Learning and Systems* 5 (2023), 606–624.

[68] Gemini Pro. 2024. Our best model for general performance across a wide range of tasks. https://deepmind.google/technologies/gemini/pro/.

[69] Chenchen Qi, Wenfei Wu, Yongcan Wang, Keqiang He, Yu-Hsiang (Sean) Kao, Zongying He, Chen-Yu Yen, Zhuo Jiang, Feng Luo, Surendra Anubolu, Yanjin Gao, Bingfeng Lin, Wenda Ni, Yiming Yang, Donglin Wei, Boyang Zhou, Jian Wang, and Shan Ding. 2025. SGLB: Scalable and Robust Global Load Balancing in Commodity AI Clusters. In *Proceedings of the ACM SIGCOMM 2025 Conference*. Association for Computing Machinery, New York, NY, USA, 626–644.

[70] Ruoyu Qin, Zheming Li, Weiran He, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2024. Mooncake: A kvcache-centric disaggregated architecture for llm serving. *arXiv preprint arXiv:2407.00079* (2024).

[71] Qwen, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. 2025. Qwen2.5 Technical Report. arXiv:2412.15115 [cs.CL] https://arxiv.org/abs/2412.15115

[72] Qwen Team. 2026. Qwen3.5: Towards Native Multimodal Agents. https://qwen.ai/blog?id=qwen3.5

[73] Markus N Rabe and Charles Staats. 2021. Self-attention does not need $O(n^2)$ memory. *arXiv preprint arXiv:2112.05682* (2021).

[74] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving Language Understanding by Generative Pre-Training. In *OpenAI Technical Report*. San Francisco, CA, USA, San Francisco, CA, USA.

[75] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[76] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2022. GPT-4. https://openai.com/index/gpt-4/. OpenAI Technical Report.

[77] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: Memory Optimization Towards Training A Trillion Parameter Models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, Atlanta, GA, USA, 1–16.

[78] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. Association for Computing Machinery, New York, NY, USA, 3505–3506.

[79] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtárik. 2021. Scaling distributed machine learning with In-Network aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, N/A, 785–808.

[80] ShareGPT Teams. 2023. ShareGPT. https://sharegpt.com/.

[81] Noam Shazeer. 2019. Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150* (2019).

[82] Vasudev Shyam, Jonathan Pilault, Emily Shepperd, Quentin Anthony, and Beren Millidge. 2024. Tree attention: Topology-aware decoding for long-context attention on gpu clusters. *arXiv preprint arXiv:2408.04093* (2024).

[83] Haoyu Song. 2024. In-Network AllReduce Optimization with Virtual Aggregation Trees. In *Proceedings of the 2024 SIGCOMM Workshop on Networks for AI Computing*. Association for Computing Machinery, New York, NY, USA, 54–60.

[84] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. *arXiv preprint arXiv:2302.13971* (2023).

[85] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[86] Feiyu Wang, Guoan Wang, Yihao Zhang, Shengfan Wang, Weitao Li, Bokai Huang, Shimao Chen, Zihan Jiang, Rui Xu, and Tong Yang. 2025. iFairy: the First 2-bit Complex LLM with All Parameters in $\{\pm 1, \pm i\}$. *arXiv preprint arXiv:2508.05571* (2025).

[87] Hongyu Wang, Shuming Ma, Li Dong, Shaohan Huang, Huaijie Wang, Lingxiao Ma, Fan Yang, Ruiping Wang, Yi Wu, and Furu Wei. 2023. Bitnet: Scaling 1-bit transformers for large language models. *arXiv preprint arXiv:2310.11453* (2023).

[88] WCCFTECH. 2025. NVIDIA Rubin Is The Most Advanced AI Platform On The Planet. https://wccftech.com/nvidia-rubin-most-advanced-ai-platform-50-pflops-vera-cpu-5x-uplift-vs-blackwell.

[89] IJsbrand Wijnands, Eric C. Rosen, Andrew Dolganow, Tony Przygienda, and Sam Aldrin. 2017. Multicast Using Bit Index Explicit Replication (BIER). RFC 8279. doi: 10.17487/RFC8279

[90] Bingyang Wu, Shengyu Liu, Yinmin Zhong, Peng Sun, Xuanzhe Liu, and Xin Jin. 2024. LoongServe: Efficiently Serving Long-Context Large Language Models with Elastic Sequence Parallelism. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. Association for Computing Machinery, New York, NY, USA, 640–654.

[91] Bingquan Xia, Bowen Shen, Cici, Dawei Zhu, Di Zhang, Gang Wang, Hailin Zhang, Huaqiu Liu, Jiebao Xiao, Jinhao Dong, Liang Zhao, Peidian Li, Peng Wang, Shihua Yu, Shimao Chen, Weikun Wang, Wenhan Ma, Xiangwei Deng, Yi Huang, Yifan Song, Zihan Jiang, Bowen Ye, Can Cai, Chenhong He, Dong Zhang, Duo Zhang, Guoan Wang, Hao Tian, Haochen Zhao, Heng Qu, Hongshen Xu, Jun Shi, Kainan Bao, Kai Fang, Kang Zhou, Kangyang Zhou, Lei Li, Menghang Zhu, Nuo Chen, Qiantong Wang, Shaohui Liu, Shicheng Li, Shuhao Gu, Shuhuai Ren, Shuo Liu, Sirui Deng, Weiji Zhuang, Weiwei Lv, Wenyu Yang, Xin Zhang, Xing Yong, Xing Zhang, Xingchen Song, Xinzhe Xu, Xu Wang, Yihan Yan, Yu Tu, Yuanyuan Tian, Yudong Wang, Yue Yu, Zhenru Lin, Zhichao Song, and Zihao Yue. 2025. MiMo: Unlocking the Reasoning Potential of Language Model–From Pretraining to Posttraining. *arXiv preprint arXiv:2505.07608* (2025).

[92] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. 2023. Efficient streaming language models with attention sinks. *arXiv preprint arXiv:2309.17453* (2023).

[93] Amy Yang, Jingyi Yang, Aya Ibrahim, Xinfeng Xie, Bangsheng Tang, Grigory Sizov, Jeremy Reizenstein, Jongsoo Park, and Jianyu Huang. 2024. Context Parallelism for Scalable Million-Token Inference. *arXiv preprint arXiv:2411.01783* (2024). https://arxiv.org/abs/2411.01783

[94] Zhewei Yao, Cheng Li, Xiaoxia Wu, Stephen Youn, and Yuxiong He. 2023. A comprehensive study on post-training quantization for large language models. *arXiv preprint arXiv:2303.08302* (2023).

[95] Jingyang Yuan, Huazuo Gao, Damai Dai, Junyu Luo, Liang Zhao, Zhengyan Zhang, Zhenda Xie, Yuxing Wei, Lean Wang, Zhiping Xiao, Yuqing Wang, Chong Ruan, Ming Zhang, Wenfeng Liang, and Wangding Zeng. 2025. Native sparse attention: Hardware-aligned and natively trainable sparse attention. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 23078–23097.

[96] Yifan Yuan, Omar Alama, Jiawei Fei, Jacob Nelson, Dan RK Ports, Amedeo Sapio, Marco Canini, and Nam Sung Kim. 2022. Unlocking the power of inline {Floating-Point} operations on programmable switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 683–700.

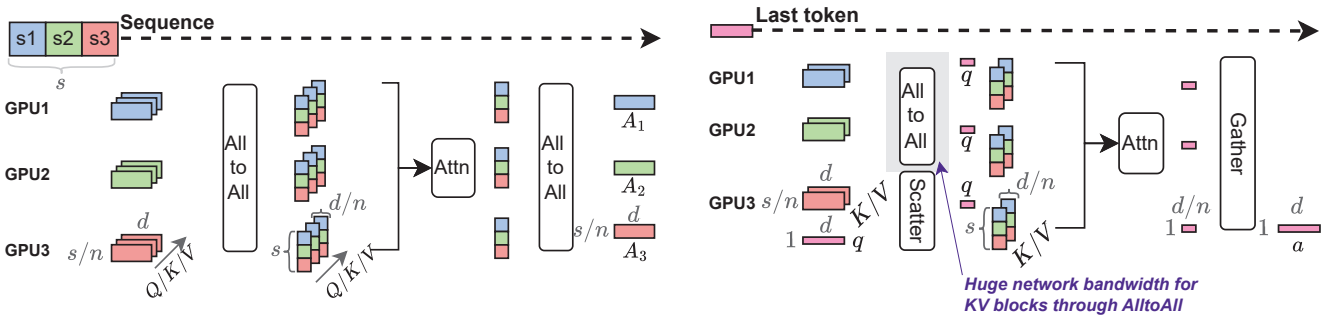
[97] Yu Zhang, Zongyu Lin, Xingcheng Yao, Jiaxi Hu, Fanqing Meng, Chengyin Liu, Xin Men, Songlin Yang, Zhiyuan Li, Wentao Li, Enzhe Lu, Weizhou Liu, Yanru Chen, Weixin Xu, Longhui Yu, Yejie Wang, Yu Fan, Longguang Zhong, Enming Yuan, Dehao Zhang, Yizhi Zhang, T.Y. Liu, Haiming Wang, Shengjun Fang, Weiran He, Shaowei Liu, Yiwei Li, Jianlin Su, Jiezhong Qiu, Bo Pang, Junjie Yan, Zhejun Jiang, Weixiao Huang, Bohong Yin, Jiacheng You, Chu Wei, Zhengtao Wang, Chao Hong, Yutian Chen, Guanduo Chen, Yucheng Wang, Huabin Zheng, Feng Wang, Yibo Liu, Mengnan Dong, Zheng Zhang, Siyuan Pan, Wenhao Wu, Yuhao Wu, Longyu Guan, Jiawen Tao, Guohong Fu, Xinran Xu, Yuzhi Wang, Guokun Lai, Yuxin Wu, Xinyu Zhou, Zhilin Yang, and Yulun Du. 2025. Kimi linear: An expressive, efficient attention architecture. *arXiv preprint arXiv:2510.26692* (2025).

[98] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, Zhangyang "Atlas" Wang, and Beidi Chen. 2024. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems* 36 (2024), 34661–34710.

[99] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2024. SGLang: efficient execution of structured language model programs. In *Proceedings of the 38th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '24)*. Curran Associates Inc., Red Hook, NY, USA, Article 2000, 27 pages.

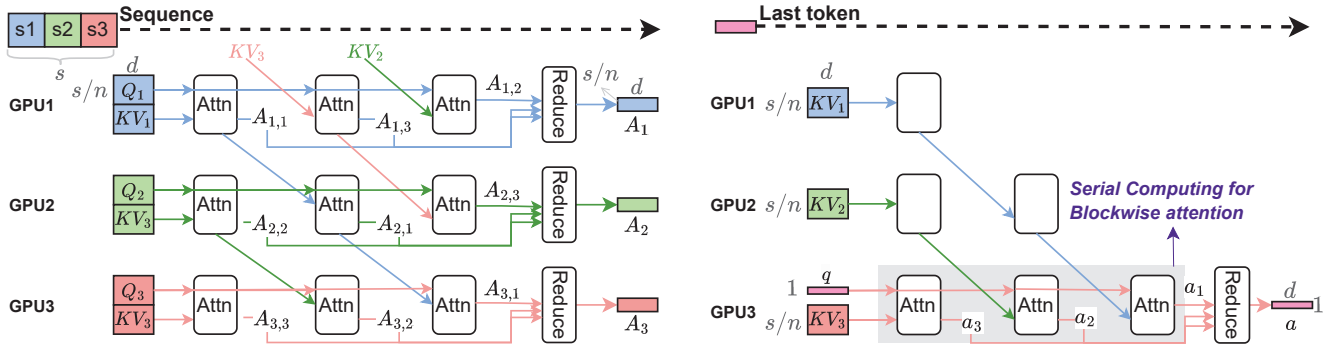
[100] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 193–210. https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin

[101] Minxuan Zhou, Weihong Xu, Jaeyoung Kang, and Tajana Rosing. 2022. TransPIM: A Memory-based Acceleration via Software-Hardware Co-Design for Transformer . In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 1071–1085. doi: 10.1109/HPCA53966.2022.00082



(a) Prefill phase of SP-Ulysses (b) Decoding phase of SP-Ulysses

Figure 16: Working flows of SP-Ulysses in LLM inference.



(a) Prefill phase of SP-Ring (b) Decoding phase of SP-Ring

Figure 17: Working flows of SP-Ring in LLM inference.

Appendices are supporting material that has not been peer-reviewed.

A Sequence parallelism

In this section, we introduce two mature approaches for SP-based attention computing: SP-Ulysses, represented by Ulysses-Attention [32], and SP-Ring, i.e., pass-KV (ring), represented by Ring-Attention [41, 42, 48]. In this section, we introduce some backgrounds that are not mention in the main body.

A.1 SP-Ulysses

The core idea of SP-Ulysses is performing head parallel (i.e., TP), where each node is responsible for the attention of h/n complete heads, where h is the number of heads in multi-head attention and n is the number of SP nodes, i.e., the degree of SP. As shown in Figure 16(a), SP-Ulysses first allows node i to calculate $\mathbf{Q}_i, \mathbf{K}_i, \mathbf{V}_i$ tensors in the shape of $[s/n, d]$ for its input sequence block. Then, SP-Ulysses adopts `AlltoAll` collective to ensure that each node receives the complete $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ tensors of its expected heads, which are in the shape of $[s, d/n]$, so that each node can perform standard attention computing just as under TP. Finally, SP-Ulysses performs another AlltoAll to gather the results across the head dimension and reshape them along the sequence dimension.

SP-Ulysses essentially converts sequence parallel into head parallelism (i.e., TP) through the AlltoAll operation. As shown in Figure 16(b), in the decoding phase, the master node should `scatter` $\mathbf{q}$ along the head dimension to other workers, while the transmission of KV tensors is still through AlltoAll. Next, all nodes compute the head-parallel attentions. Finally, the master node will `gather` the attentions to form the complete attention $\mathbf{a}$. Hence, we can get the inference time per layer as below:

$$T = T_{\text{all2all}} + T_{\text{head_attn}} + T_{\text{gather}}. \tag{8}$$

where T_{all2all} denotes the network transmission delay of AlltoAll, $T_{\text{head_attn}}$ denotes the computing delay of the head-parallel attention, and T_{gather} denotes the network delay of the gather collective.

Although SP-Ulysses achieves distributed KV blocks on the sequence dimension and parallel attention computation in the decoding phase, it has the following issues.

Limited scalability. The maximum sequence parallel degree of SP-Ulysses is limited by the number of heads in the model. Notably, the widely used GQA [3] greatly reduces the number of heads in the key/value tensors, e.g., 8 heads in LLAMA3-8B with GQA. As a result, it cannot adapt to an extremely long sequence. This limited scalability also lower the inference speed, since each GPU at least calculates an attention of one head, whose time delay ($T_{\text{head_attn}}$ in Eq. (8)) still grows as sequence length increases.

Large amount of network traffic. The network traffic in SP-Ulysses is large, as the AlltoAll collective for exchanging KV cache is inevitable. Let B denote the size of a KV block and n denote the number of nodes, the total network traffic is $O(nB)$, incurring massive network traffic and even incast.

A.2 SP-Ring

SP-Ulysses is limited by the degree of SP n, as n cannot exceeds the number of heads h. SP-Ring stems from the *blockwise attentions* [15,

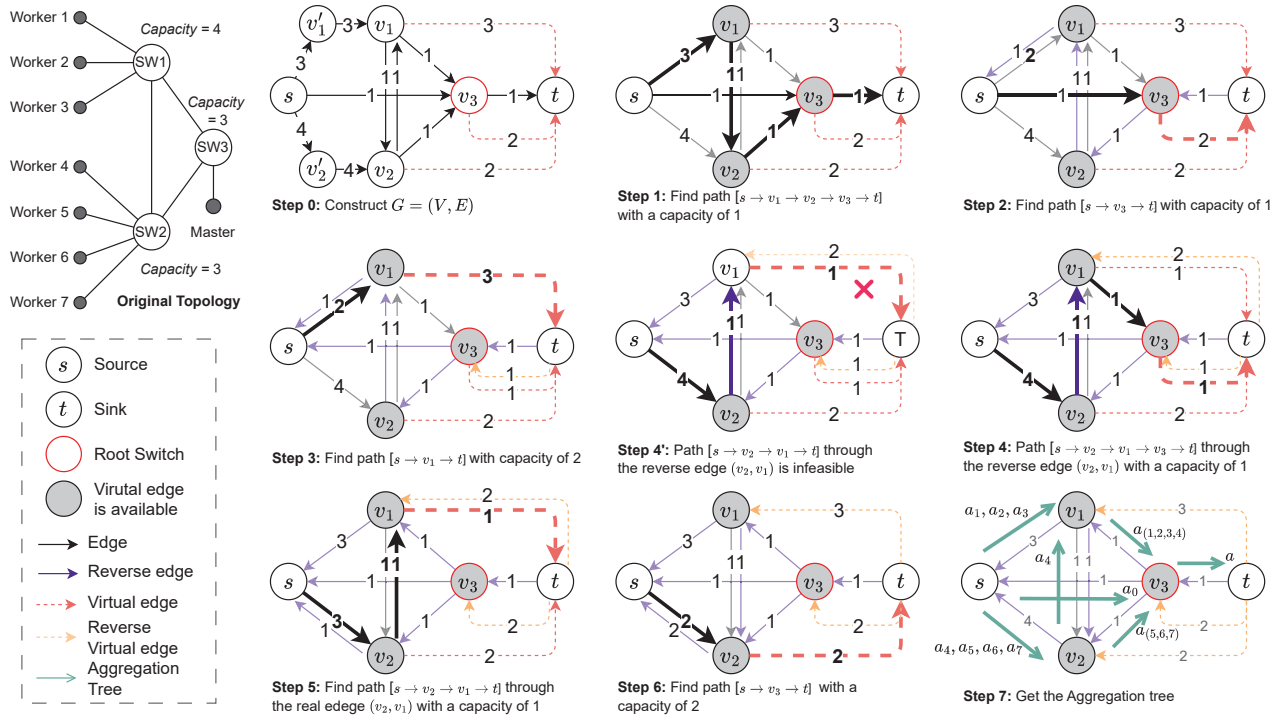


Figure 18: A detailed example to illustrate the process of constructing a flow aggregation tree: first convert the topology to a flow network $G = (V, E)$ with a capacity function $c : E \to \mathbb{R}^+$, then find the maximum flow $f_{\max} : E \to \mathbb{Z}^+$ based on the Ford-Fulkerson method, and finally extract the aggregation tree from $f_{\max}$.

47, 73], which allows to compute attentions for each sequence block and then aggregate them by the scaling-reduce operation to get the final results.

As shown in Figure 17(a), SP-Ring first computes $\mathbf{Q}_i$, $\mathbf{K}_i$, and $\mathbf{V}_i$ tensors along the sequence dimension distributively. However, it adopts the *Q-stable* policy, where the query Q_i is fixed in node i and the KV blocks are delivered in a ring fashion. In this way, for any KV block j, node i can compute its blockwise attention $\mathbf{A}_i^{(j)}$, and then aggregate all of them to get the attention $\mathbf{A}_i$ of block i by the scaling reduce. Note that there is no tensor re-partition in SP-Ring, and $\mathbf{Q}_i$, $\mathbf{K}_i$, $\mathbf{V}_i$ and $\mathbf{A}_i$ keeps the same shape of $[s/n, d]$. In the case that the communication is hidden in computation, SP-Ring can realize high performance in LLM training and inference's prefill. Besides, each node needs an extra buffer to receive the KV block from its neighbor. To address the computing imbalance in causal attention, [9, 40] distribute the sequence blocks in a back and forth order, and in this paper, we assume such method is enabled.

B Details of the maximum flow based aggregation tree construction

We use the topology shown in the Figure 18 to illustrate our algorithm of constructing the aggregation tree, which mainly consists of tree phases: (*i*) construct the flow network, (*ii*) find the maximum flow and (*iii*) convert the maximum flow to the aggregation tree.

B.1 Construct the flow network

The flow network is denoted by a directed graph $G = (V, E)$ with a capacity function $c : E \to \mathbb{Z}^+$, where V and E are the vertex and edge sets, respectively. We first need to convert the topology to the flow network as shown in the step 0 in Figure 18.

Intermediate vertices: In the beginning, each switch or worker maps to an intermediate vertex one-by-one, e.g., v_1, v_2 and v_3 corresponding to switch 1,2, and 3. For each physical worker-switch link, we set a directed edge from the worker to the switch with a capacity of 1. Note that vertices and edges can be aggregated, such as v_1' corresponding to worker 1, 2, and 3, and edge $c(v_1', v_1) = 3$. In contrast, each physical link between switches maps to a pair of directed edges with a capacity of 1, e.g., edges (v_1, v_2) and (v_2, v_1), but for simplicity, we do not show directed edges (v_3, v_1) and (v_3, v_2) since there is no flow through them in the optimal solution. Besides, we need to select a switch node as the root vertex, e.g., v_3, which is responsible for receiving a unit flow from the master.

Source and sink vertices: We abstract a source vertex s and a sink vertex t for the target flow. The source s has a directed edge to each worker vertex with a capacity of the number of aggregated workers there, e.g., $c(s, v_1') = 3$. For the topology in Figure 18, each worker is only connected to a switch, so we can simplify the network by omitting v_1' and v_2' without influence the correctness. However, for network topologies where a worker is connected to multiple switches, such simplification is not allowed. Since the master also has an attention to be aggregated on the root switch, s has a directed edge to the root vertex with a capacity of 1, e.g., (s, v_3). In ideal, only the final aggregated attention a is sent to the master from the

root switch, we set a directed edge with a capacity of 1, e.g., (v_3, t).
Virtual edges: For each switch vertex v whose aggregation capacity $C(v)$ is greater than 1, we construct a virtual directed edge to t with $c(v, t) = C(v) - 1$. This is crucial for reducing the problem to a maximum flow problem, as it ensures that the in and out flow sizes of each intermediate node are equal.

B.2 Find the maximum flow

Once the flow network is constructed, we search for the augmenting paths iteratively based on the Ford-Fulkerson method with an extra constraint: for each switch vertex, its virtual edges are available only if at least one of its real edges is occupied. Next, we detail the algorithm with the above constraint through steps 1 to 6 in Figure 18[6].
Step 1: Find an augmenting path $P_1 = [s \to v_1 \to v_2 \to v_3 \to t]$ with a capacity of 1. Then, we update the residual network $G_f(V, E_f)$ by reducing the capacity of each edge on the augmenting path by 1 and increasing the capacity of its reverse edge by 1. Specifically, since v_1, v_2, and v_3 all have flow through non-virtual edges, we update their states to indicate that their virtual edges are now available.
Step 2: Find an augmenting path $P_2 = [s \to v_3 \to t]$ with a capacity of 1. Note that since the virtual edge on V3 is available, the flow from v_3 to t can go through the virtual edge. We then update the residual network accordingly.
Step 3: Similar to step 2, find an augmenting path $P_3 = [s \to v_1 \to t]$ with a capacity of 2 and update the residual network.
Step 4': An augmenting path $[s \to v_2 \to v_1 \to t]$, where (v_2, v_1) is through the reverse edge, is infeasible. This is because this path will withdraw the flow from v_1 to v_2 in P_1, but its segment (v_1, t) is through a virtual edge. Consequently, switch vertex v_1 no longer has out flow through real edges, violating the aforementioned constraint.
Step 4: In contrast, the path $P_4 = [s \to v_2 \to v_1 \to V_3 \to t]$ with a capacity of 1 is feasible, where (v_2, v_1) is still through the reverse edge. This is because though the flow from v_1 to v_2 in P_1 is withdrawn, the added real edge (v_3, t) maintains the constraint.
Step 5: Find an augmenting path $P_5 = [s \to v_2 \to v_1 \to t]$, where (v_2, v_1) is through the real edge, with a capacity of 1.
Step 6: Find the last augmenting path $P_6 = [s \to v_2 \to t]$ with a capacity of 2. So far, no augmenting paths exist from s to t, and the paths for the maximum flow are obtained.

B.3 Extract the aggregation tree

The total capacity of the obtained maximum flow is eight, equal to the upper limit flow from s, meaning all blockwise attentions can be aggregated in network and the root switch can deliver the final single attention to the master. Next, we build the aggregation tree by uniting the six augmenting paths as shown in Figure 18. Specially, paths share the opposite segments may cancel each other, e.g., the edge (v_1, v_2) in path P_1 and (v_2, v_1) in path P_4. The flow capacity on each virtual edge plus 1 is the number of attentions to be aggregated on the switch, but which attentions to be aggregated is free, e.g., any combination of three attentions in a_3, a_4, a_5 and a_6 is feasible in switch 2.

If enabling the master to aggregate the remaining attentions, we should reconstruct the flow network as master does not need to send its attention out and it can receive attentions from other physically connected switches. For the example in Figure 18, the edges (s, v_3) should be deleted, and edges (s, v_1) and (s, v_2) with capacities of 1 should be inserted, if the master is connected to switch 1 and 2.

[6]For ease of understanding, we adopt the basic version of Ford-Fulkerson method and do not show edges with zero residual capacity.

C Turbo's reliability handling mechanism

Host-side mechanism. Turbo's network stack applies a window protocol with an ACK acknowledgment mechanism. When a worker receives an ACK, it records the successful transmission and the in-network aggregation of the data packet with the same PSN. Then, the worker advances the window consecutively to the first unacknowledged data packet and continues to send new packets in the window. It also checks the waiting time of the unacknowledged packets to detect packet loss. If any in-flight packet experiences a timeout, indicating possible packet loss, the worker selectively retransmits it and resets the timeout clock.
Switch-side logic. The switch maintains an array `seen[]` to handle packet loss and guarantee reliability. When a packet arrives at the switch, the switch first matches the packet header fields to find its states (including PSN) and then indexes the packet arrival state in `seen[pkt.psn%S]`, where `S` is the array length. If the bit is 0, the switch marks the packet as `new`; otherwise, the packet is retransmitted, and is marked as `resent`; in both cases, the bit `seen[pkt.psn%S]` is set as 1, indicating the observation of the packet. Additionally, to recycle the bit array `seen`, a future bit "one-window" away is reset as 0: `seen[(pkt.psn+W)%S]=0`, which is analyzed in prior work [39, 49, 79] and proved reliability.

The switch then proceeds with the packet. The packet is addressed to an aggregator `agg[pkt.psn%S]` and its corresponding degree counter `deg[pkt.psn%S]`. If the packet is marked `new`, the aggregator performs in-network aggregation, and the degree counter increments 1. The switch then checks whether the aggregation is complete by comparing the degree with the fan-in degree. If the aggregation is not complete, the packet is dropped; otherwise, the aggregated attention tensor is copied to the packet payload. When the aggregation at one index is complete, the switch also clears the aggregator and degree counter one window away to recycle the arrays (i.e., at `(psn+W)%S`).

D LLM serving with Turbo

Based on the above description on Turbo's methodology and techniques, we propose the system design with Turbo as shown in Figure 19, which consists of four parts:

- A centralized control node, responsible for distributing inference task and managing the distributed resources;
- A master node, the main process to run the inference task with the complete model loaded;
- Multiple worker nodes, offloading the blockwise attention computation and KV block storage;
- A network, offloading the attention aggregation.

Apparently, the computation is heterogeneous in the master node and workers, so we can deploy some cheap GPUs with lower com-

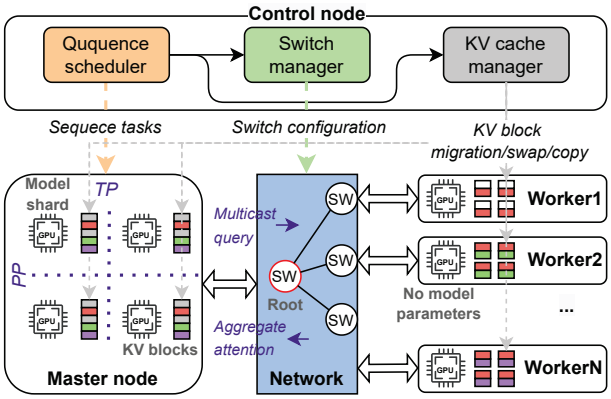


Figure 19: The inference architecture with Turbo.

puting power but large memory as workers, thereby reducing the overall price of the system.

Control process. There are three kinds of workflows in the control process, which are all launched by the control node.

First, each inference step starts with the sequence scheduler, which selects a batch of sequences, according to a specific scheduling algorithm such as First-Come-First-Served (FCFS), and distributes them to the master node.

Second, when a new long sequence is scheduled, the switch manager should build a new aggregation tree (see §4.3), and update the switch configurations. The switch update is also needed when a long sequence is finished.

Third, if the last sequence block exceeds c, the maximum length of each sequence block, the KV cache manager will migrate it to a worker, thus scaling to infinite sequence as long as worker memory is sufficient. Specially, for a long sequence, we can store its first and last KV block in the master node to improve the memory utilization under the prefix sharing [37]. Note that the master node is a logical concept, which supports TP and PP for running large models, so the KV blocks in workers are also organized with respect to the TP and PP settings to simplify the migration process. Besides, the KV cache manager also supports swapping KV tensors between one node's GPU and CPU memory, and copying KV tensors in the same GPU memory.

Decoding process. The attention computation in the decoding phase consists of three steps: in-network broadcast, parallel computing and in-network aggregation.

In the first step, the master node should broadcast query $\mathbf{q}$ to other workers. The master packs the query $\mathbf{q}$ into multiple request packets and sends them to the root switch of the aggregation tree. The packet header includes some fields to identify the sequence, query segment, and target KV blocks in the workers (see §5). Upon receiving a request packet, each switch parses the sequence ID and multicasts the packet to the corresponding child switches. This process continues until all request packets arrive at the workers.

Once a worker i receives a complete $\mathbf{q}$, it will compute the blockwise attention $\mathbf{a}^{(i)}$ using its local $\mathbf{K}_i$ and $\mathbf{V}_i$ tensors, and then send multiple response packets back. Each response packet contains a segment of $\mathbf{a}^{(i)}$ and the scaling-reduce parameter for the online aggregation (see §3.2). Note that the attention computation is model free, i.e., workers do not need to load the model parameters. If the network can aggregatate all attentions, the master node also computes its own blockwise attention and sends it to the root switch.

In the last in-network aggregation step, switches perform the online scaling-reduce operation for the attention segment and update its local states for each received response packet from their configured ports. Similar to ATP [39], bitmaps are used to determine if aggregation is complete, and a response packet is sent to the parent switch if so. The root switch finally delivers the attention a to the master for subsequent transformer processes.