Uncovering SIMD parallelism in CPU Packet Processing using the NIC as a Vector Co-Processor

Kunal Umaji
IIT Madras
Chennai, India

Shreyas Bargale
IIT Madras
Chennai, India

Yashvardhan Toshniwal
IIT Madras
Chennai, India

Yash Mehta
IIT Madras
Chennai, India

Anirudh Sivaraman
New York University
New York, USA

Srinivas Narayana
Rutgers University &
IIT Madras
Chennai, India

Abstract

The performance of packet processing on general-purpose CPUs remains as consequential as ever, as many well-known and emerging workloads rely on the expressiveness of general-purpose CPUs. To accelerate packet processing, data parallelism across packets is widely exploited by spreading packets across CPU cores. Yet, the community has only narrowly leveraged packet parallelism on individual CPU cores, with Single-Instruction Multiple-Data (SIMD) vectorization applied to limited fragments of device drivers and to specific data structures. The broader application of SIMD vectorization has the potential to accelerate packet processing on individual CPU cores by an order of magnitude. However, two longstanding challenges have prevented such wide usage: memory bandwidth limits and control flow divergence.

The increasing prevalence of programmable network interface cards (NICs) near CPUs presents an opportunity to overcome this impasse. By re-imagining the NIC as a packet vector co-processor, we can leverage its compute/memory characteristics complementary to CPUs to enable efficient SIMD parallelism that would be arduous to achieve with software-only techniques. We present encouraging preliminary results and outline directions for further research.

CCS Concepts

• **Networks** → *Programming interfaces*; **Programmable networks**.

Keywords

Packet Processing, SIMD, Vector Co-Processor



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

HotNets '26, Salt Lake City, UT, USA

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

ACM ISBN 979-8-4007-3029-0/2026/11
https://doi.org/10.1145/3847290.3859972

ACM Reference Format:
Kunal Umaji, Shreyas Bargale, Yashvardhan Toshniwal, Yash Mehta, Anirudh Sivaraman, and Srinivas Narayana. 2026. Uncovering SIMD parallelism in CPU Packet Processing using the NIC as a Vector Co-Processor. In *The 25th ACM Workshop on Hot Topics in Networks (HotNets '26), November 16–17, 2026, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3847290.3859972

1 Introduction

The performance of packet processing on general-purpose CPUs remains as consequential as ever, as many well-known and emerging workloads rely on the expressiveness of general-purpose CPUs. For example, network functions such as those used for DDoS protection, live video transcoding, and bump-in-the-wire data compression benefit from faster packet processing on CPUs. Emerging AI agents communicate with "tools" by acting as network clients, using protocols like MCP [22] to communicate with MCP servers [18]. MCP servers are increasingly containerized to enforce isolation and resource control [20, 25], where each agentic request-response traverses the container network stack on a CPU.

In this paper, we are interested in the classic question of accelerating CPU-bound packet-processing applications requiring high throughput and low latencies. Fast CPU-oriented packet processing is a deeply researched area (§2.1). However, despite the plethora of work, improvements in the packet-processing throughput of a single CPU core remain surprisingly elusive. Recent work suggests (§2.1) that deep structural changes to application and driver code are necessary to optimize packet-processing application code by up to 2–3×, specifically the interactions between the CPU, the NIC, and cache memory.

Complementary to these existing systems, we think that the community has overlooked a significant class of optimizations with the potential to produce order-of-magnitude gains in CPU packet-processing performance, with minimal structural changes. Data parallelism across packets, and the fact that packet-processing applications execute isomorphic

instructions across many packets, motivates the use of single-instruction multiple-data (SIMD) parallelism [40] and vector instructions to process packets truly in parallel. If the CPU architecture offers a vector variant to each scalar instruction in the program, the throughput of the program can in principle be improved by the factor of the vector length. In today's CPU architectures, vector lengths of 8–16 are common for standard data types [6, 15], enabling the execution of an instruction on as many packets in parallel. Some architectures offer even longer vector lengths [4, 13], suggesting an even higher performance amplification.

Two challenges stand in the way of fruitfully capitalizing on SIMD parallelism (§2.2). First, not every scalar instruction offers a vector variant of comparable performance. For example, implementing k disjoint memory accesses with latency comparable to one access requires a k-fold increase in memory bandwidth, which is not easily afforded. Second, not all instructions execute for all packets, due to control flow in the program. Typical packet-processing code is heavy on both memory access and control flow; we think that previous packet-processing systems have only narrowly adopted SIMD parallelism due to these challenges (§2.2). For example, some user-space device drivers use limited packet-parallel SIMD execution in straight-line code fragments operating on continuously-allocated packet descriptors. Other prior efforts incorporate SIMD piecemeal to design specific data structures. Our goal is to bring the benefits of SIMD packet parallelism to a broad class of packet-processing code.

This paper leverages the increasing prevalence of programmability in the network interface card (NIC) to mitigate the two key challenges above (§3). The NIC is naturally on the data path to and from the CPU, and exhibits compute and memory characteristics complementary to general-purpose CPUs (§3.1), in particular exposing high-bandwidth access to packet memory and the capability to read/write (a small amount of) high-speed NIC-local memory at line rate. We show how these characteristics can be exploited to enable more efficient SIMD execution on CPU, through field scatter/gather across packets (§3.2) and packet data regrouping based on control flow (§3.3). These NIC operations effectively turn discontinuous memory accesses into continuous ones and unify control flow across packets. We outline how a line-rate NIC pipeline can implement such operations (§3.4).

Preliminary results show the effectiveness of the aforementioned techniques for CPU packet-processing performance (§4). We also outline open research directions (§5) surrounding the ability to apply SIMD execution to network device drivers and designing an automatic parallelizing compiler.

2 Background and Motivation

2.1 Fast CPU Packet Processing

With the demise of Moore's law, packet processing in the kernel network stack is considered too slow for many high-performance applications [31, 55]. The community has investigated methods to pare down the critical packet-processing path between the hardware network interface card (NIC) and the CPU, specializing the software stack to fast networking, through designs such as kernel bypass, *e.g.* [7], kernel extensions, *e.g.* [41], and hardware offloads, *e.g.* [9, 27]. Several systems improve packet-processing throughput by exploiting parallelism across CPU cores, *e.g.* [1, 28, 52].

Improving packet-processing throughput on a single CPU core remains a challenging problem. Recent works show that improving single-CPU-core performance requires deep structural changes to packet-processing application code, specifically the interactions between the CPU, the NIC, and the cache memory hierarchy [36, 38, 42, 47, 54, 59, 60]. These efforts report throughput improvements of up to 2–3x over the prior state of the art, while requiring changes to the data structures that underpin CPU-NIC communication.

The rest of this paper focuses on a complementary approach to improving single-CPU-core performance: exploiting data parallelism across packets.

2.2 Opportunities and Challenges in Leveraging Packet-level Parallelism

We start by asking a basic question: is it possible to process packets in parallel (at the instruction level) on a CPU?

An important obstacle to processing packets in parallel is *dependency*: for example, applications may read and update memory across packets, with later packets depending on the results of computing over previous packets. However, we believe this doesn't need to be a showstopper. Many network functions exhibit the characteristic that the fragment of computation that requires sequential execution across packets is small, enabling the rest of the program to be parallelized [34, 57, 58]. Many stateful computations are associative, allowing results to be computed in parallel and recombined. Some network functions explicitly shard state and compute over flows of data in parallel [1, 52, 61]. Hence, we believe that significant chunks of packet-processing code can be executed independently over packets.

This paper explores how to leverage data parallelism across packets by executing isomorphic instructions across packets in parallel through single-instruction multiple-data (SIMD) vector parallelism [40, 50]. Consider a simple scenario where instructions $I_1, I_2, \cdots, I_n$ execute on packet data $p_1, p_2, \cdots, p_m$ in sequence. If the underlying CPU architecture offers a vector instruction vI_j corresponding to each scalar instruction I_j, and every packet can execute independently of others, we

can execute computations over k packets in parallel, where k is the number of data items that each vector instruction can operate on (the architectural vector length). Theoretically, this can provide a k-fold improvement in throughput. The x86 and ARM instruction set architectures today offer vector registers that are at least 128–512 bits wide [6, 15], with extensions in ARM and RISC-V architectures allowing a larger, configurable length [4, 13]. Depending on the data type operated on, these architectures can support parallel operation on up to k=8–16 packets in one vector instruction.

Despite their potential, few previous efforts have deployed vectorization to exploit packet-parallelism in general packet-processing applications. Implementations have been constructed within specific data structures such as hash tables for IP or ACL lookups in Intel DPDK [6], specific low-level operations (*e.g.* memory copies) in Cisco's VPP framework [14], and the design of Bloom filters and lookup tables [46, 51]. Some user-space device drivers can perform a subset of processing on 4–8 packet descriptors at a time, by shuffling data between continuously-allocated NIC descriptor rings and vector registers holding packet fields from those descriptors [16, 17, 21]. However, packet data not available on the descriptors cannot be readily processed with SIMD parallelism. Moreover, the width of the NIC descriptor limits the number of packets whose data fits a single vector register, hence limiting the achievable parallelism. In summary, existing systems do not broadly deliver the performance benefits of SIMD vectorization to general packet-processing code.

There are at least two challenges that impede the benefits of SIMD vectorization, *i.e.* achieving a k-fold throughput improvement, in general packet-processing code. First, not every scalar instruction offers a vector variant, at least not with similarly good performance. Memory access instructions are a prime example: reading and writing to k discontinuous locations in memory requires a k-fold increase in memory bandwidth, which is not affordable on most CPU architectures [40]. Second, not all instructions execute on all packets, due to control flows that cause different packets to take different branches in the application. Networking-heavy applications typically incorporate significant memory access (*e.g.* lookup tables, reading and writing discontiguous packet fields) and control flow divergence (*e.g.* due to parsing and disjoint processing of different protocols and different kinds of packets even within a protocol). Hence, realizing a k-fold throughput improvement on a machine with vector length k is non trivial.

3 Programmable NIC as a Co-Processor for vectorization

3.1 Why can line-rate NICs help?

Line-rate NICs can serve as effective co-processors for vectorizing CPU computations across packets for two reasons. First, they are naturally located on the critical path of data movement between the network and the CPU. Second, programmable NICs [10, 11, 19, 37], especially those that include PISA pipelines [29, 30] or massively-parallel datapath accelerators, exhibit certain computational and memory characteristics that are complementary to general-purpose CPUs. Our goal is to co-design NIC and CPU programs, exploiting NICs for operations that suffer poor performance on CPUs.

Two performance characteristics of PISA pipelines stand out in contrast to CPUs. First, ALUs in a PISA pipeline have high-speed access to packet memory in the fast data path, keeping packet contents in packet header vectors (PHVs), which are similar to registers in a general-purpose CPU. On the other hand, CPUs themselves hold packets in a much slower memory hierarchy. Second, it is possible to store (small amounts of) packet contents in NIC-local high-speed memory, reading these contents out into packets as needed and sending them subsequently to the CPU. The ability to read and write state memory at high speeds enables flexible aggregatation and spreading of data across packets, an operation challenging on CPUs due to the limited memory bandwidth for discontinuous reads and writes across packets.

The aforementioned characteristics enable us to re-imagine a programmable NIC as a *coprocessor* for executing vector computations across packets on CPUs. The coprocessor computational model we envision is distinct from a simple partition of computation between NIC and CPU, as some prior systems have done, *e.g.* [53, 62], since *we add new computations not part of the original packet-processing application* to the NIC and the CPU program, to better support CPU parallelism across packets (§3.2, §3.3). The co-processor computational model also differs from full hardware offload, *e.g.* [9], because the bulk of the computations still execute on CPU.

3.2 Mitigating Memory Bandwidth Limits

Memory access, *i.e.* load and store instructions, are a significant class of instructions which vectorize poorly. Consider Figure 1 (a), where a packet-processing program is vectorized 1:1 from its scalar counterpart, and requires reading from k packets to move to its next computation. This is an example of a *gather*, which requires a k-fold increase in memory bandwidth. When the data items read are from discontinuous locations—as happens when header fields must be fetched from different packets—vector memory accesses are known to incur high latencies [5, 40, 46]. Similar impediments occur when data must be written out across packets (*scatter*).

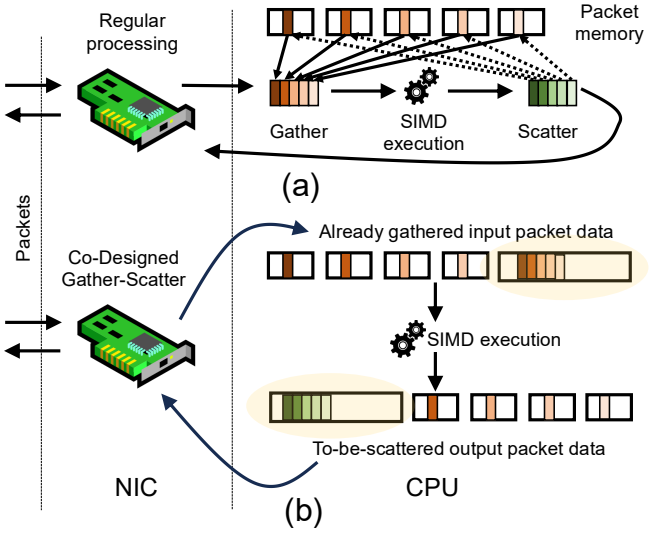


Figure 1: Lack of memory bandwidth holds back direct SIMD execution (a), but the required memory bandwidth can be effectively provided through a co-designed NIC program that gathers and scatters the relevant packet data pre- and post-the SIMD execution (b).

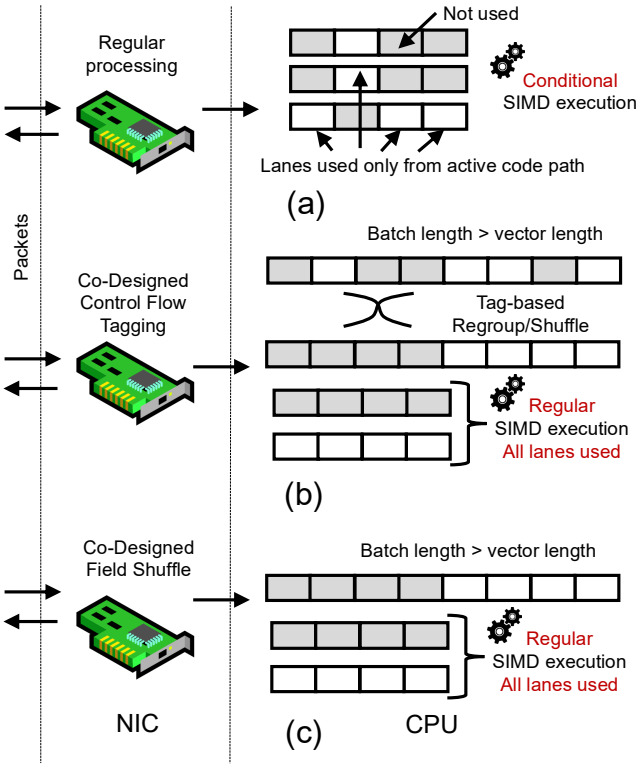


Figure 2: When packets take different control flow paths, a naive application of SIMD would use conditional execution, with some vector lanes turned off (a), impairing performance. When the control flow path is computable using packet fields, it is possible for the NIC to provide tags that allow CPU code to recover the full vector utilization by regrouping packets either in software (b) or on NIC hardware (c).

A co-designed NIC program can effectively mitigate the poor performance of scatter-gather memory accesses using the high bandwidth that NICs have to read and write packets. Consider Figure 1 (b). By gathering the packet data relevant to instructions in high-speed stateful memory on the NIC, and writing this gathered data into the last packet of a batch, the NIC can effectively turn discontinuous memory accesses into continuous ones, with the CPU reading the necessary data for computation from the last packet of the batch. A similar optimization can be implemented on the packet's way out of the CPU through the NIC, by having the NIC scatter data written into the first packet of a batch to the remainder of the packets in the batch. The memory hierarchy on vector CPUs is much easier to engineer for high performance for continuous, aligned memory accesses, than for discontinuous memory accesses [40], even when the former access is wider.

Implementing this co-design requires changes to the program running on the CPU and the creation of a new application-specific program running at the NIC. In particular, the CPU program must read packet field data for an entire batch from the last packet, and write packet field data into the first packet of the batch. The NIC must gather and scatter data from the appropriate packets in the batch (§3.4).

3.3 Mitigating Control Flow Divergence

When packets don't follow the same branch in the code, you cannot simply vectorize each scalar instruction 1:1 over data across packets, since some instructions only apply to some packets. A naive option to handle this divergence is to fall back to scalar execution whenever branches are encountered. However, this leaves out significant potential for performance, since packet-processing programs are branching-heavy (§2.2). Another possibility is to implement *conditional SIMD execution* (Figure 2 (a)), where an additional bitmask operand is used to selectively turn on the specific vector lanes that must be updated by the instruction [48], with the remaining lanes unchanged. Vector instruction set architectures on CPUs, as well as GPUs, make heavy use of predicated execution. However, high control flow divergence results in poor performance due to the underutilization of vector compute hardware [32, 33].

An alternative to conditional execution is to execute vector instructions unconditionally over sufficiently large batches of packets that have been regrouped (shuffled) into smaller batches following the same control flow paths, and sized close to the architectural vector length k. Such a reordered

computation preserves correctness for those fragments of the application that are independently executed across packets (§2.2), which is a prerequisite for vectorization anyway. The regrouping operation can be implemented in software, *i.e.* by identifying which code branch will be taken for each packet, using a temporary buffer to accumulate packets subject to the same control flow path into batches, and executing the path-relevant vector instructions unconditionally on each accumulated batch [40].

Packet-processing applications offer significant scope to determine, ahead of any processing, the code paths taken by a packet, simply by parsing the packet. Code branches often involve tests that identify protocols and specific kinds of packets within a protocol. A programmable NIC can assist packet regrouping by tagging batches of packets with information about their control flow paths, with software implementing the regrouping (Figure 2 (b)). We could go even one step further, having the NIC implement the shuffle, rearranging packet data to an order that enables direct SIMD execution on vector-length-sized batches (Figure 2 (c)). To keep up line-rate operation, the packets themselves do not need to be reordered—just the data on those packets transmitted as metadata on the last packet of the batch. Such NIC co-processing enables the CPU to recover the full advantage of vectorized execution by using all vector lanes on every instruction executed.

3.4 NIC pipeline design

We introduce a candidate NIC hardware design that uses stateful memory and ALUs in a PISA pipeline [29] to implement data gather/scatter and transposition operations without compromising line rate. The NIC should incorporate the following capabilities: (i) implement stateful memory persisted across packets (*e.g.* registers [8]) (ii) contain ALUs that implement predicated read-write stateful operations on this memory at line rate [58]; (iii) parse and deparse programmable custom packet formats [30]; and (iv) copy register data into programmed packet fields. Recent programmable NIC designs support these capabilities [10, 11, 19, 37], and we believe our design could be feasible on PISA and DPU programmable NICs.

Gather. The goal is to accumulate packet fields relevant for vectorized memory operations in the CPU in a continuous location on the last packet of a batch of packets. See Figure 3. For simplicity, suppose we allocate each available stateful register on the pipeline to record a specific field (index J) of a specific packet (index I) in the batch. Each such register maintains packet[I].field[J] for a fixed I, J. We also maintain a running counter *count* of the number of packets in the batch seen so far, with the counter rolling over to 0 once the pipeline has seen a batch of B packets. To gather data

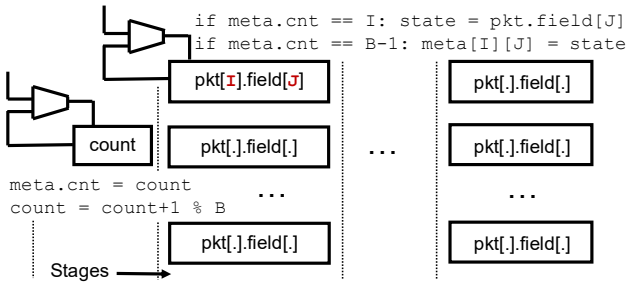


Figure 3: Hardware gather on a line-rate NIC: The NIC maintains a rolling packet counter (*count*) and records relevant packet fields (index J) of each packet (index I) in a batch (size B) in designated stateful registers. The registers are updated using the simplified code shown.

from all packets, B is set to k, the architectural vector length. Each register (corresponding to I and J) is updated to contain the J'th field whenever the counter *count* hits I, and read out into a metadata field carried on the B'th packet. Each conditional register read/write operation is simple enough to fit into the tight clock cycle budget [58].

Scatter. Scattering conceptually implements the "reverse" of the actions of the hardware gather. An incoming packet carries metadata corresponding to fields that must be scattered into the the next B packets of the batch (current packet included). This metadata is written out into the designated stateful registers and copied into the J'th packet field of the I'th packet as appropriate.

Regrouping. Line-rate regrouping is implemented through a generalization of the gather operation. We designate a register to record field J of packet I that takes a specific code branch br. When a sufficient number of total packets B (typically set to a multiple of the architectural vector length) has accumulated, the register values corresponding to each (field J, branch br, packet I) are written out as metadata into a continuous area of the packet.

We believe these operations are feasible given the recent success of related in-network computing operations that maintain packet histories [44, 61]. If the NIC capability permits, a new packet may be generated to report the recorded state after the completion of each batch, to retain the packet MTU limits unchanged between the NIC and the CPU.

3.5 A toy example

We present a simple example to walk through the end-to-end life of a packet processed using the ideas from the preceding subsections. Suppose all packets contain an Ethernet header and are either IPv4 or IPv6. A stateless middlebox works as follows. If the packet has the IPv4 ethertype, the middlebox rewrites the destination IPv4 address to a fixed value. If

the packet has the IPv6 ethertype, the gateway decrements the IPv6 hop limit. All packets are forwarded out the same interface the packets arrived on. This application reads the Ethernet ethertype (all packets) and the hop limit (only on IPv6 packets), and writes to the IPv4 destination or IPv6 hop limit. Each packet follows one of two possible code paths which can be determined by parsing the packet. The NIC implements both scatter-gather and packet field regrouping in line-rate hardware (§3.2–§3.4).

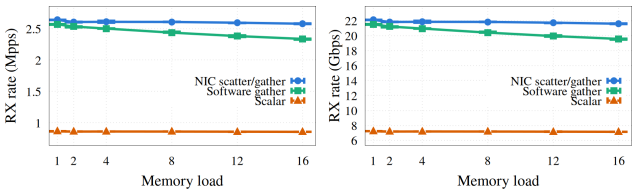
From each incoming packet, the NIC gathers four pieces of data, (i) the packet's ethertype, (ii) the hop limit of IPv6 packets, (iii) the packet index of IPv4 packets, and (iv) the packet index of IPv6 packets, into dedicated stateful registers. The contents of these registers are copied into continuously laid-out metadata on the last packet of the batch. The batch size may be set to twice the architectural vector length to provide high lane utilization, but smaller sizes are also possible.

The CPU software receives the last packet of the batch and its metadata. It implements a continuous memory load to bring each piece of metadata ((i)–(iv) above) into vector registers, possibly in multiple chunks of architectural vector length. The CPU software initializes a vector register containing copies of the fixed output IPv4 destination address corresponding to the number of entries in (iii). This register is stored to a dedicated continuous metadata region (v) on the first packet of the batch. The software also implements a vector decrement on register (ii), and stores it back to the memory region corresponding to (ii). Metadata (i) is removed from the packet. Metadata (ii)–(v) are moved from the last to the first packet of the batch.

The NIC strips the metadata from the first outgoing packet in the batch and writes them into dedicated stateful registers. For each outgoing packet, the NIC scatters the metadata from (ii) into the packets indexed by (iv), and metadata from (v) into the packets indexed by (iii).

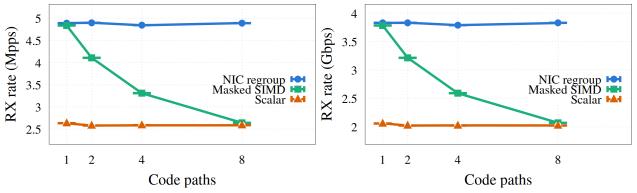
4 Preliminary Results

Setup. Our experimental setup consists of two servers on CloudLab [35] each with an Intel E5-2640 (v4) processor (10 cores) clocked at 2.4 GHz, and a Mellanox ConnectX-4 25 Gbit NIC. One machine serves as the device under test (DUT) running a DPDK packet-processing application on a single CPU core. The application receives a packet, runs a computation of interest, and immediately transmits the packet out the same network interface. Vectorized variants use AVX2 intrinsics. The other machine runs a traffic generator which measures the packet-processing throughput of the DUT by implementing a binary search to identify the highest offered traffic rate that results in a loss rate smaller than a given threshold (we use 2%), similar to the methodology used to benchmark hardware packet processors [2].



(a) Throughput (pkt/sec) **(b) Throughput (Byte/sec)**

Figure 4: Improvement in single-CPU-core throughput with a NIC co-processor that implements gather/scatter, versus a software-only vectorized implementation, as the number of discontinuous memory accesses varies.



(a) Throughput (pkt/sec) **(b) Throughput (Byte/sec)**

Figure 5: Improvement in single-CPU-core throughput with a NIC co-processor that regroups packet data to coalesce the processing of packets on the same code path on the CPU, versus a software-only implementation using conditional (masked) execution, as the number of code paths varies.

We use minimum-sized packets unless specified otherwise and report the median and 1st-to-3rd quartile range (error bars) across 10 runs of the full throughput measurement. Our experiments assume the NIC capabilities needed for line-rate scatter/gather and regrouping (§3.4). Our traffic generator transmits packets with the appropriate packet metadata.

Mitigating the memory bottleneck. We develop a DUT program that includes a compute-heavy hash function that keeps the program CPU-bottlenecked on a single core. With a packet size of 1024 bytes, this program reads up to 16 distinct cache lines (64B) on each packet, simulating disjoint memory accesses during execution. In Figure 4, we compare the throughput of a software-only vectorized implementation against an implementation relying on NIC-pre-gathered and post-scattered packet metadata. We report both packets/second (a) and bytes/second (b) throughput. The co-processor-based implementation processes packets at close to line rate, while the performance of the software-only implementation degrades with each additional memory access.

Mitigating control flow divergence. We develop a DUT program that processes each packet through one of a configurable number of code paths. In Figure 5, we compare a software-only implementation that uses conditional SIMD execution against a co-processor-based implementation that leverages hardware regrouping to keep the vector utilization full. We report both packets/second (a) and bytes/second (b) throughput. The performance of the purely-software

vectorized implementation (masked SIMD) degrades with additional divergence, while a solution relying on NIC regrouping maintains stable throughput.

Estimating NIC stateful memory usage to hold packet contents. For an architectural vector length k and batch size $B \geq k$ across nbr branches, holding n packet fields each b bits large on the NIC requires $B * n * b * nbr$ bits of memory. For instance, with $B = k = 8$, $n = 10$, and $b = 32$, holding data across $nbr = 4$ code branches requires 1280 bytes—about as large as a single packet. We believe that while the aggregate memory requirement for packet state on the NIC will remain likely small, specific implementations of pipelines or datapath processors may limit the total number of stateful storage addresses and bytes available to rein in the overheads of other related circuitry (*e.g.* multiplexers to fetch packet inputs). We leave a full exploration of the constraints of NIC implementation to future work.

We leave the full evaluation of the impact on system resource contention (PCIe bandwidth, LLC) to future work.

5 Discussion and Open Research

Building on the ideas outlined so far, we discuss some research directions for the community to pursue.

Designing parallel device drivers. A key bottleneck we uncovered in our experiments is the device driver which supplies packets to our main packet-processing application loop. Our throughput and processing time measurements of the application loop indicate that roughly 20% of the time is spent in the DPDK poll-mode driver (up to 100 nanoseconds on our system). Of course, this fraction depends on the application and the system. However, through Amdahl's law, it presents an upper bound on the performance improvement possible from infusing parallelism in the application. We think that NIC device drivers must be designed from the outset to process packets in parallel.

In a kernel-mode driver running a bypassed application with both polling and interrupts [41], the main tasks of the device driver and kernel interrupt-handler include performing some error checks, clearing the ring of received packet descriptors to prepare to send data to the application's loop, recycling or allocating new buffers to receive subsequent data from the NIC, and scheduling the application's main packet-processing loop [3, 43]. The operations on each packet are largely independent from those on others. Research is needed to determine the best methods to parallelize per-packet actions while maintaining correctness and generality. The kernel typically avoids vector operations to reduce context-saving overheads and maximize portability [23, 24], but selective vector execution could be profitable for end-to-end performance.

High-speed user-space drivers incorporate some SIMD parallelism, which, however, is limited by the choice of packet data carried by the NIC ring descriptor (§2.2). How best to re-design the NIC-CPU communication interface (*e.g.* the descriptor structure) and NIC hardware capabilities to improve SIMD packet parallelism in drivers, and more generally CPU packet-processing software, remains an open question.

Developing automatic vectorizing compilers. Vectorizing compiler passes exist today, which consume input scalar code and output vectorized versions [12, 26, 45, 49, 56]. A typical approach used in such compilers is to search for independent and isomorphic operations that can be packed into SIMD/vector instructions, by starting from "seed" operations that are easily seen as vectorizable (such as stores to continuous memory locations), and following the chain of operand definitions to uncover new sets of independent and isomorphic instructions that may also be packed [45, 56].

Designing a vectorizing compiler that exploits a NIC co-processor changes the underlying compilation problem, in at least three ways. First, the execution of every instruction for a packet is isomorphic with the execution of that instruction for a different packet. This presents a substantial increase in the choice of instructions that can be packed into vector form. Second, there is a much broader choice of operations to serve as seeds for the search algorithm. We are not restricted to stores to adjacent locations in the CPU program, since the NIC co-processor can turn discontinuous stores into continuous ones, and the effects of control flow divergence can be substantially mitigated to permit seed operations that span basic blocks. Finally, only a subset of the possible design space of CPU-oriented optimizations may be available or profitable, since NIC resources are constrained (*e.g.* memory, compute) and performance is affected by contention to system resources (PCIe bus bandwidth, LLC).

Modeling NIC constraints and the performance impacts of system resource contention, and developing efficient search algorithms for the much-larger space of vector transformations, presents a rich opportunity to develop domain-specific vectorizing compilers. Further, the presence of a NIC co-processor presents an opportunity to rethink compilers for GPU packet-processing kernels (*e.g.* [39]), which also benefit from minimizing predicated SIMT execution [32, 33].

Acknowledgments

We thank the anonymous HotNets reviewers and our shepherd Daehyeok Kim for valuable comments which improved this paper. This work is supported in part by the IIT Madras Institute Fellowship (HTRA), U.S. National Science Foundation grants FMitF-II 2422076 and CAREER 2340748, and gifts from the eBPF foundation, Nutanix, Xilinx, and Facebook.

References

[1] [n. d.]. Receive Side Scaling. [Online, Retrieved Feb 21, 2023.] https://www.kernel.org/doc/Documentation/networking/scaling.txt.

[2] 1999. Benchmarking Methodology for Network Interconnect Devices. [Online, Retrieved Sep 17, 2023.] https://www.rfc-editor.org/rfc/rfc2544.

[3] 2016. Interrupt handler: Monitoring and Tuning the Linux Networking Stack: Receiving Data. [Online, Retrieved Jun 09, 2026.] https://blog.packagecloud.io/monitoring-tuning-linux-networking-stack-receiving-data/#\protect\protect\leavevmode@ifvmode\kern+.2222em\relax~\protect\protect\leavevmode@ifvmode\kern+.2222em\relaxtext=packet%20arrived%20on).-,Interrupt%20handler,-In%20general%2C%20the.

[4] 2018. The RISC-V Vector ISA Tutorial. [Online, Retrieved Jun 15, 2026.] https://riscv.org/wp-content/uploads/2024/12/15.20-15.55-18.05.06.VEXT-bcn-v1.pdf.

[5] 2021. When vectorization hits the memory wall: investigating the AVX2 memory gather instruction. [Online, Retrieved Jun 09, 2026.] https://johnnysswlab.com/when-vectorization-hits-the-memory-wall-investigating-the-avx2-memory-gather-instruction/.

[6] 2022. Intel® AVX-512 - Instruction Set for Packet Processing. [Online, Retrieved Jul 15, 2026.] https://builders.intel.com/docs/networkbuilders/intel-avx-512-packet-processing-with-intel-avx-512-instruction-set-solution-brief-1678190247.pdf.

[7] 2023. Data Plane Development Kit. [Online, Retrieved Jul 22, 2023.] https://www.intel.com/content/www/us/en/developer/topic-technology/networking/dpdk.html.

[8] 2023. Open Tofino: P4 Intel Tofino Native Architecture - Public version. [Online, Retrieved Sep 17, 2023.] https://github.com/barefootnetworks/Open-Tofino/blob/master/PUBLIC_Tofino-Native-Arch.pdf.

[9] 2023. TCP Offload Engine: Chelsio Communications. [Online, Retrieved Sep 17, 2023.] https://www.chelsio.com/nic/tcp-offload-engine/.

[10] 2024. AMD Pensando 2nd generation plus ("giglio") data processing unit. [Online, Retrieved Jun 09, 2026.] https://www.amd.com/content/dam/amd/en/documents/pensando-technical-docs/product-briefs/pensando-giglio-product-brief.pdf.

[11] 2024. Intel Infrastructure Processing Unit (Intel IPU) SoC E2100 Product Brief. [Online, Retrieved Jun 09, 2026.] https://www.intel.com/content/www/us/en/content-details/818147/intel-infrastructure-processing-unit-intel-ipu-soc-e2100-product-brief.html.

[12] 2026. Auto-vectorization in LLVM. [Online, Retrieved Jun 09, 2026.] https://llvm.org/docs/Vectorizers.html#the-loop-vectorizer.

[13] 2026. Configurable vector length (ARM developer documentation). [Online, Retrieved Jun 15, 2026.] https://developer.arm.com/documentation/102476/0101/SVE-architecture-fundamentals/Configurable-vector-length.

[14] 2026. FD.io VPP: memcpy_avx512.h. [Online, Retrieved Jul 15, 2026.] https://s3-docs.fd.io/vpp/21.06.0/df/d3e/memcpy__avx512_8h_source.html.

[15] 2026. From Arm Neon to SVE. [Online, Retrieved Jun 15, 2026.] https://learn.arm.com/learning-paths/servers-and-cloud-computing/sve/sve_basics/.

[16] 2026. Intel ICE raw packet receive routine (AVX 512). [Online, Retrieved Oct 22, 2026.] https://github.com/DPDK/dpdk/blob/v26.07/drivers/net/intel/ice/ice_rxtx_vec_avx512.c#L37.

[17] 2026. Intel vPMD raw packet receive routine. [Online, Retrieved Oct 22, 2026.] https://github.com/DPDK/dpdk/blob/v26.07/drivers/net/intel/ixgbe/ixgbe_rxtx_vec_sse.c#L260.

[18] 2026. Model Context Protocol Servers. [Online, Retrieved Jul 12, 2026.] https://github.com/modelcontextprotocol/servers.

[19] 2026. Nvidia BlueField-3 DPU programable data center infrastructure on a chip. [Online, Retrieved Jun 09, 2026.] https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-3-dpu.pdf.

[20] 2026. Runtime Observability and Enforcement for Opaque AI Agents. [Online, Retrieved Jun 15, 2026.] https://eunomia.dev/blog/2026/05/25/runtime-security-for-ai-agents/.

[21] 2026. Vector PMD for ixgbe. [Online, Retrieved Oct 22, 2026.] https://doc.dpdk.org/guides-26.07/nics/ixgbe.html#vector-pmd-for-ixgbe.

[22] 2026. What is the Model Context Protocol (MCP)? [Online, Retrieved Jul 12, 2026.] https://modelcontextprotocol.io/docs/getting-started/intro.

[23] 2026. Why Floating Point Is Rarely Used in the Linux Kernel: Challenges and Manual Handling Explained. [Online, Retrieved Jul 09, 2026.] https://linuxvox.com/blog/use-of-floating-point-in-the-linux-kernel/.

[24] 2026. Why not use vector instructions in the kernel? [Online, Retrieved Jun 09, 2026.] https://www.programmersought.com/article/866111490866/.

[25] 2026. Zero trust AI agents on Kubernetes: What I learned deploying multi-agent systems on Kagenti. [Online, Retrieved Jul 15, 2026.] https://next.redhat.com/2026/03/05/zero-trust-ai-agents-on-kubernetes-what-i-learned-deploying-multi-agent-systems-on-kagenti/.

[26] Randy Allen and Ken Kennedy. 1987. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (1987).

[27] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. 2020. Enabling Programmable Transport Protocols in High-Speed NICs. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[28] Tom Barbette, Georgios P. Katsikas, Gerald Q. Maguire, and Dejan Kostić. 2019. RSS++: Load and State-Aware Receive Side Scaling. In *Conference on Emerging Networking Experiments And Technologies (CoNEXT)*.

[29] Pat Bosshart et al. 2013. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. *ACM SIGCOMM* (2013).

[30] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* (2014).

[31] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. 2021. Understanding host network stack overheads. In *ACM SIGCOMM*.

[32] Bruno Coutinho, Diogo Sampaio, Fernando MQ Pereira, and Wagner Meira Jr. 2013. Profiling divergences in gpu applications. *Concurrency and Computation: Practice and Experience* (2013).

[33] Bruno Coutinho, Diogo Sampaio, Fernando Magno Quintao Pereira, and Wagner Meira Jr. 2011. Divergence Analysis and Optimizations. In *International Conference on Parallel Architectures and Compilation Techniques*.

[34] Alexandros Daglis, Mark Sutherland, and Babak Falsafi. 2019. RPC-Valet: NIC-Driven Tail-Aware Balancing of us-Scale RPCs. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[35] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel hh0Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *USENIX Annual Technical Conference (ATC)*.

[36] Alireza Farshin, Tom Barbette, Amir Roozbeh, Gerald Q Maguire Jr, and Dejan Kostić. 2021. PacketMill: toward per-Core 100-Gbps networking. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[37] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, et al. 2018. Azure accelerated networking: Smartnics in the public cloud. In *NSDI*.

[38] Hamid Ghasemirahni, Alireza Farshin, Mariano Scazzariello, Gerald Q Maguire Jr, Dejan Kostić, and Marco Chiesa. 2024. Fajita: Stateful packet processing at 100 million pps. *Proceedings of the ACM on Networking (PACMNet)* (2024).

[39] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. 2010. PacketShader: a GPU-accelerated software router. In *ACM SIGCOMM*.

[40] Christopher J Hughes. 2022. *Single-instruction multiple-data execution*. Springer Nature.

[41] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. 2018. The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel. In *Conference on emerging Networking EXperiments and Technologies (CoNEXT)*.

[42] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. 2021. The nanopu: A nanosecond network stack for datacenters. In *Usenix Symposium on Operating Systems Design and Implementation (OSDI)*.

[43] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. 2005. *Linux Device Drivers, 3rd edition*. O'Reilly.

[44] Pravein Govindan Kannan, Nishant Budhdev, Raj Joshi, and Mun Choon Chan. 2021. Debugging transient faults in data centers using synchronized network-wide packet histories. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[45] Samuel Larsen and Saman Amarasinghe. 2000. Exploiting superword level parallelism with multimedia instruction sets. *ACM SIGPLAN Notices* (2000).

[46] Hejing Li, Juhyeng Han, and Dongsu Han. 2020. Leveraging SIMD parallelism for accelerating network applications. In *Asia-Pacific Workshop on Networking (APNet)*.

[47] Bowen Liu, Xinyang Huang, Qijing Li, Zhuobin Huang, Yijun Sun, Wenxue Li, Junxue Zhang, Ping Yin, and Kai Chen. 2025. CEIO: A cache-efficient network I/O architecture for NIC-CPU data paths. In *ACM SIGCOMM*.

[48] Scott A. Mahlke, Richard E. Hank, James E. McCormick, David I. August, and Wen-Mei W. Hwu. 1995. A comparison of full and partial predicated execution support for ILP processors. In *International Symposium on Computer Architecture (ISCA)*.

[49] Charith Mendis and Saman Amarasinghe. 2018. goSLP: globally optimized superword level parallelism framework. (2018).

[50] Samuel Midkiff. 2022. Automatic parallelization: an overview of fundamental compiler techniques. (2022).

[51] Vladimiro Paschali, Andrea Monterubbiano, Francesco Fazzari, Sebastiano Miano, and Salvatore Pontarelli. 2025. X2DP: eXtended eXpress Data Path. In *ACM SIGCOMM 2025 Posters and Demos*.

[52] Francisco Pereira, Fernando M.V. Ramos, and Luis Pedrosa. 2024. Automatic Parallelization of Software Network Functions. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[53] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. 2018. Floem: A Programming System for NIC-Accelerated Network Applications. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[54] Boris Pismenny, Adam Morrison, and Dan Tsafrir. 2023. {ShRing}: Networking with shared receive rings. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[55] Luigi Rizzo. 2012. Netmap: a novel framework for fast packet I/O. In *USENIX annual technical conference (ATC)*.

[56] Ira Rosen, Dorit Nuzman, and Ayal Zaks. 2007. Loop-aware SLP in GCC. In *GCC and GNU Toolchain Developers' Summit*.

[57] Hugo Sadok, Miguel Elias M. Campista, and Luís Henrique M. K. Costa. 2018. A Case for Spraying Packets in Software Middleboxes. In *ACM Workshop on Hot Topics in Networks (HotNets)*.

[58] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. 2016. Packet transactions: High-level programming for line-rate switches. In *ACM SIGCOMM*.

[59] Athinagoras Skiadopoulos, Zhiqiang Xie, Mark Zhao, Qizhe Cai, Saksham Agarwal, Jacob Adelmann, David Ahern, Carlo Contavalli, Michael Goldflam, Vitaly Mayatskikh, et al. 2024. High-throughput and flexible host networking for accelerated computing. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[60] Mark Sutherland, Siddharth Gupta, Babak Falsafi, Virendra Marathe, Dionisios Pnevmatikatos, and Alexandros Daglis. 2020. The nebula rpc-optimized architecture. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*.

[61] Qiongwen Xu, Sebastiano Miano, Xiangyu Gao, Tao Wang, Adithya Murugadass, Songyuan Zhang, Anirudh Sivaraman, Gianni Antichi, and Srinivas Narayana. 2025. State-Compute Replication: Parallelizing High-Speed Stateful Packet Processing. In *Usenix Symposium on Networked Systems Design and Implementation (NSDI)*.

[62] K. Zhang, D. Zhuo, and A. Krishnamurthy. 2020. Gallium: Automated software middlebox offloading to programmable switches. In *ACM SIGCOMM*.