Scale-up networking research: the next frontier

Andrei Stan[★◇*], Alex Pantucu[★], William X. Zheng[○], Zhongjie Chen[□], Ziming Mao[○], Rishabh Iyer[○], Ion Stoica[○], Costin Raiciu[●★]

[★]*Politehnica of Bucharest* [●]*Broadcom* [○]*UC Berkeley* [□]*Microsoft Research* [◇]*Jump Trading*

ABSTRACT

Scale-up networks are typically single-tier multi-plane networks connecting GPUs in the same node (e.g. NVL 72). Despite being central to scaling AI applications, they are both understudied and poorly understood. We take a first look at proposed standards for scale-up networks and measure the behaviour of NVLink to understand the performance implications of existing and future transport designs for scale-up. In contrast to scale-out transport, scale-up transports are heavily constrained by chip area which reduces the functionality of the scale-up transport to the bare minimum. We experimentally show that both load balancing and flow control can badly affect performance in scale-up domains, and discuss the extent to which these issues may be addressed in software instead. We conclude with open research questions for scale-up networking research.

CCS CONCEPTS

• **Networks** → **Network architectures**; **Network performance evaluation**.

KEYWORDS

GPU Scale-Up Networks, Congestion Control, Latency Measurement

ACM Reference Format:

Andrei Stan, Alex Pantucu, William X. Zheng, Zhongjie Chen, Ziming Mao, Rishabh Iyer, Ion Stoica, and Costin Raiciu. 2026. Scale-up networking research: the next frontier. 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, 8 pages. https://doi.org/10.1145/3847290.3859937

[*]The research and writing for this paper were completed prior to the author's affiliation with Jump Trading.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

HotNets '26, November 16-17, 2026, Salt Lake City, UT, USA

© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-3029-0/2026/11
https://doi.org/10.1145/3847290.3859937

1 INTRODUCTION

Training frontier AI models requires datacenters hosting hundreds of thousands of GPUs [7]. All these GPUs must work together to train and then serve large models, making the network the critical element to AI scaling.

Modern GPUs perform data movement through two distinct networks. The scale-out network connects all XPUs in the datacenter at speeds of 400-800Gbps per XPU through RDMA-based protocols initially over Infiniband and more recently over Ethernet networks (e.g. RoCEv2, MRC [2] or UET [5, 9]). Similar to cloud networks, scale-out network topologies are multi-tier Clos as they aim to provide a fully provisioned interconnect to hundreds of thousands of GPUs.

The scale-up network connects the XPUs in the same node at higher per XPU speeds (in excess of 3.2Tbps). In contrast to the scale-out network where the verbs API is used to move data, scale-up networks offer a unified memory abstraction across the GPUs in the same node and thus data transfer happens implicitly via memory access (e.g. load or store instructions, or DMA transfers). The size of the scale-up domain has been increasing from 8 GPUs for H100 to 72 GPUs for the current NVL72 hardware, and is expected to grow to 576 in the next generation.

There are two key differences between scale-up and scale-out networks beyond the one-order-of-magnitude difference in throughput and different GPU count. Latency requirements are much stricter for scale-up networks, thus mesh or single-tier topologies are preferred over multi-tier Clos. Secondly, the scale-up network adapters are on the same die as the XPU, as opposed to the scale-out NICs, which are accessible via PCIe. On-die adaptors reduce latency and power consumption (e.g., no PCIe, among others) but further restrict the area available to implement the transport.

Scale-out networking has been the focus of networking research and practice for years, which highlighted the issues due to ECMP single-path flow collisions[1, 8, 13, 18] and head-of-line blocking due to lossless operation in RoCEv2 [22]. As a result, multipath protocols (e.g. host or switch-based spraying) and best effort operation (e.g., selective retransmission, packet trimming) have recently been standardized and hardware is already available from major vendors (MRC [2] and UET [5]).

In contrast, scale-up networking has received little attention from the networking research community, despite its key role in scaling AI workloads; its implications in terms of

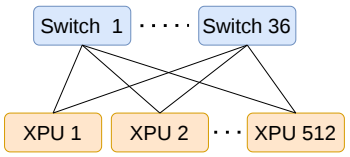


Figure 1: Single-tier multi-plane scale-up topology.

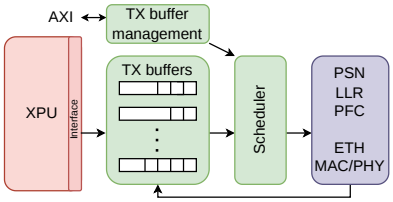


Figure 2: XPU integration for SUE [6].

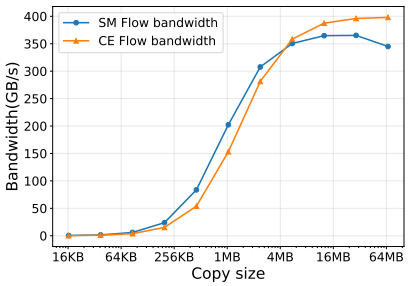


Figure 3: Copy-engine and SM based persistent kernels (all SMs) bandwidth utilization.

application performance, overall energy consumption, cost of ownership and operation complexity are largely overlooked. There are many reasons for this, but a key one is that these networks have been, until very recently based on proprietary interconnects such as NVLink or xGMI, and because it has evolved from host interconnects such as PCIe.

We summarize recent standardization efforts for scale-up networks including ESUN [20] / Sue-T [6] and UALink [4] and experimentally measure NVLink, the leading proprietary scale-up network in §2. We find that all rely on flow control to deal with congestion, and on XPU-based static load balancing to distribute traffic across the scale-up planes. These architectural choices enable very simple transports (e.g. Sue-T or UALink), but what are their performance implications? Our experiments show that flow control can lead to head-of-line blocking and that imperfect load balancing wastes bandwidth.

In brief, the same anomalies that have plagued scale-out networks also appear in scale-up networks. The obvious solutions (multipath transport, congestion control and best effort operation) are too expensive for on-chip implementations. We show how congestion control could be implemented in software to alleviate HoL blocking, but that still leaves many tough challenges for the networking community to solve.

2 SCALE-UP NETWORKING

Early scale-up domains used direct links between accelerators (e.g. NVIDIA Pascal and Volta architectures [14, 15], or Google's TPU). Each GPU-GPU connection requires multiple parallel lanes to deliver sufficient bandwidth, but load balancing (round robin) can be used to bundle these into a single high performance port while ensuring no reordering can appear; congestion control (CC) is not needed given the mesh connectivity, but bandwidth is wasted for many traffic patterns, including point-to-point.

As scale-up domains have been increasing in both bandwidth and size, mesh interconnects no longer scale. Instead, the leading interconnects use a single tier of switches and many planes as shown in Fig. 1; in the example each XPU connects to 36 different switches (e.g. a Broadcom Tomahawk 6, one per plane), with 200Gbps per plane, resulting in a total of 7.2Tbps per XPU. This topology is more efficient as it supports arbitrary traffic patterns at full XPU bandwidth, but is more difficult to use. Load-balancing across planes is harder - simple round-robin is no longer viable as it can lead to out-of-order arrival due to unbalanced queues on the different planes, which will impact application semantics. Instead, XPUs must load balance traffic across planes while ensuring consistency guarantees. Additionally, mechanisms are needed to cope with congestion appearing when multiple sources send to the same destination simultaneously.

Single-tier multi-plane topologies similar to this one are explored across the board: they are already used by Ampere, Hopper and Blackwell-based nodes (with NVSwitch switches), ESUN [20] relies on Ethernet switches, while UALink [4] relies on UALink switches.

There are two leading efforts to standardize scale-up networks: (1) Ethernet based, where the switch functionality is an evolution of Ethernet called ESUN [20] with proposed transports including Sue-T [6] and Unified Local Network profile for UET [5] and (2) UALink [4], a clean-slate solution which evolved from xGMI and relies on new switching hardware and a new transport. The UALink PHY is based on a modified Ethernet PHY.

Fig. 2 shows how scale-up transports (called I/O) integrate into the XPU, in the case of Sue-T. The interconnect is typically a chip-to-chip interconnect (like AXI), and the I/O exposes a series of per destination queues to the XPU, one queue per lane. The XPU load balances memory transactions into these queues, and the job of the transport is to batch the transactions, build packets and then deliver the data in-order and reliably to the destination. Memory semantics are handled by the XPU by relying on in-order delivery on each queue / plane. UALink takes a similar approach.

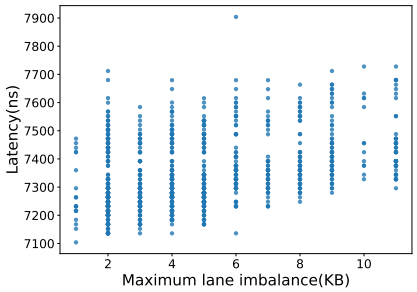


Figure 4: NVLink lane scheduling imbalance of different memory-copies with different source buffers and its impact on copy latency.

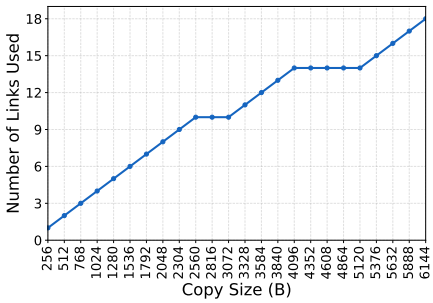


Figure 5: Number of NVLink lanes used for P2P transfer.

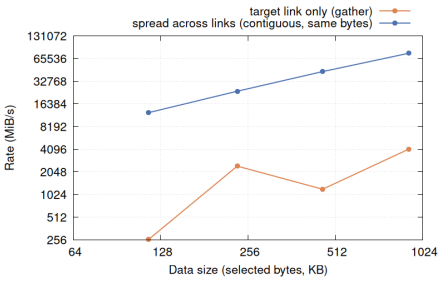


Figure 6: NVLink throughput when same data is sent over one link vs multiple links.

The key constraint for scale-up I/O is chip area: as the I/O sits on the same die as the XPU compute, area for I/O directly reduces compute: the transport must be as simple as possible, which means delegating work to the network. To deal with congestion, ESUN / SUE, and UALink rely on in-network flow control. For Ethernet, this is either PFC or the newer Credit Based Flow Control (CBFC [20]). To ensure reliability, link layer retransmission is supported by the switches (LLR [20]) where packets corrupted in flight are retransmitted by the link peer. End-to-end retransmission is optional in Sue-T (and missing in UALink) and is done via a go-back-N approach [6]; removing retransmission saves area for retransmission buffers but will lead to domain failures when packets are dropped due to insufficient headroom, or are corrupted in switch or destination memory and fail the CRC.

What are the performance effects of these choices? And what do deployed systems actually do in practice? We analyze NVLink next to answer these questions.

2.1 Experimental study of NVLink scale-up

NVIDIA GPUs can transfer data between other peer GPUs through the scale-up domain using the following methods: (1) SM based load/store operations, utilizing compute resources (threads and ld/st units) and increasing register pressure (2) Copy-Engine (CE) driven DMA, dedicated hardware component allowing for async transfers between GPUs (3) Tensor Memory Accelerator (TMA) async transfers between shared and global memory with limited data transfer size. CE-based transfers can saturate NVLink bandwidth with a single flow. Fig. 3 compares the bandwidth utilized by a persistent kernel utilizing all the SMs on the GPU copying.

2.2 NVLink load balancing

How does NVLink map memory transactions onto the 18 NVSwitch lanes? NVIDIA provides hardware counters on the NVLink traffic sent on each NVLink lane through NVML [16]. Using these counters we observe the scheduling patterns of NVLink and seek to find the mapping between memory addresses and lanes. We start by sending data between two GPUs in increasingly larger sizes. In Fig. 5 the results show that chunks of 256 bytes are scheduled on the same NVLink lanes. Even when repeating multiple writes in different access patterns the same scheduling can be observed:

Takeaway #1: Chunks of 256B are scheduled on the same lane deterministically.

By allocating multiple pages of memory (2MB) and remapping them into multiple regions in the virtual address space, we can repeat the same access patterns to determine whether the virtual or physical addresses are used when dispatching NVLink packets. The results show that regardless of the virtual memory region, the chunks map to the same lanes.

Takeaway #2: Physical addresses are used when mapping source GPU addresses to NVLink lanes.

As the page offset bits (first 21 bits) in both physical and virtual addresses are identical, the load balancing function could be based solely on these bits. But when writing to the same offset in different pages we get different lane scheduling. We conclude that the plane load balancing function uses bits from both physical frame number and the page offset bits.

How well does the load balancing function spread traffic across lanes? We conduct a series of experiments using different source GPU buffers of 2 MB (the size of a GPU memory page), 4, 8 and 16MB. We first run a discovery step in which we map every 256 byte-aligned address to its lane. For all buffer sizes, we find that traffic is not balanced perfectly: one link always carries more traffic than others. We refer to this as the target link and focus the rest of the experiment on testing two transfer patterns. The first one, referred to as gather, only transfers data from the addresses mapped to the target link (these are scattered throughout the buffer). The second method transfers the same amount of data, but reads from a

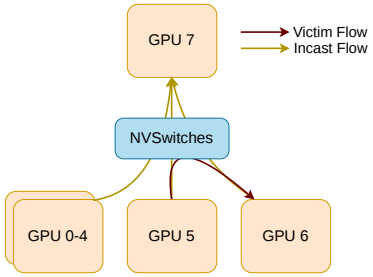


Figure 7: HoL blocking test on a DGXH100.

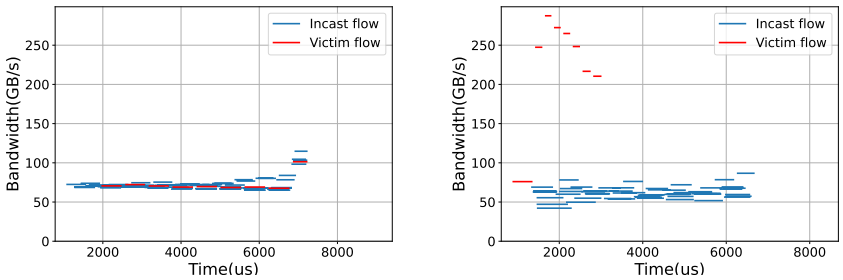


Figure 8: Incast flows and *victim flow* overlapping during subsequent copies: without CE congestion control (left) and with CE congestion control (right).

contiguous memory zone. We measure both transfer methods using a resident kernel launched once, and calculate the mean over 50,000 consecutive repetitions of the transfer, so that the measurement is continuous and unaffected by re-launch overhead. Fig. 6 plots for each data volume the target link throughput during the gather transfer against the aggregate throughput of the same size contiguous transfer. The latter multi-link series increases smoothly and benefits from data being sent over multiple lanes, as expected. The gather transfer's target link throughput varies non-monotonically across the tested sizes. The results confirm that load-balancing is load-insensitive, and using a single lane out of 18 reduces performance 18-fold.

Does this matter for actual traffic? When a load balancing function fails to spray traffic uniformly across all lanes, the lanes with the highest load will degrade the flow throughput. We test NVLink's load balancing performance by issuing inter-GPU CE writes on different source and destination memory regions. As the source physical address is used for determining the scale-up lane, each memory region will have a different scheduling and a different imbalance. For example, when using 2MB buffers, throughput can vary depending on the source buffer by up to 10%. [1]. We observe significant differences in the load balancing during these tests. We run 1000 such experiments and plot the latency of the transfer versus the amount of imbalance in Fig. 4; a more balanced scheduling correlates with higher throughput. FabricPerf [10] has independently discovered this imbalance in NVLink load balancing.

Takeaway #3: Load imbalance of addresses onto lanes reduces performance by upto 10%.

2.3 NVLink behaviour in congestion

Both ESUN / SUE-T and UALink rely on lossless operation and in-network flow control to cope with congestion, but this can lead to head-of-line blocking and reduced performance. What does NVLink do?

[1]We use Nsight to time the memory copies with 2% error.

We run an incast [3] experiment illustrated in Fig. 7 with many-to-one flows and an additional *victim flow*. Ideally the incast flows should share the ingress bandwidth of GPU 7 and the additional *victim flow* use the remaining egress bandwidth of GPU 5. Since inter-GPU CE transfers are serialized at the source, we will run the additional flow using SMs while the incast flows will use the CE. In Fig. 8 we observe the *victim flow* achieving equal throughput as the incast flows, and as soon as the switch egress contention disappears the throughput rises to line-rate. This behavior indicates that NVLink uses a flow control mechanism that paces all traffic on the ingress buffers leading to HOL blocking of additional flows to other destinations. Within large-scale datacenter networks these effects have been addressed by having end-to-end congestion control pace senders before the flow control is triggered (e.g. DCQCN [22]).

3 FIXING SCALE-UP NETWORKING

Head-of-line blocking and load balancing issues affect even single-tier multi-plane networks, as we showed earlier, but industry trends point to these problems worsening in the near future. NVL72, for instance, comes also in a two rack form factor (36x2), each rack with its own NVSwitches which are then connected via 36 switch to switch links; NVL576 is rumoured to be a two-tier DragonFly network [11]. In such networks, issues due to load balancing and HoL blocking will become more pressing, just as they did in scale-out domains. How can we solve these without increasing chip area?

The scale-up domain has multiple methods of initiating transfers (§ 2.1) while overload is handled via network backpressure in the form of flow control. Implementing congestion control can help avoid triggering flow control for common cases (and thus avoid HoL blocking) but requires handling each transfer method independently while ensuring fairness across flows. In this section we discuss implementing congestion control in scale-up networks without any hardware support for CE and load/store (SM based) transfers. Future work should include TMA congestion control as well.

Measuring congestion. Implementing any congestion controller requires a congestion signal. Since XPUs do not see packet loss and do not have access to packet-level telemetry (e.g. ECN), **delay** is the only usable congestion signal.

Scale-up domains are closely bundled (e.g. same tray, rack or adjacent racks) and use high speed interconnects; this means that queuing and host-related delays are the main contributors to end to end delay, but measuring delay is not easy: scale-up NICs do not have precise hardware timestamps supported on scale-out NICs [12], and obvious approaches to delay measurements are quite noisy.

In-kernel measurements. One way to measure delay is by timing read operations in kernels. GPUs have in-order threads retiring instructions one after the other, allowing for accurate measurements of a block of instructions. By issuing read operations to remote memory and enforcing a Read-After-Write (RAW) memory dependency we can accurately measure the round trip delay between two XPUs; this is simple to implement but will conflate forward and reverse path delays and make congestion control on the forward path react to reverse path congestion, which is obviously undesirable. Another option is to use write operations to measure the one-way delay to the destination by subtracting the receive time of the write (measured via polling at the destination) from the send time and maintaining an estimate of the clock drift between the source and destination. The implementation also requires transmitting the measurement to the source. For timing purposes CUDA exposes the *globaltimer*, a clock shared by all the SMs, and the *clock64* register containing the current clock cycle of the SM. *globaltimer* has a refresh rate of $1us$ which is too coarse for precise delay measurements. In our prototype we use reads for in-kernel RTT measurements (leaving one-way delay estimation to future work) and use *clock64* for accurate timing.

Timing copy-engine transfers. CUDA API calls such as *cudaMemcpyAsync* support stream semantics, allowing for *CUDA events* to be recorded before and after the transfer which enables measurements. However, the events add overhead to the recorded duration. We instead use NVIDIA Nsight which offers a more accurate measurement ($3us$ shorter) consistent across different transfer sizes. We compute a size-independent delay sample as:
$delay = total - serialization_delay(size) - timing_overhead$
(where $timing_overhead$ is a constant we measured)

3.1 Copy-engine congestion control

The copy-engine receives variable sized transfer requests from the application and then issues them serially; it goes as fast as the network allows it. We implement congestion control for CE by intercepting transfer requests, splitting them into chunks and then altering their transmission rate. We identify and implement two ways of performing congestion control for the copy-engines: (1) pacing of fixed sized chunks and (2) using variable sized chunks.

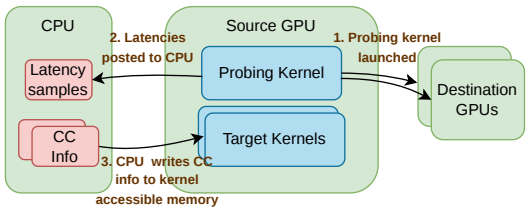


Figure 9: Load/store congestion control design.

Pacing fixed-size chunks. The straightforward way to implement congestion control is to pace the transmission for fixed-sized chunks; we implemented the pacer using a leaky bucket and use the rate computed by the congestion control algorithm (see below). The fixed chunk size for this approach should be large enough for the system to reach line-rate (at least 32MB based on the results in Fig. 3). If the application requested size is smaller than the chunk size the controller still waits for enough credits to be accumulated.

The congestion controller only adjusts the spacing between the chunks; a chunk will still be delivered at line-rate by the hardware. The downside of this approach is that when multiple flows start at the same time, their chunks will collide (each 32MB chunk lasts 8us at 3.2Tbps), triggering NVLink flow control. We can reduce the occurrence of this effect by adding jitter before chunk transmissions. This does reduce the collisions but does not eliminate them.

Variable sized chunking. An alternative approach is to vary the chunk size dynamically; when the rate is low due to congestion, smaller chunks should be used to reduce the aliasing effects discussed above. Conventional wisdom says that we should still use pacing, but this would result in a more complex system as congestion control would need to adjust both rate and chunk size. Instead, we observe that the CE posting overheads (spent in the CUDA runtime, driver, hardware queuing [21]) can be used as an implicit way to achieve rate limiting. Fig. 3 showcases the performance of the copy-engine for various chunk sizes; we use this information and the target rate from the congestion controller to select the smallest chunk size which achieves that rate (eg. 1MB for 150GB/s). We limit the range of the chunk sizes between our desired minimum and maximum bandwidth. As the maximum incast in the domain can reduce the bandwidth downto 60GB/s, the minimum chunk size would be 512KB, while the maximum is 32MB (390GB/s).

To ensure we still utilize the total I/O bandwidth when we have a mix of congested and uncongested flows, we post multiple chunks onto the same stream (a configuration parameter set to 4 in our experiments). The chunks from uncongested flows will be larger, increasing total goodput while

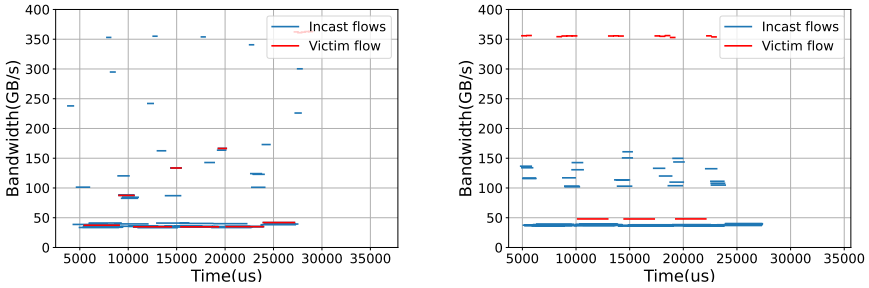


Figure 10: Incast flows and *victim flow*: without congestion control (left) and with load/store in-kernel congestion control (right).

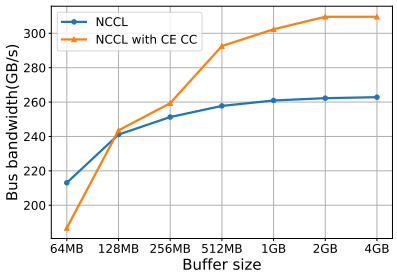


Figure 11: NCCL all-to-all on 4 H100 GPUs using copy-engines.

still pacing congested flows. While queuing multiple transfers helps minimize the idle time between chunks, it also delays the congestion response time. The impact depends on the workloads themselves, but we observe that periodic and repeatable patterns converge to a stable chunk size. On the other hand, sudden changes in network congestion take effect only when the committed chunks queue is drained.

For the congestion controller itself we use a simplified version of Swift [12] algorithm which has the nice property that the target latency is stable when no contention is present and the flow based scaling can accommodate higher queuing delays when congestion builds up.

Both pacing and chunk-size approaches need to track the queued chunks and to update the state of the controller on transfer completions. We record a CUDA event after each transfer and we query the events for transfer completions.

Integration. Ideally adding CC should require no code changes and be easily tested with any application. We implement the copy-engine congestion controller as a dynamic library interposing the CUDA runtime API (eg. *cudaMemcpyAsync*, *cudaMemcpyBatchAsync*, etc.) calls. The CE transfers are managed by background threads running the congestion controller. In order to maintain the stream sematics, we enqueue memory stream operations *cuStreamWaitValue32* / *cuStreamWriteValue32* that are signaled by the threads when the last chunk is delivered.

Evaluation. We use the (2) *variable sized chunking* for all experiments in this section as it performed better due to the flow control getting triggered on burst collision in the (1) *pacing fixed-size chunks* approach. On the initial testing setup Fig. 7 using the running CE driven incast flows and a load/store kernel for the *victim flow* we successfully paced the CE flows without noticing any flow control anymore with the additional flow running up to 4x faster (Fig. 8).

In figure Fig. 11 we run unmodified NCCL all-to-all collectives with *nccl-tests* over the copy-engine. To enable congestion control, we LD_PRELOAD our implementation when launching NCCL. The results show that using congestion control improves performance by up to 20% for large collectives. For collectives smaller than 128MB, software congestion control reduces performance due to chunking overheads. Further optimizations are possible, but a quick fix is to enable congestion control only for large collectives.

3.2 Load/store congestion control

Implementing congestion control for GPU kernels in software is harder than for copy-engines. Each GPU thread issues load/store via scale-up that dispatch just 1-8 bytes. Congestion control requires gathering the congestion signal, updating the internal state and enforcing the congestion window; without hardware support all these operations will use GPU resources (e.g. SMs), need kernel changes and will thus slow down the main workload. To minimize the added overhead we run the congestion control algorithm on the CPU, and perform latency measurements and instruction pacing on the GPU. The architectural diagram is shown in Fig. 9

Traditional protocols use per packet feedback (ACKs) to drive congestion control which then limits the amount of data in flight. Load/store semantics do not provide explicit completion notifications and there is no explicit way to limit the amount of data in flight from software. Instead, our congestion control loop uses out-of-band congestion measurements (via a separate probing kernel) and computes an *credit rate* that is used by the target kernel to slow down memory accesses. In more detail, the target GPU kernel uses cooperative thread arrays or CTAs [17] to implement credit accumulation (at the rate computed by congestion control) which can then be spent on memory movement operations. The kernel is modified to poll for credits before each chunk of the total workload is started (e.g. a tile transfer), which effectively paces operations at the target rate. In our prototype we only pace writes; pacing reads (and maintaning a separate CC context for them) is left for future work.

The CC interface has two methods called by a single thread in the kernel CTA: *cc_pace*, which blocks the other threads from entering the next chunk and accumulates credits and *cc_poll*, which polls the CPU written congestion information

and updates the admission rate. Each GPU kernel has its corresponding congestion state updated independently.

Probing kernel. A persistent kernel runs on all GPUs that probes the latency to all other possible destination GPUs as described previously. The kernel publishes the latencies in a host memory buffer. The CPU based congestion controller samples this buffer at a fixed rate (20*us* in our implementation) and applies an EWMA over the gathered latencies. While copy-engine congestion control only measured latency once per chunk, a dedicated probing kernel provides a more accurate latency signal at the cost of using an SM.

Evaluation. In Fig. 10 we run the same incast plus victim flow scenario (shown in Fig. 7) with the many to one flows running on SMs with congestion control enabled. The victim flow is running the same kernel but with no congestion control. When no CC is running, the common case is that flow control is triggered and all the flows have equal bandwidth, the outliers where the bandwidth is greater than 60GB/s are the cases where flows do not overlap. When running CC on the incast flows the *victim flow* manages to occupy all of the available bandwidth most of the time.

Notice, however, that the outcome of congestion control is not perfectly stable, with periods where the victim flow's bandwidth is reduced to that of the incast flows. We speculate that this is due to congestion control not always doing a good job in reducing the incast queue size at the bottleneck which leads flow control to kick in. A full exploration of the interaction between congestion control and flow control is required in future work.

4 DISCUSSION

As scale-up domains increase in size and the topologies are changing from single-tier to multi-tier the previous assumptions that oblivious XPU-based load balancing and network flow control is good enough no longer hold. We've shown that flow control can lead to HOL blocking (§ 2.3) and a 20% performance drop for all-to-all even on current hardware; we expect this will get worse for future hardware.

Our work shows that it is possible to approximate congestion control in software for the copy-engine, and with some intrusive kernel changes also for the SM-based workloads. This is a stop-gap solution until hardware support is added, but it raises a few challenges that need to be addressed:

- One-way latency estimations are needed to separately measure forward and reverse path congestion. This can be achieved via time-synchronising GPUs as proposed in FabricPerf [10] at a fairly high compute resource cost. Are there more efficient ways of achieving the same goal?
- Reads and writes must be congestion controlled independently, thus congestion information must be known to both senders and receivers of probes. Efficiently measuring, disseminating and then using congestion information in XPUs is a new area of exploration that will need to balance of benefits of congestion control versus the compute and bandwidth resources it uses.
- Host latencies due to HBM and memory controller contention must be measured and isolated from network latencies.

Most existing congestion control solutions targetted at AI workloads focus on best-effort networks, with or without support for packet trimming (e.g. RCCC and NSCC [2, 5] or Swift [12, 19]) or networks with PFC (e.g. DCQCN [22]). Flow control schemes such as CBFC offer finer grained control over switch buffer utilization than PFC, but their interaction with congestion control is poorly understood. Understand the interaction between congestion control and credit-based flow control may result in new ways to use existing congestion control algorithms in flow controlled networks, or novel congestion and flow control algorithms.

Implementing hardware congestion control in XPUs is another interesting area of research. A split CPU / hardware architecture, with the hardware providing the congestion signals per destination and implementing pacing and the software implementing congestion control seems a step in the right direction.

Fixing load balancing also requires a hardware / software split to reduce area, but appears a harder problem to solve; the current approach of statically mapping from addresses to lanes is very limiting, but dynamic load balancing (as in MRC [2], for instance) has memory semantic implications which will require XPU co-design.

Finally, given the growing size and complexity of scale-up domain networking, there are multiple missing pieces that would benefit scale-up communication by improving observability: (1) Current userspace APIs add high overhead for timing GPU events, reducing the performance of the controller. Better observability in hardware queues, latencies across the network and at the scale-up NIC could uncover both resource contention and help implement better in-software congestion control. (2) Support for network telemetry is required for both dynamic load balancing and congestion control. ECN marks and maybe newer signals such as congestion signaling should be accessible to congestion control to improve accuracy and reduce the need for accurate timestamps.

ACKNOWLEDGEMENTS

This work was partly funded by HRIA (project no. 351416, funded by the Romanian Ministry of Investment and European Funds) and by Broadcom Gift Funding for the Netsys group at Politehnica of Bucharest.

REFERENCES

[1] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. 2010. Hedera: dynamic flow scheduling for data center networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation* (San Jose, California) *(NSDI'10)*. USENIX Association, USA, 19.

[2] Joao Araujo, Alex Chow, Mark Handley, Ryder Lewis, Christoph Paasch, Jitendra Padhye, Michael Papamichael, Greg Steinbrecher, Amin Tootoonchian, Lihua Yuan, S. Anantharamu, Abhishek Dosi, Mohit Garg, Mahdieh Ghazi, Torsten Hoefler, Deepal Jayasinghe, Jithin Jose, Abdul Kabbani, Guohan Lu, Yang Wang, K. Doddapaneni, Murali Garimella, Vipin Jain, Yanfang Le, H. Nagulapalli, S. Narayanan, Rong Pan, Rathina Sabesan, Raghava Sivaramu, Rip Sohan, Eric Davis, Dragos Dumitrescu, Mohan Kalkunte, Bhaswar Mitra, Guglielmo Morandin, Adrian Popa, Costin Raiciu, Eric Spada, John Spillane, Niranjan Vaidya, Aviv Barnea, Idan Burstein, Elazar Cohen, Yamin Friedman, Noam Katz, Masoud Moshref, Yuval Shpigelman, Shahaf Shuler, Shy Shyman, and Sayantan Sur. 2026. Resilient AI Supercomputer Networking using MRC and SRv6. arXiv:2605.04333 [cs.NI] https://arxiv.org/abs/2605.04333

[3] Yanpei Chen, Rean Griffith, Junda Liu, Randy H. Katz, and Anthony D. Joseph. 2009. Understanding TCP incast throughput collapse in datacenter networks. In *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking* (Barcelona, Spain) *(WREN '09)*. Association for Computing Machinery, New York, NY, USA, 73–82. https://doi.org/10.1145/1592681.1592693

[4] Ultra Accelerator Link Consortium. 2025. UALink Common 2.0 Specification. https://ualinkconsortium.org/specification/

[5] Ultra Ethernet Consortium. 2026. Ultra Ethernet Specification v1.1. https://ultraethernet.org/wp-content/uploads/sites/20/2026/10/UE-Specification-1.1.pdf

[6] Broadcom Corporation. 2025. https://www.opencompute.org/documents/ocp-sue-spec-final-pdf-1

[7] Alexandru M. Gherghescu, Vlad-Andrei Bădoiu, Alexandru Agache, Mihai-Valentin Dumitru, Iuliu Vasilescu, Radu Mantu, and Costin Raiciu. 2024. I've Got 99 Problems But FLOPS Ain't One. In *Proceedings of the 23rd ACM Workshop on Hot Topics in Networks* (Irvine, CA, USA) *(HotNets '24)*. Association for Computing Machinery, New York, NY, USA, 195–204. https://doi.org/10.1145/3696348.3696893

[8] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. 2017. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (Los Angeles, CA, USA) *(SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 29–42. https://doi.org/10.1145/3098822.3098825

[9] Torsten Hoefler, Karen Schramm, Eric Spada, Keith Underwood, Cedell Alexander, Bob Alverson, Paul Bottorff, Adrian Caulfield, Mark Handley, Cathy Huang, Costin Raiciu, Abdul Kabbani, Eugene Opsasnick, Rong Pan, Adee Ran, and Rip Sohan. 2025. Ultra Ethernet's Design Principles and Architectural Innovations. arXiv:2508.08906 [cs.NI] https://arxiv.org/abs/2508.08906

[10] Songlin Huang, Chenshu Wu, Eric Keller, Sangtae Ha, Laurent Vanbever, and Theophilus Benson. 2026. FabricPerf: Measuring NIC-less Scale-Up Network through GPU Communication Kernel Profiling. In *Proceedings of the ACM SIGCOMM 2026 Conference* (Denver, Colorado, USA). 642–656. https://doi.org/10.1145/3789240.3829172

[11] John Kim, Wiliam J. Dally, Steve Scott, and Dennis Abts. 2008. Technology-Driven, Highly-Scalable Dragonfly Topology. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA '08)*. IEEE Computer Society, USA, 77–88. https://doi.org/10.1109/ISCA.2008.19

[12] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. 2020. Swift: Delay is Simple and Effective for Congestion Control in the Datacenter. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 514–528. https://doi.org/10.1145/3387514.3406591

[13] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. 2018. Homa: a receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 221–235. https://doi.org/10.1145/3230543.3230564

[14] NVidia. 2016. NVIDIA Tesla P100 Whitepaper. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

[15] NVidia. 2017. NVIDIA Tesla V100 Whitepaper. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[16] NVidia. 2026. NVIDIA Management Library (NVML). https://developer.nvidia.com/management-library-nvml

[17] NVIDIA. 2026. Parallel Thread Execution ISA Version 9.3. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#cooperative-thread-arrays

[18] Costin Raiciu, Christopher Pluntke, Sebastien Barre, Adam Greenhalgh, Damon Wischik, and Mark Handley. 2010. Data center networking with multipath TCP. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks* (Monterey, California) *(Hotnets-IX)*. Association for Computing Machinery, New York, NY, USA, Article 10, 6 pages. https://doi.org/10.1145/1868447.1868457

[19] Arjun Singhvi, Nandita Dukkipati, Prashant Chandra, Hassan M. G. Wassel, Naveen Kr. Sharma, Anthony Rebello, Henry Schuh, Praveen Kumar, Behnam Montazeri, Neelesh Bansod, Sarin Thomas, Inho Cho, Hyojeong Lee Seibert, Baijun Wu, Rui Yang, Yuliang Li, Kai Huang, Qianwen Yin, Abhishek Agarwal, Srinivas Vaduvatha, Weihuang Wang, Masoud Moshref, Tao Ji, David Wetherall, and Amin Vahdat. 2025. Falcon: A Reliable, Low Latency Hardware Transport. In *Proceedings of the ACM SIGCOMM 2025 Conference* (Coimbra, Portugal) *(SIGCOMM '25)*. Association for Computing Machinery, New York, NY, USA, 248–263. https://doi.org/10.1145/3718958.3754353

[20] ESUN Workstream. 2026. OCP ESUN - Network Operator Requirements - Base Specification. https://www.opencompute.org/documents/ocp-esun-network-operator-requirements-base-specification-rev-1-0-final-pdf

[21] Yuang Yan, Ian Karlin, and Ryan Grant. 2026. Revealing NVIDIA Closed-Source Driver Command Streams for CPU-GPU Runtime Behavior Insight. arXiv:2604.26889 [cs.PF] https://arxiv.org/abs/2604.26889

[22] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion Control for Large-Scale RDMA Deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (London, United Kingdom) *(SIGCOMM '15)*. Association for Computing Machinery, New York, NY, USA, 523–536. https://doi.org/10.1145/2785956.2787484