Espresso: Exploiting the Sparsity Property in Event Sensors with Spatiotemporal Ordering

Leshan Li†, Hongyi Li†, Qingyuan Yang, Mingtao Ou, Rong Zhao*, and Xinglong Ji*
Tsinghua University, Beijing 100084, China †Contribute equally
*Corresponding Authors (xinglongji@tsinghua.edu.cn, r_zhao@tsinghua.edu.cn)

***Abstract*—Event-based vision sensors are novel cameras inspired by human eyes, capable of capturing the rapid motion of objects with the high-speed sparse event stream. However, it is challenging to efficiently stream process event data without demolishing its sparsity. In this paper, we design Espresso, an architecture for event-based vision processing that can efficiently stream spatiotemporal events while preserving sparsity. We implement Espresso on the FPGA platform and design experiments to compare the performance with embedded GPU and line-buffer-based accelerator. In real-world scenarios, Espresso achieves throughput up to 5000 fps, which is 5.1× higher than the embedded GPU.**

***Index Terms*—Event-based Vision Sensor, Accelerator, Sparsity, Dynamic Vision**

I. Introduction

Event cameras are bio-inspired imaging sensors that capture the changes in light intensity in the scene with high temporal resolution and low latency. Unlike traditional frame-based cameras, event cameras output a stream of asynchronous events, each of which contains the spatial location and the timestamp of the change in the scene [18]. State-of-the-art bio-inspired cameras can integrate events to generate images with a frame rate of up to 10,000 fps [10], [22]. The high temporal resolution and low latency make event data suitable for vision processing in high-speed motion scenes such as autonomous driving and robotics [13]. In addition, the sparsity of event data can significantly reduce the amount of data to be processed and stored, which is believed to benefit edge devices with limited resources and power [14].

However, the events generated by typical event cameras like DVS [18] are ordered temporally without the spatial correlation between events. Such temporal order hinders the efficient processing of event data because events without spatial correlation can hardly be processed in the stream. For traditional frame-based images with spatially continuous pixels (Fig. 1A), stream processing is a widely applied method to deploy vision algorithms, a typical design being the line-buffer [5], [20], [21], which enables spatially adjacent pixels to be streamed continuously into the computation blocks and achieves very high level of data reusing and very low latency as a result (processing each pixel takes only one clock cycle). Stream processing has also been proved promising for efficient processing on sparse data [16]. However, such a design struggles to work for events in temporal order. As for traditional vision processing algorithms which typically work on groups of neighboring pixels (such as 3×3 Sobel or 5×5 Gaussian), spatially out-of-order events can hardly be directly streamed into the computation blocks since the arrival of a single event offers no information about its vicinity (Fig. 1B).

Current works on event-based vision processing mainly follow two methods: 1. integrate the events to traditional frame-based dense images and pose frame-based vision algorithms on them [1], [2], [15]; 2. design novel vision algorithms customized to temporal-order event stream for real-time stream processing on events [4], [7]. The former can be aided with architecture design like line-buffer to achieve relatively low latency, but the act of framing inevitably demolishes the sparsity of event data (Fig. 1C), which hinders the further reduction of latency and memory. The latter maintains sparsity of event data, by which the latency can be cut down to 130ns/event, but the fundamental different mechanisms of algorithms may cause incorrect results under certain conditions (overall false rate over 10% on event-camera dataset [4]). Recently, some works on sensors have noticed the shortcomings of temporal-ordering events and have been trying to integrate spatial features directly into the hardware to output events in spatiotemporal order (events within every frame are in spatial order) [22] (Fig. 1.D). However, **to the best of our knowledge, there has been no dedicated design to fully utilize such a spatiotemporal order feature.**

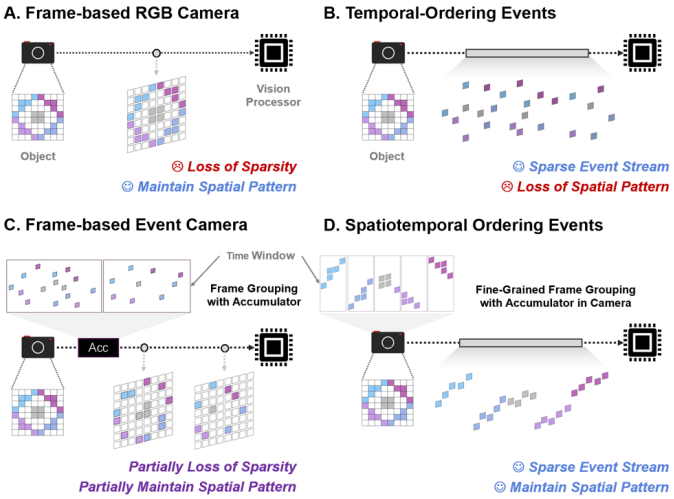

Fig. 1. The motivation of Espresso: the variation of event stream coding.

To address this issue, we propose Espresso, a vision processing architecture for spatiotemporal ordering events, which al-

lows the flexible implementation of traditional computer vision algorithms. Espresso employs an order-maintaining method and is able to efficiently stream process events. Besides, Espresso completely preserves and leverages the sparsity of event data by skipping computation on zeros and saving the memory of zeros. As a result, Espresso achieves high-level data reusing and very low latency. Specifically, our main contributions are as follows:

- We propose the N-Pending-FIFOs design to solve the challenge of order mismatching between input and output streams, enabling the efficient stream process of spatiotemporal ordering events.
- We propose the Shift-Hash Table design for the storing of events, which greatly reduces the memory access latency and preserves the sparsity of event data for low memory consumption.
- Based on N-Pending-FIFOs and Shift-Hash Table, we build Event-Scheduler which takes in events and outputs windows. Applying the Event-Scheduler to each layer of a vision algorithm, we establish the Espresso architecture to achieve efficient event stream processing.

We implement Espresso on an embedded FPGA platform and evaluate its performance. As a result, Espresso achieves a speedup of 5.1× than Jetson Orin NX (with Cuda-based OpenCV) and 3.3× than the line-buffer-based accelerator on FPGA. Such design sheds light on efficient dynamic vision processing with both low overhead and high speed.

II. Background

A. Event Camera

Event cameras are bio-inspired imaging sensors that can respond to the change in light intensity. Instead of generating frame-based images at a fixed rate, event cameras asynchronously detect pixel brightness change and generate an event stream, each event encoded with its position, timestamp, and value [13]. Compared to traditional cameras, event cameras offer several key advantages: 1. High temporal resolution (less than $1\mu s$) and high dynamic range (up to 140dB), enabling event cameras to perform effectively in high-speed or high-dynamic-range scenarios such as high-speed drones and autonomous driving [8], [13]. 2. Low power consumption and the inherent sparsity of event data, making event cameras well-suited for edge computing devices with limited power and resources [14] [8]. State-of-the-art event cameras can generate 4.6G events per second [12] or up to 10K event frames per second [10]. Some works have noticed the shortcomings of temporal-ordering event streams and integrated spatial features to generate spatiotemporal-ordering event streams [22]. Spatiotemporal order potentially enables the efficient stream processing of events, which is regarded as the trend of event-based sensor development. However, this feature has not been well utilized in computing devices.

B. Event-based Vision Processing

Since the invention of event cameras, plenty of studies on event-based vision processing have been carried out. [15] track the 6-DoF fast camera motions based on event frame using a geometric method. [2] fuses measurement from the event camera with IMU, realizing Visual Inertial Odometry on the event camera dataset. [1] further combines standard frame with events and inertial measurement and achieves robust visual SLAM with high accuracy and high speed. Recently, there have also been works applying DNN to event-based vision algorithm [17]. However, most of these works pre-process the events by accumulating them into frame-like representation, which demolishes the asynchronous and sparse feature of event data. There have also been works to avoid accumulating events to frames, trying to further exploit the feature of event data by software or hardware design. [7] proposes a novel feature detection algorithm that can process events asynchronously using a Surface of Active Events. [4] further develops a novel tracking algorithm, and achieves real-time process on event (130ns/event for detection and 1970ns/event for tracking) on CPU. Unfortunately, the novel processing mechanism for temporal-ordering events sees a drop in accuracy. Some works also try to deploy spiking neural networks (SNN) on neuromorphic hardware such as SpiNNaker [9], TrueNorth [3], and Loihi [11], in order to accelerate the event processing. However, the inference latency of the heavy network still proves unsatisfactory (105ms on DVS Gesture Recognition dataset [14]).

III. Method

Spatiotemporal ordering events maintain spatial order, which brings possibilities for efficient streamed processing. Fig. 2 illustrates the ideal architecture for spatiotemporal ordering events processing. For stream processing of dense images, previous works use the line-buffer shown in Fig. 2A to realize pixel-level streaming and pipelining. For the spatiotemporal ordering events, in order to maintain the sparsity in a frame-based image, all zero points are skipped and each event contains the value (Val) and the position (Pos) (Fig. 2B).

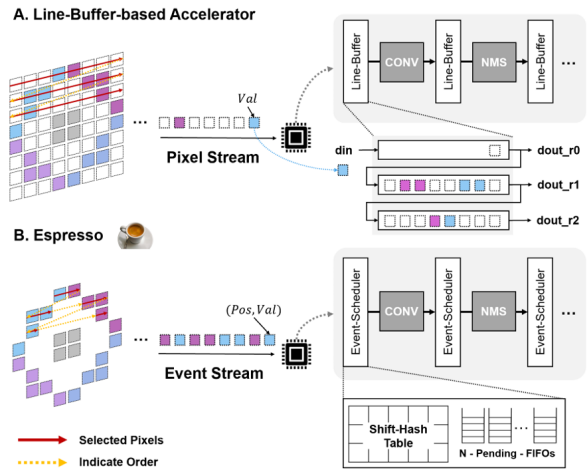

Fig. 2. Design goal of Espresso.

However, compared with the processing of dense pixel streams, two key challenges prevent us from achieving the

efficiency of stream processing and fully utilizing the sparsity of event cameras at the same time. Fig. 3 concludes these two issues:

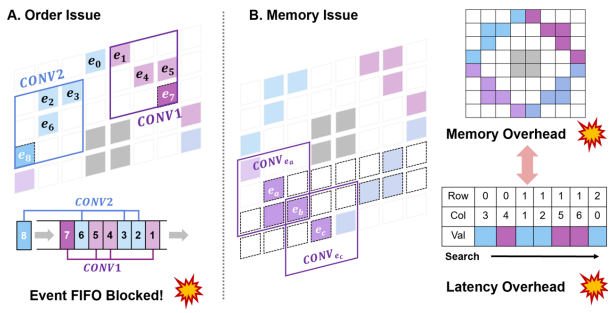

Fig. 3. Challenges of sparse event stream processing.

1) **Order Issue**: The ideal stream processing hardware implements communication between computational operators based on FIFOs [6]. However, for the convolution (CONV) example in Fig. 3A, when e_7 enters the queue[1], all the events for CONV1 are ready. But at this time, e_2 is still waiting for e_8 to complete the subsequent CONV2, and therefore it cannot exit the queue immediately, the FIFO is blocked. In a nutshell, *order mismatching exists between input and output stream*.
2) **Memory Issue**: As shown in Fig. 3B, e_b is needed for both e_a centered CONV and e_c centered CONV, indicating searching for previous events is inevitable. *The sparse memory of event data with discontinuous spatial addresses implies an address-matching process is possibly required during memory access*, which may bring great latency overhead.

For the order issue, we bridge the gap between input order and output order by maintaining the "Window Address" (WinAddr) order with N-Pending-FIFOs in section III-A. For the memory issue, we develop a hash-based data structure, dubbed as Shift-Hash Table in section III-B to balance the trade-off between memory and latency overhead. Finally, section III-C gives the whole picture of the collaboration between N-Pending-FIFOS and Shift-Hash Tables.

A. N-Pending-FIFOs for Order Issue

Fortunately, we note that window sliding is the most widely used computational scheme in vision processing with the following two advantages:

1) Fixed window size for a certain algorithm.
2) If window sliding is ordered, then for a particular pixel on an image, the windows that sweep over that pixel remain in the same order spatially.

Fig. 4A illustrates the second advantage. We use ★ to represent the event, ♦ to mark the center of the window sliding over the ★. WinAddr is defined as the address of ♦. For each ★, the number of its ♦ is the same as the window size.

[1]The subscript of e_i indicates the order in which the events arrive.

To solve the order issue, we leverage the feature of window sliding and design hardware of N rows of Pending FIFOs. A key design philosophy is: **For each event ★, focus only on the order and calculation of its associated WinAddr ♦.**

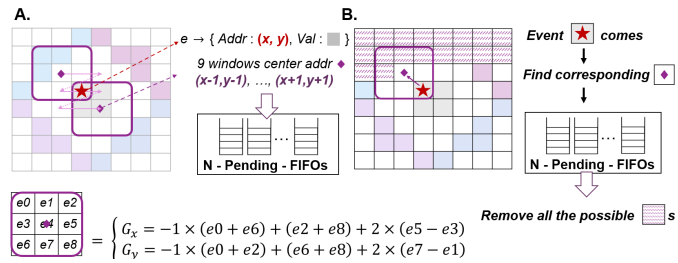

Fig. 4. The relationship between event order at an image view and N-Pending-FIFOs operation.

For the case of 3×3 sobel kernel (a computer vision operator widely used in edge detection [19]) in Fig. 4A, when a new event e (marked as ★) with spatial address (x, y) arrives, the nine associated WinAddrs (marked as ♦) will be triggered, pushed into the N-Pending-FIFOs. This operation implements the conversion of all subsequent operations from the event perspective to the WinAddr perspective, essentially **updating all event-related calculations rather than the event itself**.

The pop mechanism of N-Pending-FIFOs is also event-driven, as shown in Fig. 4B. For the coming events ★, all the WinAddr within the N-Pending-FIFOs with a smaller address than the smallest address of the ★'s ♦s will be popped. Because the arrival of the event ★ is spatially ordered, so is its smallest ♦. The arrival of the event ★ reveals that locations smaller than the address of ♦ will no longer be calculated in subsequent computations, and can be removed from the FIFOs.

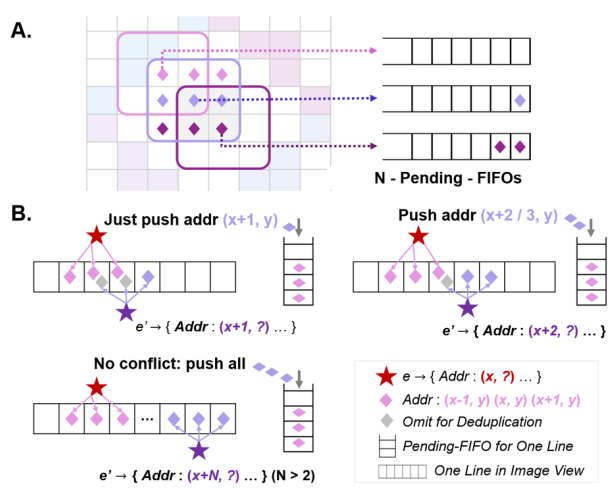

Fig. 5. Mechanism of N-Pending-FIFOs in details.

The operation above successfully bridges the gap between input order and output order. However, a single FIFO still faces the problems mentioned in Fig. 3A, and a more dedicated design of the FIFO is also needed. Our design idea is to

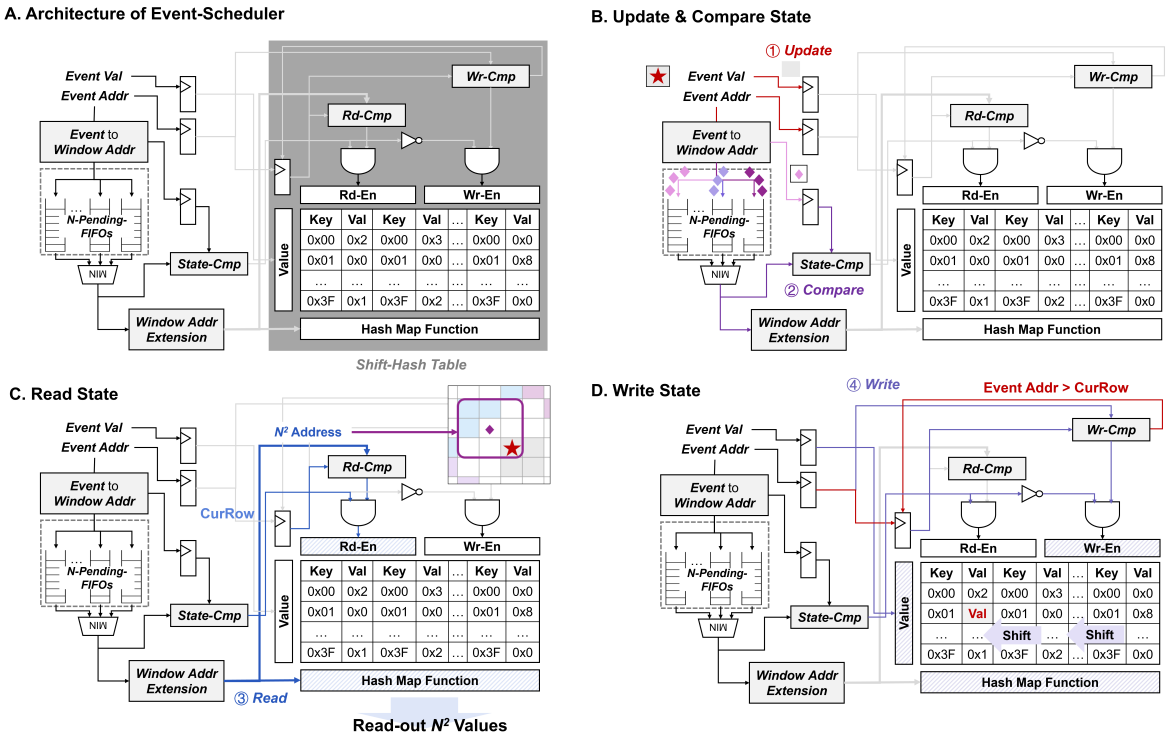

Fig. 6. The design of Event-Scheduler.

simplify the discussion of the problem by decomposing the order in two-dimensional space into one dimension.

Fig. 5 shows the design details of the N-pending-FIFOs. For the Sobel kernel in Fig. 5A, there should be 3 Pending FIFOs. Each of the N FIFOs is responsible for storing the addresses of one row of the WinAddrs ♦ triggered by an event. The advantages of this design are two-fold:

1) By comparing the current event's spatial address with the last event's spatial address, we can design logic to avoid repeated WinAddr in each FIFO (Fig. 5B).
2) Besides, the window addresses in each FIFO will always be in spatial order for both push and pop.

Therefore, we can simply complete the pop operation by comparing the addresses at the head of the FIFO, avoiding complex address lookups and searches.

In summary, we use WinAddr as the medium, supplemented by N-Pending-FIFOs, to realize a solution to the problem of inconsistent order of input and output.

B. Shift-Hash Table for Memory Issue

To solve the memory challenge, the Shift-Hash Table, an N-row hash table based on the shift register, is designed. Considering the sparsity of events, we set the length of each row less than the image width. The privilege of only N rows instead of the whole image being necessary to store also comes from the stream processing mechanism.

Specifically, upon the arrival of event ★ with spatial address (x, y), all the computation of pixels with address row less than $y - N$ has been completed, so that the memory of pixels over N rows ahead of ★ can be released. The releasing process is implemented by the shift register design in practice. The above design avoids excessive memory overhead.

TABLE I
OVERALL AND MEMORY READ LATENCY PERFORMANCE OF HASH AND NON-HASH DESIGN

	Sobel3		Sobel3 + Gaussian5	
	Non-Hash	Hash	Non-Hash	Hash
Overall Latency	$209\mu s$	$66\mu s$	$524\mu s$	$112\mu s$
Mem-Rd Latency	$137\mu s$	$11\mu s$	$422\mu s$	$38\mu s$
Ratio	**65.6%**	**16.7%**	**80.5%**	**33.9%**

Another important benefit brought by the Shift-Hash Table is the low latency of memory reading, which allows the reading of $N \times N$ values within one clock cycle. Compared with other non-hash discontinuous memory design which requires an address-matching process for memory read, the latency improvement can be substantial. As Tab. I shows, for example only containing Sobel3, memory based on non-hash design causes much greater latency overhead than Hash design. The case is even worse when the number of layers grows, where the memory read process accounts for over 80% of the overall latency. On the contrary, the shift-hash design achieves an acceptable latency overhead, an overall speedup being $3.2\times$ and $4.7\times$ faster than the non-hash design.

From the perspective of the system, the Mem-Rd latency, which directly influences the average time for processing one event, can be of great influence on the overall latency, and the Shift-Hash Table which limits Mem-Rd latency to 1 clock cycle is the optimal choice in this design.

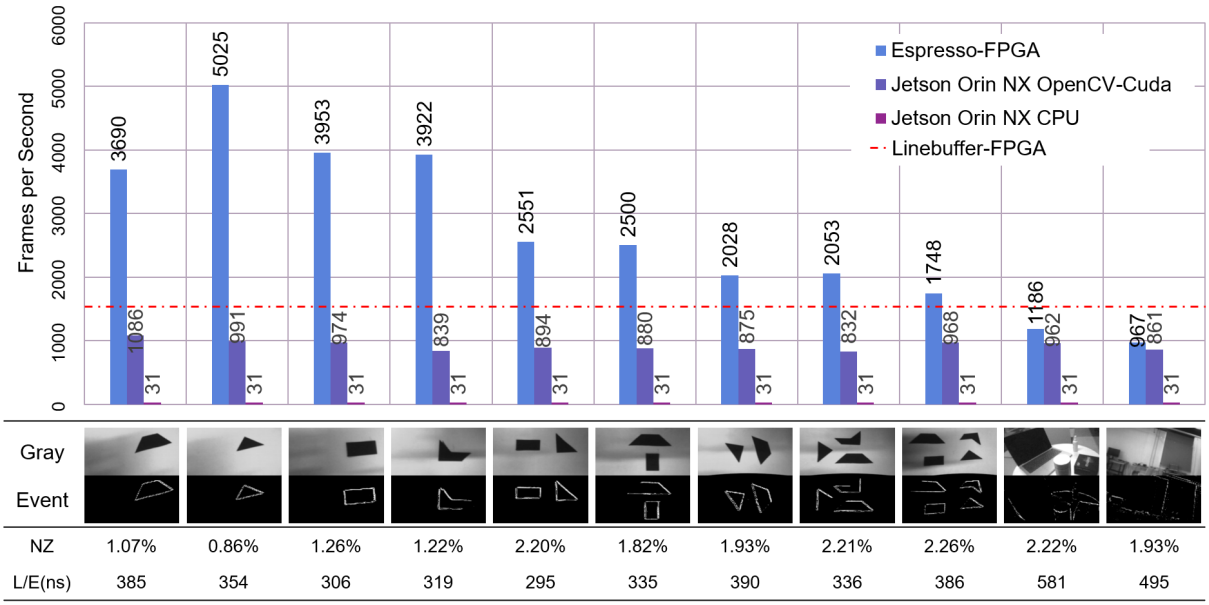

Fig. 7. Evaluation on different scenarios. (NZ: percentage of non-zero values, L/E: latency/event, resolution: 256×256)

C. Event-Scheduler Architecture

Event-Scheduler enables the cooperation of N-Pending-FIFOs and Shift-Hash Table with a finite state machine (FSM) with 4 states: update, compare, read, and write. Fig. 6A illustrates the overall architecture; Fig. 6B, C, and D show the execution behavior within Event-Scheduler in different states.

The update state (Fig. 6B) is the default one. Upon receiving the event ★, Event-Scheduler first converts it to the corresponding WinAddrs and pushes them into the N-Pending-FIFOs. The smallest WinAddr ♦ will be copied into separate registers to implement the pop mechanism of Fig. III-AB. Then, the FSM transfers to the compare state.

In the compare state (Fig. 6B), the minimum address of the heads of N-Pending-FIFOs is selected and compared with the copied WinAddr ♦ to judge whether to read the Shift-Hash Table or not. If the minimum address is smaller, it indicates that all events for the convolutional computation centered on this address are ready. Therefore, the FSM will go into the read state. Otherwise, it goes into the write state and writes the value of the event value to the shift-hash table.

In the read state (Fig. 6C), the WinAddr is first extended to its $N \times N$ surrounding addresses (illustrated in the upper right of the subfigure) which are then sent simultaneously to the Shift-Hash Table to read $N \times N$ values within one clock cycle. After these, revert back to the compare state.

In the write state (Fig. 6D), the current event's value and address are sent to the Shift-Hash Table, and after the writing is done in one or multiple clock cycles (which happens if the shifting is needed; shifting p times requires p extra cycles), the state turns back to the update state.

The Event-Scheduler design allows the number of layers in Espresso to be added. In other words, any vision algorithms with windows sliding operators can be realized on Espresso.

IV. Evaluation

We implement Espresso architecture on the FPGA platform and test the system performance of Espresso in real-world scenarios IV-B. Based on the results of real-world performance, we design and conduct three parameter sensitivity experiments IV-C to further analyze the behavior of Espresso.

A. Experiment Setup

To test the performance of Espresso architecture, we established Espresso-Harris, a Harris Corner Detection system based on Espresso architecture, which has a three-layer structure. The first layer is a 3×3 Sobel; the second layer is a 5×5 Gauss + Harris Response, the third layer is a 5×5 NMS (Non-Maximum Suppression). We synthesize and implement the Espresso-Harris on the Zynq UltraScale+ MPSoC ZCU104 device, with a global clock frequency of 100MHz. The baseline device is a Jetson Orin NX with an 8-core Arm Cortex-A78AE 64-bit CPU (2MB L2 + 4MB L3 and 16GB 128-bit LPDDR5 RAM) and a 1024-core NVIDIA Ampere architecture GPU with 32 Tensor Cores. Besides, we also build a line-buffer-based harris-corner-detection accelerator on the same FPGA device and compare its throughput and latency with Espresso-Harris. Real-world data have been collected for the throughput test, including both simple-shape scenarios and complex-environment scenarios. Besides, a series of parameter sensitivity experiments have been conducted on randomly generated data to further analyze the performance of Espresso-Harris under different conditions.

B. Systematic Performance of Espresso

Fig. 7 shows the latency performance of Espresso-Harris in simple-shape scenarios and complex-environment scenarios compared with CPU-Harris, GPU-Harris (OpenCV Cuda Lib)

and line-buffer-Harris. In simple-shape scenarios, Espresso-Harris's throughput is 2.4× higher than line-buffer-Harris, 3.8× higher than GPU-Harris and 116× higher than CPU-Harris. As the scenarios become more complex, the throughput of Espresso-Harris drops while others' remain the same level. In complex-environment scenarios, Espresso-Harris's throughput is 1.2× higher than GPU-Harris, but 1.3× lower than line-buffer-Harris. Additionally, the latency/event of Espresso-Harris increases as the scenarios become more complex. The results show that Espresso-Harris can achieve a higher throughput than line-buffer-Harris and GPU-Harris in simple-shape scenarios, but the advantage fades away as the scenarios become more complex.

We hypothesize there are two main attributes of complex-environment scenarios that challenge the Espresso design. First, a complex environment with more curves and edges will definitely cause higher NZ within a certain time period, which causes more computing times and longer latency/frame. Second, a complex environment with heavier noise will cause more separated single-dot events. Such a separated pattern of events increases the computing times at an incredible rate, strongly magnifying the latency/event and also causing longer latency/frame as a result. To confirm our hypothesis, a series of parameter sensitivity experiments have been conducted and the results are shown in Fig. 8 and 9.

C. Parameter Sensitivity Analysis

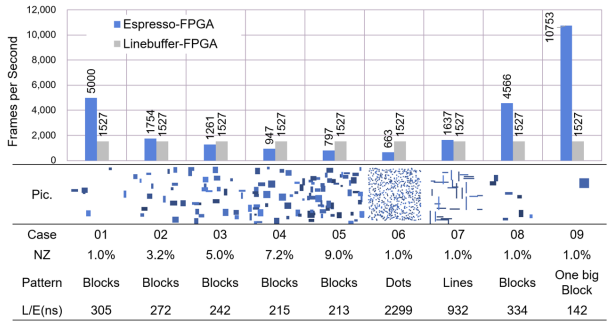

Fig. 8. Evaluation on input with different sparsity and different patterns.

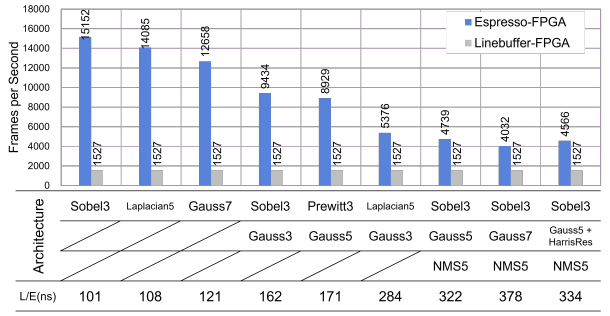

Fig. 9. Evaluation on different architectures. (Pattern: Blocks, NZ: 1%)

To further analyze the performance and behavior of Espresso architecture, three sensitivity experiments have been conducted. The input data are randomly generated by scripts, with different NZ levels and different aggregation levels. As case 01 to case 05 in Fig. 8 show, with the same level of aggregation (pattern=block), higher NZ actually results in lower throughput (from 9% NZ to 1% NZ, the throughput increases by 6.3×). However, inputs with lower NZ do not see a drop in latency/event; on the contrary, latency/event tends to increase as NZ drops. That, actually, is because at high NZ such as 9%, blocks overlap to form bigger blocks with higher aggregation levels, which pull down the latency/event dramatically. The conclusion that latency/event descends as aggregation level rises is also illustrated by case 06 to case 09 in Fig. 8, which demonstrate that the time Espresso-Harris takes to process inputs with the same NZ(1%) but different aggregation level can vary significantly. As the pattern changes from separated dots to one single block, the latency/event drops by more than 16.1×, and the throughput increases also by 16.2×. This result accounts for why the latency advantage of Espresso-Harris may weaken in complex real-world scenes with strong noise, which typically appears as separated dots.

In particular, we test the performance of different architectures of Espresso with different vision processing kernels, and the results are shown in Fig. 9. With the same input of 1%NZ and block pattern, as the layers of the architecture increase, the throughput drops accordingly with an almost constant proportion. That is because the NZ of input rises as the data are processed through layers. Since every layer has to wait for the completion of its following layer's calculation before sending out a new event, the bottleneck of the latency lies in the last layer with the input of the highest NZ. Also, kernels with larger size raises NZ faster than smaller ones (3×3 kernel up to 9× while 5×5 kernel up to 25×), thus resulting in longer latency unless they are at the ending layer in the architecture. The results above inspire us to minimize the number of layers as much as possible and avoid using large kernels at non-ending layers when designing Espresso accelerators so that Espresso architecture can show the greatest power in terms of throughput.

V. Conclusion

In this work, we design Espresso, a novel architecture for event-based vision processing. Espresso enables efficient stream processing of spatiotemporal ordering events. We design N-Pending-FIFOs and Shift-Hash Table to address the challenge of order mismatching and the latency overhead of discontinuous memory reading, respectively. Evaluations show that in real-world scenarios Espresso can achieve throughput up to 5000 fps, which is 5.1× higher than the embedded GPU.

Acknowledgment

This work was supported by the STI 2030—Major Projects (Grant No. 2021ZD0200300) and was financially supported by State Grid Corporation Headquarters Science and Technology Research Program (5700-202340289A-1-1-ZN). Taoyi Wang from Tsinghua University provided valuable suggestions on this work.

REFERENCES

[1] T. H. A. R. Vidal, H. Rebecq and D. Scaramuzza, "Ultimate slam? combining events, images, and imu for robust visual slam in hdr and high-speed scenarios," *IEEE Robotics and Automation Letters*, 2018.

[2] N. A. A. Zhu and K. Daniilidis, "Event-based visual inertial odometry," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[3] F. Akopyan, J. Sawada, A. Cassidy, R. Alvarez-Icaza, J. Arthur, P. Merolla, N. Imam, Y. Nakamura, P. Datta, G.-J. Nam, B. Taba, M. Beakes, B. Brezzo, J. B. Kuang, R. Manohar, W. P. Risk, B. Jackson, and D. S. Modha, "Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 10, p. 1537–1557, Oct. 2015.

[4] I. Alzugaray and M. Chli, "Asynchronous corner detection and tracking for event cameras in real time," *IEEE Robotics and Automation Letters*, 2018.

[5] K. Z. J. T. Y. D. C. Wang, Y. Liu and P. Ren, "ac2slam: Fpga accelerated high-accuracy slam with heapsort and parallel keypoint extractor," *International Conference on Field-Programmable Technology (ICFPT)*, 2021.

[6] V. Dadu, J. Weng, S. Liu, and T. Nowatzki, "Towards general purpose acceleration by exploiting common data-dependence forms," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. ACM, Oct. 2019, p. 924–939. [Online]. Available: http://dx.doi.org/10.1145/3352460.3358276

[7] C. B. E. Mueggler and D. Scaramuzza, "Fast event-based corner detection," *Proceedings of the British Machine Vision Conference(BMVC)*, 2017.

[8] A. A. et al., "A low power, fully event-based gesture recognition system," *IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

[9] E. P. et al., "Spinnaker: A 1-w 18-core system-on-chip for massively-parallel neural network simulation," *IEEE Journal of Solid-State Circuits*, 2013.

[10] K. K. et al., "1.22m 35.6mpixel rgb hybrid event-based vision sensor with 4.88m-pitch event pixels and up to 10k event frame rate by adaptive control on event sparsity," *IEEE International Solid-State Circuits Conference (ISSCC)*, 2023.

[11] M. D. et al., "Loihi: A neuromorphic manycore processor with on-chip learning," *IEEE Micro*, 2018.

[12] M. G. et al., "A 3-wafer-stacked hybrid 15mpixel cis + 1 mpixel evs with 4.6gevent/s readout, in-pixel tdc and on-chip isp and esp function," *IEEE International Solid-State Circuits Conference (ISSCC)*, 2023.

[13] G. Gallego, T. Delbruck, G. Orchard, C. Bartolozzi, B. Taba, A. Censi, S. Leutenegger, A. J. Davison, J. Conradt, K. Daniilidis, and D. Scaramuzza, "Event-based vision: A survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, no. 1, p. 154–180, Jan. 2022.

[14] Y. Gao, B. Zhang, Y. Ding, and H. K.-H. So, "A composable dynamic sparse dataflow architecture for efficient event-based vision processing on fpga," in *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '24. ACM, Apr. 2024, p. 246–257. [Online]. Available: http://dx.doi.org/10.1145/3626202.3637558

[15] G. G. H. Rebecq, T. Horstschaefer and D. Scaramuzza, "Evo: A geometric approach to event-based 6-dof parallel tracking and mapping in real time," *IEEE Robotics and Automation Letters*, 2017.

[16] O. Hsu, M. Strange, R. Sharma, J. Won, K. Olukotun, J. S. Emer, M. A. Horowitz, and F. Kjølstad, "The sparse abstract machine," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 710–726. [Online]. Available: https://doi.org/10.1145/3582016.3582051

[17] S. Klenk, M. Motzet, L. Koestler, and D. Cremers, "Deep event visual odometry," in *2024 International Conference on 3D Vision (3DV)*. IEEE, Mar. 2024, p. 739–749.

[18] P. Lichtsteiner, C. Posch, and T. Delbruck, "A 128×128 120 db 15 μs latency asynchronous temporal contrast vision sensor," *IEEE Journal of Solid-State Circuits*, vol. 43, no. 2, p. 566–576, 2008. [Online]. Available: http://dx.doi.org/10.1109/JSSC.2007.914337

[19] M. K. Vairalkar and S. Nimbhorkar, "Edge detection of images using sobel operator," *International Journal of Emerging Technology and Advanced Engineering*, vol. 2, no. 1, pp. 291–293, 2012.

[20] V. Vemulapati and D. Chen, "Fslam: an efficient and accurate slam accelerator on soc fpgas," *International Conference on Field-Programmable Technology (ICFPT)*, 2022.

[21] B. Y. W. Fang, Y. Zhang and S. Liu, "Fpga-based orb feature extraction for real-time visual slam," *International Conference on Field Programmable Technology (ICFPT)*, 2017.

[22] Z. Yang, T. Wang, Y. Lin, Y. Chen, H. Zeng, J. Pei, J. Wang, X. Liu, Y. Zhou, J. Zhang, X. Wang, X. Lv, R. Zhao, and L. Shi, "A vision chip with complementary pathways for open-world sensing," *Nature*, vol. 629, no. 8014, p. 1027–1033, May 2024.