TOWARDS HIGH-PERFORMANCE HANDLERS ON THE PSPIN PLATFORM

Tiancheng Chen, Yunxin Sun, Pengcheng Xu

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

The PsPIN platform offers a complete implementation of the sPIN in-network computing paradigm, allowing practical handlers to be developed for real-world scenarios. However, the existing use-cases for PsPIN do not sufficiently demonstrate handlers' capabilities and performance bottlenecks. In this work, we introduce three new use-cases to further diversify the handler portfolio under different scenarios of in-network computing. Furthermore, we study the performance bottlenecks and optimizations for such use cases. Finally, we propose three pieces of advice for developing high-performance handlers on the PsPIN platform.

1. INTRODUCTION

Streaming processing in the network (sPIN) [1] defines a unified programming model and architecture for network acceleration beyond simple RDMA. It provides a user-level interface similar to CUDA for compute acceleration, taking the specialities and constraints of low-latency line-rate packet processing into consideration. It also defines a flexible and programmable network instruction set architecture (NISA) that lowers the barrier of entry and supports a large number of use-cases.

MCS [2, 3] lock is of wide use in parallel computing. MCS lock is a standard algorithm for providing locks with FIFO guarantees (and thus fairness) and scalability. MCS lock is typically used in high-contention use cases. Although locks and contention levels should be avoided as much as possible in PsPIN, it is crucial to understand how basic lock algorithms like MCS perform under the framework.

Collective operations [4] are some of the most critical operations at scale [5]. AllReduce is a collective where all nodes contribute with some data aggregated according to a given operator. At the end of the collective, all nodes know the aggregation result.

Particularly relevant for ML/AI [6], is the use of Sparse allreduce [7, 8], where the input data is sparse, and where it might be more efficient to transmit only the non-zero values. Such usage poses some computational challenges, depending on the format used to transmit, store, and access the data in the NIC.

Learning applications in real-time scenarios such as robotics and autonomous vehicles require low-latency inference of neural networks [9, 10]. The usage of small models in these scenarios is also common due to the tight energy footprint required in embedded deployment scenarios [11]. A single-layer perceptron (SLP) [12] is a feed-forward network based on a threshold transfer function that, given an input vector, returns either 0 or 1 according to the threshold function. While mathematically simple, such a model provides adequate computational intensity during inference and synchronization challenges during training. Experience from implementing and analyzing the simple model will facilitate the development of more complicated neural networks in PsPIN.

In this work, we implement and analyze the three following use cases as packet handlers on the PsPIN platform: a) MCS locks; b) Sparse reduction; c) Single-layer perceptron. We explain the design choices made in detail when implementing these handlers. We then present the performance measurements of the handlers under synthesized workloads of different scales. We also analyze the measurements to locate the bottlenecks of each handler. We deliver several observations on how to implement high-performance packet handlers with PsPIN:

- In general, PsPIN is not a framework designed for sharing data structures among data packets. The length and the strength of critical sections should be minimized as much as possible.
- Good utilization of ISA extensions in the HPUs deliver throughput increase when the workload is compute-heavy;
- Contention on remote locks kills performance. Hierarchical locking or batched synchronization may help by eliminating such contentions.

The authors are listed in alphabetical order of their last names.

Related work. The original PsPIN paper [13] introduced a set of handlers as examples to characterize the capabilities of the platform. However, most handlers introduced implement small building blocks for larger applications, e.g. scatter, aggregation, and reduction. Few of the original example handlers have a high *operational intensity*, featuring primarily data manipulation and simple accumulation. Furthermore, the examples also did not make good use of the available synchronization mechanisms provided by the platform. In contrast, handlers for larger-scale use cases usually require some degree of synchronization between processing elements.

2. BACKGROUND

In this section, we formally define the three use cases discussed in this work. We also include a brief analysis of any data dependencies involved.

MCS locks. The MCS lock uses a linked list, and each node spins on its own flag. The lock is fair because it is equal to a FIFO queue. It is also contention free because each thread spins on its own local variable. In PsPIN (or, more generally, the in-network computing paradigm) framework, we put sharing data structures on L1 (local to the smart NIC) or L2 memory (shared among smart NIC or clusters). When a data packet arrives, the smart NIC handles the packet. The critical section is where the handler touches the sharing data structures. In this use case, we modify the sharing data structures in the critical section. We use the MCS lock algorithms to coordinate between threads. For example, the sharing data structures could be as simple as an integer. When a packet arrives, the handler adds the shared integer by one. The use case is simply equal to counting the number of data packets received.

Sparse reduction. The smart-NIC switches are organized in a tree manner. Each switch has several children and one parent, except for the root switch, which has no parent. The leaf switches are connected to hosts. AllReduce operation starts as the leaf switches send messages to parent switches. Their parent finishes collective operation on all incoming messages and sends them to its parent. Once the root NIC finishes aggregation, the result is broadcast to hosts. Since it is trivial to achieve line-rate processing in the broadcasting phase, only the aggregation phase is benchmarked.

Single-layer perceptron. A single-layer perceptron [12] is an algorithm for learning a binary classifier called a threshold function: a function that maps its input $\vec{x}$ to an output value $f(\vec{x})$:

$$f(\vec{x}) = \begin{cases} 1 & \text{if } \vec{w} \cdot \vec{x} + b > 0, \\ 0 & \text{otherwise} \end{cases}$$

where $\vec{w}$ is a vector of real-valued weights, $\vec{w} \cdot \vec{x}$ is the dot product $\sum_{i=1}^{m} w_i x_i$, and b is the bias. In neural networks, such a perceptron is an artificial neuron with the Heaviside step function as the activation function. The term *single-layer* perceptron distinguishes the model from general feed-forward ANNs, also referred to as *multilayer* perceptrons.

The learning algorithm for single-layer perceptrons is simpler than general ANNs, which require backpropagation or more sophisticated algorithms. We define the following variables: • $r \in (0, 1]$: learning rate of the perceptron; • $y = f(\vec{z})$: output of perceptron with input $\vec{z}$; • $D = \{(\vec{x}_1, d_1), \ldots, (\vec{x}_s, d_s)\}$: training set of s samples; • $x_{j,i}$: the i-th feature of the j-th training input. For simplicity, make $x_{j,0} = 1$; • w_i: the i-th value in the weight vector, to be multiplied by the value of the i-th input feature. As $x_{j,0} = 1$, w_0 is effectively the bias, eliminating the need for a separate bias b;

The learning algorithm comprises the following steps: 1. Initialize $\vec{w}$ to $\vec{0}$. 2. For each training sample in D: (a) Calculate the actual output: $y_j(t) = f[\vec{w}(t) \cdot \vec{x}_j]$ (b) Update the weights: $\vec{w}(t+1) = \vec{w}(t) + r \cdot (d_j - y_j(t))\vec{x}_j$ 3. Stop after running for a set number of iterations. Without proof, we claim that it always converges on *some* solution in the case of a linearly-separable training set.

3. IMPLEMENTATION

In this section, we describe the implementation details of the use cases as handlers on the PsPIN platform.

3.1. MCS Locks

We implement the MCS lock in the following way. The MCS lock is implemented in a linked list. When a thread wants to acquire a lock, it inserts itself into the linked list in a FIFO manner. Each thread just spins on its thread-local variable indicating the lock status, so there is minimal contention. If the node becomes the tail of the queue, it means the thread has become the oldest thread, and it is qualified to acquire the lock. In this case, the local variable will be locked, and the thread will be woken up. Leaving lock is simple; we simply unlock the thread-local variable indicating the lock status and atomically leave the queue. The operation of the critical section involves a loop iteratively modifying the sharing data structures. We denote the strength of the critical section by the length of the loop.

3.2. Sparse Reduction

The implementation is based on FLARE [14] project. An AllReduce vector is separated into several blocks; the range

of data index in each block and the size of each block are determined in the configuration phase. The scheduler schedules packets of the same block to the same cluster so that the buffer for the block can be stored in low-latency L1 memory, avoiding costly remote access. After aggregating data from all children, the NIC sends data in the buffer to the parent.

Sparse data. The sparse data is represented as index and data. There are two data representations in the local buffer, dense array and hash table. Dense arrays treat sparse data as dense data, faster but occupying too much memory at high sparsity. The hash table approach is not as fast, and the buffer simply sends the data to the parent if the entry is occupied. *Sparsity is defined as the inverse of sparsity ratio.*

Avoiding stalls. When there is only one buffer in use, messages of the same block have to be processed sequentially, which may cause HPU stalls when many children switch are sending messages simultaneously. The double buffer version eases this problem. From the benchmark result, it reaches the balance of efficient processing and memory usage. If the block size is too large so that there are not enough blocks scheduled to all clusters, some HPUs are idle, which prevents the NIC from reaching max bandwidth. Ideally, this can be solved by dynamic block size or having multiple AllReduce operations in parallel.

3.3. Single-layer perceptron

We implement the two phases of operation, *training* and *inference*, as two different types of messages; different types of messages are specified in Table 1. Each packet contains a header with the type as a tag for the handler to act. The message payload contains input data in batches. The exact packet layout for all three types of messages is shown in Figure 1.

We show the packet processing routine in Figure 2. The init handler `slp_hh` first initializes the training weight lock used by all clusters. Then, during the training phase, all HPUs in all the clusters update a single copy of the weight vector on the first cluster exclusively, competing for the training weight lock stored in the scratchpad of the first cluster. This serializes the update of the weight vector to ensure correctness while still allowing the error to be computed in parallel. After processing the fit message, the finish handler of the fit phase broadcasts the master weight to all clusters for use during the inference phase. Finally, after finishing the processing of all inference packets, the tail handler performs clean-up and report statistics.

4. EVALUATION

In this section, we present the experiment results of the use cases implemented. We analyze the results to identify different factors of the handler implementation that affect performance.

Type	Direction	Description
`FIT`	Ingress	For calculate loss and updating weight vector
`PREDICT`	Ingress	For running prediction and generating response packet
`RESPONSE`	Egress	For returning prediction results

Table 1: Three different types of messages involved.

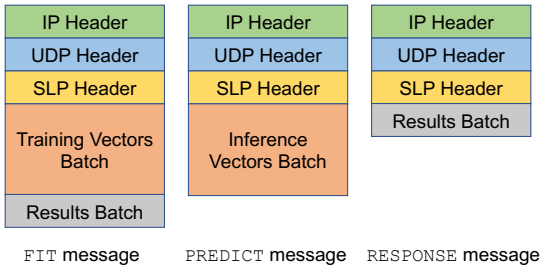

Fig. 1: The layout of training, inference, and response messages. The SLP header indicates the type and length of the following payload.

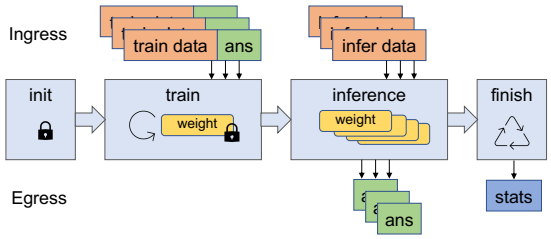

Fig. 2: The overall flow of handler execution during fit and prediction.

Experimental setup. All performance measurements are taken using the PsPIN cycle-accurate Verilator [15] model with 4 clusters and 8 HPUs each and an L1 cache size of 1 KB. The RIC5Y core [16] used in the platform supports integer SIMD with at most 32-bit vectors, allowing at most 4-way SIMD with `int8_t`. The target clock speed is 1 GHz, and the target line rate is 400 Gbps. We obtain the time information from the timestamps in simulation traces of instruction issuance.

4.1. MCS Locks

Different strength of critical section. In Figure 3 left, we present the feedback throughput and the packet latency wrt the strength of the critical section (we denote the strength of the critical section by the number of loops used to mod-

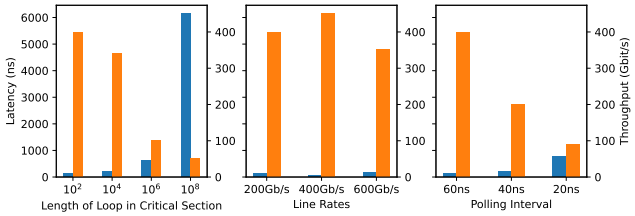

Fig. 3: **Left**: Packet Latency and Feedback Throughput under Different Critical Sections; **Middle**: Packet Latency and Feedback Throughput under Different Line Rates; **Right**: Packet Latency and Feedback Throughput under Different Polling Interval

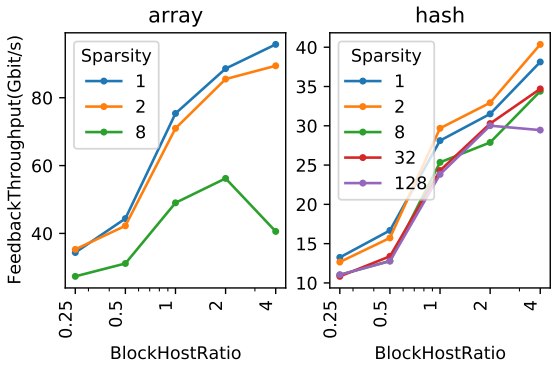

Fig. 4: Input throughput (Gbps) of single buffer sparse reduction with dense array (left) or hash table (right).

ify the sharing data structures iteratively). We could conclude that PsPIN is not designed for dependent data packet processing. The contention between data packets should be minimized as much as possible.

Different line rates. We also test how MCS locks perform under different line rates. Line rates are the theoretical maximum throughput of the system. We test the MCS lock with the length in the critical section to be $i = 10^6$. We show the result in Figure 3 middle. The result shows that when the line rate increases modestly, the achieved throughput could catch up with the benefits from the line rates. However, the critical section becomes the bottleneck when the line rates increase past a threshold, and the throughput saturates.

Different polling interval. The polling interval is the frequency the PsPIN fetches the data packet. The bigger this number, the more frequently the data packet will be fetched. It is easy to see that the more frequent the data packet arrives, the higher the contention will be under the MCS lock side. We show the result in Figure 3 right. We could see that PsPIN is not scalable in high-contention scenarios. The contention between different data packets should thus be minimized as much as possible. We should also contain the polling interval to receive the maximal performance.

4.2. Sparse Reduction

Block/host ratio. The input bandwidth at $\{8, 16, 32\}$ hosts and $\{8, 16, 32\}$ blocks is investigated. Figure 4 shows the comparison of single buffer and dense array/hash table at different sparsity. Notice that this result is valid for the current hardware setting, which is 4 clusters with 8 HPUs per cluster. The drop of dense single buffer at sparsity 8, block/host ratio 4 is as expected because the large dense buffer eats up too much memory. The scheduler will not schedule blocks that arrive later, as there is not enough memory left on the cluster.

Data representation. Two cases are examined as shown in Figure 5. Under low contention (8 hosts 32 blocks), the array representation outperforms hash representation except for the double buffer case at sparsity 8, where the large buffer size prevents scheduling of other blocks. Double buffer still outperforms single buffer because the next packet of the same block can still arrive before the previous one finishes aggregation. The hash representation is not sensitive to sparsity, as expected. Under high contention (32 hosts 8 blocks), double buffer achieves averagely 60% higher input bandwidth than single buffer case (both array and hash), but still significantly slower than low contention case.

Multiple streams. When there are multiple AllReduce operations, the behaviour is similar to that of #streams times of blocks (e.g. 1 stream 32 blocks equivalent to 4 streams 8 blocks). The performance plots are omitted for conciseness. However, we need to notice that if previous operations exhausted the L1 memory, the new AllReduce would not be scheduled until the old block finishes, even though there might be free HPUs to be dispatched.

Cases not considered in simulation: (a) Switch topology might be different for these AllReduce operations, meaning the NIC workload of each operation might vary, which affects the bandwidth. (b) Different datatypes or collective operations might take different cycles to complete and break the balance of the workload of different streams.

Data type and SIMD. See Figure 6. On the RI5CY core, 2 int16/int8 addition are vectorized into one. As SIMD is performed on contiguous memory addresses, the only use case in sparse reduction is to aggregate two buffers in the flushing phase. As it is not a major step of block aggregation, the SIMD provides little acceleration.

Compression ratio. It is defined as $\#\text{outPackets}/\#\text{inPackets}$. We observed hash-based method suffers from a high compression ratio ($> 50\%$) under high sparsity ($>= 8$), which indicates the low efficiency of aggregation. This is caused by both high sparsity and the splashing of hash table. Increasing the hash table size is only mitigation. It is more reasonable to send the messages to parents with a dense

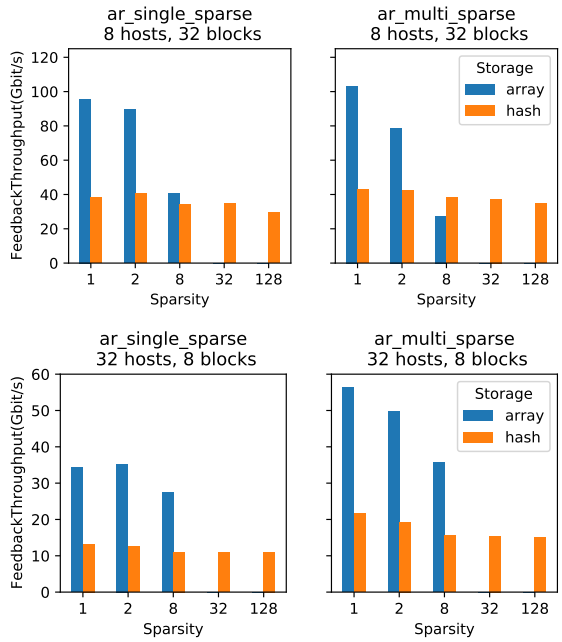

Fig. 5: Input throughput (Gbps) of sparse reduction with different data representation under low (**up**) and high (**bottom**) contention.

Dimension	Range of values
Execution modes	{`predict`, `fit`}
#Packets per msg	$\{2^n \mid n \in [3, 10]\}$
Packet size	$\{2^n \mid n \in [7, 10]\}$
Data type	{`int8`, `int16`, `int32`, `float`}
Vector length	$\{2^n \mid n \in [3, 7]\}$

Table 2: The dimensions for evaluation of the SLP handlers.

buffer directly.

4.3. Single-Layer Perceptron

Experiment setup. We study the performance of the two phases of execution separately due to their vastly different characteristics: the prediction phase is contention-free and computation-heavy, while the fit phase involves significant contention on the lock for weight update. The evaluation space is defined as the Cartesian product of the dimensions shown in Table 2, with points taking too long to execute or require too much memory omitted[1].

Linear scalability of predict. Figure 7 shows that during prediction the near-linear scalability of the prediction phase execution. The increasing packet counts saturate the

[1]As discussed with one of the mentors, the SLP use case builds upon a version of PsPIN with a workaround to one of the known bugs. The results presented here may not be reliable.

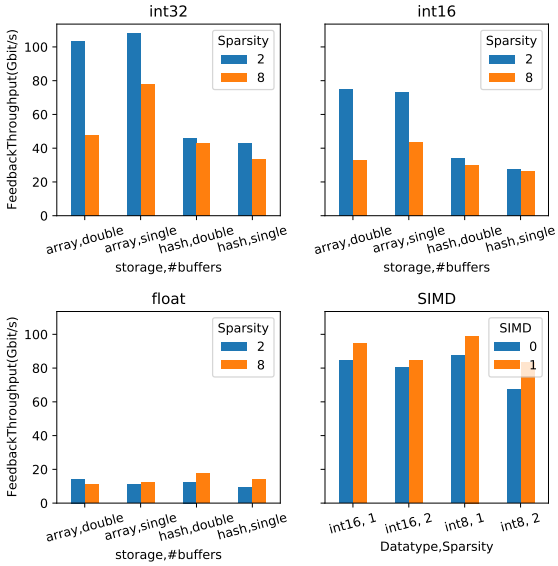

Fig. 6: Input throughput (Gbps) of sparse reduction with int32 (upper left), int16 (upper right), float (lower left), and SIMD result (lower right).

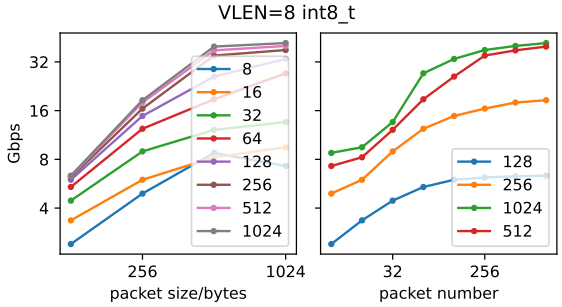

Fig. 7: Throughput of the predict phase with different packet sizes and counts. Lines in the left denote different packet counts and in the right different packet sizes.

number of HPUs available, while the increasing packet sizes demonstrate the increase of computation ratio to other overhead in handler execution. The scaling of packet size ends at 1024 bytes, the size of the L1 packet buffer.

ISA extensions. ISA extensions help increase throughput by lowering non-compute overhead. The utilization of ISA extensions for different data types and vector lengths are shown in Table 3. Figure 8 shows the best-case throughput for different DTYPEs and VLENs. The compiler failed to automatically vectorize the compute loop for all cases, resulting in 2- and 4-fold lower throughput than theoretically possible in 8-bit and 16-bit integer compared to the 32-bit situation.

We also note the performance drop for VLEN=32 due to the compiler not fully unrolling the hot loop but turning

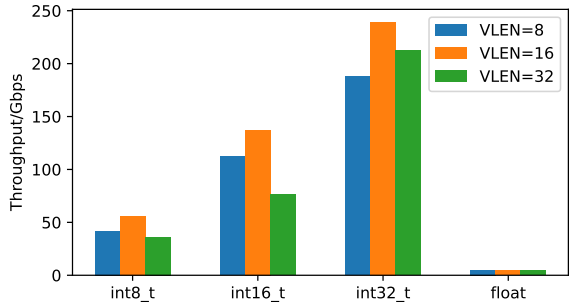

Fig. 8: Best-case prediction throughput for given vector length and data type. Note that the `float` results are significantly lower due to the use of software floating point as of the lack of FPU.

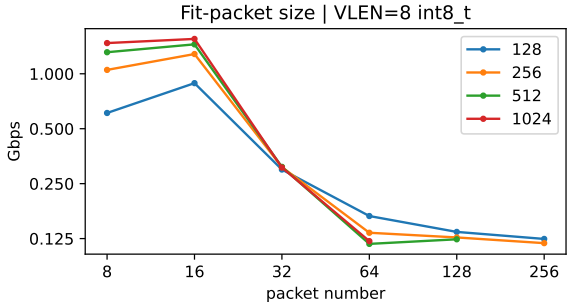

Fig. 9: Throughput during fit phase with different packet counts.

to use the hardware loop instructions. Despite this being faster than without the hardware loop support, overheads still incur compared to the fully unrolled cases.

DTYPE	VLEN	Extensions				
		pi	**hl**	**alu**	**mac**	**vec**
`int8_t`	8	✓	✗	✓	✓	✗
	16	✓	✗	✓	✓	✗
	32	✓	✓	✓	✓	✗
`int16_t`	8	✓	✗	✓	✓	✗
	16	✓	✗	✓	✓	✗
	32	✓	✓	✓	✓	✗
`int32_t`	8	✓	✗	✓	✓	✗
	16	✓	✗	✓	✓	✗
	32	✓	✓	✓	✓	✗

Table 3: ISA extensions utilized by different vector length (VLEN) and data type (DTYPE) handlers. Extensions available from RI5CY [17]: post-increment (**pi**), hardware-loop (**hl**), ALU extensions (**alu**), multiply-accumulate (**mac**), vectoral (**vec**).

Lock contention in fit. We observe that the contention over locks, especially remote locks, kills the performance of the handlers. Figure 9 shows the severe performance degradation in the fit phase with the increase of packet count due to increased contention on the lock for updating the weight vector. Figure 10 further shows that the more HPUs participate, the worse the contention becomes, with most of the time spent spinning and waiting for the lock. We also observe that remote HPUs (those in a different cluster from the main cluster) wait on average ten times longer than local HPUs. This is due to atomic operations on remote memory being significantly slower: the `amoor.w` instruction during spinning takes 5 cycles on a local scratchpad and 24 cycles

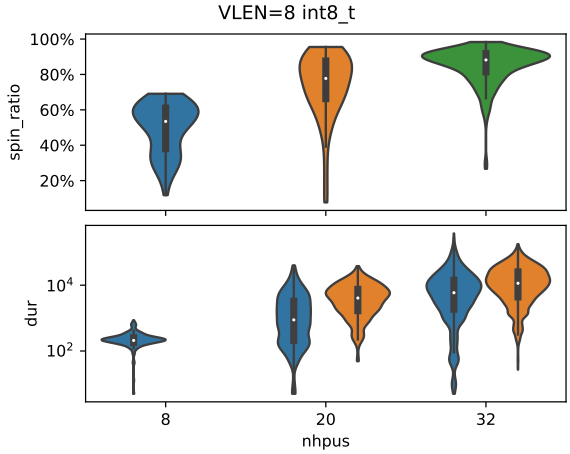

Fig. 10: **Up**: Ratio of the time HPUs spent spinning compared to overall time of execution. **Bottom**: Wait duration of local and remote HPUs. When `nhpus`=8, all scheduled HPUs are from the home cluster, thus no remote HPUs exist. `nhpus` denote the number of HPUs competing for the lock.

on a remote scratchpad.

5. CONCLUSION

We developed three more use cases for the PsPIN platform [13] and performed extensive performance analysis under different settings. We also provided advice towards optimizing handlers based on the analysis of bottlenecks of the three use cases. We hope that the addition of these use cases and their detailed performance analysis will further promote the adoption of in-network computing with PsPIN.

6. REFERENCES

[1] Torsten Hoefler, Salvatore Di Girolamo, Konstantin Taranov, Ryan E. Grant, and Ron Brightwell, "Spin:

High-performance streaming processing in the network," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, New York, NY, USA, 2017, SC '17, Association for Computing Machinery.

[2] Theodore Johnson and Krishna Harathi, "A simple correctness proof of the mcs contention-free lock," *Information processing letters*, vol. 48, no. 5, pp. 215–220, 1993.

[3] John M Mellor-Crummey and Michael L Scott, "Synchronization without contention," *ACM SIGPLAN Notices*, vol. 26, no. 4, pp. 269–278, 1991.

[4] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard Version 4.0*, June 2021.

[5] Sudheer Chunduri, Scott Parker, Pavan Balaji, Kevin Harms, and Kalyan Kumaran, "Characterization of mpi usage on a production supercomputer," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018, pp. 386–400.

[6] Tal Ben-Nun and Torsten Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *ACM Comput. Surv.*, vol. 52, no. 4, aug 2019.

[7] Huasha Zhao and John Canny, "Kylix: A sparse allreduce for commodity clusters," in *2014 43rd International Conference on Parallel Processing*, 2014, pp. 273–282.

[8] Huasha Zhao and John Canny, "Sparse allreduce: Efficient scalable communication for power-law data," 2013.

[9] Bichen Wu, Alvin Wan, Forrest Iandola, Peter H. Jin, and Kurt Keutzer, "Squeezedet: Unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving," 2019.

[10] Chang Gao, Antonio Rios-Navarro, Xi Chen, Tobi Delbruck, and Shih-Chii Liu, "Edgedrnn: Enabling low-latency recurrent neural network edge inference," in *2020 2nd IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, 2020, pp. 41–45.

[11] Yu Wang, Lixue Xia, Tianqi Tang, Boxun Li, Song Yao, Ming Cheng, and Huazhong Yang, "Low power convolutional neural networks on a chip," in *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2016, pp. 129–132.

[12] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain.," *Psychological Review*, vol. 65, no. 6, pp. 386–408, 1958.

[13] Salvatore Di Girolamo, Andreas Kurth, Alexandru Calotoiu, Thomas Benz, Timo Schneider, Jakub Beránek, Luca Benini, and Torsten Hoefler, "Pspin: A high-performance low-power architecture for flexible in-network compute," 2021.

[14] Daniele De Sensi, Salvatore Di Girolamo, Saleh Ashkboos, Shigang Li, and Torsten Hoefler, "Flare: Flexible in-network allreduce," 2021.

[15] "Welcome to verilator," https://www.veripool.org/verilator/, Accessed: 2022-01-14.

[16] Michael Gautschi, Pasquale Davide Schiavone, Andreas Traber, Igor Loi, Antonio Pullini, Davide Rossi, Eric Flamand, Frank K. Gürkaynak, and Luca Benini, "Near-threshold risc-v core with dsp extensions for scalable iot endpoint devices," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2700–2713, 2017.

[17] "Ri5cy: User manual," https://www.pulp-platform.org/docs/ri5cy_user_manual.pdf, Accessed: 2022-01-14.