

Programmable Scale-Up Switches for Software-Defined GPU Fabrics

Shouxu Lin
Cornell University
Ithaca, USA

Daehyeok Kim
University of Texas at Austin
Austin, USA

Ran Shu^{*}
Microsoft Research
Beijing, China

Jiaxin Lin
Cornell University
Ithaca, USA

Abstract

An AI cluster now runs two networks. Alongside the Ethernet scale-out fabric, a scale-up fabric such as NVLink or UALink carries accelerators' most bandwidth-intensive traffic, and it has grown from a few GPUs in a server into a rack- and cluster-scale network of hundreds of accelerators. Yet unlike the Ethernet fabric beside it, it is a network software cannot program. Accelerators inject load/store transactions that traverse links and switches with no software interposition, leaving offload, observability, and resilience beyond software's reach.

We argue that scale-up fabrics should become software-defined, as programmable switches did for Ethernet over the past decade, and that the switch is the natural place to begin. The packet match-action model of P4 and RMT does not carry over, because a scale-up switch must operate on load/store transactions rather than independent packets, stay expressive for emerging GPU communication yet bounded for line rate, and never corrupt GPU memory. We take a first step with *SUMO*, a transaction-aware programming model that composes bounded, verifiable operators over load/store transactions to express offload, routing, and telemetry.

CCS Concepts

• **Networks** → **Network protocols**; **Network architectures**; **Data center networks**.

Keywords

Programmable switches; GPU interconnects

^{*}Ran Shu is now at Broadcom.



This work is licensed under a Creative Commons Attribution 4.0 International License.

HotNets '26, Salt Lake City, UT, USA

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-3029-0/2026/11

<https://doi.org/10.1145/3847290.3859967>

ACM Reference Format:

Shouxu Lin, Ran Shu, Daehyeok Kim, and Jiaxin Lin. 2026. Programmable Scale-Up Switches for Software-Defined GPU Fabrics. In *The 25th ACM Workshop on Hot Topics in Networks (HotNets '26)*, November 16–17, 2026, Salt Lake City, UT, USA. ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/3847290.3859967>

1 Introduction

Modern AI training and inference span hundreds to thousands of accelerators, with communication demands increasingly exceeding the capacity of conventional Ethernet networks. Accelerator systems are therefore adopting tightly coupled *scale-up* fabrics, such as NVIDIA NVLink [2], UALink [55], and Google TPU's Inter-Chip Interconnect (ICI) [19, 27]. Once confined to a few accelerators within a server, these fabrics now span racks and clusters with hundreds of accelerators [1, 7]. Unlike Ethernet, scale-up fabrics integrate directly with accelerator memory systems, enabling communication through fine-grained load/store transactions rather than packets [55]. These operations bypass the host CPU, OS, NIC, and much of the protocol stack, delivering high bandwidth and low latency [13, 22].

This performance comes at a cost: today's fabrics are fixed-function and opaque. Accelerators inject load/store transactions directly into the fabric, but no software layer can match, transform, replicate, route, or observe them in transit. We call this the *programmability gap*. This gap is a design choice, not a fundamental limit: memory semantics do not preclude in-fabric programmability, but preserving these semantics and performance is challenging, so current fabrics omit it. This gap leads to three limitations:

(1) Restricted communication offload:

Existing scale-up switches such as NVSwitch support collective offloads, but are limited to fixed-function primitives and support only static communication patterns [38]. As shown in §2.2.1, this rigidity limits offload performance and cannot support emerging dynamic patterns such as MoE token routing.

- (2) **Limited performance observability:** Today’s scale-up fabrics expose only coarse GPU and switch counters, while end-to-end tools offer collective traces and kernel timelines [34, 36, 39]. These tools detect communication slowdowns but cannot distinguish congestion, queuing, path imbalance, or slow memory endpoints, making root-cause diagnosis difficult (§2.2.2).
- (3) **Inflexible routing and recovery:** Scale-up topologies often provide substantial path diversity [1, 7], but software has little control over how memory transactions use those paths. As we show in §2.2.3, software cannot steer traffic around degraded links, and the fabric offers limited runtime failure detection and recovery.

We argue that closing this gap requires making scale-up fabrics *software-defined*, as programmable switches did for Ethernet. There, programmability enabled software-defined networking [9, 16, 32], in-network computation [18, 47], and flexible network management [21, 28, 44]. Scale-up fabrics need the same substrate. We focus on the switch, which sits on the datapath and observes traffic among accelerators, making it a natural choice (§3).

Scale-up switches cannot simply reuse the packet match-action model of P4 and RMT [10] designed for Ethernet, for three reasons. First, they operate on memory transactions, requiring explicit load/store semantics. Second, the programming model must support emerging GPU communication patterns while sustaining line rate. Third, correctness requirements are stricter, as faulty programs could violate protocols or corrupt GPU memory.

We take a first step toward software-defined scale-up fabrics with SUMO (Scale-Up Memory Ops), a transaction-aware switch programming model. Unlike packet-based models, SUMO operates on load/store transactions, with operators typed by requests and completions to preserve request-completion semantics. Programmers compose bounded operators for replication, address translation, reduction, routing, and telemetry into chains triggered by transactions. Before installation, a verifier checks memory correctness, resource bounds, and predictable line-rate execution. We show that SUMO supports efficient AllReduce, dynamic MoE token routing, and fine-grained in-network telemetry for rerouting. SUMO is one point in a broader design space for software-defined scale-up fabrics, and we close with the open questions it raises.

2 Background and Motivation

2.1 Scale-up Networks

A scale-up network connects accelerators into a unified logical compute unit via dedicated memory interconnects. These fabrics use accelerator links (e.g., NVLink [2], xGMI [3], and

Multimem Primitives	Description
<code>st_multicast(addr, val)</code>	Store <code>val</code> to memory location <code>addr</code> on every GPU in the group.
<code>ld_reduce(addr, d)</code>	Reduce the values loaded from memory location <code>addr</code> of all GPUs, and store the reduced result to register <code>d</code> on the calling GPU.

Table 1: NVLS multimem primitives [38].

UALink [55]) and switches (e.g., NVSwitch [15, 38] and Ultra Accelerator Switch [55]) to connect accelerator memories. These links carry cache-line-level transactions, including load/store requests and their completions [55].

Network Architecture: Early scale-up deployments were largely confined to a single server, typically connecting 4–8 accelerators through a local switched fabric [33]. More recently, scale-up connectivity has expanded beyond a single server to cluster-scale configurations [1, 7, 27, 55]. As shown in Fig. 1, NVIDIA GB200 NVL72 [1] uses a single-stage, non-blocking fabric connecting 72 GPUs to 18 NVSwitches. Each GPU connects to every switch via one NVLink, forming 18 parallel switch planes that provide rack-wide all-to-all connectivity. Beyond NVIDIA, Google TPU Pods [19, 27] use hierarchical ICI and optical switches to connect thousands of accelerators. UALink supports switched fabrics with all-to-all connectivity among 1024 accelerators [55].

Scale-up Switch and Offloads: Beyond forwarding, scale-up switches can offload GPU communication; we use NVLink SHARP (NVLS) [15, 38] as an example. As shown in Table 1, NVLS provides switch-offload primitives via *multimem* instructions. A *multimem* address maps symmetric GPU memory locations to a single logical address, enabling multicast or reduction with one instruction. Fig. 2 illustrates *multimem.st_multicast*. Without *multimem*, sender GPU G_0 must separately store data to N receivers. With *multimem*, G_0 issues one store, and the switch replicates the data to all receivers. Similarly, *ld_reduce* performs many-to-one reduction: the switch replicates a load request across GPUs, reduces the returned data, and sends the result to the requester. These primitives reduce GPU-injected traffic and GPU cycles for replication and reduction. Each instruction operates on 4–16 bytes.

2.2 Programmability Gap

Scale-up fabrics achieve high performance through hardware-native load/store semantics that avoid software overhead. However, this creates a programmability gap: software cannot route or observe in-flight memory transactions, and switch offloads are fixed-function. For example, NVLS supports in-switch multicast and reduction only as fixed hardware primitives. We show how this gap limits communication offload, observability, and resilience.

2.2.1 Limited Collective Expressiveness. Current scale-up switches support a narrow set of fixed-function operations,

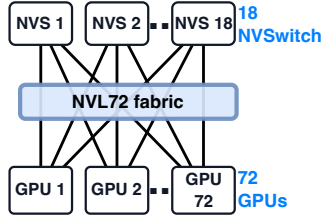


Figure 1: NVIDIA GB200 NVL72.

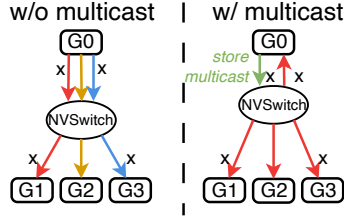
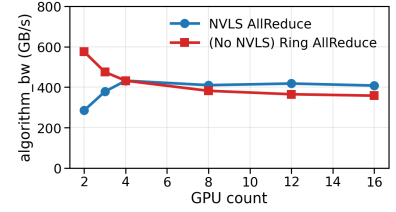

 Figure 2: Operation of *st_multicast*.


Figure 3: AllReduce on NVIDIA GB200 NVL72.

which limits both how well existing collectives perform and which MoE patterns can be offloaded.

Slow AllReduce Performance: We evaluate NVLS-offloaded AllReduce on a GB200 NVL72 system (Fig. 1). As Fig. 3 shows, NVLS delivers only modest gains over ring AllReduce: it provides no benefit with four or fewer GPUs, and even with 8–16 GPUs it achieves only $1.07\times$ – $1.15\times$ higher algorithm bandwidth, reaching just 44.4% of the 900 GB/s NVLink bandwidth. This limited benefit is consistent with prior observations [24, 53, 57, 62].

To understand why, we examine how NCCL implements AllReduce with NVLS. NCCL uses two multimem primitives, *ld_reduce* for ReduceScatter and *st_multicast* for AllGather. As shown in Fig. 4, the S -byte input tensor is partitioned into N shards across N GPUs. Each GPU reduces its assigned shard across all GPUs with *ld_reduce* and multicasts the result with *st_multicast*. During ReduceScatter, each GPU sends S bytes and receives S/N bytes; during AllGather, the reverse. This design has two inefficiencies.

First, it incurs *redundant data transfers*. Each GPU sends and receives $(1 + 1/N)S$ bytes, whereas an ideal switch-based AllReduce needs only S bytes in each direction: the local data to the switch and the reduced result back. The extra $1/N$ overhead grows as N shrinks, which is why NVLS provides no benefit with four or fewer GPUs and underperforms ring AllReduce below four GPUs (Fig. 3).

Second, bandwidth degrades when *the GPU issues multiple multimem operations*. The two phases generate asymmetric traffic: *ld_reduce* is dominated by GPU-to-switch transfers and *st_multicast* by switch-to-GPU transfers, so running them sequentially underutilizes the full-duplex NVLink. NCCL therefore splits each shard into chunks and pipelines the two primitives to overlap transfers in both directions. However, our measurements show that issuing *ld_reduce* and *st_multicast* concurrently on the same GPU reduces bandwidth from about 700 GB/s, when either primitive runs alone, to 550 GB/s, matching the measured NVLS bus bandwidth $((1 + 1/N) \times \text{algorithm bandwidth})$ in Fig. 3. We observe no such degradation when the two primitives run on different GPUs or when regular loads and stores overlap, which points to contention for GPU-side multimem resources rather than the switch. NVLS thus appears limited by the GPU overhead

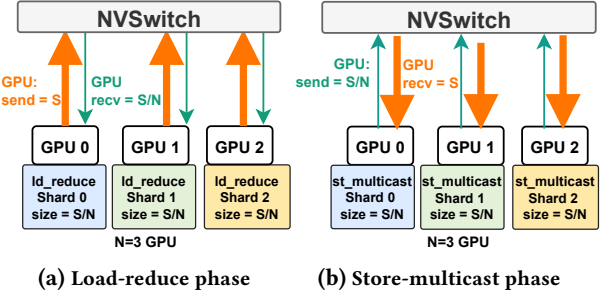


Figure 4: Two-phase implementation of AllReduce in NVLS.

of issuing fine-grained multimem operations and by GPU-side resources, such as load/store buffers, that cap in-flight multimem requests.

Together, these results expose the core limitation of today’s scale-up collective offload: GPUs must orchestrate reduction and multicast through separate fine-grained memory operations, incurring redundant transfers and GPU-side contention. Without programmability, the switch cannot perform AllReduce itself, coordinating reduction and multicast internally as scale-out in-network aggregation does [29, 47]. **Static Multimem Cannot Support Dynamic MoE Pattern:** Beyond conventional collectives, the current interface lacks the flexibility that emerging workloads require. Applications such as MoE [50] and speculative decoding [30] exhibit dynamic, asymmetric communication patterns, whereas *multimem* assumes static destination sets and symmetric addresses, *i.e.*, it operates only on memory locations at the same offset on every GPU.

MoE dispatch is a representative example. Multicast can send a token to multiple experts across GPUs, but destination sets vary by token. As shown in Fig. 5, tokens 0 and 1 target $\{G_0, G_1\}$, while token 2 targets $\{G_1, G_2\}$. Moreover, each GPU’s receive buffer is typically partitioned by source GPU, with tokens from G_0 occupying one contiguous segment, followed by those from G_1, G_2 , and so on. Since GPUs receive different token counts from each sender, the same token may occupy different offsets across destination GPUs. For example, token 2 occupies slots 0 and 3 in the receive buffers of G_1 and G_2 , respectively. NVLS supports neither dynamic destination sets nor asymmetric offsets, so implementing

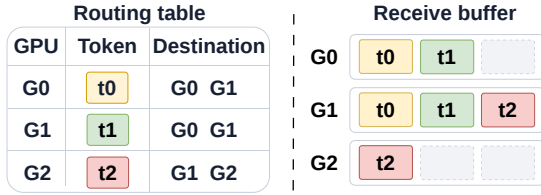


Figure 5: MoE token dispatch.

Causes of congestion	Bandwidth (GB/s)	Slowdown (%)
Sender	170.2	57.5
Receiver	190.1	52.5
Link	132.3	66.9

Table 2: Bandwidth degradation caused by different types of congestion on an NVIDIA DGX H100 server (450 GB/s NVLink).

dispatch with its multicast primitive incurs substantial redundant traffic [61].

2.2.2 Limited Fabric Observability. Scale-up memory transactions are largely invisible to software, preventing developers from tracing individual transactions. Applications observe only kernel or collective completion times and aggregate switch counters, while per-link traffic, request latency, backpressure, and switch queueing remain opaque.

This opacity complicates performance diagnosis. Suppose bandwidth between two GPUs drops from 400 GB/s to 200 GB/s. GPU probe kernels [46], application profilers such as Nsight Systems [39], and collective traces such as NCCL Inspector [12, 59] can detect the slowdown but not directly identify its cause. Possible causes include: (1) the source GPU cannot inject data quickly enough because the communication kernel is delayed due to SM contention; (2) the destination GPU cannot drain incoming data because of HBM contention; (3) a GPU-to-switch uplink or switch-to-GPU downlink experiences degradation or congestion; or (4) an internal switch path is congested or impaired.

We reproduce three of these causes and find that they produce similar end-to-end slowdowns (Table 2). For case (1), we co-locate a communication kernel sending from G_0 to G_1 with a computation kernel on G_0 whose combined thread blocks exceed the GPU’s resident-block capacity. Contention for SMs, registers, and shared memory cuts the communication bandwidth from 400 GB/s to 170.2 GB/s. For case (2), a communication kernel on G_0 sends data to G_1 while a memory-read kernel on G_1 loads data from HBM into shared memory. The incoming writes and local reads contend for HBM bandwidth, reducing the communication bandwidth by 52.5%. For case (3), three GPUs simultaneously send to one receiver GPU. The transfers contend for the switch-to-receiver downlink, reducing per-transfer bandwidth by 66.9%.

Because these causes look alike end to end, pinpointing the origin of a slowdown requires fine-grained telemetry from inside the fabric.

2.2.3 Inflexible Routing without Resilience. Scale-up fabrics offer substantial path diversity but little application control over how memory requests map to physical paths. For example, in NVL72, each GPU connects to 18 NVSwitch ASICs via 18 NVLinks, providing multiple routes between GPU pairs [1]. However, CUDA peer-memory access exposes remote GPU memory as an address: kernels issue loads and stores but cannot select or weight the physical route of individual requests [37].

This separation is particularly problematic under *grey failures*, where a path remains reachable but offers substantially lower bandwidth. Consider two parallel path groups: one healthy and one degraded from 900 GB/s to 100 GB/s [6, 8]. An equal traffic split makes the degraded group the bottleneck, whereas a capacity-aware 9:1 split allows both groups to finish at roughly the same time and preserves more aggregate bandwidth. Because kernels can neither map individual memory requests to physical paths nor adjust the split as link conditions change, they cannot perform such rebalancing.

3 Programmable Scale-Up Switch

The three limitations above share a common root cause: today’s scale-up fabrics lack the in-network flexibility needed to support evolving use cases.

Why not extend NVLS? In principle, extending existing fixed-function NVSwitch could support the use cases we consider. However, this approach limits extensibility, as existing offloads rely on specialized GPU memory instructions. Supporting new switch operations may therefore require changes to the GPU instruction set or even the interconnect protocol. For example, DySHARP [61] proposes a new specialized switch architecture for MoE collective offloads but also requires new GPU memory instructions, packet format modifications, and architectural changes to GPUs. Thus, extending switch functionality can require upgrading the entire accelerator-interconnect stack.

We instead advocate a programmable switch that operates on ordinary memory transactions while preserving existing GPU instructions and interconnect protocols. Our goal is not to enable functionality that fixed-function switch fundamentally cannot support, but to decouple the evolution of in-network functionality from accelerator architectures. And the main benefit of programmability on the switch is that it reduces the cost and complexity of supporting new use cases without repeatedly redesigning the hardware, similar to the motivation behind SDN [32].

Design Requirements: The programming model for such a switch must satisfy four requirements:

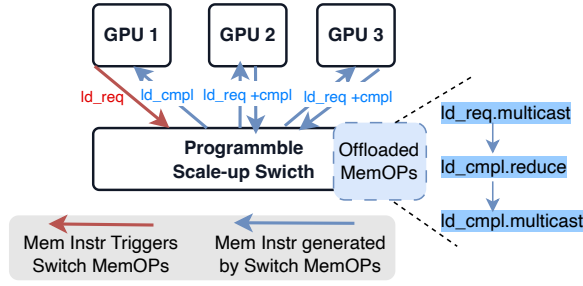


Figure 6: Proposed programmable scale-up switch.

1. **Compatibility with load/store protocols.** The switch must preserve endpoint-visible memory semantics, including request–completion dependencies, ordering constraints and memory access permissions. For example, every load issued by a GPU must eventually receive a valid completion with a matching tag [55].
2. **Expressive but bounded semantics.** The programming model must support the use cases in §2.2, yet keep its semantics bounded so that execution remains efficient and predictable.
3. **Line-rate throughput and bounded latency.** The switch must process transactions at the aggregate fabric line rate. Per-transaction work and latency must be bounded to avoid GPU stalls and excessive buffering.
4. **Verifiable correctness and performance.** A faulty switch program must not corrupt GPU memory or violate protocol or memory-protection rules, and a verifier should check this before installation (§5).

Why Not P4? Emerging proposals such as ESUN [41], UALoE [5], and SUE-T [42] carry memory transactions over Ethernet, potentially enabling P4-programmable switches in scale-up fabrics. However, P4’s programming model is a poor fit. P4 processes packets independently, whereas scale-up switches must preserve dependencies between load/store requests and completions. Expressing these cross-packet dependencies through per-packet abstractions complicates memory correctness verification and bounded execution. ML collectives also require stateful coordination across transactions, which is costly on RMT-based P4 switches due to limited stateful processing and packet-centric execution [17, 43]. Scale-up switches instead need a transaction-centric programming model.

Our Design: SUMO. We propose *SUMO*, a constrained, transaction-aware programming model built on distributed shared-memory operators. These operators process fine-grained load/store requests and completions, supporting only efficiently implementable and verifiable operations.

Fig. 6 illustrates the architecture. When a memory transaction arrives, its type triggers a programmer-defined chain of operators. Each operator may inspect or transform the transaction, replicate or route it, aggregate it with related

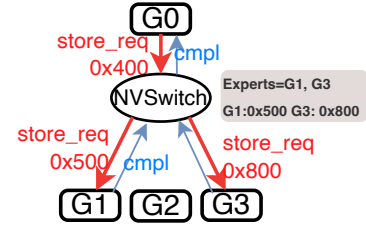


Figure 7: MoE token dispatch with SUMO.

transactions, generate new requests or completions, or update telemetry state. The switch executes the chain while preserving endpoint-visible load/store semantics and enforcing bounded resource usage.

4 SUMO Design and Use Cases

Table 3 summarizes the operators, which cover transaction replication, address translation, data reduction, explicit routing, and telemetry collection. We adopt a top-down approach to identify a minimal set of operators required by our target use cases. They apply to four classes of memory transactions, matching the UALink specification [55]: load requests (*load_req*) and data (*load_cmpl_data*), store requests with data (*store_req_data*) and completions (*store_cmpl*).

A programmer maps each transaction class to an operator chain, executed sequentially upon a matching transaction. Operators can modify transactions, emit new ones, wait for and combine multiple transactions, or export information to the control plane. Chains can span multiple passes: for example, *multicast(req)* returns handles to completions, whose operators execute only after the switch receives them. This model requires stateful processing and transaction generation, both partially supported by existing hardware, as NVSwitch’s in-network reduction and multicast demonstrate. We aim to expose and generalize these capabilities through a programmable interface.

Use Case 1: MoE Token Routing: SUMO offloads dynamic MoE token routing (Fig. 7) through in-switch address translation and dynamic multicast, which existing scale-up switches lack (§2.2.1).

```
trigger(store_req_data) {
    expert_ids[] = experts_lookup(store_req_data.addr);
    addrs[] = lookup_dst_gpu_addr(expert_ids[]);
    store_cmpls[] =
        store_req_data.multicast(addrs[]).send();
    // wait until all stores complete
    reduce(store_cmpls[]);
    store_req_data.cmpl().send();
}
```

The chain is triggered when the switch receives a GPU-issued *store_req_data*. The request address indexes an expert routing table, configured when the MoE model is deployed, that

Operator	Interface	Semantics	Applies to
send	$t' = t.send$	send t , if t is a req, return the handler of its completions.	All
multicast	$t.multicast(addr[])$	Replicate t to multiple destination addresses.	LR, SR
update_address	$t.update_address(new_addr)$	Update the transaction address.	LR, SR
reduce	$reduce(t[], \{f\})$	Wait until all $t[]$ are ready; combine using the reduction function f .	LC, SC
cmpl	$t.cmpl(\{data\})$	Generate completion for a request t .	LR, SR
telemetry_lat	$\ell = t.telemetry_lat()$	Measure request-to-completion latency.	LR, SR
count	$c.count(t)$	Increment a counter for matching transactions.	All
bind	$t.bind(port)$	Select the outbound switch port.	Outbound
ctrl_send	$ctrl_send(data)$	Export data or telemetry to the control plane.	All

Table 3: SUMO operators supported by the programmable scale-up switch. LR denotes a load request, SR a store request carrying data, LC a load completion carrying data, and SC a store completion. The symbol t denotes a memory transaction.

identifies the GPUs hosting the selected experts. For each destination GPU, the switch computes where the token should be written, based on the number of tokens previously routed to that GPU. The *multicast* operator then replicates the store request, with each replica writing to its destination address.

Each replica produces a completion, and the *reduce* operator waits until all of them have arrived. The switch then uses *cmpl* to generate a completion for the original store request and *send* to return it to the requesting GPU. The sender thus observes a single completed store, only after the token has reached every target expert.

Use Case 2: AllReduce Offload: Where NVLS splits AllReduce into two GPU-driven phases (§2.2.1), SUMO lets a single chain fuse load reduction with store multicast.

```

trigger(load_req) {
  dest_addrs[] = {load_req.addr, G2_addr, G3_addr};
  // Gather and reduce values from all participants.
  load_cmpls[] = load_req.multicast(dest_addrs[]).send();
  reduced_load_cmpl = reduce(load_cmpls[], +);
  // Construct a store
  store_req = new_store_req(reduced_load_cmpl.data);
  // Multicast the result to all GPUs.
  store_cmpls[] =
    store_req.multicast(dest_addrs[]).send();
  // Wait until all result stores complete.
  reduce(store_cmpls[]);
  // Complete the triggering load request.
  load_req.cmpl(reduced_load_cmpl.data).send()
}

```

The switch multicasts a GPU-issued *load_req* to all participants, reduces the returned data, and multicasts the result through store requests. Intermediate reduced shards never travel to GPUs, so per-GPU traffic drops to the minimum of S bytes sent and received. And because the switch both reduces and multicasts, GPUs no longer issue concurrent multimem instructions, which could relieve the GPU-side contention we measured and improve bandwidth utilization.

Use Case 3: In-Network Telemetry and Rerouting: SUMO can measure request-completion latency inside the fabric. For selected transactions, the switch timestamps the request, matches its completion, and records the latency with *telemetry_lat*. It counts slow transactions with *count* and exports only threshold violations through *ctrl_send*, enabling fine-grained diagnosis at low overhead.

These signals can also support grey failure recovery (§2.2.3): in multi-tier fabrics, the control plane can reweight *bind* port assignments to steer traffic away from degraded paths. Such rerouting must preserve memory ordering. For example, UALink’s non-strict mode [54, 55] requires ordering among requests sharing the same source, destination, virtual channel, and 256-byte aligned memory region.

Assuming two valid next-hop paths and this non-strict ordering model, the following example distributes memory requests across two ports with 2:1 hash-bucket weights:

```

// Quiesce and drain traffic before installation.
trigger(t) {
  key = (t.src, t.dst, t.vc, t.addr >> 8);
  bucket = hash(key) % 3;
  port = (bucket < 2) ? p0 : p1;
  t.bind(port).send();
}

```

Hashing address at 256-byte granularity ensures that requests targeting the same destination memory region follow the same port within a routing configuration. To safely install this new routing function, the control plane should quiesce affected traffic, drain outstanding transactions, and install the new routing configuration consistently across switches, following principles from consistent network updates [45]. This prevents dependent requests from being reordered across old and new routing configurations.

Use Case 4: Computation-communication Overlap: ML systems fuse computation and communication within a kernel, but each kernel may require push- or pull-based communication to match its memory access patterns [61]. For example, a fused AllGather–GEMM kernel pulls remote data through loads, whereas NVLS multicast only pushes data through stores. Programmable scale-up switches can achieve flexible combinations of memory operations (e.g., load/store) and collective operations (e.g., multicast/reduction), enabling communication to follow the memory access patterns of different fused kernels and facilitating finer-grained compute-communication overlap.

Feasibility analysis. Executing SUMO programs requires switch buffers to track transaction dependencies and store intermediate results. Buffer usage depends on the size and lifetime of intermediate variables. For example, in use case 2, completion generation retains original request metadata

until dependent operations finish, while reduction buffers partial results and metadata. Thus, switch SRAM limits concurrent stateful operations and dependency chain lengths. For example, each Blackwell GPU connects to 18 NVSwitches via 50 GB/s links. For a 72-port NVSwitch, assuming a 400 ns switch-to-memory round-trip latency, the aggregate bandwidth-delay product (BDP) is $72 \times 50 \text{ GB/s} \times 400 \text{ ns} = 1.44 \text{ MB}$, modest relative to modern switch SRAM capacity [23]. Moreover, operators such as completion generation retain only metadata rather than full payloads. Buffers can be released once their operations complete. For example, after reduction, the switch can free the accumulation buffer while retaining metadata for outstanding dependencies. The verifier should ensure state requirements fit within switch SRAM, while admission control bounds concurrent executions.

For packet generation, we envision an ASIC-based packet-template engine that constructs protocol-compliant requests and completions by populating predefined formats with fields such as addresses, transaction tags, and completion status, avoiding arbitrary packet construction overhead.

Future work includes exploring larger on-switch buffers (e.g., DRAM with prefetching), low-latency packet-processing architectures, and program optimizations to reduce peak state consumption.

5 Program Verification

SUMO programs explicitly encode request-completion dependencies, enabling verification before installation. The verifier builds on data-plane verification techniques [31, 51, 52, 56], but unlike prior work targeting packet correctness or cross-packet stateful properties [58], it checks memory transaction dependencies, ensuring generated requests, completions, and inter-request ordering comply with the scale-up protocol. Programs violating these requirements are rejected to prevent GPU memory corruption. Specifically, it checks:

1. **Request-completion correctness.** Requests and completion obligations are linear resources. Every generated request must use a valid tag and produce exactly one matching completion or follow a bounded error path.
2. **Memory protection.** Generated and translated addresses must remain within authorized GPU memory regions. Dynamic lookups require runtime bounds checks.
3. **Bounded state and buffering.** The verifier bounds outstanding transactions, concurrent operator instances, and switch-local state. For stateful operators such as *reduce*, it derives the buffering bound from the maximum response latency, concurrent group count, and timeout.
4. **Bounded performance.** Each operator chain must have bounded execution latency and sustain line-rate processing. Operations awaiting remote completions must define a bounded timeout or error path.

6 Discussion and Future Work

SUMO is a first step, not a finished design. Turning it into a practical substrate for software-defined scale-up fabrics raises open questions that seed a research agenda.

Architecture design. How should SUMO be mapped onto switch hardware while preserving line-rate throughput? Is an RMT-style pipeline [10, 11] sufficient? What hardware mechanisms can efficiently support collectives [20]?

Runtime integration. How should the compiler and runtime stack compile, install, schedule, update, and revoke SUMO programs? How should programmable switches integrate with communication frameworks such as DeepEP [14], NCCL [35], and NVSHMEM [40], and with collective synthesis systems [4, 49, 62]?

Stateful data structures and affinity. Efficient execution requires scalable in-switch structures, including hash and transaction tables [25, 26], telemetry sketches [60], and reduction buffers [29, 48]. These structures must support high concurrency under tight area, timing, and latency constraints. To avoid cross-switch state synchronization, SUMO should enforce per-chain switch affinity: all requests generated by a chain, and their completions, must traverse the same switch, keeping intermediate state local.

Memory ordering and cache coherence. SUMO programs may redirect transactions or delay completions, potentially violating fabric ordering guarantees. Under UALink’s non-strict mode, requests from the same source to the same destination targeting the same 256-byte aligned region must remain ordered, while others may be reordered [54]. SUMO preserves these guarantees through per-ordering-domain sequencing and buffering, with the verifier checking ordering dependencies. Most scale-up interconnects (e.g., UALink) lack cross-accelerator hardware cache coherence but support I/O coherence within destination nodes [54]. Thus, SUMO need not implement distributed cache coherence but must preserve the I/O-coherence semantics of ordinary memory accesses. SUMO assumes a software-dataplane co-design model in which GPU software knows which memory regions participate in switch-side computation, understands the resulting memory semantics, and performs the necessary synchronization and software cache management. Supporting transparent compatibility with arbitrary GPU applications remains an open challenge and is left as future work.

Acknowledgments

We thank the anonymous HotNets reviewers for their constructive feedback, which helped improve this paper. This work is supported by gifts from Google and NVIDIA, Lambda, and the NAIRR Pilot. Daehyeok Kim was supported by NSF Awards 2403026 and 2443203.

References

- [1] 2024. *NVIDIA Blackwell GB200 NVL72 Technical Overview*. Technical Report. NVIDIA Corporation. <https://www.nvidia.com/en-us/data-center/gb200-nvl72/> Official Technical Specification.
- [2] 2024. *NVIDIA NVLink and NVSwitch: The World's First High-Speed GPU Interconnect*. Technical Report. NVIDIA Corporation. <https://www.nvidia.com/en-us/data-center/nvlink/> Technical Whitepaper.
- [3] Advanced Micro Devices, Inc. 2024. XGMI Configuration — Virt-driv Documentation. https://instinct.docs.amd.com/projects/virt-driv/en/latest/userguides/XGMI_configuration.html. Accessed: 2026-02-03.
- [4] Advanced Micro Devices, Inc. (AMD). 2024. Radeon Collective Communication Library (RCCL). <https://github.com/ROCm/rccl> (2024).
- [5] AMD Networking. 2026. Resilient Scale-Up Networking Built for Production AI. <https://www.amd.com/en/blogs/2026/amd-helios-resilient-scale-up-networking-for-ai.html>
- [6] Daiyaan Arfeen, Dheevatsa Mudigere, Ankit More, Bhargava Gopireddy, Ahmet Inci, and Gregory R Ganger. 2025. Nonuniform-tensor-parallelism: Mitigating gpu failure impact for scaled-up llm training. *arXiv preprint arXiv:2504.06095* (2025).
- [7] Rohil Bhargava, Taylor Allison, and Harry Petty. 2026. NVIDIA Vera Rubin POD: Seven Chips, Five Rack-Scale Systems, One AI Supercomputer. NVIDIA Technical Blog. <https://developer.nvidia.com/blog/nvidia-vera-rubin-pod-seven-chips-five-rack-scale-systems-one-ai-supercomputer/> See section “NVIDIA Vera Rubin Ultra NVL576.” Accessed: 2026-07-16.
- [8] Paul Borrill. 2026. The Ghost in the Datacenter: Link Flapping, Topology Knowledge Failures, and the FITO Category Mistake. *arXiv preprint arXiv:2603.03736* (2026).
- [9] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.
- [10] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando A. Mujica, and Mark Horowitz. 2013. Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*.
- [11] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargatik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, et al. 2017. drmt: Disaggregated programmable switching. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 1–14.
- [12] Sirshak Das, Jason Sewall, Giuseppe Congiu, Pavel Shamis, and Gargi Prasad. 2025. Enhancing Communication Observability of AI Workloads with NCCL Inspector. NVIDIA Technical Blog. <https://developer.nvidia.com/blog/enhancing-communication-observability-of-ai-workloads-with-nccl-inspector/> Accessed: 2026-07-14.
- [13] Daniele De Sensi, Lorenzo Pichetti, Flavio Vella, Tiziano De Matteis, Zebin Ren, Luigi Fusco, Matteo Turisini, Daniele Cesarini, Kurt Lust, Animesh Trivedi, et al. 2024. Exploring gpu-to-gpu communication: Insights into supercomputer interconnects. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.
- [14] DeepSeek-AI et al. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. *arXiv:2501.12948 [cs.CL]* <https://arxiv.org/abs/2501.12948>
- [15] Ashraf Eassa, Alex Ishii, and Ryan Wells. 2022. Upgrading Multi-GPU Interconnectivity with the Third-Generation NVIDIA NVSwitch. NVIDIA Technical Blog. <https://developer.nvidia.com/blog/upgrading-multi-gpu-interconnectivity-with-the-third-generation-nvidia-nvswitch/>
- [16] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. 2018. Azure accelerated networking: {SmartNICs} in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 51–66.
- [17] Nadeen Gebara, Alberto Lerner, Mingran Yang, Minlan Yu, Paolo Costa, and Manya Ghobadi. 2020. Challenging the stateless quo of programmable switches. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*. 153–159.
- [18] Richard L Graham, Lion Levi, Devendar Burreddy, Gil Bloch, Gilad Shainer, David Cho, George Elias, Daniel Klein, Joshua Ladd, Ophir Maor, et al. 2020. Scalable hierarchical aggregation and reduction protocol (sharp) tm streaming-aggregation hardware design and evaluation. In *International Conference on High Performance Computing*. Springer, 41–59.
- [19] Diwakar Gupta and Sabastian Mugazambi. 2026. Inside the Eighth-Generation TPU: An Architecture Deep Dive. <https://cloud.google.com/blog/products/compute/tpu-8t-and-tpu-8i-technical-deep-dive>. Published April 22, 2026; accessed July 16, 2026.
- [20] Zhenhao He, Dario Korolija, Yu Zhu, Benjamin Ramhorst, Tristan Laan, Lucian Petrica, Michaela Blott, and Gustavo Alonso. 2024. {ACCL+}: an {FPGA-Based} collective engine for distributed applications. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 211–231.
- [21] Thomas Holterbach, Edgar Costa Molero, Maria Apostolaki, Alberto Dainotti, Stefano Vissicchio, and Laurent Vanbever. 2019. Blink: Fast connectivity recovery entirely in the data plane. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 161–176.
- [22] Changho Hwang, KyoungSoo Park, Ran Shu, Xinyuan Qu, Peng Cheng, and Yongqiang Xiong. 2023. ARK: GPU-driven Code Execution for Distributed Deep Learning. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 87–101. <https://www.usenix.org/conference/nsdi23/presentation/hwang>
- [23] Intel. 2023. Intel Tofino Programmable Ethernet Switch ASIC Family. In *Technical Product Brief*. <https://www.intel.com/content/www/us/en/products/details/network-io/data-center-ethernet-switches/tofino-series.html>
- [24] jiachen17713. [n.d.]. *Unnecessary Traffic in Load-Reduce and Store Operations on a Multicast Object*. NVIDIA Developer Forums. <https://forums.developer.nvidia.com/t/unnecessary-traffic-in-load-reduce-and-store-operations-on-a-multicast-object/356726> CUDA Programming and Performance forum post.
- [25] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. NetChain: Scale-Free Sub-RTT Coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 35–49.
- [26] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*. 121–136. <https://doi.org/10.1145/3132747.3132764>
- [27] Norman P. Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Prabhu, et al. 2023. TPUv4: An Ocs-Enabled 6e82 Distributed Deep Learning System. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23)*. ACM, 1–14. <https://doi.org/10.1145/3579371.3589357>

- [28] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, Lawrence J Wobker, et al. 2015. In-band network telemetry via programmable dataplanes. In *ACM SIGCOMM*, Vol. 15. 1–2.
- [29] ChonLam Lao, Yanfang Le, Kshitij 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)*. USENIX Association, 741–761.
- [30] Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*. PMLR, 19274–19286.
- [31] Jed Liu et al. 2018. p4v: Practical Verification for Programmable Data Planes. In *Proceedings of ACM SIGCOMM*. 490–503. <https://doi.org/10.1145/3230543.3230582>
- [32] Nick McKeown. 2009. Software-defined networking. *INFOCOM keynote talk 17*, 2 (2009), 30–32.
- [33] NVIDIA. 2022. Introducing NVIDIA HGX H100: An Accelerated Server Platform for AI and High-Performance Computing. <https://developer.nvidia.com/blog/introducing-nvidia-hgx-h100-an-accelerated-server-platform-for-ai-and-high-performance-computing/>. Accessed: 2025-11-25.
- [34] NVIDIA Corporation. 2024. NCCL-Inspector: A Performance Analysis Tool for NCCL. <https://github.com/NVIDIA/nccl-inspector>. Tool for analyzing collective communication performance.
- [35] NVIDIA Corporation. 2024. NVIDIA Collective Communication Library (NCCL). <https://github.com/NVIDIA/nccl> (2024).
- [36] NVIDIA Corporation. 2024. NVIDIA Nsight Systems. <https://developer.nvidia.com/nsight-systems>. System-wide performance analysis tool.
- [37] NVIDIA Corporation. 2024. CUDA C++ Programming Guide: Programming Systems with Multiple GPUs. <https://docs.nvidia.com/cuda/cuda-programming-guide/03-advanced/multi-gpu-systems.html>. Accessed July 16, 2026.
- [38] NVIDIA Corporation. 2026. NVIDIA Collective Communication Library User Guide: NVLink SHARP (NVLS). NCCL 2.30.7 Documentation. <https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/env.html#nccl-nvls-enable> Section: NCCL_NVLS_ENABLE.
- [39] NVIDIA Corporation. 2026. *NVIDIA Nsight Systems User Guide*. NVIDIA Corporation. <https://docs.nvidia.com/nsight-systems/UserGuide/index.html> Accessed: 2026-07-14.
- [40] NVIDIA Corporation. 2026. NVSHMEM Documentation. <https://docs.nvidia.com/nvshmem/api/index.html>. Accessed July 16, 2026.
- [41] Open Compute Project. 2026. *OCF ESUN: Network Operator Requirements, Base Specification, Rev. 1.0*. Technical Report. Open Compute Project. <https://www.opencompute.org/documents/ocp-esun-network-operator-requirements-base-specification-rev-1-0-final-pdf/>
- [42] Open Compute Project. 2026. *SUE-T: Base Specification, Version 1.0.0*. Technical Report. Open Compute Project. <https://www.opencompute.org/wp-content/uploads/2026/10/Scale-Up-Ethernet-Transport-SUE-T-Base-Specification-v1.0.0.pdf>
- [43] P4.org Architecture Working Group. 2026. P4₁₆ Portable Switch Architecture (PSA). P4.org Specification. <https://p4lang.github.io/p4-spec/docs/PSA.pdf> Version 1.2.
- [44] Jeff Rasley, Brent Stephens, Colin Dixon, Eric Rozner, Wes Felter, Kanak Agarwal, John Carter, and Rodrigo Fonseca. 2014. Planck: Millisecond-scale monitoring and control for commodity networks. *ACM SIGCOMM Computer Communication Review* 44, 4 (2014), 407–418.
- [45] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. 2012. Abstractions for network update. *ACM SIGCOMM Computer Communication Review* 42, 4 (2012), 323–334.
- [46] Junyeol Ryu, Ming Liu, and Matthew D. Sinclair. 2026. Understanding and Profiling the Accelerator Chiplet Network Using PingPoint. In *Proceedings of the ACM SIGCOMM 2026 Conference*.
- [47] 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)*. 785–808.
- [48] 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)*. USENIX Association, 785–808.
- [49] Aashaka Shah, Vijay Chidambaram, Meghan Cowan, Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, Jacob Nelson, Olli Saarikivi, and Rachee Singh. 2023. {TACCL}: Guiding collective algorithm synthesis using communication sketches. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 593–612.
- [50] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538* (2017).
- [51] Anirudh Sivaraman et al. 2016. Packet Transactions: High-Level Programming for Line-Rate Switches. In *Proceedings of ACM SIGCOMM*. 15–28. <https://doi.org/10.1145/2934872.2934900>
- [52] Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. 2018. Debugging P4 Programs with Vera. In *Proceedings of ACM SIGCOMM*. 518–532. <https://doi.org/10.1145/3230543.3230548>
- [53] telala. 2024. Allgather Performance Using NVLS Is Poor. GitHub issue, NVIDIA/nccl, Issue 1506. <https://github.com/NVIDIA/nccl/issues/1506> Opened November 7, 2024; accessed July 16, 2026.
- [54] UALink Consortium. 2025. *UALink 200G 1.0 Specification*. UALink Consortium. <https://ualinkconsortium.org/specification/> Sec. 2.7.9.
- [55] UALink Consortium. 2025. UALink — open standard for AI accelerator interconnect. <https://ualinkconsortium.org/>. Accessed: 2025-11-25.
- [56] Vasco T. Vasconcelos. 2009. Session Types for Linear Multithreaded Functional Programming. In *Proceedings of PPDP*. 1–6. <https://doi.org/10.1145/1599410.1599411>
- [57] visualxu. 2026. [RFE]: NVLS/Symmetric Multimem Algorithm Optimization. GitHub issue, NVIDIA/nccl, Issue 2072. <https://github.com/NVIDIA/nccl/issues/2072> Opened March 26, 2026; accessed July 16, 2026.
- [58] Qinshi Wang, Mengying Pan, Shengyi Wang, Ryan Doenges, Lennart Beringer, and Andrew W. Appel. 2023. Foundational Verification of Stateful P4 Packet Processing. In *Proceedings of ITP*. 32:1–32:20. <https://doi.org/10.4230/LIPIcs.ITP.2023.32>
- [59] Tianyuan Wu, Wei Wang, Yinghao Yu, Siran Yang, Wenchao Wu, Qinkai Duan, Guodong Yang, Jiamang Wang, Lin Qu, and Liping Zhang. 2025. GREYHOUND: Hunting Fail-Slows in Hybrid-Parallel Training at Scale. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. USENIX Association. <https://www.usenix.org/conference/atc25/presentation/wu-tianyuan>
- [60] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. 2018. Elastic Sketch: Adaptive and Fast Network-Wide Measurements. In *Proceedings of the ACM SIGCOMM 2018 Conference*. 561–575. <https://doi.org/10.1145/3230543.3230544>
- [61] Qijun Zhang, Chen Zhang, Zhuoshan Zhou, Haibo Wang, Zhe Zhou, Zhipeng Tu, Guangyu Sun, Zhiyao Xie, Yijia Diao, Zhigang Ji, et al. 2026. Accelerating MoE with Dynamic In-Switch Computing on Multi-GPUs. *arXiv preprint arXiv:2605.05607* (2026).

- [62] Liangyu Zhao, Saeed Maleki, Yuanhong Wang, Zezhou Wang, Ziyue Yang, Hossein Pourreza, and Arvind Krishnamurthy. 2026. {ForestColl}:{Throughput-Optimal} Collective Communications on

Heterogeneous Network Fabrics. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*. 2067–2093.