Programmable Scale-Up Switches for Software-Defined GPU Fabrics

Shouxu Lin
Cornell University

Ran Shu
Microsoft Research

Daehyeok Kim
UT Austin

Jiaxin Lin
Cornell University

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 *MemOps*, a transaction-aware programming model that composes bounded, verifiable operators over load/store transactions to express offload, routing, and telemetry.

1 INTRODUCTION

Modern AI training and inference now span hundreds to thousands of accelerators, and their communication demands increasingly exceed what conventional Ethernet-based networks can provide. Accelerator systems are therefore adopting tightly coupled *scale-up* fabrics, such as NVIDIA NVLink [2], the UALink standard [46], and the Inter-Chip Interconnect (ICI) used in Google TPUs [17, 24]. Once confined to links among a few accelerators inside a single server, these fabrics have grown into rack- and cluster-scale networks of hundreds of accelerators [1, 6]. Unlike Ethernet, a scale-up fabric is integrated directly with accelerator memory systems, so accelerators communicate through fine-grained load/store memory transactions rather than packets [46]. These accelerator-initiated memory operations bypass the host CPU, OS, NIC, and much of the traditional protocol stack, delivering ultra-high bandwidth and low latency [12, 20].

These fabrics deliver high performance, but today they are also fixed-function and opaque by design. Accelerators inject load and store transactions directly into the fabric, where they traverse links and switches with no software interposition, and no software layer can match, transform, replicate, route, or observe these transactions. We call this the *programmability gap*. This gap is a design choice, not an inherent property of load/store fabrics. Nothing about memory semantics forbids in-fabric programmability; rather, making a fabric programmable *while preserving* those semantics and performance is hard, which is why current fabrics omit it. Left unaddressed, the gap limits communication offload, observability, and resilience in three concrete ways:

(1) **Restricted communication offload:** Existing scale-up switches like NVSwitch provide collective offload primitives, but these primitives are fixed-function and support only predetermined, static communication patterns [35]. As we show in §2.2.1, this rigidity can limit the performance of collective offloads and cannot express the 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 provide collective traces and kernel timelines [31, 33, 36]. These mechanisms can detect communication slowdown, but provide limited insight into whether it was caused by congestion, queuing, path imbalance, or a slow memory endpoint. As we show in §2.2.2, the lack of telemetry makes root-cause diagnosis hard.

(3) **Inflexible routing and recovery:** Scale-up topologies often provide substantial path diversity [1, 6], but expose little software 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 calls for making scale-up fabrics *software-defined*, much as programmable switches did for Ethernet over the past decade. In the Ethernet world, programmability became the common substrate behind software-defined networking [8, 15, 29], in-network computation [16, 40], and flexible network management [19, 25, 38].

Scale-up fabrics now need the same substrate to support flexible offload, observability, and management. We focus on the switch, which sits on the communication datapath and observes traffic between many accelerators, as the natural site for this functionality (§3).

Realizing such a switch is hard, and it cannot simply reuse the packet match-action model of P4 and RMT [9], built for Ethernet, because scale-up fabrics make the problem distinct: First, the switch operates on memory transactions, so the programming model must explicitly capture load/store semantics. Second, its interface must be expressive enough for emerging GPU communication patterns yet constrained enough to run at line rate. Third, correctness requirements are stricter, since a faulty program could violate or corrupt GPU memory.

In this paper, we take a first step toward software-defined scale-up fabrics with *MemOps*, bounded transaction-level operators that run on scale-up switches. MemOps are typed by load/store requests and completions, preserving request–completion semantics. Programmers compose a small set of bounded operators, spanning replication, address translation, reduction, routing, and telemetry, into transaction-triggered chains. Before installation, each program is verified for memory correctness, bounded resource usage, and predictable line-rate execution. We show that MemOps can express efficient AllReduce, dynamic MoE token routing, and fine-grained in-network telemetry. MemOps represent one point in a broader design space for software-defined scale-up fabrics, and we conclude by outlining open research questions.

2 BACKGROUND AND MOTIVATION

2.1 Scale-up Networks at Scale

A scale-up network connects multiple accelerators into a unified logical compute unit through dedicated memory interconnects. These fabrics use accelerator links (*e.g.,* NVLink [2], xGMI [3], and UALink [46]) and accelerator switches (*e.g.,* NVSwitch [14, 35] and Ultra Accelerator Switch [46]) to enable communication among accelerator memories. These links carry cache-line-level memory transactions such as load requests, store requests, and their completions [46].

Network Architecture: Early scale-up deployments were largely confined to a single server, typically connecting 4–8 accelerators through a local switched fabric [30]. More recently, scale-up connectivity has expanded beyond a single server to cluster-scale configurations [1, 6, 24, 46].

For example, as shown in Fig. 1, the NVIDIA GB200 NVL72 [1] forms a single-stage, non-blocking scale-up fabric in which 72 GPUs are connected to 18 NVSwitches. Each GPU contributes one NVLink connection to each switch, and the 18 parallel switch planes collectively provide all-to-all connectivity across the rack. The larger NVL576 topology [6] extends this design to 576 GPUs using a two-tier NVSwitch fabric. Beyond NVIDIA architectures, Google TPU Pods [17, 24] use hierarchical ICI and optical switches to interconnect thousands of accelerators. UALink also supports switched fabrics that provide all-to-all connectivity among 1024 accelerators [46].

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

Table 1: NVLS multimem primitives [35].

Scale-up Switch and Offloads: Scale-up switches can offload communication from GPUs. We use NVLink SHARP (NVLS) [14, 35] as a state-of-the-art example.

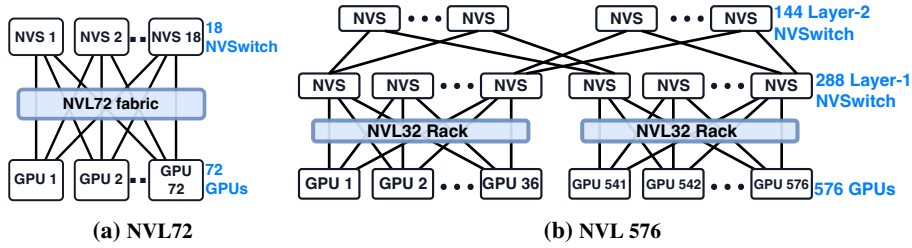
As shown in Table 1, NVLS exposes switch-offload primitives through *multimem* instructions. A *multimem* address maps symmetric memory locations across multiple GPUs to one logical address, allowing a GPU to use a single memory instruction to invoke a multicast or reduction over multiple GPUs' memory through the NVLink switch fabric. Fig. 2 shows how to offload communication using *multimem.st_multicast*. Traditionally, the sender GPU G_0 must store the same data separately to each of the N receivers. With *multimem*, G_0 issues a single store operation, and the switch replicates the data to all receivers. Similarly, *ld_reduce* performs a many-to-one reduction: the switch replicates one load request to multiple GPUs, reduces the data across GPUs, and sends the aggregated result to the requesting GPU. These primitives reduce GPU-injected traffic and the GPU cycles required for replication and reduction.

2.2 Programmability Gap

Scale-up fabrics achieve high performance through hardware-native load/store semantics that avoid much of the software-mediation overhead. However, current fabrics remain fixed-function and opaque: software cannot route or observe memory transactions in flight, and switch offloads expose only limited, predefined functionality. We call this the *programmability gap*. In this section, we show how this gap limits communication offload, observability, and resilience.

2.2.1 Restricted Switch Offload. The first limitation is that current scale-up switch offloads expose only a narrow set of primitives, providing limited performance benefits to collective and MoE workloads.

Slow Collective Performance: We evaluate NVLS-offloaded AllReduce on a GB200 NVL72 system, whose topology is shown in Fig. 1a. As shown in Fig. 3, NVLS only delivers modest gains over ring AllReduce. NVLS provides no benefit with four or fewer GPUs; even with 8–16 GPUs, it achieves only 1.07×–1.15× higher algorithm bandwidth, reaching just



(a) NVL72 (b) NVL 576

Figure 1: Scale-up network architecture.

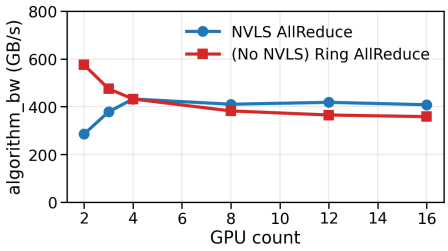
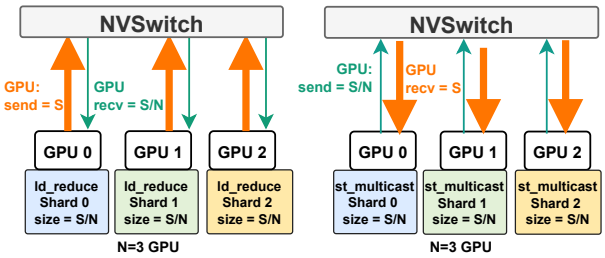


Figure 3: NVLS benchmark in NVL72 GB200 cluster.



(a) Load-reduce phase (b) Store-multicast phase

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

44.4% of the 900 GB/s NVLink bandwidth. This limited benefit is consistent with prior observations [21, 45, 48, 52].

The reason is that NVLS lacks a native in-switch AllReduce primitive and instead realizes AllReduce using two sequential primitives: *ld_reduce* (ReduceScatter) followed by *st_multicast* (AllGather). During each phase, however, each GPU sends and receives asymmetric amounts of data, which under-utilizes the bidirectional link bandwidth.

Fig. 4 illustrates this process. In the first phase, the input data of size S is partitioned into N equal-sized shards. For each shard, its destination GPU issues *ld_reduce*, causing the NVSwitch to reduce the corresponding shards across all GPUs and return the result to that GPU. Thus, each GPU sends S bytes and receives one $\frac{S}{N}$-byte reduced shard. In the second phase, each GPU uses *st_multicast* to send its local reduced shard to the NVSwitch, which distributes it to all GPUs. Thus, each GPU sends $\frac{S}{N}$ bytes and receives S bytes.

The key limitation is that communication is directionally imbalanced in each phase. The first phase is dominated by GPU-to-switch traffic, whereas the second is dominated by

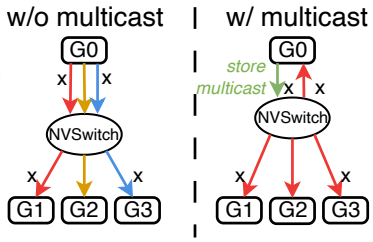


Figure 2: Operation of *st_multicast*.

switch-to-GPU traffic. Since each phase is bottlenecked by its heavier direction, the two directions of the full-duplex GPU–switch link are not fully utilized. Let B denote the per-direction link bandwidth. Each phase therefore takes approximately $\frac{S}{B}$, yielding a total time of $\frac{2 \cdot S}{B}$ and an algorithm bandwidth of $\frac{S}{\frac{2S}{B}} = \frac{B}{2}$, which matches the result in Fig. 3.

In contrast, an ideal in-switch AllReduce would integrate reduction and multicast into a single streaming pipeline. As each data block arrives at the switch, it is reduced and immediately multicast back to all GPUs, fully utilizing both directions of the full-duplex link. This would achieve an algorithm bandwidth of B. Unfortunately, the primitives exposed by the current NVLS interface (Table 1) cannot implement such a design because of its semantic constraints [48].

Static Multimem Cannot Support Dynamic MoE Pattern: Beyond the suboptimal performance of conventional collectives, the current interface lacks the flexibility required by emerging communication workloads. Applications such as MoE [43] and speculative decoding [27] exhibit dynamic and asymmetric communication patterns, whereas the current *multimem* interface assumes static destination sets and symmetric addresses across GPUs, *i.e.,* it can operate only on memory locations at the same offset on different GPUs. This mismatch makes the interface ill-suited to such workloads and can cause substantial performance degradation.

MoE dispatch is a representative example. Multicast can send the same token to multiple experts located on different GPUs, but the destination set varies across tokens according to expert placement. As shown in Fig. 5, tokens 0 and 1 are sent to $\{G_0, G_1\}$, whereas token 2 is sent to $\{G_1, G_2\}$. Moreover, each GPU's receive buffer is typically partitioned by source GPU: tokens from G_0 occupy one contiguous segment, followed by those from G_1, G_2, and so on. Because each GPU receives different numbers of tokens from different senders, the same token may be placed at different offsets in the receive buffers of its destination GPUs. For example, token 2 occupies slots 0 and 3 in the receive buffers of G_1 and G_2, respectively. The NVLS interface cannot directly express either these dynamic destination sets or asymmetric offsets. Implementing such communication with the existing multicast primitive therefore introduces large redundant traffic [51].

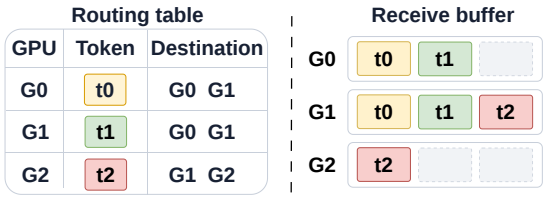


Figure 5: MoE token dispatch.

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 with 450 GB/s NVLink bandwidth.

2.2.2 Limited Fabric Observability. Scale-up memory transactions are largely invisible to software instrumentation, and developers cannot insert custom probes to monitor or trace individual memory transactions. Applications can observe only kernel or collective completion times and aggregate switch counters, while the fabric's internal behavior, such as per-link traffic, request latency, backpressure, and switch-level queueing, remains opaque.

This opacity complicates performance diagnosis. Suppose the observed bandwidth between two GPUs drops from 400 GB/s to 200 GB/s. End-to-end measurements using GPU probe kernels [39], application profilers such as Nsight Systems [36], or collective-level traces such as NCCL Inspector [11, 49] can detect the degradation, but cannot directly identify its root 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.

As shown in Table 2, we reproduce three of these causes and show that they produce similar end-to-end slowdowns.

For case (1), we co-locate a communication kernel transferring data from G_0 to G_1 with a computation kernel on G_0. Their combined thread-block count exceeds the GPU's maximum concurrent capacity, *i.e.,* the number of SMs multiplied by the maximum resident blocks per SM. The two kernels therefore contend for SM residency and resources such as registers and shared memory. As a result, the communication bandwidth drops 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 data to a single receiver GPU. The transfers contend for the switch-to-receiver downlink, reducing per-transfer bandwidth by 66.9%.

These results show that without fine-grained, lower-layer telemetry from the scale-up fabric, pinpointing the origin of the slowdown is difficult.

2.2.3 Inflexible Routing without Resilience. Scale-up fabrics provide substantial physical path diversity, but expose limited application-level control over how fine-grained memory requests are mapped onto those paths. For example, in an NVL72 system, each GPU connects to the NVSwitch fabric through 18 NVLinks, one to each of the 18 NVSwitch ASICs, providing multiple switch-mediated routes between GPU pairs [1]. CUDA peer-memory access, however, exposes a remote GPU's memory as an address: kernels issue ordinary loads and stores to that address, but cannot select or weight the physical route used by an individual request [34].

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 [5, 7]. 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. Since kernels cannot map individual memory requests to physical paths or dynamically adjust the traffic split as link conditions change, such rebalancing is impossible.

3 PROGRAMMABLE SCALE-UP SWITCH

The three limitations in the previous section share a common root cause: today's scale-up fabrics provide no way to program how memory transactions are processed in flight. Realizing the software-defined fabric we envision therefore begins with making the switch programmable. Programmability could be introduced at several points in the datapath, including the GPUs, the links, and the switches. We focus on the switch because it lies directly on the communication datapath, holds a global view of traffic from many accelerators, and can observe, transform, aggregate, and route distributed memory transactions without modifying GPU architectures.

Design Requirements: A programmable scale-up switch's programming model must therefore satisfy four requirements:

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 [46].
2. **Expressive but bounded semantics.** The programming model must support use cases described in §2.2. At the

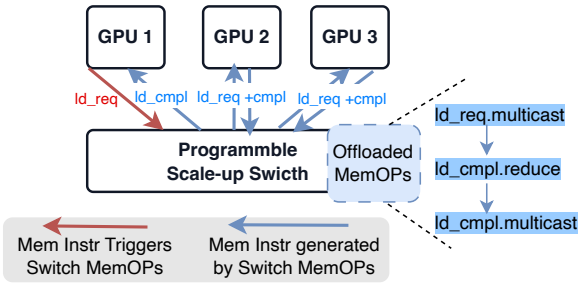


Figure 6: Proposed scale-up switches.

same time, the programming semantics should remain bounded to enable efficient and predictable execution.

3. **Line-rate throughput and bounded latency.** The switch must process transactions at the aggregate fabric line rate. Per-transaction work and execution 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 memory protocol and memory-protection rules. Before installation, a verifier should ensure that each program terminates, uses bounded resources, accesses only authorized address ranges, generates protocol-compliant transactions, and sustains line-rate execution with predictable latency.

Why Not P4? These requirements make a traditional P4-style programming model a poor fit. P4 processes packets independently, whereas scale-up fabrics must preserve dependencies across load/store requests and completions. P4's general per-packet abstractions therefore do not naturally represent dependent memory transactions, which complicates verification of memory correctness and bounded execution. Moreover, packet-level actions poorly express ML patterns such as reduction that coordinate many memory transactions.

Our Design: Programmable Switch with MemOps. We propose a constrained, transaction-aware programming architecture based on distributed shared-memory operators, which we call *MemOps*. MemOps execute directly on fine-grained load and store requests and completions, while exposing only operations that can be efficiently implemented and verified.

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

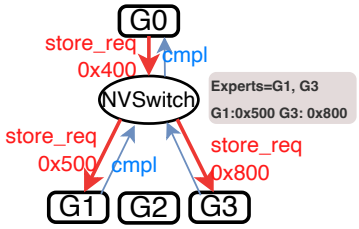


Figure 7: MoE token dispatch with MemOps.

4 MEMOPS DESIGN AND USE CASES

As summarized in Table 3, the proposed MemOp set includes operators for transaction replication, address translation, data reduction, explicit routing, and telemetry collection. MemOps may be applied to four classes of memory transactions: load requests (*load_req*), load completions carrying data (*load_cmpl_data*), store requests carrying data (*store_req_data*), and store completions (*store_cmpl*). These transaction classes match the UALink specification [46].

A programmer associates each transaction class with a chain of MemOps. The switch starts executing the chain sequentially when a matching transaction arrives. A MemOp may modify the current transaction, emit one or more new transactions, wait for and combine multiple transactions, or export selected information to the control plane. A MemOp chain can execute across multiple passes. For example, *multicast(req)* returns handles to the resulting completions, and the MemOps that process those completions run only once the switch receives them. Next, we walk through three use cases that MemOps express: MoE token routing, AllReduce, and in-network telemetry.

Use Case 1: MoE Token Routing: We now show how MemOps can offload dynamic MoE token routing; Fig. 7 shows the workflow. MemOps overcome the limitation of existing scale-up switches (§2.2.1) using in-switch address translation and dynamic multicast.

```
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 on-switch computation 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 identifies the GPUs hosting the selected experts. For each destination GPU, the switch then computes the address where the token should be written, based on the number of tokens previously routed to that GPU. Then, the *multicast* MemOp replicates the original store request,

MemOp	Interface	Semantics	Applies to
`send`	$\mathbf{t}' = \mathbf{t}$.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`	$\ell = \mathbf{t}$.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: MemOps 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.

with each replica writing to the corresponding destination address.

Each transmitted store request produces a completion. The *reduce* MemOp waits until all of these completions 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. Thus, the host observes a single completed store only after the token has been delivered to target experts.

Use Case 2: AllReduce Offload: We now show an AllReduce implementation. Here, MemOps overcome the limitation of existing scale-up switches by letting programmers write a chain that fuses 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 load data, and multicasts the result through store requests. After all stores complete, it returns a load completion to the requester. This gather–reduce–broadcast executes within one MemOp chain and can be pipelined across cache lines for concurrent processing.

Use Case 3: In-Network Telemetry. MemOps can measure request–completion latency. For selected transactions, the switch timestamps the request, matches its completion, and records the latency using *telemetry_lat*. It can aggregate slow transactions with *count* and export only threshold violations through *control_plane_send*, enabling fine-grained diagnosis with low telemetry overhead.

5 PROGRAM VERIFICATION

Before installation, each MemOp program must be checked for protocol correctness, memory safety, bounded resource usage, and predictable performance. The verifier combines linear types, SMT-based safety checks, and a target-aware resource model, building on techniques from session types and programmable data-plane verification [28, 44, 47]. MemOps are amenable to compositional verification because each operator explicitly exposes its transaction types and dependencies. The verifier constructs a transaction graph and 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 MemOp 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

MemOps are a first step, not a finished design. Turning them into a practical substrate for software-defined scale-up fabrics raises open questions that seed a broader research agenda.

Architecture design. How should MemOps be mapped onto switch hardware while preserving line-rate throughput? Is an RMT-style pipeline [9, 10] sufficient? What hardware mechanisms can efficiently support collective operations [18]?

Runtime integration. How should the compiler and runtime stack compile, install, schedule, update, and revoke MemOp programs? How should programmable switches integrate with communication frameworks such as DeepEP [13], NCCL [32], and NVSHMEM [37], as well as collective synthesis systems [4, 42, 52].

Stateful data structures. Efficient MemOp execution requires scalable in-switch structures, including hash and transaction tables [22, 23], telemetry sketches [50], and reduction

buffers [26, 41]. These structures must support high concurrency under tight area, timing, and latency constraints.

Beyond programmable switches. Switch programmability represents only one point in the design space. Future work should explore complementary programmable interfaces in accelerators, links, and memory controllers, enabling an end-to-end software-defined scale-up fabric.

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-drv Documentation. https://instinct.docs.amd.com/projects/virt-drv/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] 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).

[6] 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.

[7] Paul Borrill. 2026. The Ghost in the Datacenter: Link Flapping, Topology Knowledge Failures, and the FITO Category Mistake. *arXiv preprint arXiv:2603.03736* (2026).

[8] 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.

[9] 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)*.

[10] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, 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.

[11] 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.

[12] 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.

[13] 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

[14] 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/

[15] 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.

[16] Richard L Graham, Lion Levi, Devendar Burredy, 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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

[21] 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.

[22] 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.

[23] 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

[24] 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

[25] 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.

[26] 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)*. USENIX Association, 741–761.

[27] Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*. PMLR, 19274–19286.

[28] 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

[29] Nick McKeown. 2009. Software-defined networking. *INFOCOM keynote talk* 17, 2 (2009), 30–32.

[30] 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.

[31] NVIDIA Corporation. 2024. NCCL-Inspector: A Performance Analysis Tool for NCCL. https://github.com/NVIDIA/nccl-inspector. Tool for analyzing collective communication performance.

[32] NVIDIA Corporation. 2024. NVIDIA Collective Communication Library (NCCL). *https://github.com/NVIDIA/nccl* (2024).

[33] NVIDIA Corporation. 2024. NVIDIA Nsight Systems. https://developer.nvidia.com/nsight-systems. System-wide performance analysis tool.

[34] NVIDIA Corporation. 2026. 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.

[35] 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.

[36] NVIDIA Corporation. 2026. *NVIDIA Nsight Systems User Guide*. NVIDIA Corporation. https://docs.nvidia.com/nsight-systems/UserGuide/index.html Accessed: 2026-07-14.

[37] NVIDIA Corporation. 2026. NVSHMEM Documentation. https://docs.nvidia.com/nvshmem/api/index.html. Accessed July 16, 2026.

[38] 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.

[39] 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*.

[40] 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.

[41] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. 2021. Scaling Distributed Machine Learning with In-Network Aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 785–808.

[42] 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.

[43] 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).

[44] 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

[45] 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.

[46] UALink Consortium. 2025. UALink — open standard for AI accelerator interconnect. https://ualinkconsortium.org/. Accessed: 2025-11-25.

[47] Vasco T. Vasconcelos. 2009. Session Types for Linear Multithreaded Functional Programming. In *Proceedings of PPDP*. 1–6. https://doi.org/10.1145/1599410.1599411

[48] 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.

[49] 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

[50] 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

[51] 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).

[52] 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.