NetSparse: In-Network Acceleration of Distributed Sparse Kernels

Gerasimos Gerogiannis
University of Illinois at Urbana-Champaign
Urbana, Illinois, USA
gg24@illinois.edu

Dimitrios Merkouriadis
University of Illinois at Urbana-Champaign
Urbana, Illinois, USA
dm68@illinois.edu

Charles Block
University of Illinois at Urbana-Champaign
Urbana, Illinois, USA
coblock2@illinois.edu

Annus Zulfiqar
University of Michigan
Ann Arbor, Michigan, USA
zulfiqaa@umich.edu

Filippos Tofalos
University of Illinois at Urbana-Champaign
Urbana, Illinois, USA
tofalos2@illinois.edu

Muhammad Shahbaz
University of Michigan
Ann Arbor, Michigan, USA
msbaz@umich.edu

Josep Torrellas
University of Illinois at Urbana-Champaign
Urbana, Illinois, USA
torrella@illinois.edu

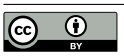
Abstract

Many hardware accelerators have been proposed to accelerate sparse computations. When these accelerators are placed in the nodes of a large cluster, distributed sparse applications become heavily communication-bound. Unfortunately, software solutions to optimize network communication are inefficient.

In this paper, we introduce novel hardware mechanisms to optimize network communication in distributed sparse computations. Our proposal, called *NetSparse*, consists of four mechanisms. Communication is offloaded to new processing units in the NIC that support efficient *remote indexed gather* operations, minimizing host-NIC communication. Moreover, these units have the ability to identify and eliminate redundant requests, which minimizes traffic. Further, new hardware modules in the NICs and switches concatenate multiple requests with the same destination node in a single packet, saving traffic and header overheads. Finally, switches are augmented with a hardware cache that stores fetched data from remote racks, making it available to all the nodes in the local rack on demand. Our evaluation on a simulated 128-node cluster with per-node sparse accelerators running sparse workloads reveals that NetSparse improves performance substantially. When the cluster uses traditional software-based communication, the workloads run only 3x faster than on a single-node system; when it is augmented with the NetSparse hardware, the workloads run 38x faster than on the single-node system—attaining more than half of the performance of an ideal system that has no communication overheads.

CCS Concepts

• **Networks** → **In-network processing**; **Programmable networks**; • **Hardware** → **Networking hardware**; • **Computer systems organization** → **Distributed architectures**; **Special purpose systems**.



MICRO '25, Seoul, Republic of Korea

ACM ISBN 979-8-4007-1573-0/25/10
https://doi.org/10.1145/3725843.3756076

Keywords

High-performance Computing, In-network Computing, Sparsity, Sparse Computations, SmartNIC, Programmable Switch, RDMA, Hardware Acceleration

ACM Reference Format:
Gerasimos Gerogiannis, Dimitrios Merkouriadis, Charles Block, Annus Zulfiqar, Filippos Tofalos, Muhammad Shahbaz, and Josep Torrellas. 2025. NetSparse: In-Network Acceleration of Distributed Sparse Kernels. In *58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25), October 18–22, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3725843.3756076

1 Introduction

Sparse computations such as the multiplication of a sparse matrix and a dense matrix (SpMM) or a dense vector (SpMV), and the sampled multiplication of two dense matrices (SDDMM) [24] are widely-used in graph analytics and machine learning [6, 14, 46, 64, 70, 90]. These computations use sparse datasets such as social networks or computational genomic graphs that can have terabyte-scale memory footprints [25] and compute needs that require execution on a large-scale cluster platform.

Over the last few years, there has been significant interest in the development of hardware accelerators for sparse computations [1, 28, 29, 31, 32, 55, 57, 61, 81, 82]. Although these architectures are effective, they mostly target a single node. When these accelerators are placed on each node of a cluster, computation gets sped-up, and cluster-level programs become heavily bound by network communication [11]. One could optimize network communication with hardware support, but the research in this area has focused on supporting distributed *dense* kernels, such as those in dense machine learning [75]. Network-centric hardware optimizations for distributed *sparse* computations remain underexplored.

Sparse communication algorithms typically follow one of two approaches [9, 11]. In the *Sparsity-Unaware* (SU) approach, communication is scheduled as if the kernel was dense. Data is transferred in coarse granularities using all-to-all transfers (e.g., collectives [77]), disregarding whether every data item is actually needed by the requester. As this approach minimizes the software overheads of generating many small network messages [11], it leads to high

utilization of the network's bandwidth. However, by ignoring the sparsity pattern of the data, it induces redundant data transfers.

The second alternative is the *Sparsity-Aware* (SA) approach [34, 62, 72]. Here, the indices of the nonzero values of the sparse matrix (or edges in a graph) are scanned and trigger the transfer of small data items (e.g., with RDMA [65]). We refer to these data items as *properties* since, in graph problems, they represent properties of the vertices connected to each edge (e.g., embeddings [46]). Similarly, we refer to the fine-grained remote requests as *Property Requests* (PRs). Although SA results in fewer redundant transfers, it induces significant software overheads for PR generation [11]. These overheads lead to network bandwidth underutilization.

In this work, we first quantify the inefficiencies of software-only implementations of the two communication approaches. On average, the SU approach generates 720 redundant property transfers for every useful one. Although the SA approach leads to fewer redundant transfers, it can drop the line utilization to below 1% in a large cluster. In addition, due to the small sizes of the SA payloads (4B-1KB), the network traffic is dominated by packet headers, reducing the goodput [86]. Overall, both approaches are inefficient, leaving orders of magnitude of performance on the table.

To drastically improve performance, this paper presents *NetSparse*, a set of hardware techniques to accelerate sparse communication. NetSparse extends SmartNICs (SNICs) [4, 5, 37, 66, 68, 87, 88] and programmable switches [15, 17, 18, 38, 39] with hardware extensions catered to the sparse domain. NetSparse enhances the SA approach by (1) increasing the network bandwidth utilization, (2) reducing the traffic, and (3) increasing the goodput by reducing the header overheads of network packets.

NetSparse introduces four hardware mechanisms. First, it extends RDMA to support one-sided *Remote Indexed Gather* (RIG) operations. Using coarse-grained RIG commands, the host offloads PR generation for batches of nonzeros to the NIC, reducing the overhead of frequent host-NIC communication through the PCIe bus. PRs are generated and served by special hardware *RIG Units* in the NIC without host involvement. The second mechanism enables RIG Units to *filter out* redundant PRs at runtime. With both techniques, PRs are generated at higher rates, the network bandwidth is better utilized, and useless traffic is minimized.

The third mechanism is a low-level network protocol that concatenates different PRs with the same destination node in a single packet. This is implemented with *(De)Concatenator* hardware modules in NICs and Switches. The goal is to increase the goodput by sharing the header overhead over multiple PRs. Finally, the fourth mechanism augments Top-of-Rack (ToR) switches with a hardware cache that stores properties fetched from remote racks. Subsequent PRs from the local rack can quickly obtain the property from the cache. In contrast to P4-based [12, 13] switch caching mechanisms [42], the NetSparse cache is updated in the data plane to accommodate short-lived distributed sparse kernels.

We evaluate NetSparse with simulations of a 128-node cluster with a sparse accelerator in each node while running distributed sparse computations on matrices from SparseSuite [25]. Our results show that NetSparse substantially improves performance over ideal versions of SU and SA software approaches (where SA is augmented with the Conveyors framework [59]). For example, with SA software, the workloads run only 3x faster than on a single-node system; with NetSparse, the workloads run 38x faster than on the single-node system—attaining more than half of the performance of an ideal system that has no communication overheads.

This paper makes the following contributions:

- A characterization of software solutions for communication in distributed sparse kernels, exposing major inefficiencies.
- The four hardware techniques of NetSparse to accelerate the communication of sparse workloads in a distributed machine.
- A simulation-based evaluation of NetSparse in a 128-node cluster.

2 Background

2.1 Distributed Sparse Communication

Many sparse computations involve the multiplication of a sparse matrix with an input array to produce an output array. The matrix may represent the edges of a graph, while the input and output arrays may represent features or properties of the vertices of the graph. The input and/or output arrays can be dense vectors (e.g., in SpMV), or dense matrices that are typically tall and skinny (e.g., in SpMM and SDDMM). Figure 1 shows a pictorial representation. We refer to the number of elements in a property as K. Typically, sparse kernels involve multiple iterations, with the output property array of one iteration becoming the input of the next iteration. Across iterations, the structure of the sparse matrix may change (e.g., in graph neural network applications with sampling [30]).

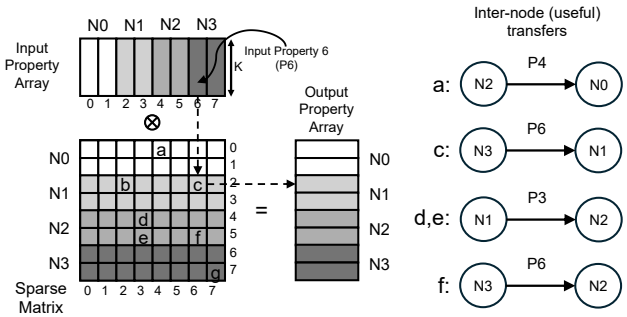


Figure 1: Communication in a distributed sparse kernel.

In distributed sparse computations, data structures are partitioned across nodes. Although different partitioning methods exist, 1D works well, especially when the input/output property arrays are tall and skinny [72]. Figure 1 shows how the 3 structures are 1D-partitioned across 4 nodes of a distributed system (i.e., N0, ... N3), with elements in the same node depicted with the same color.

The coordinates of the nonzeros in the sparse matrix dictate the accesses to the input and output property arrays. Consider nonzero c in Figure 1. Its row index (rid) is 2, which leads to a write to the output property at index 2. Its column index (cid) is 6, which leads to a read of the input property at index 6. Note that both c and the output property that it accesses reside in the same node (i.e., N1). With 1D partitioning, writes are always local. However, the read for input property 6 is remote. The right part of Figure 1 shows all the necessary inter-node input property transfers for the specific nonzero pattern in the example sparse matrix. Notice that nonzeros d and e can be covered using a single property transfer from N1 to N2. No remote transfers are needed for nonzeros b and g, as the properties they access are already local.

Since writes are always local, from now on, we refer to input properties simply as properties. Also, we use *Property Requests* (PRs) to refer to inter-node reads and read responses for these properties.

2.2 Sparse Communication Methods

Distributed sparse communication algorithms typically follow one of two approaches. The *Sparsity-Unaware* (SU) approach [9, 11, 77] schedules property transfers as if the kernel was dense. The matrix-specific nonzero pattern is ignored, and all the nodes receive all of the input properties—e.g., through coarse-grained collective communications [77]. This leads to redundant property transfers. For example, in Figure 1, N0 would receive all 6 remote properties, even though it only needs P4. Despite the redundant transfers, the SU approach can typically achieve high utilization of the network's bandwidth, since it avoids the software overheads of generating many small network messages [11].

The alternative approach, called *Sparsity-Aware* (SA) [34, 62, 72], requires scanning the sparse matrix and issuing a PR for each nonzero that needs a remote property (e.g., through one-sided RDMA). SA can eliminate most of the redundant transfers in SU. Still, a request for the same property may be issued more than once if multiple nonzeros in a node have the same cid (e.g., d and e in Figure 1). This causes some redundant transfers. Also, SA suffers significant software overhead to generate many fine-grained PRs.

2.3 Lifetime of an SA Property Request

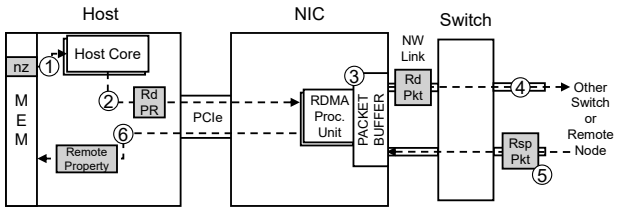


Figure 2: The lifetime of a sparsity-aware property request.

Figure 2 shows the lifetime of an SA PR. First, a core reads one of the nonzeros assigned to its node ①, to determine what remote property needs to be fetched. Recall that the nonzero's cid represents the index of the remote property that needs to be fetched. Hence, in the rest of the paper, we refer to cid as *idx* (for index). After the idx is read from memory, the destination node is determined and a read PR is generated ②. This read PR will be directed through RDMA to the remote node that hosts the target property. Specifically, the read PR is forwarded to an RDMA processing unit in the NIC and a network packet is generated ③. The packet travels through the network and eventually reaches its destination ④. The destination node generates a read response containing the necessary property data, which travels back to the original node ⑤ and is written to its memory ⑥.

This process introduces significant inefficiencies. The host cores and the NIC communicate via MMIO writes for every nonzero of the sparse matrix, leading to the serialization of PR generation for different idxs. Furthermore, given that typical K values (Section 2.1) range from 1 to 256 (i.e., 4B–1KB payloads, assuming single precision), this process creates many fine-grained small-payload network messages. The resulting software overheads limit the rate at which new PRs are generated and limit network bandwidth utilization.

3 Motivation

In this section, we characterize the communication patterns exhibited by the SpMM, SDDMM, and SpMV kernels for five large sparse matrices from the popular SparseSuite matrix collection [25] (details in Section 8) running on the NCSA Delta supercomputer [63].

• SU algorithms issue three orders of magnitude more redundant transfers than useful ones. The first row of Table 1 displays the ratio of useful to redundant property transfers for the SU approach in a 128-node system. On average, 720 redundant transfers are generated for every useful one. Hence, an optimal system that avoids redundant transfers could potentially offer up to 720x average performance gains. This is a fundamental inefficiency of SU.

Table 1: Ratio of useful to redundant property transfers in a 128-node system for the SU and SA algorithms.

Matrix	arabic	europe	queen	stokes	uk
SU	1 : 1947	1 : 582	1 : 74	1 : 32	1 : 966
SA	1 : 27	1 : 0.02	1 : 25	1 : 3.6	1 : 4.5

• SA can also issue redundant transfers. The second row of Table 1 shows that, with the exception of the europe matrix, the SA approach can also lead to redundant transfers. This can happen if nonzeros in the same node share the same idx—e.g., the nonzeros d and e in Figure 1 both have idx=3. Note that, although some SA algorithms such as [11, 72] eliminate these redundant transfers, they typically require costly preprocessing that needs to be repeated every time the sparse matrix changes.

• The software overhead of SA leads to severe underutilization of the network bandwidth. As explained in Section 2.2, SA transfers have software overheads that limit network bandwidth utilization. To quantify this effect, we use the source code of the SOTA distributed SpMM algorithm [11], configure it to SA-only mode, and evaluate it for a property size of K=32 (i.e., 128B). We measure the data transfer rates for a 2-node setup in the Delta supercomputer, which is interconnected with Cray Slingshot [79]. The measurements are shown in Table 2. On average, the achieved transfer rate is a little less than 0.5Gbps. We then compute the Line Utilization, which is the ratio of the transfer rate to the maximum line bandwidth (i.e. line rate). From the table, we see that the average line utilization is 0.24%, which leaves a lot of performance on the table.

Table 2: SA transfer rate and related metrics for a 2-node setup in the Delta machine (K=32).

Matrix	arabic	europe	queen	uk
Transfer Rate	0.5 Gbps	0.2 Gbps	0.7 Gbps	0.5 Gbps
Line Util.	0.26%	0.09%	0.36%	0.25%
Goodput	0.11%	0.04%	0.16%	0.11%

• The header overhead of small-payload packets impacts goodput. SA suffers from a high packet header overhead. Table 3 displays the contribution of the packet headers in the total SA network traffic for different K values. For low K, this contribution is as high as 97.6%. The last row of Table 2 shows that, for K=32, less than half of the achieved throughput is actually goodput.

Table 3: Contribution of packet headers to the total SA network traffic for properties with different K.

K	1	2	4	8	16	32	64	128	256
%	97.6	95.2	90.9	83.3	71.4	55.6	38.5	23.8	13.5

• PRs generated close in time are often directed to the same node. We find that PRs generated close in time by a node often request properties hosted by the same destination node. We refer to this property as *temporal remote destination locality*. As shown in Table 4, for queen, on average, all 64 consecutive PRs from a source node are directed to the same destination node (out of the 127 possible destination nodes). The reason is that queen has its nonzeros largely concentrated around the diagonal. PRs in other matrices also show temporal remote destination locality—even in europe, 64 consecutive PRs have only 7.4 different destinations out of the 127 possible ones. Later, we show how we exploit this property to concatenate network messages and improve goodput.

Table 4: Average number of unique remote destination nodes in 64 consecutive PRs from a node in a 128-node system.

Matrix	arabic	europe	queen	stokes	uk
Nodes	2.51	7.43	1.00	1.85	5.61

• Remote properties fetched by a node in a rack can be reused by other nodes in the same rack. Most of today's high performance computing and datacenter networks are hierarchical [45, 49, 51]. For example, datacenter clusters are organized in racks, with nodes in the same rack connected to the same ToR switch. Similarly, the Delta supercomputer is interconnected with a Dragonfly topology [45], where nodes are organized in small groups and intra-group communication is faster. Hence, we investigate the potential for sharing properties across nodes of the same rack/group, assuming groups of size 16. Excluding redundant transfers, we find that, on average, 85% of the PRs are for properties useful to more than one node in the same group. We later show how we employ in-network caching to utilize this sharing potential.

• Putting it all together. These results reveal that distributed sparse kernels are missing out on substantial performance potential. Fortunately, they also hint at how to attain some of this performance. To avoid the excessive redundant transfers of SU approaches, one must (1) exploit the sparsity patterns and adopt an SA approach. At the same time, one should: (2) eliminate any remaining redundant transfers of SA approaches at no preprocessing cost, (3) increase the SA line utilization by eliminating the software overheads associated with PR generation, and (4) decrease network header overheads. Throughout the process, one should leverage (5) temporal remote destination locality and (6) intra-group property sharing.

4 NetSparse Architecture

Based on the previous findings, we introduce four hardware techniques to accelerate distributed sparse communication. We call the resulting architecture *NetSparse*. NetSparse extends SNICs and programmable switches with hardware extensions catered to the sparse domain. In this section, we give a high-level overview of these mechanisms; we describe the mechanisms in more detail in Sections 5 and 6. Our design uses the vanilla SA approach of Figure 2 as a starting point.

1. Hardware Support for One-Sided Remote Indexed Gathers. The SA communication pattern involves reading a list that indexes properties needed by the nonzeros and that are hosted in various remote nodes. This operation as a whole is a large *Remote Indexed Gather (RIG)*. This operation is not supported natively by RDMA today (e.g., as defined by Infiniband verbs [58]). In a vanilla SA execution, the host cores break-down the RIG into fine-grained per-nonzero RDMA reads, and issue each one of them to the NIC. As a result, the host and the NIC need to communicate and synchronize through the PCIe bus very frequently. This leads to a high software cost and high latencies that cannot be hidden, limiting the effective rate at which new PRs can be generated.

To increase this rate, we propose extending RDMA with native support for RIG operations. The host cores issue RIG operations that are handled by novel specialized processing units (*RIG Units*) in an SNIC. Each RIG command contains the information needed for the SNIC to fetch the properties for a coarse-grained batch of nonzeros, without requiring involvement from the cores of either the host or remote nodes. RIG operations are one-sided. A RIG completes and the host core is informed when: (1) all the PRs in the batch have been generated, and (2) all the remote properties have been fetched and written back to the host memory (through DMA). By coarsening the granularity of NIC-host communication and eliminating general-purpose overheads with the specialized RIG Units, the PR generation rate increases, leading to a substantial *increase in the network bandwidth utilization*.

2. Property Request Filtering and Coalescing. Our second technique *eliminates redundant SA PRs at runtime*. Recall that nonzeros with the same idx generate PRs for the same property. Since the input property array is not updated during an iteration of the sparse kernel, all the requests to a given property after the first one are redundant and waste network resources. To eliminate as many of them as possible, we use a per-node bitvector with as many bits as different idxs the sparse matrix has. We call this bitvector *Idx Filter* and allocate it in the DRAM memory available in modern SNICs [4]. A bit in the Idx Filter is set by the RIG Units after a remote property arrives and is written to the host's memory. Also, before the RIG Units forward a PR for packetization, they check if the bit for the specific idx is set in the Idx Filter. If it is, the PR is redundant and is dropped. The PR is also dropped if another PR with the same idx is currently outstanding, but the requested property has not yet arrived. This is achieved with a request coalescing mechanism supported by the RIG Units.

3. Property Request Concatenation. This technique is grounded on the temporal remote destination locality (Section 3). When multiple fine-grained PRs generated close in time are directed to the same node, we concatenate them in a single network packet. For this, we introduce a low-level NetSparse network protocol implemented on top of RDMA. Our protocol has two layers: the *Concatenation* layer and the *PR* layer. The headers of the concatenation and higher-level networking layers can be shared across multiple PRs. Hence, the effective per-PR header overhead decreases, *increasing goodput*. PR concatenation is realized using a mechanism based on delay queues (Section 6.1). We augment SNICs and switches with special *Concatenator* units that implement this mechanism in hardware.

4. In-Switch Property Caching. To *share properties* fetched from remote racks within nodes of the same rack, we augment ToR switches with a hardware cache. We call it *Property Cache*. In contrast with P4-based switch caching mechanisms [42], the Property Cache is updated in the data plane, to accommodate the short-lived distributed sparse kernels. The Property Cache is placed in a new layer of *middle pipes*, that complement the existing ingress/egress pipes in Tofino-like switches (Section 6.2).

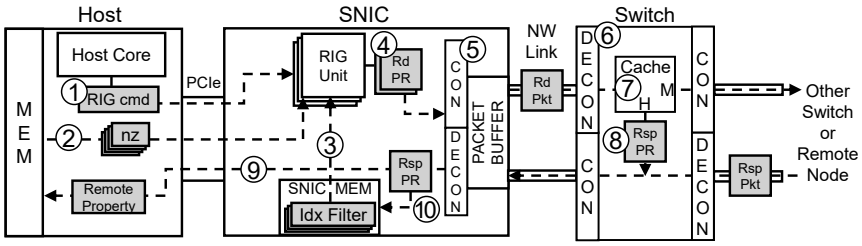


Figure 3: The lifetime of a Sparsity Aware (SA) Property Request (PR) with NetSparse.

Figure 3 shows how the lifetime of a PR changes with NetSparse in comparison with Figure 2. At step ①, one of the host cores sends a RIG command to one of the RIG Units of the SNIC. The RIG Unit then reads the appropriate idxs for a batch of nonzeros from the host's memory with DMA (step ②). At ③, for every idx, the RIG Unit checks the corresponding bit in the Idx Filter. If the bit is set, it means that the property for this idx has already been fetched by this or another RIG Unit. In this case, a PR is not generated and the idx is dropped. An idx is also dropped if a PR with the same idx is currently outstanding. This is determined through a coalescing mechanism before step ③ (not shown). For the non-dropped idxs, read PRs are generated ④. At step ⑤, multiple PRs may be concatenated in a single packet, which is forwarded to the network. When the packet reaches the ToR switch, it is de-concatenated ⑥ and the Property Cache is looked-up using the idx of each PR ⑦. For read PRs that hit, response PRs containing the requested properties are generated and forwarded back to the node ⑧. The generated response PRs can be concatenated with others to form response packets. When a response packet arrives back at the SNIC, it is deconcatenated, and the properties it carries are written to the host's memory ⑨. Finally, the Idx Filter is updated ⑩, marking the corresponding properties as already fetched.

On Property Cache misses, read PRs are concatenated into packets and forwarded further into the network. When a read PR reaches its destination, a remote RIG Unit loads the requested property from its host's memory and creates a response PR for the original node.

A RIG command completes when all the PRs for the non-dropped idxs of the batch are generated and all their responses are received at the requesting node. Parallelism increases when multiple commands are concurrently active in different RIG Units of an SNIC.

5 NIC-centric NetSparse Mechanisms

This section describes the NIC-centric NetSparse mechanisms: how the RIG operation is offloaded to the NIC and how redundant PRs are eliminated. We also describe the architecture of the RIG Units.

We assume an RDMA-capable SNIC with an NoC interconnecting the different SNIC components, including DRAM memory, packet buffer, existing RDMA Processing Units, and other accelerators or lightweight cores. This architecture, shown in Figure 4, is based on an AMD Pensando SNIC [4, 5]. However, our mechanisms are not tied to this type of SNIC. The RIG Units are attached to the SNIC's NoC and are connected directly to the packetization and buffering logic through hardware queues.

5.1 Offloading the RIG Operation to the NIC

Although SA communication can be expressed as RIGs, RDMA today lacks native support for this coarse-grained operation. In a naive SA execution, the host issues fine-grained RDMA reads with different destinations. For each of these reads, the host and the NIC need to communicate. This is an expensive process that leads to high software costs and limits the PR generation rate.

To eliminate these costs, we propose extending RDMA hardware with the new RIG Units, capable of performing RIG operations. A RIG Unit can be configured to operate as a client thread or as a server thread. Client threads receive RIG commands from the host's cores, generate the necessary read PRs for a coarse-grained batch of idxs, and inform the host when the operation is complete. Server threads respond to requests from the network by fetching properties from their host's memory, and generating response PRs.

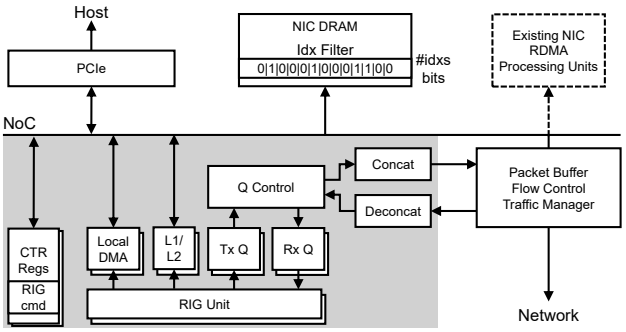


Figure 4: NetSparse SNIC, with our hardware extensions highlighted. Concat/Deconcat are discussed in Section 6.

Figure 4 shows a NetSparse SNIC, with our hardware extensions highlighted. The RIG Units have memory-mapped control registers to communicate with the host. A client thread begins its execution when the host writes a RIG command to a register. The command contains the host address that the client thread should read the nonzero idxs from, the host address to write the gathered remote properties, the number of idxs, and the size of a property. After the thread receives the command, it issues a local DMA transfer that copies the idxs from the host memory to a buffer in the RIG Unit. It then generates PRs and forwards them to a hardware queue (Tx Q) for packetization. When a response PR arrives from the network through an Rx Q, the property it carries is written to the host's memory at the appropriate location based on its idx. When all the properties have been received, the host is notified. The intermediate concatenator/deconcatenator modules are analyzed in Section 6.

The nonzeros processed by a node are grouped into batches, and multiple RIG commands are active concurrently in different RIG Units. This improves parallelism and enhances the overlapping of computation and communication. However, there is a tradeoff in the RIG batch size. A very small batch size can expose the communication overhead between the SNIC and the host core and can

limit the effectiveness of SNIC offloading. In fact, for a batch size of 1, the RIG command is very similar to a vanilla RDMA read. A very large batch size limits parallelism and can cause intra-node load imbalance between the different client threads. We explore this tradeoff in our evaluation.

5.2 Redundant PR Elimination

An Idx Filter bitvector is a software structure used to eliminate redundant PRs due to idx repetitions. It has one bit for every column in the sparse matrix and is allocated in the SNIC's DRAM memory (Figure 4). Given that modern SNICs can easily support 16GB of DRAM memory [67], they have enough memory capacity to fit Idx Filters for sparse matrices with 100 billion columns—well over the largest SparseSuite matrices. A bit is set in the filter when the property for the corresponding idx has been received and written to the host memory. The same filter is shared by all client RIG Units, which access the SNIC DRAM through a small coherent L1/L2 cache subsystem (and additionally through a shared LLC attached to the SNIC NoC, if there is one available, as in Pensando [4, 5]).

A RIG Unit also contains a private hardware *Pending PR Table*, which is a CAM that tracks the currently outstanding PRs issued by this particular RIG Unit. The table is updated in hardware when the RIG Unit issues a PR and when a PR response is received.

With these two structures, NetSparse eliminates redundant PRs. Specifically, after a client thread running on a RIG Unit reads an idx, it checks the Pending PR Table and the Idx Filter, to see if there is an outstanding or an already completed PR, respectively, for the same idx. If any condition is true, the PR is not issued—an event we call *coalescing* and *filtering*, respectively, for the two cases. Note that only PRs originating from the same RIG Unit can be coalesced. We made this design decision because implementing coalescing across RIG Units in a node requires synchronization support.

5.3 Architecture of a RIG Unit

Figure 5 shows the architecture of a RIG Unit in client mode. The RIG Unit contains: (i) the Logic Engine ①, which includes logic to check for idx coalescing and filtering, and to perform PR generation, and (ii) its memory structures ②, namely, the SRAM Idx and Rx Property Buffers, and the Pending PR Table. The figure also shows the RIG Unit's auxiliary structures: a DMA engine that accesses the host's memory ③, L1 and L2 caches to access the SNIC DRAM ④, and the Tx and Rx Queues to communicate with the network interface ⑤.

The DMA engine accesses the host's memory to fetch batches of idxs into the Idx Buffer. For each idx, the Logic Engine checks the Pending PR Table and the Idx Filter (through its LSQ and L1/L2 cache hierarchy) for redundant PR elimination. If the current idx is not eliminated, the Destination Solver calculates the destination node of the PR, and the local host's memory address to store the remote property when it arrives. Then, the PR Generator registers an entry in the Pending PR Table, creates a read PR, and pushes it to the Tx Queue. When the corresponding response PR is received at the Rx Queue, the fetched property is stored in the Rx Property Buffer. Then, the property is written to the host's memory through DMA, and the Idx Filter and Pending PR Table are updated appropriately.

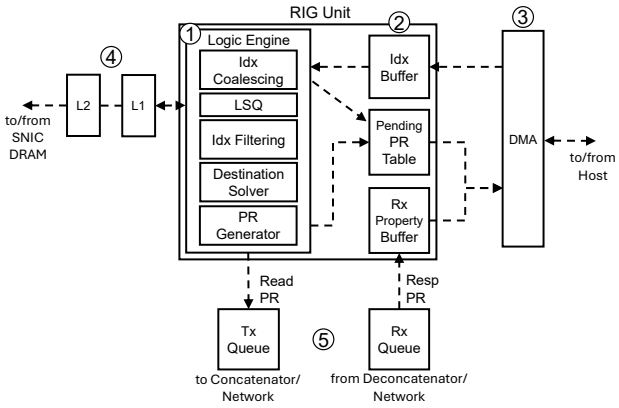


Figure 5: RIG Unit in client mode and auxiliary structures.

A RIG Unit does not stall while processing an idx. It processes multiple idxs in a pipelined manner. It can read a new idx from the Idx Buffer every cycle, and overlap the Idx Filter accesses, the DMA operations, and the communication with the network interface across multiple PRs. This overlapping hides latencies that prevent current software solutions from generating PRs at high rates. A RIG Unit only stalls if it has issued the maximum number of network requests that its Pending PR Table can track, or if its LSQ or Tx Queue are full.

The same RIG Unit can be reconfigured to implement a server thread as follows. First, the Rx Queue receives read PRs and the PR Generator generates response PRs that are pushed to the Tx Queue. Further, the Destination Solver calculates the local host's memory address where the requested properties reside. Finally, the Rx Property Buffer receives properties from the host and deposits them in the Tx Queue. The Idx Buffer and the Idx Filtering/Coalescing logic are unused. Incoming Read PRs are assigned to Server RIG Units dynamically through the Q Control module (Figure 4).

5.4 Host-Side Software Changes for RIG

In our design, RIG offload is exposed as a seamless extension to the existing RDMA-Verbs API [58] rather than as a separate library or a new system call. We introduce a new opcode *IBV_WR_RIG* alongside standard opcodes such as *IBV_WR_RDMA_READ* in the *ibv_send_wr* union. This requires only small changes to the *libibverbs* driver to recognize the new opcode and program the control registers of the RIG Units in the SNIC. Along with the opcode, the host passes all the other fields necessary for a RIG command (Section 5.1).

6 Network-centric NetSparse Mechanisms

We now detail the network-centric NetSparse mechanisms: Property Request (PR) Concatenation and In-Switch Property Caching.

6.1 Property Request Concatenation

6.1.1 Protocol. With NetSparse, multiple PRs of the same type (read or response PRs) that share the same destination can be concatenated into a single network packet. This is achieved by using our proposed application-level NetSparse network protocol (Figure 6). The protocol is structured into two layers: a Concatenation and a PR layer. We use RDMA as the base protocol before the NetSparse header; however, other alternatives are possible.

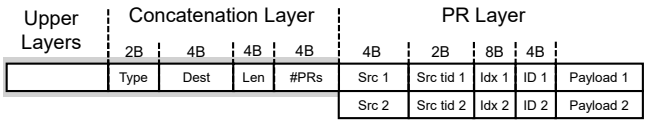


Figure 6: NetSparse 2-layer network protocol.

As shown in Figure 6, the concatenation layer header includes fields for the PR type (i.e., read or response), destination, property length (which is the same for all PRs in a given kernel), and the number of concatenated PRs. The PR layer header specifies, for each individual PR, the source node, source RIG Unit ID (i.e., tid), property idx, and request ID. A single NetSparse packet can encapsulate multiple PRs, along with their respective PR layer headers and payloads, as long as the packet remains within the maximum transmission unit (MTU) size limit. Importantly, NetSparse packets have a single destination and are not multicast. Hence, they do not alter the network's routing.

By concatenating PRs, the header overhead of the concatenation and upper layers is shared across multiple PRs. This results in a large reduction in header overhead. To see why, assume a header of 50 bytes for the upper layers. Without concatenation, each PR packet requires a header of 50+10+18 = 78 bytes. Thus N PRs in separate packets have a header of 78N bytes. With concatenation, a packet with N PRs has a header of 50+14+18N = 64+18N bytes. For even small values of N, this is a significant reduction in overhead, which improves goodput—particularly for small property sizes. In addition, the resulting reduction in the number of network packets reduces network contention and improves performance.

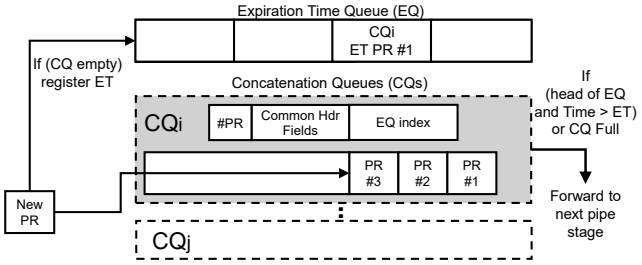


Figure 7: PR concatenation mechanism.

6.1.2 Mechanism. The concatenated PRs can come from the same or different threads, and from the same or different nodes. Concatenation happens transparently, without requiring the threads to synchronize. It occurs in the data plane, in both the ToR switches and the SNICs. We call these points *Concatenation Points*.

The concatenation mechanism is shown in Figure 7. A Concatenation Point has multiple *Concatenation Queues* (CQs), which are delay queues that collect PRs of the same type and destination. There is one CQ for reads and one CQ for responses per possible destination (in the worst case, for a total of 2(N-1) CQs, where N is the number of nodes). Each CQ is MTU sized (i.e., 1500B). When a PR arrives, it is pushed into the appropriate CQ, based on its type and destination. In a CQ, PRs wait to be concatenated. When a CQ gets full, the PRs it contains are concatenated and forwarded for packetization. However, the PRs in a CQ can also be forwarded for packetization before the CQ gets full. We describe this case next.

Each non-empty CQ is associated with an Expiration Time (ET), which is the time when its first PR entered the CQ plus a *DelayCycles* parameter. DelayCycles is the maximum allowed time that any PR can spend waiting to concatenate with others. If the ET of a CQ is exceeded, the PRs it contains are concatenated and forwarded for packetization.

To efficiently identify CQs close to expiration, we employ a circular queue called *Expiration Time Queue* (EQ). When a PR enters an empty CQ, the index of the CQ plus the computed ET are inserted at the tail of the EQ. The EQ is shown in Figure 7, where each entry has a pair $\{CQ_i$, ET PR #1$\}$. At every cycle, the hardware checks the entry at the head of the EQ for expiration—all the other CQs are guaranteed to expire later.

Beyond a set of PRs, each CQ holds some metadata. As shown in Figure 7, the metadata is the number of PRs currently in the CQ, the common header fields of the PRs in the CQ, and the position in the EQ that holds the entry for the CQ (i.e., the EQ index). The latter is needed in case the CQ gets full before its ET expires. In this case, the corresponding EQ entry needs to be cleared.

PR concatenation delays individual PRs for some cycles. This is tolerable within some limits. What we care about is when the whole kernel completes—not when individual PRs do. Typically, this delay is negligible compared to network round trip times.

PR deconcatenation breaks down a concatenated package into its component PRs. It also happens in the SNICs and ToR Switches. Its implementation is straighforward.

6.1.3 Implementation. Since (de)concatenation happens in the data plane, one approach would be to implement it using the P4 pipelines included in Tofino switches and Pensando SNICs. Unfortunately, the register arrays of the P4 pipelines cannot handle data types larger than 32 bits. As a result, the number of register accesses that would be needed per cycle for concatenation to work would quickly exhaust the number of register accesses that the hardware can support per stage and cycle. For this reason, we extend the switch/SNIC architecture with new hardware *Concatenator* and *Deconcatenator* modules. Their integration in the SNIC is shown in Figure 4. To integrate them in switches, we can follow an approach similar to Taurus [85] and implement them in separate chiplets. Networking chips such as Broadcom's Tomahawk [16] have already started moving towards a multi-chiplet direction.

P4 supports *externs* [23] to extend the language and provide operations not natively supported. To use (De)Concatenators, one approach is to introduce new dedicated *control-block* [85] P4 types in P4. Within these new control blocks, (De)Concatenator units can be invoked using the *Concatenate* and *Deconcatenate* constructs, similar to how hashing units and stateful memory (meters, counters, and registers) are made available in the Tofino chip [38, 39].

6.2 In-Switch Property Caching

To share properties fetched from remote racks within the nodes of the same rack, we augment ToR switches with a cache. Although prior work such as NetCache [42] has implemented caching in Tofino switches by utilizing the P4 register arrays and Match-Action Tables (MATs), such an approach would not work for sparse kernels. NetCache targets very long-running Key-Value Stores (KVS) and, due to P4 pipeline limitations, NetCache is periodically updated

with values through the control plane. Instead, an iteration of a distributed sparse-kernel is much shorter-lived than KVS. The control plane-based cache update rate is not sufficient. Hence, we include a new hardware cache in the switch called *Property Cache*, intended to be used in sparse supercomputers. Similar to the Concatenators, this is a new hardware module that could be invoked, for example, in P4 as a control-block using Lookup and Insert constructs. Before the kernel begins, the *Property Cache* is configured by the control plane and any data it contains is marked as invalid.

6.2.1 Switch Architecture. To support our technique, we modify the architecture of a switch to handle arriving packets as follows. An arriving read packet is deconcatenated and each of its PRs checks the Property Cache to see if it contains the desired property. If a PR finds its property (i.e., it hits in the cache), the PR becomes a response PR and its destination node is changed to be the node that issued the PR. Otherwise, no action is taken. Then, all the PRs (irrespective of whether they hit or missed in the cache) go through a concatenation step and are then issued to the correct output port.

An arriving response packet is deconcatenated and each of its PRs checks the Property Cache to see if it contains the corresponding property. If a PR finds the property, no action is taken. Otherwise, the PR's property is saved in the cache for future use. Then, all the PRs go through a concatenation step and are issued to the correct switch output port.

NetSparse supports this algorithm with high performance by adding a second crossbar to the network switch. The design is shown in Figure 8. The figure shows a switch with 16 input ports (leftmost part of the figure) grouped into four *ingress* pipes and 16 output ports (rightmost part of the figure) grouped into four *egress* pipes. Rather than having a single 16x16 crossbar, we add four more pipes in the middle and one additional 16x16 crossbar between the middle and egress pipes. Each middle pipe has a deconcatenator, a Property Cache, and a concatenator.

To prevent the access latency of the Property Cache from blocking the pipe, the cache itself is pipelined. Further, a shift queue (not shown) is used to host PRs while they check the cache. At every cycle, a new PR can enter this queue, without applying backpressure.

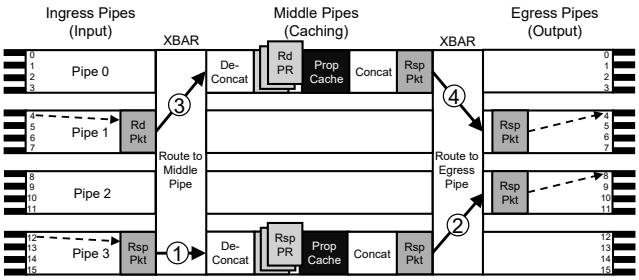


Figure 8: NetSparse switch with two example packet flows.

Figure 8 shows two example packet flows. First, a response packet arrives at Port 12 in ingress Pipe 3 and must be routed to Port 8 in egress Pipe 2. NetSparse first routes it to the middle pipe at the same offset as the ingress pipe (i.e., Pipe 3) ①, where it will update the Property Cache. In the middle pipe, the packet is first deconcatenated and then each of its PRs deposits its property in the Property Cache (unless it is already there). Then, all the PRs go through a concatenation step, and are then routed to their requested output port (Port 8 in egress Pipe 2) ②.

The second example is a read packet that arrives at Port 4 in ingress Pipe 1 and must be routed to Port 0 in egress Pipe 0. NetSparse first routes it to middle pipe at the same offset as its egress pipe (in the example, Pipe 0) ③. This is because middle Pipe 0 has the cache where prior PRs that can source the response would have saved the property. Indeed, for a pair of {response, read} PRs to communicate a property in a deterministic routing network, the egress port of the read must match the ingress port of the response.

In the example, as the read packet reaches middle Pipe 0, it is deconcatenated and each of its PRs accesses the Property Cache. Assume that only one PR hits (shown in the figure). That read PR gets the property from the cache, becomes a response PR and changes its destination to be its source node. Consequently, NetSparse routes it to the output port that matches its input port: Port 4 in egress Pipe 1 ④. Before reaching the egress pipe, however, the PR goes through a concatenation step. The read PRs that missed in the Property Cache remain unchanged, go through a concatenation step, and are routed to their initial output port (Port 0 in egress Pipe 0).

An alternative design that we considered was to place the Property Cache in the Egress Pipes. This approach does not need the middle pipes or the second crossbar. However, it disrupts the packet flow for both read and response PRs.

First, for read PRs, if a read PR finds its property in the Property Cache and needs to change its destination node, it must be recirculated to an input port of the switch. It is then routed through the crossbar a second time to reach the new destination. Second, for response PRs, a response PR must be mirrored: it must be routed to both its destination egress pipe and to the egress pipe with the same index as its ingress pipe (since this is the egress pipe that contains the correct Property Cache to update).

6.2.2 Property Cache Architecture. The Property Cache is a set-associative hardware cache that is accessed with the property idx bits (i.e., the cid bits of the sparse matrix) and returns the property associated with the target idx. In practice, different kernels may have different property sizes. Hence, we envision the Property Cache to support different property sizes and be configurable. Specifically, before a kernel executes, the control plane configures the Property Cache for the single property size to use.

To support this configurability, the Property Cache is implemented in *Segments*. To see how it works, assume that it needs to support property sizes of {16B, 32B, 64B, ... 512B}. In this case, the Property Cache is designed with 32 segments, each able to provide 16B of property data. If the cache is configured for 16B properties, a cache access will only activate one segment, which will provide the 16B. If the cache is configured for 32B properties, a cache access will activate two adjacent segments, each of which will provide 16B. If the cache is configured for 512B properties, a cache access will activate all 32 segments, which together will provide 512B.

Figure 9 shows the Property Cache described. Because it has 32 segments, each with a certain number of sets and ways, the property idx bits include a 5-bit *Segment bits* field, in between some Set bits and the Tag bits. These Segment bits will identify the segment(s) with the desired property value. The Segment bits are routed to a *Segment Selector* that is configured with a *Mode* and generates

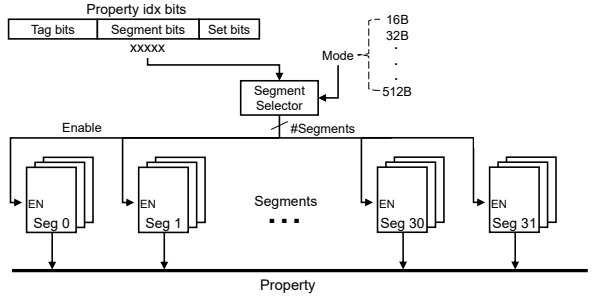


Figure 9: Segmented design of the Property Cache.

a bitmask of 32 *Enable* bits to enable the desired segment(s). For example, if Mode is 16B, the bitmask has only 1 Enable bit set; if Mode is 32B, the bitmask has only 2 adjacent Enable bits set; if Mode is 512B, all the Enable bits are set. The Set bits and Tag bits are routed to all the segments, and prompt the segments that are enabled to dump the property bits onto an output network.

As an example, assume that the Property Cache is configured in 32B Mode. This means that the Segment Selector will ignore the LSB of the Segment bits. Assume that the Segment bits are 1110X. In this case, we are looking for the pair of segments that are one before last. Hence, the Segment Selector will set the Enable bits to 0...01100. Overall, with this Property Cache design, the whole cache capacity can be utilized regardless of the property size.

If the property size of the kernel is larger than the largest one (S_{max}) supported by the Property Cache, the input property array can be tiled into chunks, so that the maximum property size in each chunk does not exceed S_{max}.

7 Other Considerations

1. Network Packet Loss. NetSparse is designed for a lossless RDMA-based networking environment (i.e., Infiniband-like), which applies backpressure when network queues get full. This limits packet losses to hardware failures mostly. Detection of packet losses is accomplished by setting up a watchdog timer when a RIG Unit initates a RIG operation and resetting it when the operation terminates. If the watchdog timer times-out, the RIG operation is considered failed. Then, the host is informed, and the whole buffer in the host's memory to which the RIG Unit has possibly written partial results is discarded.

2. Scalability of the Concatenation Mechanism. Since our concatenation mechanism allocates one Concatenation Queue (CQ) for each possible destination node, the total SRAM requirement scales with the total number of nodes in the network. We show in our evaluation that this does not cause issues for 128 nodes. However, it could cause excessive memory overhead and low queue utilization under large-scale deployments. To solve this issue, a possible approach is to *virtualize* the CQs. With this approach, each concatenation point includes a fixed number (i.e., independent of the number of nodes) of smaller sub-MTU sized (e.g., 128B) "physical" CQs. These physical CQs are dynamically assigned on demand, whenever a PR for a new destination node appears or a physical CQ for a desired destination node is found to be full. Multiple physical CQs with PRs with the same destination are linked into a "virtual" CQ. If the total occupancy of a virtual CQ reaches the MTU, the contents of its physical CQs are concatenated together into a single packet that is issued into the network, and the corresponding physical CQs are freed.

3. Deployment. NetSparse is backward-compatible with state-of-the-art existing RDMA-capable HPC infrastructure. NetSparse headers appear as payloads of the RDMA layer, and thus they are opaque to the existing network routing infrastructure. However, to be able to generate and decode NetSparse packets, at a minimum, the end-host SNICs need to include RIG Units that can handle RIG commands. Both the source and destination nodes should be RIG-capable. We believe that RIG operations can be useful for other throughput-oriented kernels with fine-grained irregular accesses. NetSparse switches can be deployed incrementally.

8 Methodology

8.1 Software Baselines

We compare NetSparse to an SU and an SA software-only baseline. We make idealistic assumptions for both of them. For the SU baseline (*SUOpt*), the communication time is assumed to be equal to only the time needed for a single node to receive all of the data bytes needed from the network at 100% line bandwidth utilization and without any header overheads. There is no network or SNIC latency and all nodes perfectly overlap their message reception. This is the optimal performance limit of the SU approach.

For the SA baseline (*SAOpt*), we assume an environment where some of the NetSparse mechanisms are implemented in software. Specifically, we augment the SA algorithm with the Conveyors framework [59] and implement in software the three NetSparse mechanisms that have a reasonable design in software: communication batching, message concatenation, and filtering.

To mimic the first two mechanisms, we rely on Conveyors. With Conveyors, each core aggregates batches of PR idxs with the same destination into a software buffer, and issues them to the network as a single message. Since RDMA does not support one-sided Remote Indexed Gathers, Conveyors use two-sided communication (i.e., send and receive messages) that require synchronization between cores in different nodes. Note also that sharing the headers across PRs for message concatenation is only possible if they originate from the same node. Finally, to mimic filtering, we optimistically assume that the software perfectly pre-filters redundant PRs offline without any preprocessing or runtime cost. The other NetSparse techniques (concatenation of messages originating from different nodes and in-network caching) are not feasible with software. Further, we also assume that there is no network or SNIC latency. Overall, for SAOpt, we only account for the software overheads of PR generation, book-keeping, synchronization, and buffering.

To isolate the software overheads of SAOpt, we measured the rate of data transfers under an ideal scenario: all communication happens between cores of the same node and is perfectly balanced. Specifically, we evaluated such a setting on a high-performance NCSA Delta node with 64 out-of-order cores. By dividing the total traffic by the execution time, we computed the rate of data transfers. Then, we divided this rate by the line rate, ignoring headers, to calculate the goodput as a percentage of the line rate. The results are shown in Figure 10 for two different property sizes K (Section 2) and different numbers of cores in the node. We observe that the

(optimistic) goodput scales almost linearly with the number of cores. We see that the maximum achievable goodput—even with 64 high-performance cores, perfectly balanced communication load, and no network overheads—is far from the optimal 100%. From this experiment, we measure the total software overhead per PR. We then use this value to calibrate our simulations of SAOpt in the evaluation section.

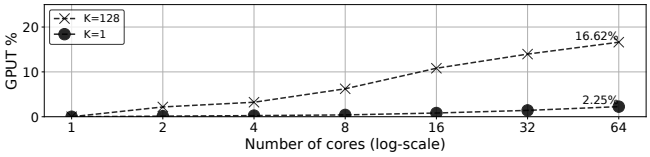


Figure 10: Ideal SAOpt goodput vs number of cores in a node.

In our evaluation, in SAOpt, we use all 64 CPU cores in a node to support the cross-node communication; in NetSparse, we use a single CPU core per node to control the RIG Units of the SNIC. We do this to highlight the capabilities of NetSparse when most CPU cores are not available to speed-up communication.

8.2 Simulated System and Benchmarks

To evaluate NetSparse, we use the SST Simulator [73], with the Merlin [33] component library for the network and switches, and DRAMSim3 [53] for DRAMs. We modify the rdmaNIC component of the simulator to model a NetSparse-augmented Pensando-like SNIC. Further, we augment the Merlin switches to model a Tofino-like architecture with NetSparse extensions. Our communication experiments use communication patterns that are representative of the SpMM, SDDMM and SpMV kernels. For experiments including computation, we run SpMM with SPADE accelerators [29] integrated with each host, again simulated with SST.

Table 5: System Parameters.

Cluster	
Topology	128 nodes; 8 racks; 16 nodes/rack; Leaf-Spine
Node	
Node	1 socket; 64 cores; 2.2GHz; DDR 256GB, 270GB/s
In-Node Accel.	128 SPADE PEs; 1GHz; HBM 64GB, 800GB/s
PCIe	Gen6; 256GB/s; 200ns one-way latency
SNIC	
General	AMD Pensando-like; 2.2GHz DDR Memory 16GB, 64GB/s; 16MB LLC
RIG Units	32 RIG Units; 16 32KB L1s; 16 128KB L2s; 64/256-entry LSQ/Pending PR Table; 4KB Property Buffer/Idx Buffer
Network Interface	2MB RxBuffer/TxBuffer; 400Gbps; 1500B MTU
Network	
Bandwidth	400Gbps per link
Zero-load Latency	Switch: 300ns; Network link: 450ns one-way; Intra/inter-rack RTT: 2.4us/5.4us
Switches	Intel Tofino-like; 32 ports x 400Gbps; 8 pipes, 2GHz; 96MB Packet buffer; NetSparse extensions only in ToR switches
NetSparse Header	Upper/Concat/PR-layers: 50B/12B/18B
Concatenation	512KB SRAM per switch pipe and per SNIC; NIC/Switch concat. delay cycles: 500/125; Deconcatenation has no delay cycles;
Property Cache	32MB total per switch; 16/512B min/max cache line; 32 segments; 16-way; 16 cycles lat; LRU

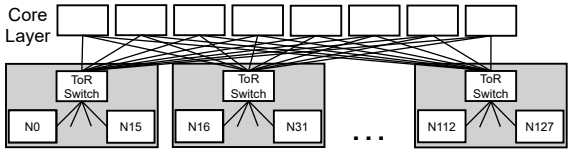


Figure 11: Simulated system topology.

We simulate a 128-node machine with 400Gbps links. Table 5 shows the system parameters. The system has a Leaf-Spine network topology, with racks of 16 nodes (Figure 11). We assume an Infiniband-like environment which applies backpressure when network queues get full. Packets are routed deterministically. In the SNIC, each pair of RIG Units shares a single L1 and L2 cache (typically, a client and a server unit).

We use 5 large sparse matrices from the SparseSuite [25] collection as benchmarks (Table 6). These matrices come from different domains such as web-crawls, road networks, and scientific applications. We evaluate them for properties with K={1,16,128} (i.e., 4B, 64B, and 512B). We use a RIG batch size of 32k nonzeros for arabic, queen, and stokes, and 8k for europe and uk. We also analyze the performance sensitivity to other values.

Table 6: Benchmark sparse matrices.

Matrix	arabic-2005	europe_osm	queen_4147	stokes	uk-2002
Rows (M)	23	51	4	11	19
NNZ (M)	640	108	317	350	298

8.3 NetSparse Hardware Overheads

NetSparse requires new hardware extensions to the SNICs and switches. To evaluate their area and power impact, we implemented the main logic structures (i.e., RIG pipelines and the Concatenators) in RTL using System Verilog. We synthesized our designs targeting a 45nm process with the Synopsys Design Compiler [36] and the open-source FreePDK45 cell library [84]. We verified that our designs easily meet timing for a frequency target of 1.5GHz at 45nm. Thus, a frequency of 2.2GHz for more modern 7-10nm process technologies is very reasonable [83]. For the caches, SRAMs, and CAMs in the NetSparse extensions, we used CACTI [8], and also validated that they meet timing. We scale down from 45nm to 10nm using the process scaling equations from [83]. We present our area and power evaluation in Section 9.5. Dynamic power is reported under maximum activity.

9 Evaluation

9.1 Performance and Scalability Analysis

In this section, we analyze the performance and scalability of *NetSparse*. We start by comparing the time taken by the inter-node communication in NetSparse, SAOpt, and SUOpt assuming that computation has no cost. Figure 12 compares the communication speedup of NetSparse and SAOpt over SUOpt for different property sizes. SUOpt is favored by small properties (K=1), since the network traffic caused by redundant transfers is lower. For this reason, we see that the speedups of SAOpt and NetSparse increase for larger properties. We observe that NetSparse outperforms both baselines for all matrices and all property sizes. In contrast, SAOpt performs worse than SUOpt for stokes (all K values) and arabic (for K=1). The performance gain of NetSparse over SUOpt correlates directly with

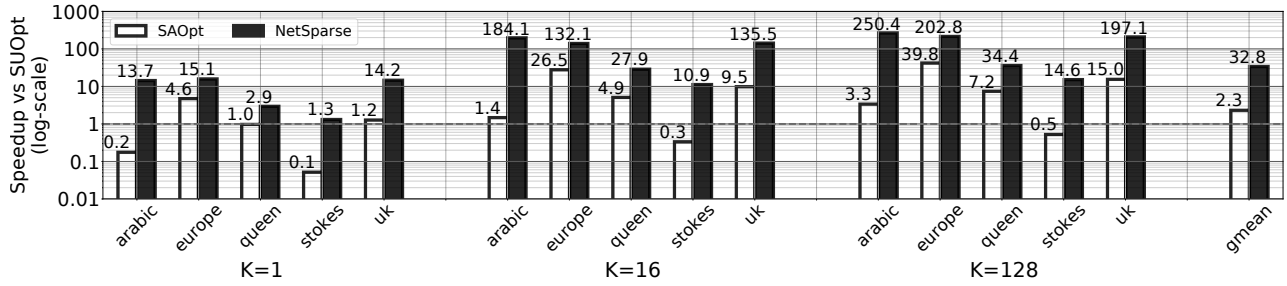


Figure 12: NetSparse and SAOpt communication speedups over SUOpt for the 128-node system.

the number of redundant transfers SUOpt issues for each matrix (Table 1). On average (i.e., gmean), Netsparse outperforms SUOpt by 33× and SAOpt by 15×.

Table 7 shows key statistics and reveals the reasons behind the communication speedups of NetSparse. We focus on the node that takes the longest to finish (*Tail Node*), since it determines the execution time. We see that *NetSparse* mechanisms work. First, PRs get filtered and coalesced, as revealed by the filtering and coalescing (F+C) rate. Second, PRs get concatenated, as shown by the high number of PRs per packet. Third, we have hits in the Property Caches. For europe, the filter and cache hit rates are low because the matrix is very sparse; it is rare to find PRs with the same idx.

Table 7: Performance statistics for the tail node in NetSparse, and comparison to SUOpt and SAOpt (K=16).

	F+C Rate	Avg #PR/ Pkt	P_{Cache} Hit Rate	Gput	Line Util.	-Trfc vs SU	Gput SA	-#PR vs SA
arabic	97%	5.7	26%	35%	65%	283x	1%	3.8x
europe	8%	4.5	5%	37%	70%	188x	10%	1.3x
queen	95%	19.6	50%	40%	66%	42x	11%	1.1x
stokes	90%	12.1	6%	38%	64%	17x	8%	4.4x
uk	61%	17	30%	30%	50%	271x	9%	2.6x

The main reason for the NetSparse speedup over SUOpt is the significant decrease in network traffic (Column 6 of Table 7). For example, arabic has 283x less traffic in NetSparse than in SUOpt. When compared to SAOpt, NetSparse eliminates software overheads and minimizes CPU involvement and synchronization. As a result, it raises the goodput to healthy levels, as can be seen by comparing Column 4 to Column 7. In addition, NetSparse also decreases the number of PRs relative to SAOpt due to more effective filtering (Column 8). For example, arabic has 3.8x fewer PRs in NetSparse than in SAOpt. This is because the Conveyors framework included in SAOpt assigns threads in the same node to different ranks and cross-rank filtering is not possible. SAOpt shows lower goodput for arabic in comparison with other matrices. This is due to intra-node load imbalance in the tail node, which gets magnified with SAOpt.

Next, we evaluate how the substantial communication speedups translate to end-to-end performance improvements. For this, we execute SpMM on the 128-node system, using the SPADE accelerators for computation. For communication, we use SUOpt, SAOpt, or NetSparse. SPADE does not support SpMM with K=1, so we exclude this property size from this experiment.

Figure 13 illustrates how the system strong-scales from 1 to 128 nodes with different communication approaches. In other words, it shows the end-to-end speedup of the 128-node system over the execution on a single node. We also include, as a performance limit, the ideal speedup of a hypothetical system without any communication. Such speedup is shown with dashed lines.

Unsurprisingly, in the presence of hardware-accelerated computation, software solutions for communication are not enough. Due to the huge communication inefficiencies of SUOpt, the 128-node system is, on average, slower (i.e., its speedup is 0.7×) than the single-node system—which does not require communication. With SAOpt, the 128-node system is only 3× faster than the single-node one. With NetSparse, hardware-accelerated computation is paired with hardware-accelerated communication, enabling end-to-end scalability. The 128-node system is 38x faster than the single-node one. This is more than half of the speedup of a hypothetical ideal system without any communications overheads, which reaches a speedup of 72.

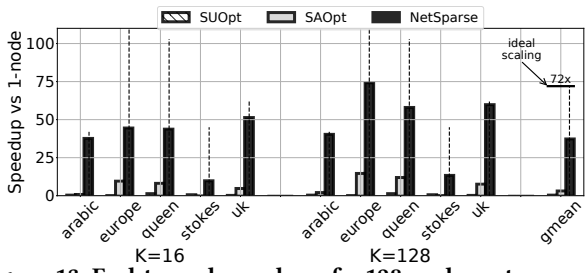


Figure 13: End-to-end speedup of a 128-node system over a single-node system for SpMM.

Figure 14 compares the communication to computation ratios for SAOpt and NetSparse. Note that, in practice, communication and computation (partially) overlap. SAOpt is dominated by non-hardware accelerated communication. With NetSparse, communication is accelerated and becomes comparable or faster than accelerated computation for arabic, queen, and uk. For europe and stokes, communication has still non-negligible potential for improvement. An extra improvement would bring the end-to-end performance of those matrices closer to the ideal scaling, displayed in Figure 13. We discuss how this remaining potential could be exploited later.

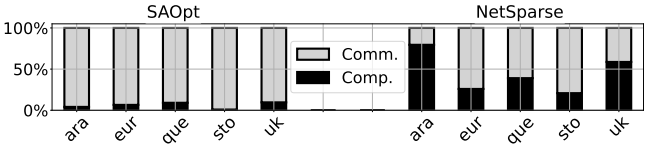


Figure 14: Communication/computation breakdown.

Table 8: Ablation analysis vs SUOpt.

	arabic (denser)								
	K=1			K=16			K=128		
Optim.	Spd	-Trfc	Gput	Spd	-Trfc	Gput	Spd	-Trfc	Gput
RIG	0.2	3.1x	0.4%	1.8	5.3x	22.9%	3.6	8.5x	39.4%
Filter	3.4	17.4x	0.8%	34.2	63.2x	22.2%	78.7	109.0x	56.7%
Coalesce	8.4	34.6x	0.9%	88.0	118.5x	23.8%	184.8	279.9x	54.0%
Conc$_{NIC}$	12.6	80.8x	1.1%	129.1	191.2x	34.0%	184.2	310.7x	52.6%
Switch	13.7	115.5x	0.9%	184.1	283.4x	35.2%	250.4	398.0x	55.1%

	europe (sparser)								
	K=1			K=16			K=128		
Optim.	Spd	-Trfc	Gput	Spd	-Trfc	Gput	Spd	-Trfc	Gput
RIG	7.4	8.8x	2.3%	82.8	99.1x	26.2%	176.0	248.0x	55.7%
Filter	7.5	8.9x	2.4%	84.8	100.8x	26.4%	175.5	252.2x	54.7%
Coalesce	8.1	9.6x	2.3%	91.3	108.7x	26.4%	190.3	272.1x	54.9%
Conc$_{NIC}$	14.1	19.1x	4.1%	122.1	159.6x	35.2%	197.8	292.9x	57.1%
Switch	15.1	24.8x	4.0%	132.1	188.4x	36.5%	202.8	301.5x	58.4%

9.2 Ablation Analysis

In Table 8, we perform an ablation study. We progressively apply each *NetSparse* mechanism, and show the resulting communication speedup (Spd), tail node traffic reduction (-Trfc), and tail node goodput for different property sizes. Speedups and traffic reductions are calculated with respect to SUOpt. We use arabic, which is denser, and europe, which is sparser, to assess the impact for different sparsity patterns. We now discuss the effect of each optimization.

First, we use the SNIC RIG Units. Although not shown in the table, they improve the line utilization and, as a result, the goodput compared with SAOpt. Since RIG operations are sparsity-aware, they naturally lead to traffic reductions vs SUOpt (e.g., Columns 5 and 8). Although they are the most impactful optimization for europe, arabic still has a lot to gain after this optimization.

Next, by additionally applying *Filtering and Coalescing*, redundant PRs are eliminated and traffic is further reduced, especially for arabic. For europe, filtering and coalescing are less effective, since as explained, it is rarer to find PRs with the same idx for this matrix. With NetSparse, filtering and coalescing do not introduce software overheads and, thus, do not hurt the goodput.

We then apply *Concatenation at the NIC*. Concatenation decreases both network traffic and increases goodput by decreasing the per-PR header overhead. Naturally, concatenation is more effective when K is small (K=1,16). It benefits both matrices.

Finally, we include the NetSparse switch, with the Property Cache, concatenators and deconcatenators. The NetSparse switch decreases traffic for two reasons: cross-node concatenation is now possible, and some PRs are now served by the Property Cache. For the sparser europe, concatenation is the main reason: notice that there is comparatively less traffic reduction for K=128, which benefits less from concatenation. For arabic, which has a higher cache hit rate (Table 7), both reasons contribute to the traffic reduction.

9.3 Sensitivity Analysis

Next, we perform a sensitivity analysis on how different values of the NetSparse parameters affect performance for N=128 and K=16.

Figure 15 shows the speedup of the execution with different RIG batch sizes (i.e., the number of nonzeros in a RIG command), over the one with a batch size of 16k. The values that we have used by default are 32k for arabic, queen, and stokes, and 8k for europe and uk. As explained in Section 5, a small batch size can introduce software overhead, while a large one can introduce SNIC thread load imbalance. This explains why, for many matrices, the best batch size is not at the extremes. We also see that the optimal choice is input-sensitive: it is not the same for all the matrices, and sometimes is different than the one we used.

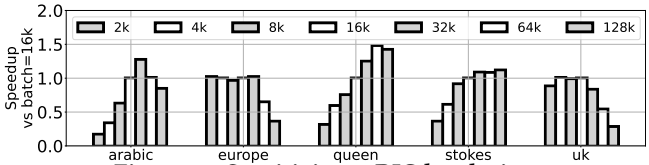


Figure 15: Sensitivity to RIG batch size.

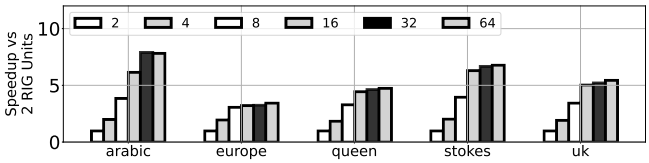


Figure 16: Sensitivity to number of RIG Units.

Figure 16 shows how performance scales with varying numbers of RIG units, using a 2-unit configuration (1 server + 1 client) as the baseline. Speedups grow until 32 units (our default design point), after which no matrix achieves significant additional gains.

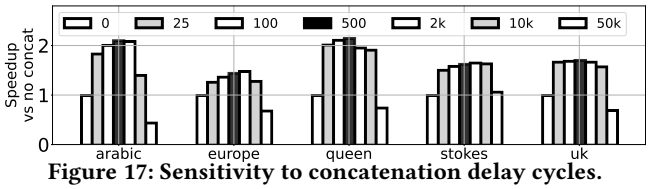


Figure 17: Sensitivity to concatenation delay cycles.

In Figure 17, we show the effect of the concatenation delay cycles. We sweep the maximum cycles a PR can be delayed in CQs from 0 to 50000, and present the speedups over no concatenation. The figure refers to the SNIC delay cycles, but the switch delay cycles are also scaled. Our default design has 500 cycles. Initially, as the cycles increase, performance increases due to more PRs getting concatenated. However, when the delay gets too large, performance gets worse than with no concatenation. Notice how concatenation effectiveness correlates with the temporal remote destination locality property (Table 4). Queen has the highest destination locality and gets the most benefit. In contrast, europe gets the least benefit.

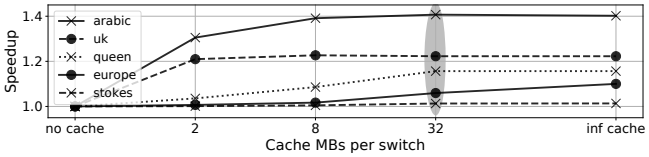


Figure 18: Speedup vs no-cache for different cache sizes.

Finally, we consider the impact of the Property Cache. Figure 18 shows the speedups vs no-cache as the cache size increases from 0 to infinite. The size we use by default is 32MB per switch. We see that caching can provide up to 40% improvement for arabic, while it does not improve stokes, regardless of the cache size. Also, we see that the cache size we chose is sufficient for most matrices, despite the fact that they are among the largest in SparseSuite.

9.4 Remaining Performance Potential Analysis

Our results in Figures 13 and 14 suggest that matrices like stokes and europe could benefit from further communication improvements. We started the paper by revealing a large performance potential in Section 3 and then we introduced NetSparse to exploit some of this potential. Thus, two natural questions are: (1) how close did we get to the optimal communication performance, and (2) how could any remaining performance potential be exploited.

Consider the europe matrix. While the optimal traffic reduction vs SU is 582x (Table 1), we achieved 188x in Table 7. Further, in Table 1, although the line utilization is 70%, the goodput is only 37%. In summary, although we accelerated communication for europe by 132x (Figure 12 for K=16), there is still room for further gains. We now discuss the inefficiencies that prevent *NetSparse* from achieving the full potential.

A first reason is that concatenation in NetSparse is imperfect. In the best case, NetSparse reduces the effective header bytes to 18B per PR—a non-negligible overhead for fine-grained properties. A second reason is that the statically-selected values for the NetSparse parameters (e.g., the RIG batch size) are often nonoptimal. Future work could investigate techniques to dynamically adjust NetSparse parameters such as the RIG batch size.

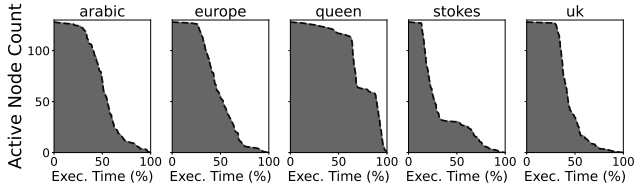


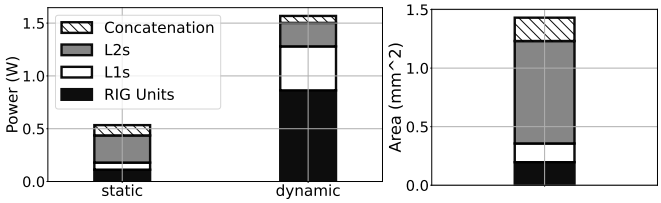
Figure 19: Inter-node load imbalance assuming no computation costs.

Another key reason is inter-node communication imbalance, which leads to suboptimal traffic reduction. Table 1 displayed the potential for *average traffic* reduction. However the NetSparse tail node, which determines performance, typically receives more than the average traffic. This limits the potential for execution time reduction. The problem is shown in Figure 19, which illustrates the number of active nodes as the execution progresses, assuming no computation costs. It reveals that, almost all matrices (except queen) have significant imbalance in communication. This imbalance is not a consequence of the NetSparse hardware, but of the way the sparse matrix is partitioned across nodes. To maximize the performance with fast sparse networking hardware, future work should rethink partitioning methods and algorithms.

9.5 Hardware Overhead Analysis

This section discusses the area and power overheads of NetSparse in SNICs and switches.

1. SNIC overheads. The main overheads are the RIG Units, their L1 and L2 caches, and the Con/De-concatenation blocks (Table 5). For a 10nm process, Figure 20 shows our estimated static power, (maximum) dynamic power, and area for each structure in an SNIC. We see that the combined (worst-case) power is 2.1W and the area is 1.43mm^2. Given that a Pensando Elba chip with a 200Gbps network link consumes up to 50W [27], the power overhead of the NetSparse extensions at maximum activity is 4% and, when idle, 1%. Regarding the area overhead, using the Elba die floorplan [78] and the relative area that the 16 ARM Cortex A-72 block [20] consumes, we extrapolate a conservative die size of 300 mm^2 for the SNIC. Hence, the NetSparse area overhead is only 0.5%. The total storage overhead of all the NetSparse extensions is only about 3.5MB.



(a) Power **(b) Area**

Figure 20: Power and area breakdown of the SNIC extensions.

The figure also shows that the L2s are the dominant source of area and static power, while the RIG Units account for most of the dynamic power. Further, Table 9 shows the contribution of each of the RIG Unit structures (Figure 5) to its area. The most area-hungry structure is the Pending PR Table, which is a CAM structure that book-keeps PRs pending in the network.

Table 9: Contribution of each structure to RIG Unit area.

Structure	Idx Buffer	Pend. PR Table	Prop. Buffer	LSQ	Rest
Area	12%	53%	12%	10%	13%

2. Switch overheads. The extensions in the ToR switches are the Property Caches, the Con/De-concatenation blocks, and the second crossbar. We estimate the area of the caches and concatenators to be 21.3 mm^2 and 1.5mm^2, respectively, and their combined power about 10W. Given a power consumption of 270W for a Tofino2 switch [2], the caches and concatenators add a 4% overhead. For area, given that modern high-performance networking switches are large ASICs that can reach 700 mm^2 [21, 22, 80], we estimate the area overhead for caching and concatenation to be about 3.2%.

Estimating the impact of the second crossbar is harder. Intel reports that the packet buffer, QoS circuitry, and ingress-to-egress routing logic account for a combined area of 25% in a Tofino2 switch [2]. However, a large fraction of this area is likely consumed by the SRAMs of the packet buffer. In our design, we have used input-queued crossbars [44] with virtual output queues [60] and split the packet buffer between the two crossbars without duplicating it. Further, the literature [19, 22] reports that a stand-alone (i.e., without cross-point buffering [74]) 32x32 switch crossbar like the one we use can be realized in less than 5mm^2 (i.e. 1% of the switch area). Nevertheless, due to limitations in publicly available data, we can only place the area overhead of the extra crossbar and inter-pipe routing at a range of 1–15%.

We believe this switch overhead is justifiable for inter-chip networking of sparse accelerators in high-value sparse applications. Dense accelerator clusters like the TPU supercomputer [43] already have their own custom networking.

9.6 Performance Analysis for Other Settings

We now evaluate NetSparse for other system settings.

1. Using NetSparse when Computation is Performed by a CPU rather than an Accelerator. In this section, we investigate the effect of replacing the SPADE accelerators with CPU cores for computation. For this, we calibrated our simulator using SpMM runs on two Intel Sapphire Rapids CPU servers [10]: a 48-core with DDR, and a 56-core that also includes HBM [35] (with similar bandwidth to the one modeled for the SPADE PEs). Measurements were taken using the optimized MKL Inspector-Executor routine [40].

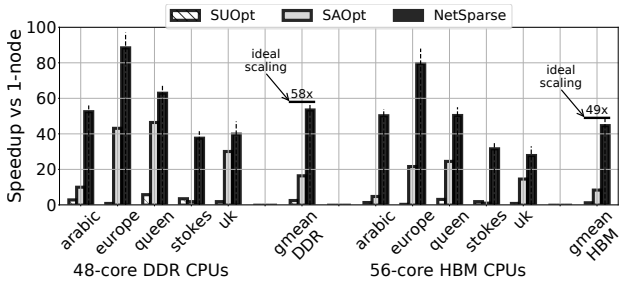


Figure 21: End-to-end speedup of a 128-node system over a single-node system for SpMM using CPUs for computation.

Figure 21 shows the end-to-end speedup of the 128-node system over the execution on a single node for SpMM (for K=128) using CPUs for computation. The figure is organized as Figure 13, with bars for SUOpt, SAOpt, and NetSparse, and dashed lines for a hypothetical system without any communication.

We see that NetSparse significantly outperforms both SUOpt and SAOpt, and approaches ideal scaling. The scalability of SAOpt for the DDR setting is still disappointing, although less than in Figure 13. This is expected, as communication inefficiencies are less exposed when computation is not accelerated as much. Since SpMM is very sensitive to the memory bandwidth, SPR+HBM accelerates local computation more than SPR+DDR. Hence, communication inefficiencies become more pronounced, and the scalability of SUOpt and SAOpt drops. On average, across K=128 and K=16 (not shown), the 128-node DDR system becomes 2.6x, 13x, and 53x faster than the single-node one with SUOpt, SAOpt, and NetSparse, respectively. The same numbers for the 128-node HBM system are 1.4x, 7x, and 42x. Overall, even with CPU computation, NetSparse delivers much higher speedups than SUOpt and SAOpt.

2. Using Other Network Topologies. We simulate machines with two other network topologies. One is HyperX [3] with 32 switches organized in a 3D 4x4x2 topology, with 4 hosts per switch, and 4 cross-switch links in every dimension (i.e., HyperX width of 4). The other is Dragonfly [45] with 32 switches divided into 4 groups, 4 hosts per switch, 4 inter-group links, and minimal routing. These 2 network configurations have similar bisection bandwidth as our original Leaf-Spine one but have a higher diameter. We evaluate all networks with the same per-link latency.

Figure 22 shows the communication speedup of NetSparse over SUOpt for the three different network topologies, for 128 nodes and K=16. We see that the performance of NetSparse remains high for all three networks. One notable exception is stokes with HyperX, where the speedup drops by more than 2x compared to the Leaf-Spine network, for which we originally designed NetSparse. This performance degradation is due to the higher number of hops in the HyperX network.

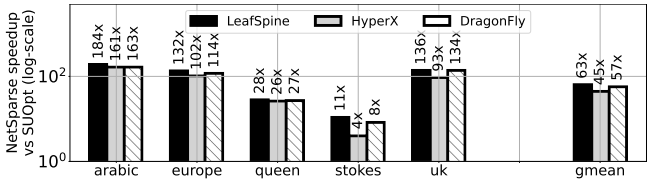


Figure 22: Communication speedup of NetSparse over SUOpt for different network topologies.

10 Related work

In-Network Computing: Modern smart networking hardware such as SNICs [4, 37, 66, 68, 87, 88] and programmable switches [13, 38, 39, 89] have enabled a wide variety of in-network computing applications. Such applications range from dense machine learning [50, 54, 69, 75] and Key-Value Stores (KVS) [42, 56], to coherence, persistence, and consistency [47, 48, 52, 71, 76]. In-network caching has been proposed before for KVS [42, 56]. However, sparse computations differ significantly from both KVS and dense machine learning. Flare [26] introduces a switch design that supports collective reduction operations of sparse data, which is useful for accumulating sparse gradients during ML training. To the best of our knowledge, *NetSparse* is the first work to introduce in-network hardware extensions for general distributed sparse kernels.

Hardware and Software for Sparsity: The importance of sparsity in computing has motivated both hardware and software research. On the hardware-front, accelerators [28, 29, 31, 32, 41, 55, 57, 61, 81, 82] aim at optimizing sparse computations on a single node. At the same time, a variety of SA [34, 62, 72], SU [6, 7, 9, 77] and hybrid [11] software efforts aim at scaling sparse computations across multiple nodes. *NetSparse* proposes hardware mechanisms that target the inefficiencies of software-based sparse communication.

11 Conclusion

This paper introduced novel hardware mechanisms to improve communication in distributed sparse computations. Our proposal, called *NetSparse*, consists of four mechanisms: remote indexed gather in the SNIC, filtering of redundant requests, concatenation of multiple requests into a single packet, and hardware caches in switches to store fetched data from remote racks. By reducing the network traffic, and improving the bandwidth utilization and goodput, *NetSparse* achieves substantial speedups over software solutions. In future work, we plan to extend NetSparse to apply to kernels with more than one sparse data structures, such as SpGeMM.

Acknowledgments

This work was supported by NSF with grants CCF 2107470, CCF 2316233, CAREER Award 2521510, and Graduate Research Fellowship DGE 21-46756; by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA; and by the IBM-Illinois Discovery Accelerator Institute. This work used the Delta system at the National Center for Supercomputing Applications through allocation ELE250007 from the Advanced Cyber infrastructure Coordination Ecosystem: Services Support (ACCESS) program, which is supported by National Science Foundation grants 2138259, 2138286, 2138307, 2137603, and 2138296. The authors thank Ariful Azad for his feedback during the early stages of this work.

References

[1] Matthew Adiletta, Jesmin Jahan Tithi, Emmanouil-Ioannis Farsarakis, Gerasimos Gerogiannis, Robert Adolf, Robert Benke, Sidharth Kashyap, Samuel Hsia, Kartik Lakhotia, Fabrizio Petrini, Gu-Yeon Wei, and David Brooks. 2023. Characterizing the Scalability of Graph Convolutional Networks on Intel® PIUMA. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Raleigh, North Carolina.

[2] Anurag Agrawal and Changhoon Kim. 2020. Intel Tofino2–a 12.9 Tbps P4-programmable ethernet switch. In *2020 IEEE Hot Chips 32 Symposium (HCS)*. IEEE Computer Society, 1–32.

[3] Jung Ho Ahn, Nathan Binkert, Al Davis, Moray McLaren, and Robert S Schreiber. 2009. HyperX: topology, routing, and packaging of efficient large-scale networks. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. 1–11.

[4] AMD. last accessed: 10/18/2024. Pensando. https://www.amd.com/en/accelerators/pensando.

[5] AMD. last accessed: 10/18/2024. Performance of a Multi-Stage SDN Pipeline on Arm® vs AMD Pensando™ Programmable Silicon. https://www.amd.com/content/dam/amd/en/documents/pensando-technical-docs/white-papers/pensando-comparison-of-dpu-hardware-strategies.pdf.

[6] Ariful Azad, Aydin Buluç, and John Gilbert. 2015. Parallel triangle counting and enumeration using matrix algebra. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE, 804–811.

[7] Ariful Azad, Oguz Selvitopi, Md Taufique Hussain, John R Gilbert, and Aydın Buluç. 2021. Combinatorial BLAS 2.0: Scaling combinatorial algorithms on distributed-memory systems. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2021), 989–1001.

[8] Rajeev Balasubramonian, Andrew B Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New tools for interconnect exploration in innovative off-chip memories. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 2 (2017), 1–25.

[9] Vivek Bharadwaj, Aydın Buluç, and James Demmel. 2022. Distributed-memory sparse kernels for machine learning. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 47–58.

[10] Arijit Biswas and Sailesh Kottapalli. 2021. Next-Gen Intel Xeon CPU-Sapphire Rapids. In *Hot Chips*, Vol. 33.

[11] Charles Block, Gerasimos Gerogiannis, Charith Mendis, Ariful Azad, and Josep Torrellas. 2024. Two-Face: Combining Collective and One-Sided Communication for Efficient Distributed SpMM. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*.

[12] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-Independent Packet Processors. In *ACM SIGCOMM CCR*.

[13] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. *ACM SIGCOMM CCR* (2013).

[14] Ulrik Brandes. 2001. A faster algorithm for betweenness centrality. *Journal of mathematical sociology* 25, 2 (2001), 163–177.

[15] BROADCOM. last accessed: 10/18/2024. BCM88690. https://www.broadcom.com/products/ethernet-connectivity/switching/stratadnx/bcm88690.

[16] BROADCOM. last accessed: 10/18/2024. Broadcom Ships Tomahawk 5, Industry's Highest Bandwidth Switch Chip to Accelerate AI/ML Workloads. https://investors.broadcom.com/news-releases/news-release-details/broadcom-ships-tomahawk-5-industrys-highest-bandwidth-switch.

[17] BROADCOM. last accessed: 10/18/2024. Trident 4 / BCM56880 Series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56880-series.

[18] BROADCOM. last accessed: 10/18/2024. Trident 5 / BCM78800 Series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm78800.

[19] Cagla Cakir, Ron Ho, Jon Lexau, and Ken Mai. 2015. Modeling and design of high-radix on-chip crossbar switches. In *Proceedings of the 9th International Symposium on Networks-on-Chip*. 1–8.

[20] Cepulis, Darren. 2015. CERN OCP & HPC on ARM. https://indico.cern.ch/event/389301/contributions/1822797/attachments/778382/1067363/CERN_OCP_HPC_ARM_1.pdf. Accessed: 2025-06-20. Originally presented at the OCP & HPC on ARM workshop.

[21] Shuangliang Chen, Saptadeep Pal, and Rakesh Kumar. 2024. Waferscale network switches. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 215–229.

[22] 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.

[23] The P4 Language Consortium. last accessed: 11/20/2024. P416 Language Specification. https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html.

[24] Timothy A Davis. 2019. Algorithm 1000: SuiteSparse: GraphBLAS: Graph algorithms in the language of sparse linear algebra. *ACM Transactions on Mathematical Software (TOMS)* 45, 4 (2019), 1–25.

[25] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1–25.

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

[27] Michael Galles and Francis Matus. 2021. Pensando distributed services architecture. *IEEE Micro* 41, 2 (2021), 43–49.

[28] Gerasimos Gerogiannis, Sriram Aananthakrishnan, Josep Torrellas, and Ibrahim Hur. 2024. HotTiles: Accelerating SpMM with Heterogeneous Accelerator Architectures. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 1012–1028.

[29] Gerasimos Gerogiannis, Serif Yesil, Damitha Lenadora, Dingyuan Cao, Charith Mendis, and Josep Torrellas. 2023. SPADE: A Flexible and Scalable Accelerator for SpMM and SDDMM. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 19, 15 pages. https://doi.org/10.1145/3579371.3589054

[30] William L Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*. 1025–1035.

[31] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. 2016. EIE: Efficient inference engine on compressed deep neural network. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 243–254.

[32] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W. Fletcher. 2019. ExTensor: An Accelerator for Sparse Tensor Algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 319–333. https://doi.org/10.1145/3352460.3358275

[33] Karl Scott Hemmert. 2018. *Merlin Element Library Deep Dive*. Technical Report. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States).

[34] Yuxi Hong and Aydin Buluç. 2024. A Sparsity-Aware Distributed-Memory Algorithm for Sparse-Sparse Matrix Multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Atlanta, GA, USA) *(SC '24)*. IEEE Press, Article 47, 14 pages. https://doi.org/10.1109/SC41406.2024.00053

[35] Huda Ibeid, Vikram Narayana, Jeongnim Kim, Anthony Nguyen, Vitali Morozov, and Ye Luo. 2025. Performance Analysis of HPC Applications on the Aurora Supercomputer: Exploring the Impact of HBM-Enabled Intel Xeon Max CPUs. In *ISC High Performance 2025 Research Paper Proceedings (40th International Conference)*. Prometeus GmbH, 1–11.

[36] Synopsys Inc. 2016. *Design Compiler User Guide*. Synopsys. Version L-2016.03.

[37] Intel. last accessed: 10/18/2024. Intel Ethernet Controller 700 Series - Open vSwitch Hardware Acceleration Application Note. https://builders.intel.com/docs/networkbuilders/intel-ethernet-controller-700-series-open-vswitch-hardware-acceleration-application-note.pdf.

[38] Intel. last accessed: 10/18/2024. Tofino: P4-programmable Ethernet switch ASIC that delivers better performance at lower power. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html.

[39] Intel. last accessed: 10/18/2024. Tofino2: Second-generation P4-programmable Ethernet Switch ASIC that Continues to Deliver Programmability without Compromise. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html.

[40] Intel Corporation. 2025. *Intel oneAPI Math Kernel Library (oneMKL)*.

[41] Hanchen Jin, Zichao Yue, Zhongyuan Zhao, Yixiao Du, Chenhui Deng, Nitish Srivastava, and Zhiru Zhang. 2024. Vesper: A Versatile Sparse Linear Algebra Accelerator With Configurable Compute Patterns. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2024), 1–1. https://doi.org/10.1109/TCAD.2024.3496882

[42] 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 *ACM SOSP*.

[43] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Clifford Young, Xiang Zhou, Zongwei Zhou, and David A Patterson. 2023. TPU v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 1–14.

[44] Mark Karol, Michael Hluchyj, and Samuel Morgan. 2003. Input versus output queueing on a space-division packet switch. *IEEE Transactions on communications* 35, 12 (2003), 1347–1356.

[45] John Kim, Wiliam J Dally, Steve Scott, and Dennis Abts. 2008. Technology-driven, highly-scalable Dragonfly topology. *ACM SIGARCH Computer Architecture News* 36, 3 (2008), 77–88.

[46] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[47] Apostolos Kokolis, Antonis Psistakis, Benjamin Reidys, Jian Huang, and Josep Torrellas. 2021. Distributed data persistency. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 71–85.

[48] Apostolos Kokolis, Antonis Psistakis, Benjamin Reidys, Jian Huang, and Josep Torrellas. 2024. HADES: Hardware-Assisted Distributed Transactions in the Age of Fast Networks and SmartNICs. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 785–800.

[49] Kartik Lakhotia, Maciej Besta, Laura Monroe, Kelly Isham, Patrick Iff, Torsten Hoefler, and Fabrizio Petrini. 2022. PolarFly: a cost-effective and flexible low-diameter topology. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[50] Kartik Lakhotia, Kelly Isham, Laura Monroe, Maciej Besta, Torsten Hoefler, and Fabrizio Petrini. 2023. In-network allreduce with multiple spanning trees on PolarFly. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures*. 165–176.

[51] Kartik Lakhotia, Laura Monroe, Kelly Isham, Maciej Besta, Nils Blach, Torsten Hoefler, and Fabrizio Petrini. 2024. PolarStar: Expanding the Horizon of Diameter-3 Networks. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures*. 345–357.

[52] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan RK Ports. 2020. Pegasus: Tolerating skewed workloads in distributed storage with in-network coherence directories. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 387–406.

[53] Shang Li, Zhiyuan Yang, Dhiraj Reddy, Ankur Srivastava, and Bruce Jacob. 2020. DRAMsim3: a cycle-accurate, thermal-capable DRAM simulator. *IEEE Computer Architecture Letters* 19, 2 (2020), 106–109.

[54] Youjie Li, Iou-Jen Liu, Yifan Yuan, Deming Chen, Alexander Schwing, and Jian Huang. 2019. Accelerating distributed reinforcement learning with in-switch computing. In *Proceedings of the 46th International Symposium on Computer Architecture*. 279–291.

[55] Zhiyao Li, Jiaxiang Li, Taijie Chen, Dimin Niu, Hongzhong Zheng, Yuan Xie, and Mingyu Gao. 2023. Spada: Accelerating Sparse Matrix Multiplication with Adaptive Dataflow. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 747–761. https://doi.org/10.1145/3575693.3575706

[56] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. 2017. Incbricks: Toward in-network computation with an in-network cache. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. 795–809.

[57] Xiaoyang Lu, Boyu Long, Xiaoming Chen, Yinhe Han, and Xian-He Sun. 2024. ACES: Accelerating Sparse Matrix Multiplication with Adaptive Execution Flow and Concurrency-Aware Cache Optimizations. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 71–85.

[58] Patrick MacArthur, Qian Liu, Robert D. Russell, Fabrice Mizero, Malathi Veeraraghavan, and John M. Dennis. 2017. An Integrated Tutorial on InfiniBand, Verbs, and MPI. *IEEE Communications Surveys & Tutorials* 19, 4 (2017), 2894–2926.

[59] F. Miller Maley and Jason G. DeVinney. 2019. Conveyors for Streaming Many-To-Many Communication. In *2019 IEEE/ACM 9th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. 1–8. https://doi.org/10.1109/IA349570.2019.00007

[60] N. McKeown. 1999. The iSLIP scheduling algorithm for input-queued switches. *IEEE/ACM Transactions on Networking* 7, 2 (1999), 188–201. https://doi.org/10.1109/90.769767

[61] Francisco Muñoz Martínez, Raveesh Garg, Michael Pellauer, José L. Abellán, Manuel E. Acacio, and Tushar Krishna. 2023. Flexagon: A Multi-Dataflow Sparse-Sparse Matrix Multiplication Accelerator for Efficient DNN Processing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 252–265. https://doi.org/10.1145/3582016.3582069

[62] Ujjaini Mukhopadhyay, Alok Tripathy, Oguz Selvitopi, Katherine Yelick, and Aydin Buluc. 2024. Sparsity-aware communication for distributed graph neural network training. In *Proceedings of the 53rd International Conference on Parallel Processing*. 117–126.

[63] National Center for Supercomputing Applications. 2024. Delta User Documentation. Retrieved 2024 from https://docs.ncsa.illinois.edu/systems/delta/en/latest/

[64] Israt Nisa, Aravind Sukumaran-Rajam, Sureyya Emre Kurt, Changwan Hong, and P Sadayappan. 2018. Sampled dense matrix multiplication for high-performance machine learning. In *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*. IEEE, 32–41.

[65] NVIDIA. 2024. RoCE v2 Considerations. Retrieved 2024 from https://enterprise-support.nvidia.com/s/article/roce-v2-considerations

[66] NVIDIA. last accessed: 10/18/2024. CONNECTX-6 DX. https://www.nvidia.com/en-us/networking/ethernet/connectx-6-dx/.

[67] NVIDIA. last accessed: 10/18/2024. NVIDIA Bluefield-3 Datasheet. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-3-dpu.pdf/.

[68] NVIDIA. last accessed: 10/18/2024. NVIDIA Bluefield Data Processing Units. https://www.nvidia.com/en-us/networking/products/data-processing-unit/.

[69] NVIDIA. last accessed: 11/20/2024. AllReduce. https://docs.nvidia.com/doca/archive/doca-v1.3/allreduce/index.html.

[70] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web.* Technical Report. Stanford infolab.

[71] Antonis Psistakis, Fabien Chaix, and Josep Torrellas. 2024. MINOS: Distributed Consistency and Persistency Protocol Implementation & Offloading to SmartNICs. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 1–17.

[72] Isuru Ranawaka, Md Taufique Hussain, Charles Block, Gerasimos Gerogiannis, Josep Torrellas, and Ariful Azad. 2024. Distributed-Memory Parallel Algorithms for Sparse Matrix and Sparse Tall-and-Skinny Matrix Multiplication. In *2024 SC24: International Conference for High Performance Computing, Networking, Storage and Analysis SC*. IEEE Computer Society, Los Alamitos, CA, USA, 730–746. https://doi.org/10.1109/SC41406.2024.00052

[73] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. Cooper-Balis, and B. Jacob. 2011. The Structural Simulation Toolkit. *SIGMETRICS Perform. Eval. Rev.* 38, 4 (mar 2011), 37–42. https://doi.org/10.1145/1964218.1964225

[74] Roberto Rojas-Cessa, Eiji Oki, Zhigang Jing, and H Jonathan Chao. 2001. CIXB-1: Combined input-one-cell-crosspoint buffered switch. In *2001 IEEE Workshop on High Performance Switching and Routing (IEEE Cat. No. 01TH8552)*. IEEE, 324–329.

[75] 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 *USENIX NSDI*.

[76] Korakit Seemakhupt, Sihang Liu, Yasas Senevirathne, Muhammad Shahbaz, and Samira Khan. 2021. PMNet: In-Network Data Persistence. In *ACM/IEEE ISCA*. IEEE.

[77] Oguz Selvitopi, Benjamin Brock, Israt Nisa, Alok Tripathy, Katherine Yelick, and Aydın Buluç. 2021. Distributed-memory parallel algorithms for sparse times tall-skinny-dense matrix multiplication. In *Proceedings of the ACM International Conference on Supercomputing*. 431–442.

[78] ServeTheHome. 2020. Pensando Distributed Services Architecture – SmartNIC. https://www.servethehome.com/pensando-distributed-services-architecture-smartnic/.

[79] Kawthar Shafie Khorassani, Chen Chun Chen, Bharath Ramesh, Aamir Shafi, Hari Subramoni, and Dhabaleswar Panda. 2022. High performance MPI over the Slingshot interconnect: Early experiences. In *Practice and Experience in Advanced Research Computing*. 1–7.

[80] Vishal Shrivastav. 2022. Stateful multi-pipelined programmable switches. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 663–676.

[81] Linghao Song, Yuze Chi, Atefeh Sohrabizadeh, Young-kyu Choi, Jason Lau, and Jason Cong. 2022. Sextans: A Streaming Accelerator for General-Purpose Sparse-Matrix Dense-Matrix Multiplication. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 65–77.

[82] Nitish Srivastava, Hanchen Jin, Shaden Smith, Hongbo Rong, David Albonesi, and Zhiru Zhang. 2020. Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 689–702.

[83] Aaron Stillmaker and Bevan Baas. 2017. Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm. *Integration* 58 (2017), 74–81.

[84] James E. Stine, Ivan D. Castellanos, Michael Wood, Jeff Henson, Fred Love, W. Rhett Davis, Paul D. Franzon, Michael Bucher, Sunil Basavarajaiah, Julie Oh, and Ravi Jenkal. 2007. FreePDK: An Open-Source Variation-Aware Design Kit. In *2007 IEEE International Conference on Microelectronic Systems Education (MSE'07)*. San Diego, CA, 173–174. https://doi.org/10.1109/MSE.2007.44

[85] Tushar Swamy, Alexander Rucker, Muhammad Shahbaz, Ishan Gaur, and Kunle Olukotun. 2022. Taurus: a data plane architecture for per-packet ML. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 1099–1114.

[86] Wikipedia. 2024. Goodput. Retrieved 2024 from https://en.wikipedia.org/wiki/Goodput

[87] Xilinx. last accessed: 10/18/2024. Alveo SN1000 SmartNICs. https://www.xilinx.com/content/dam/xilinx/publications/product-briefs/sn1000-product-brief.pdf.

[88] Xilinx. last accessed: 10/18/2024. Alveo U250 Data Center Accelerator Card. https://www.xilinx.com/products/boards-and-kits/alveo/u250.html.
[89] Mingran Yang, Alex Baban, Valery Kugel, Jeff Libby, Scott Mackie, Swamy Sadashivaiah Renu Kananda, Chang-Hong Wu, and Manya Ghobadi. 2022. Using Trio: Juniper Networks' Programmable Chipset for Emerging In-Network Applications. In *ACM SIGCOMM*.
[90] Serif Yesil, Azin Heidarshenas, Adam Morrison, and Josep Torrellas. 2020. Speeding up SpMV for power-law graph analytics by enhancing locality & vectorization. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.