Fast Pipeline 128x128 Pixel Spiking Convolution Core for Event-Driven Vision Processing in FPGAs

A. Yousefzadeh, T. Serrano Gotarredona, and B. Linares-Barranco
Instituto de Microelectrónica de Sevilla, IMSE-CNM (CSIC and Universidad de Sevilla), Sevilla, SPAIN
Email: bernabe@imse-cnm.csic.es

***Abstract*— This paper describes a digital implementation of a parallel and pipelined spiking convolutional neural network (S-ConvNet) core for processing spikes in an event-driven system. Event-driven vision systems use typically as sensor some bio-inspired spiking device, such as the popular Dynamic Vision Sensor (DVS). DVS cameras generate spikes related to changes in light intensity. In this paper we present a 2D convolution event-driven processing core with 128×128 pixels. S-ConvNet is an Event-Driven processing method to extract event features from an input event flow. The nature of spiking systems is highly parallel, in general. Therefore, S-ConvNet processors can benefit from the parallelism offered by Field Programmable Gate Arrays (FPGAs) to accelerate the operation. Using 3 stages of pipeline and a parallel structure, results in updating the state of a 128 neuron row in just 12ns. This improves with respect to previously reported approaches.**

Keywords—Spiking Convolutional Neural Networks, DVS, Artificial Retina, FPGA, Parallel Processing

I. INTRODUCTION

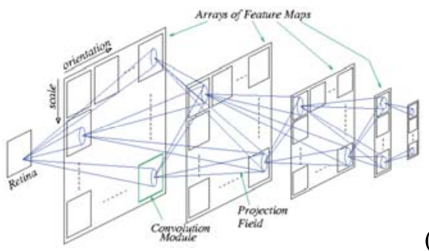
In conventional artificial vision systems, cameras capture frames at a given frame rate. In these systems, independent of the changes in the frames, the processing will be done for every single frame. Therefore, for detecting fast moving objects, a high frame rate is required, resulting in a need for very powerful processors. However, in biological brains, the procedure is quite different. Our eyes contain many neurons that are sensitive to light and each neuron will generate a spike as soon as it detects a change in light. The processing neurons in the cortex operate with time responses of several milliseconds, but there are many millions of them operating in parallel and with a very efficient spike based information encoding scheme.

Address Event Representation (AER) [2] is a way to represent spikes and synapses with artificial electronic systems. In this way, every pixel of an artificial retina chip can generate events that contain the address of the originating pixel. From the point of view of artificial AER neural networks, these events play the role of spikes in biology. For example in a 128×128 pixel retina, 14 bits are required to encode the (x, y) position. The information is mainly hidden in the timing of the generated spikes. The artificial retina that has been used in this work [1], also appends one extra bit to the AER events which shows the sign (increasing or decreasing) of the relative light intensity change of the corresponding pixel.

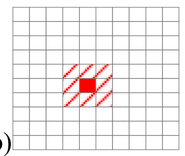
After generating and sending spikes in a retina chip, the next step is to process the spikes and extract the desired features. Work on AER systems started around twenty years ago [2], but AER processing with generic feature extraction hardware is more recent [3], [8], [11], [13], [15], [18].

The main difference between frame-based and frame-free event-driven systems is hidden in the asynchronous nature of spikes. In frame based systems, waiting for new frames to detect an object is unavoidable. For example, a normal commercial camera generates around 25 frames per second, so that the time difference between frames is 40ms. For detecting an object, the system at least needs to wait 40ms to receive a new frame. On the other hand, in frame-free event-driven vision processing, the object could be recognized as soon as enough relevant spikes are provided by the retina. Retina pixels physically need a few microseconds to detect changes in light and generate spikes. So the time for detecting an object can be as small as one millisecond [9]. Another interesting advantage of using frame-free event-driven systems is power consumption. The number of spikes generated by a spiking DVS retina is directly related to the environment visual activity. If there is no changing activity, the DVS will not generate any spike and the processing system also consumes less power during those moments. For a more detailed comparison between conventional frame-based and spiking event-driven frame-free vision processing refer to [4]. Some simple event-driven processing can be implemented in software [5]-[6]. However, software implementations suffer from high latencies due to processor resource sharing between all neurons. For event-driven processing some mixed analog-digital chips [14] have been designed that achieved good results but they suffered from high mismatch in the analog circuitry, which required expensive in-pixel calibration. Fully digital ASICs were also designed [3],[8], but only one ConvNet Feature Map could be implemented in one low cost chip.

In this paper we introduce a fully digital implementation of a spiking convolutional event-driven core that can be implemented in commercial FPGAs. We present a pipelined scheme capable of updating 128 synaptic connections in 12ns. This improves with respect to previously reported FPGA convolutional event-driven cores where 121 event synaptic updates where performed in 3μs [11], or 84 event synaptic updates in 10ns [15]. In the next Section, we will describe briefly the spiking convolutional neural network concept. Then the proposed core will be presented and finally we will provide the implementation results.



(a)



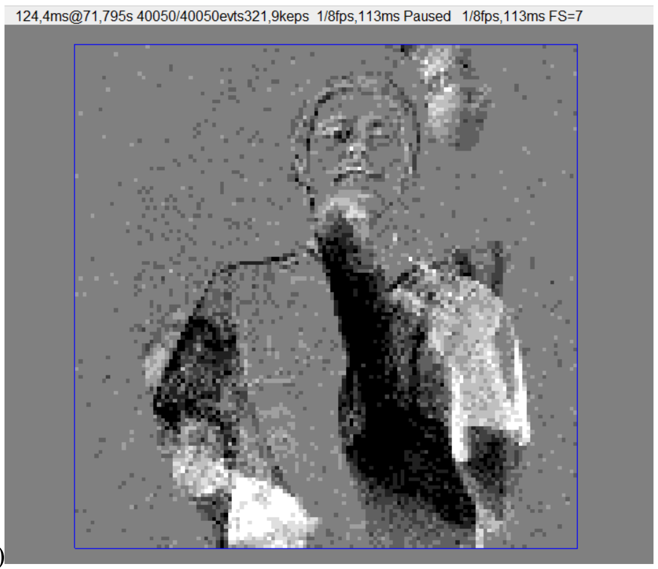
(b)

Fig. 1. (a) Illustration of event-driven convolutional computation concept. (b) Generic ConvNet with several layers, each with several Feature Maps.

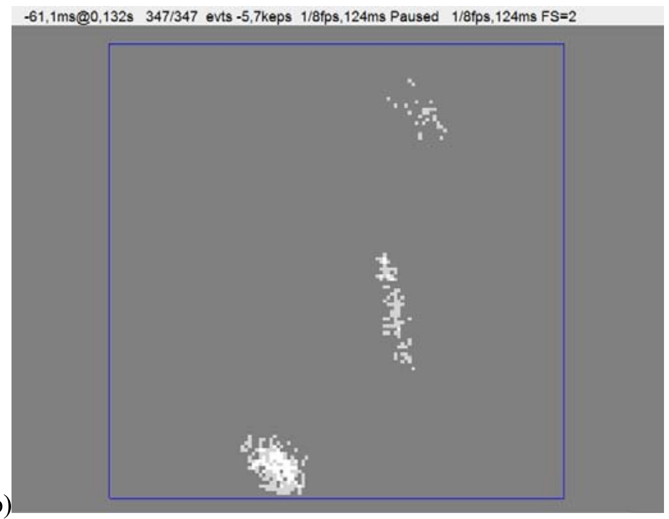
II. Convolutional Neural Network concept

In conventional frame-based image processing, one typical strategy is using kernel convolutions to extract image features, combine them progressively and perform object recognition. To exploit this concept in neural networks, the Convolutional Neural Network (ConvNet) paradigm was developed to define how to learn kernel weights (synaptic weights) [16]. If we use a kernel size of $M \times N$, each neuron in a layer connects to the next layer through $M \times N$ synapses. The distribution of synaptic weights is the same for all neurons in a previous layers. This is also known as "weight sharing". Fig. 1(a) shows a generic ConvNet structure with multiple layers, each layer with several "Feature Maps" (FM). Each Feature computes several convolutions, depending on the origins of the events. Each FM is a 2D grid arrangement of neurons receiving spike events from neurons of the previous layer FMs. A neuron in a FM sends its spikes to a "projection field" of neurons in a receiving FM in the next layer. This is done by assigning kernel weights to synapses is illustrated in Fig. 1(b). Assume that the 10×10 grid is a FM of a that received a spike from a source neuron in the coordinate in the red position. If the kernel size is 3×3, the incoming spike will convey to a 3×3 square through synapses. This square is the "projection field" of the source neuron. The weights of the synapses are extracted from the kernel weights. For more details refer to [3],[8],[14].

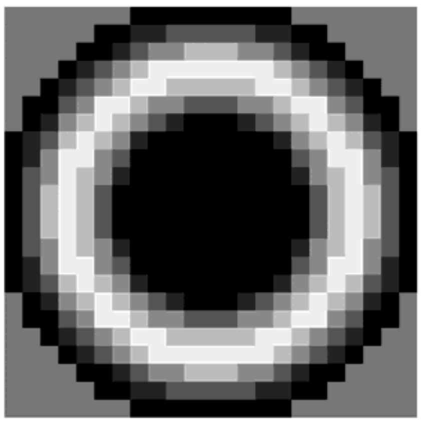
As an illustration of this event-driven convolutional processing, let us consider the case illustrated in Fig. 2. Fig. 2(a) shows a 124ms histogram obtained by collecting the events from a DVS retina while observing a person juggling with three balls. The DVS retina has 128x128 pixel resolution. Each ball has a diameter of about 16 pixels in Fig. 2(a). The retina can generate events at a rate of up to 10Meps (million events per seconds). In this particular recording, the event rate generated by the retina is 322keps (thousand events per second). If every event generated by a retina pixel is sent to our 128x128 pixel convolution processing core programmed with the 23x23 pixel convolution kernel in Fig. 2(c), as illustrated in Fig. 1, the convolution core output is as shown in Fig. 2(b). This is because the convolution kernel in Fig. 2(c) is tuned to detect circles of 16 pixel diameter: pixels on the diameter contribute positively to the center of the circle, while pixels in the center region or slightly beyond the 16 pixel diameter contribute negatively to the center. This way, the output of the event-driven convolution processing, as shown in Fig. 2(b), highlights the centers of the 16-pixel diameter balls.


124,4ms@71,795s 40050/40050evts321,9keps 1/8fps,113ms Paused 1/8fps,113ms FS=7

(a)


-61,1ms@0,132s 347/347 evts -5,7keps 1/8fps,124ms Paused 1/8fps,124ms FS=2

(b)



(c)

Fig. 2. Event-Driven 2D Convolution Processing. (a) 124ms histogram from a DVS output. (b) Ouput of reported 2D convolution processing core programmed with the kernel shown in (c).

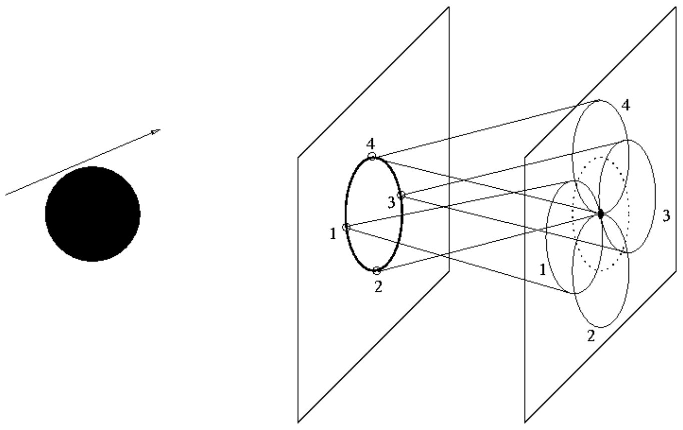


Fig. 3. Event-Driven 2D Convolution Processing when observing moving balls. Left: ball moving in reality. Center: output provided by a DVS retina, where the pixels on the periphery of the ball generate events, as those are the pixels detecting changes in light. Pixels 1, 2, 3, and 4 on this periphery project the convolution kernel on a 128x128 pixel array inside the convolution processing core. Each retina pixel contributes positively on the projecting 16 pixel diameter circumference. The positive contributions of all pixels at the retina plane add up at the center of the circumference in the convolution core plane, signaling the presence of a 16 pixel diameter circle.

Fig. 3 helps to better understand the intuition behind this event-driven convolutional processing. The left side in Fig. 3 shows a solid ball moving in the real world. The central plane in Fig. 3 represents the pixels of a DVS sensor, which detect illumination changes. Thus, only the pixels on the peripheral circumference will become active and send one or more spikes.

When a retina pixel sends a spike to the convolution core on the right of Fig. 3, it will send spikes to the projection field defined by the convolution kernel in Fig. 2(c). This projection field sends a positive contribution to the pixels at circumference of diameter equal to 16 pixels, while the contribution is negative inside and slightly outside that circumference. When adding up the contributions of all projection fields of the retina active pixels, there will be a net positive contribution in the center of the original circumference on the convolution core plane, signaling the center of the circumference of the expected size. This is what is shown in Fig. 2(b).

III. Proposed Event-Driven Convolutional Core

The implementation of the convolutional core presented in this work contains a fast parallel and pipelined hardware structure for event processing. The retina provides completely asynchronous events. In the FPGA, a complete asynchronous design is not efficient. In this paper, a Globally Asynchronous Locally Synchronous (GALS) structure is designed to share some resources and take advantage from pipeline and parallel design principles. With this approach, we obtained a high rate of event processing, while using an optimum number of cells in the FPGA. In this paper a simple leaky integrated and fire neuron model with instant synapses [9] has been used.

Fig. 4 illustrates the connections between the convolutional core and the AER modules for receiving and transmitting events through asynchronous parallel AER interfacing. The core contains 3 main modules “Convolution Core”, “AER RX”, and “AER TX”. The AER receiver and AER transmitter are designed to communicate with asynchronous AER protocol PCBs [10] and change the event flow into a fast synchronous protocol to communicate with the convolution core. Both asynchronous and synchronous protocols include flow control. The synchronous protocol can send one event per clock cycle, while the asynchronous protocol within the FPGA is slower.

The convolution core itself, has 3 major blocks that work in parallel. The first block manages input events and updates the neurons states by doing event-driven convolutions. The second block is in charge of applying a forgetting rate (leakage) to the neurons, and the last block is the block for managing generated events and make them ready to be sent out of the core.

A. Input event processing block

Fig. 5 illustrates schematically the data flow diagram of the input event processing part. The convolutional core uses a pipelined parallel scheme to convolve one row of a kernel to one row of pixels (128 pixels in our case) in one clock cycle.

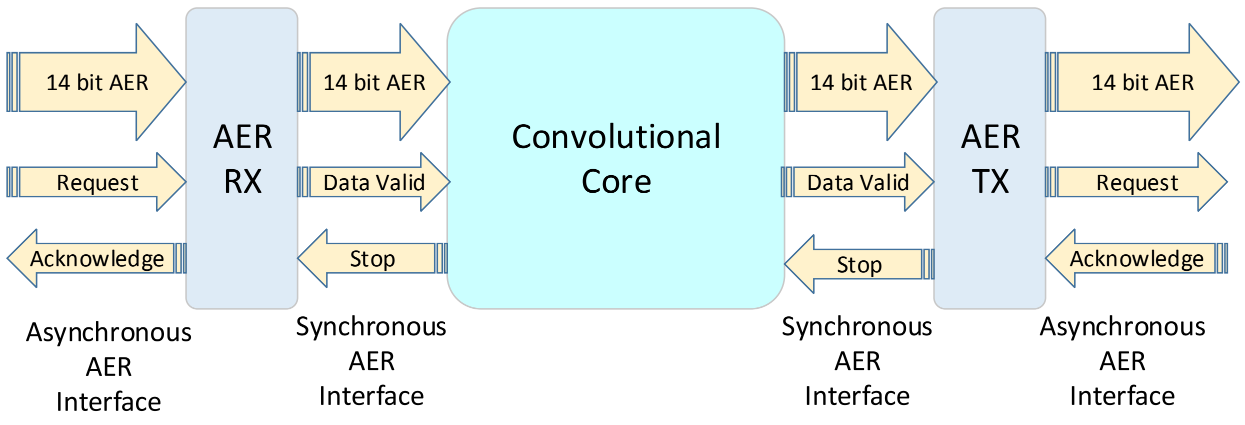


Fig. 4. Convolutional core interfaces.

Pixel arrays hold their neuron state in dual-port block RAMs of the FPGA. Fig. 5 illustrates the use of one of the ports. The second port is used for the forgetting logic part and will be explained in subsection *B*, next. The membrane voltage of the neurons is saved in pixel arrays with 10 bits per pixel. The 2's complement scheme has been used to save signed numbers. In this work, we use a 128×128 pixel arrangement for the convolutional core. Each pixel state is 10-bit and the core reads one line of pixels at the same time. Therefore, we need to use a 1280-bit bus. We used Xilinx Block RAMs that are fully synchronous. This means they need one clock cycle for reading and providing data. That is the reason for locating the first stage of the pipeline inside the pixel arrays.

The process of updating one row of pixels consists of 3 steps: reading a row of pixels from the block RAM, adding the proper row of the kernel and comparing the results against a threshold. Finally, the corresponding row of pixels, should be updated in the block RAM, which also needs an additional clock cycle for writing. For this purpose, it needs at least 2 clock cycles for reading and writing. To reach the speed of 1 row per clock cycle, 2 independent pixel arrays are used in parallel, each one containing half of the pixels. The even rows of pixels are in pixel array 1 and the odd rows are in pixel array 2. The 2 modules of block RAMs let the core read a row when it is writing in the previous row of the other RAM. Fig. 4 illustrates the sequence of reading from pixel arrays in a normal operation.

Fig. 6 shows the time sequence for reading and writing from the pixel arrays. The first, second and third clock cycles are spent for reading from pixel array 1 to fill up the 3 pipeline stages. In the fourth clock cycle, the first row of pixel array 2 will be read and the first row of pixel array 1 will be written. In the following clock cycles, one read and one write operation will be done in one clock cycle until doing the whole convolution.

The address calculator in Fig. 5 is a logic block that defines the addresses of the rows which should be read and written in the pixel arrays and kernel ROM. It calculates the proper addresses based on the input event address and its current state.

Another part in Fig. 5 is the "kernel ROM". For the kernel, the number of bits allocated to each weight is 6 bits including sign, based on the 2's complement scheme. Normally, the kernel size is smaller than the pixel array size. Therefore, a logic is designed to find out the columns of pixels that should be added with the kernel. For this purpose, the "Kernel size matching" logic, puts the kernel row in the proper columns of an empty 128 cell register, while the other cells stay at zero. Each cell contains 6 bits. Fig. 7 illustrates the output of this module. The adder simply contains 128 10-bit adder blocks that add the 128 pixels to the kernel weights which are put in proper columns.

After adding the kernel to the pixels values, the threshold logic block compares the values of each pixel to a positive and a negative threshold. This block has 2 outputs, the new pixels values and the event vector register.

New pixels values are the result of adding the kernel to the previous pixels values and the threshold logic puts the reset value for the pixels that exceed the threshold. In neural network terminology, it means that the neuron fires and generates a new spike and its state goes back to reset. The new pixel value is written in the same row of the pixel array to update the row.

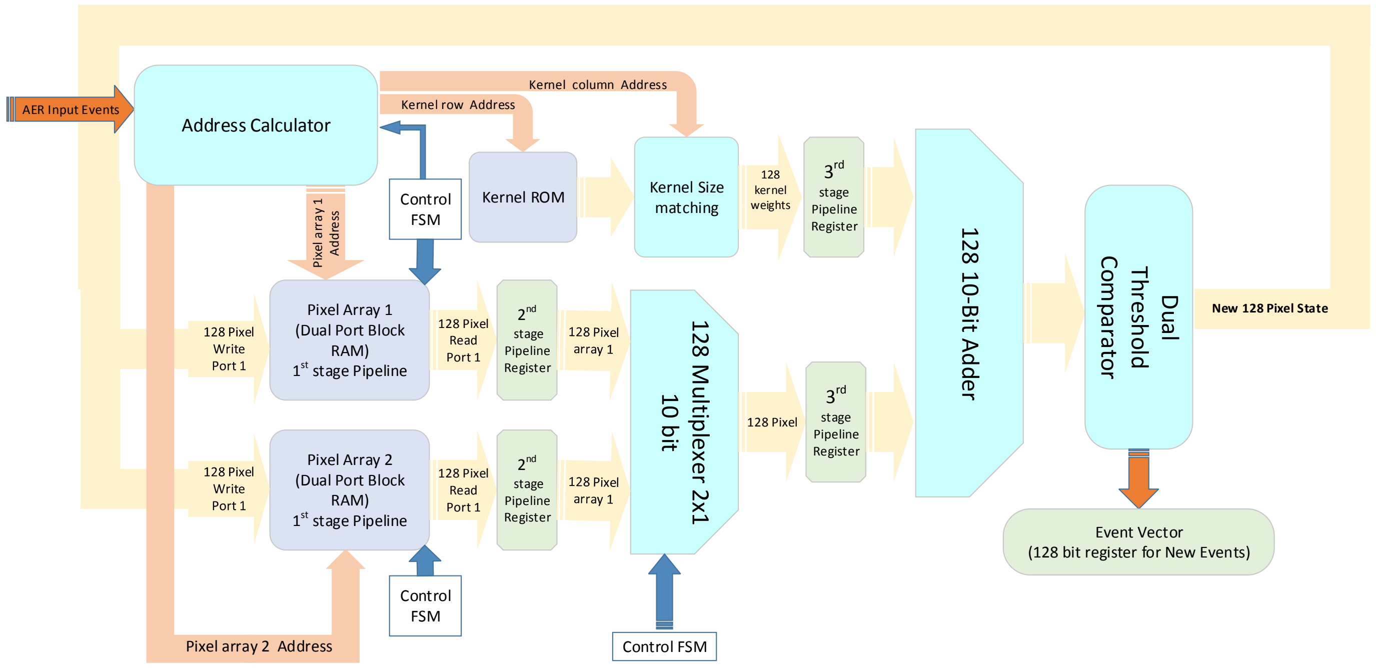
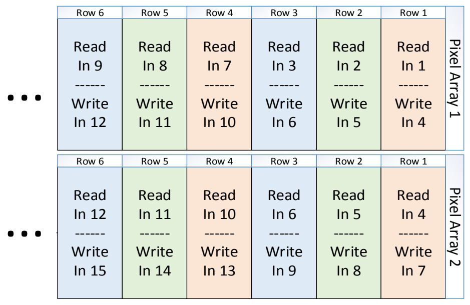


Fig. 5. Data flow diagram of the input event processing block

Another output of the threshold logic block is the event vector. In this version of the core, negative events are not saved. So, if a pixel value goes below its negative threshold, it is reset to zero but no new events are generated. However, if the pixel value goes above its positive threshold, it is reset to zero and in the 128 bit event vector register, a flag related to the place of this pixel will be set to ON. Another logic block that manages and sends generated events, uses this vector as input.



Row 6
Row 5
Row 4
Row 3
Row 2
Row 1
Read In 9 ------ Write In 12
Read In 8 ------ Write In 11
Read In 7 ------ Write In 10
Read In 3 ------ Write In 6
Read In 2 ------ Write In 5
Read In 1 ------ Write In 4
Pixel Array 1
Read In 12 ------ Write In 15
Read In 11 ------ Write In 14
Read In 10 ------ Write In 13
Read In 6 ------ Write In 9
Read In 5 ------ Write In 8
Read In 4 ------ Write In 7
Pixel Array 2

Fig. 6. Time sequence for reading and writing from the pixel arrays. It starts by reading the first row of pixel array 1 in the first clock cycle and shows the operations until the 15th clock cycl

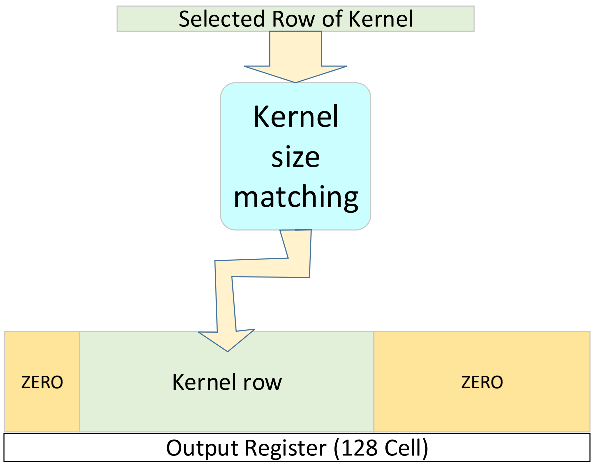


Fig. 7. Input and output of kernel size matching logic block

The number of pixel rows that should be added with kernel rows depends on the size of the kernel and the address of the incoming event. There is a control finite state machine in this part of the core that controls the flow of data and asserts the control signals (such as read and write in the pixel arrays) and selects the proper input for the multiplexer. It also controls the "address calculator" logic block and the stop signal for the synchronous interface to manage flow control. This logic block should be aware of parameters like kernel size, negative and positive thresholds and reset value of the pixels.

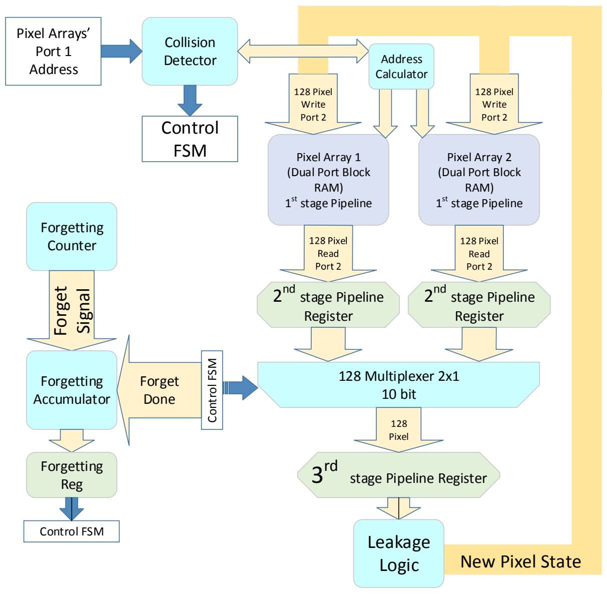
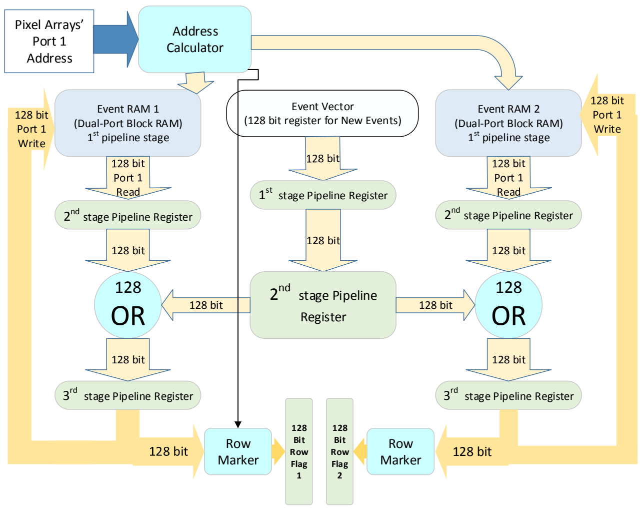


Fig. 8. Data Flow diagram for the forgetting logic block



Pixel Arrays' Port 1 Address
Address Calculator
128 bit Port 1 Write
Event RAM 1 (Dual-Port Block RAM) 1st pipeline stage
Event Vector (128 bit register for New Events)
Event RAM 2 (Dual-Port Block RAM) 1st pipeline stage
128 bit Port 1 Write
128 bit Port 1 Read
128 bit
128 bit Port 1 Read
2nd stage Pipeline Register
1st stage Pipeline Register
2nd stage Pipeline Register
128 bit
128 bit
128 bit
128 OR
128 bit
2nd stage Pipeline Register
128 bit
128 OR
128 bit
128 bit
3rd stage Pipeline Register
3rd stage Pipeline Register
128 bit
Row Marker
128 Bit Row Flag 1
128 Bit Row Flag 2
Row Marker
128 bit

Fig. 9. Process of writing events in the event RAM

B. *Forgetting logic block*

Another important part of the convolution core is the logic block in charge of applying leakage to the neurons. Fig. 8 illustrates schematically the data flow diagram for this block.

In this block, the core uses the second port of the pixel arrays RAM. The pipeline stages, multiplexer and address calculator are almost the same as in the previous part. When the forgetting logic and convolution logic blocks want to write in the same row of the pixel arrays, there is a conflict. For addressing this situation, a collision detector logic block is designed to detect this situation and notify the control logic. After detecting a collision, the pipeline stage should become empty and the forgetting logic has to start from the previous 3 rows. To minimize the number of collisions, a "forgetting process" starts from the end of the pixel array and proceeds towards the first row, while the convolution logic reads and writes in the reverse direction. Using this strategy minimizes collisions, as the maximum collision happening for 1 convolution process is just once, preventing bursts of collisions.

A complete cycle of forgetting needs 128 + 3 clock cycles if no collision happened. For each collision occurrence the forgetting logic waits for 3 clock cycles and 3 additional clock cycles are added to fill up the pipeline again.

There is a forgetting counter in Fig. 8 that generates forgetting signals based on the forgetting rate defined by the user. The "forgetting accumulator" block takes care to not lose any forgetting signal within the huge traffic of input events. Whenever a forgetting signal comes, the accumulator adds '1' to the forgetting register and whenever a "forgetting done" signal activates (that means a complete cycle of forgetting has concluded), the logic will decrease by '1' the forgetting register. The forgetting register contains the number of forgetting cycles that should be performed.

The "leakage logic" block adds or subtracts '1' to the pixel value based on the sign bit. For positive values it will decrease the number and for negative values it will add '1' to the value to set them closer to the reset value. For the value equal to reset, the "leakage logic" will do nothing.

C. *Output event generator block*

Whenever an event is generated by the "threshold logic" block of the convolution block, another part of the core takes care of these new events. This part includes 2 parallel processes. The first one writes the new events into the event RAM, and the second one reads them and send them out of the core. Event RAM is a dual-port block RAM that contains 128x128 bits of data. It means that for every pixel, there is 1 bit of data in the event RAM that indicates the corresponding pixel has generated a new event or not.

Fig. 9 illustrates the process of writing events in the event RAM. This part also uses the same pipeline and parallel techniques and 2 dual port RAMs to speed up the process. Whenever a new event vector comes, the corresponding row of the event RAM will be read. The content of the event RAM row and the new event will enter into the 128 OR gates and the result is the updated row of event RAM. The "Address calculator" logic block for this process uses the address of the pixel arrays and manipulates it to fit the new pipeline stages. The control FSM that is not shown in Fig. 9, will control the write enable signals of the RAMs.

The "Row Marker" block is designed to help another process for output event management, which increases the speed of finding events in the RAMs. Finding an event in the whole memory by scanning each line of memory one by one is not efficient. "Row marker" calculates the OR between the 128 bits of the updated event vector and puts it in the proper column of the 128 bit row flag register. This way, each flag indicates that in the corresponding row of the event RAM, there is one or more new events waiting to be sent.

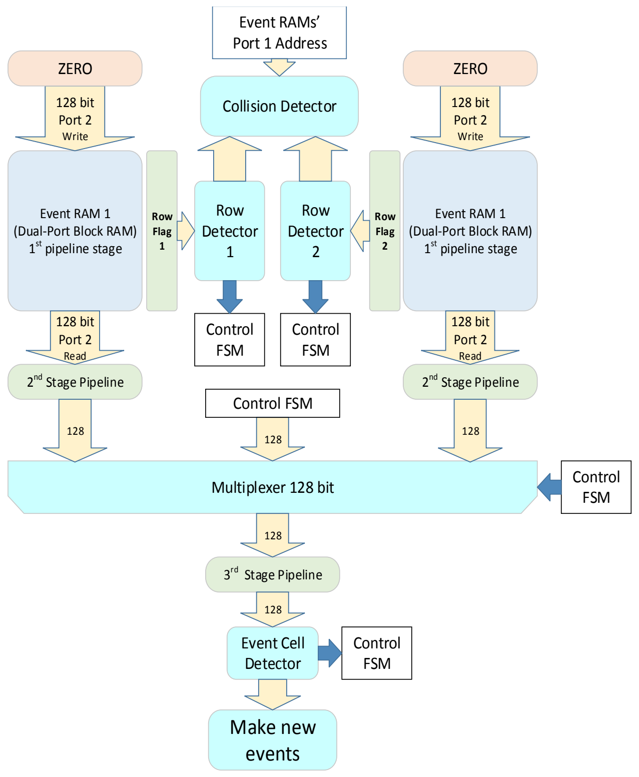


Fig. 10. Process of reading events from the event RAM and sending events out

Another process to manage the generated events is the process of reading from Event RAM and send the events out of the core. Fig. 10 illustrates this process.

Based on the information in the Row flag registers, the row address of event RAMs will be defined through a "Row detector" logic block. Another logic block that operates in the same way is the "Event cell detector", which works on the content of the event RAM to find the events. With these 2 logic blocks, the "new event maker" logic block can find out the X and Y of the new event to make an AER event package.

The "collision detector" is in charge of finding the write cycles with the same address in both ports of the event RAMs. In case of collision, the process of reading and sending events will always wait for the process of writing new events in the RAMs. The control logic block is responsible for asserting the write enable signals for the block RAM and taking care in collision situations. It also should handle stop signals from the output synchronous interface and propagate them back to the input synchronous interface.

Another important role of the control FSM is sending events one by one. Whenever an event is sent, its place in the 3rd stage pipeline register should become 0 to start sending another event. The control logic does this process by changing the 3rd stage pipeline register through the multiplexer. When all of the events in the 3rd stage pipeline register have been sent, the control register will assert the event RAM write enable signal to write zero in the selected row of the event RAM and the row flag register.

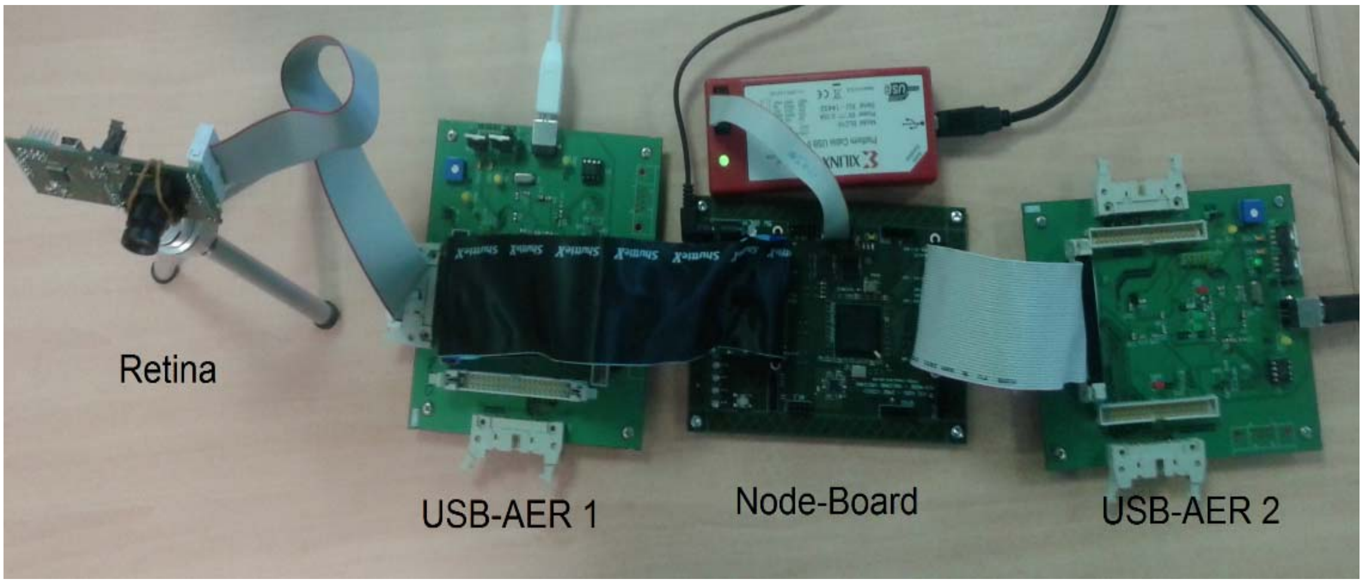


Fig. 11. Experimental Setup

IV. Implementation Results

Fig. 11 shows the setup used in this work which contains a retina camera [1], a node-board [17] (containing one Spartan6 and other necessary interfaces) and 2 USBAERmini2 boards [10] that send AER spikes (before and after processing) through USB to a computer. These boards are used to monitor DVS events, as well as convolution output events, as was shown in Fig. 2.

In this paper, we used Xilinx XST to synthesize and implement Verilog codes. With Spartan-6 we obtained a critical path of 12ns for the pipeline stages, which allowed us to use 80MHz of clock frequency.

Using block RAM in this project is unavoidable because of the huge amount of memory that is needed for saving pixel states. Although it is not a very expensive memory with respect to distributed RAM, it is slower and it cannot be used in a fully costum manner. This means it is normally offered in a special size of memory. For example, in Spartan-6 the minimum size of block RAM is 9kb [12] and the widest port for dual-port RAM contains 18 bits of data. Therefore the block RAM will be 512×18 bits. The core uses these block RAMs to make a pixel array of 64×1280 bits (each pixel array contains half of pixels), so that from each 512 rows of block RAM, just 64 rows have been used and there is a waste of memory happening here.

In Spartan-6 XC6SLX150T-3, the number of occupied slices is around 3.3k out of 23k, and the number of 8kb block RAMs used is 160 out of 536. Also, for comparison purposes, the core has been also synthesized for Virtex-6 and Virtex-7 technologies. TABLE I. shows the percentage of resources that are needed, the critical path delays and maximum frequencies in the different FPGAs.

If the kernel has L lines, the presented core needs L+3 clock cycles for calculating a convolution. As a comparison, Camuñas [3] produced a 0.35um CMOS chip for 32×32 pixels, which for a kernel size of 23×23 the processing needed 50 clock cycles or 417ns with 120MHz clock frequency. In the same situation, the presented core needs 26 clock cycles for this kernel which in Spartan-6 needs about 312ns, in Virtex-6 about 195ns and in Virtex-7 about 156ns.

Regarding other FPGA implementations of Event-Driven ConvNets, to our knowledge there are two other cases reported. Zamarreño et al. [11] used a convolution core adapted from [17], where neuron states are updated pixel by pixel, instead of row by row. This allowed for very compact convolution cores, so that many of them could be put on one signle FPGA: a total of 64 cores, each of 64x64 pixels could be put on a Virtex6, each core together with a programmable router for configuring arbitrary ConvNets. However, as synaptic update was pixel by pixel, it required about 3us to update one event of 11x11 convolution kernel. This is equivalent to requiring 3.17us to update a row of 128 pixels, as we are doing in this work.

Another recently reported example of event-driven ConvNets [15], reports a core update speed of 84 synaptic updates in 10ns, implemented on a Spartan6, thus achieving a performance which approaches the one reported in the present work.

V. Conclusions

In this paper, we present a 128×128 pixel convolutional core for event processing that can process each kernel row in one clock cycle using a parallel and pipelined structure. In spiking ConvNet designs, using this core can speed up event processing and it can be used to make a layer for neural networks. For convolving a kernel that contains L lines, the core needs L+3 clock cycles. We implemented the core in different FPGAs. For the FPGA in our Node-Board

(XC6SLX150-3), 12ns are needed to update the state of a 128 neuron row. The core also contains a leakage logic that works independently and does not interfere with the main process.

TABLE I. RESOURCES NEEDED IN DIFFERENT FPGAS AND MAXIMUM CLOCK FREQUENCY

FPGA Chip	**Resource Utilization**		
	Occupied Slices	***Occupied Block RAM***	***Critical Path***
Spartan-6 (XC6SLX45-3)	3,298 (48%)	80 x 16kb (68%)	12ns (83MHz)
Spartan-6 (XC6SLX75-3)	3,285 (28%)	80 x 16kb (46%)	12ns (83MHz)
Spartan-6 (XC6SLX150-3)	3,317 (14%)	80 x 16kb (29%)	12ns (83MHz)
Virtex-6 (XC6VLX75T-3)	4,549 (39%)	80 x 32kb (51%)	7.5ns (133MHz)
Virtex-7 (XC7VX330T-3)	4,067 (7%)	80 x 32kb (10%)	6ns (166MHz)

ACKNOWLEDGMENT

This work has been supported by EU grants 604102 HBP (the Human Brain Project) and 644096 ECOMODE, Spanish grant (with support from the European Regional Development Fund), BIOSENSE (TEC2012-37868-C04-02/01), Andalusian grant NANO-NEURO (TIC-6091) and EU CHIST-ERA grant PNEUMA (PRI-PIMCHI-2011-0768). A.Y. was supported by an FPI grant from the Spanish Ministry of Economy and Competitivity.

REFERENCES

[1] T. Serrano-Gotarredona and B. Linares-Barranco, "A 128×128 1.5% Contrast Sensitivity 0.9% FPN 3 µs Latency 4 mW Asynchronous Frame-Free Dynamic Vision Sensor Using Transimpedance Preamplifiers", *IEEE J. Solid-State Circuits*, vol. 48, no. 3, Mar. 2013.

[2] M. A. Mahowald, "VLSI analogs of neuronal visual processing: A synthesis of form and function," *Ph.D. dissertation*, Comput. Neural Syst., California Inst. Technol., Pasadena, CA, 1992.

[3] L. Camuñas-Mesa, A. Acosta-Jiménez, C. Zamarreño-Ramos, T. Serrano-Gotarredona, and B. Linares-Barranco, "A 32x32 Pixel Convolution Processor Chip for Address Event Vision Sensors with 155ns Event Latency and 20Meps Throughput," *IEEE Trans. Circuits and Systems*, vol. 58, no. 4, pp. 777-790, Apr. 2011.

[4] C. Farabet , R. Paz, J. Pérez-Carrasco, C. Zamarreño-Ramos, A. Linares-Barranco,Y. LeCun,"Comparison between frame-constrained fix-pixel-value and frame-free spiking-dynamic-pixel ConvNets for visual processing", *Frontiers in neuroscience*, April, 2012.

[5] [Online]. Available: http://jaer.wiki.sourceforge.net

[6] T. Delbrück, "Frame-free dynamic digital vision," in *Proc. Int. Symp.Secure-Life Electron.*, Adv. Electron. Quality Life Soc.,Mar. 6–7, 2008, pp. 21–26.

[7] T. Serrano-Gotarredona, A. G. Andreou, and B. Linares-Barranco, "AER image filtering architecture for vision processing systems," *IEEE Trans. Circuits Syst. I, Fundam. Theory Appl.*, vol. 46, no. 9, pp. 1064–1071, Sep. 1999.

[8] L. Camuñas-Mesa, C. Zamarreño-Ramos, A. Linares-Barranco, A. Acosta-Jiménez, T. Serrano-Gotarredona,"An Event-Driven Multi-Kernel Convolution Processor Module for Event-Driven Vision Sensors," *IEEE J. Solid-State Circuits*, vol. 47, no. 2, pp. 504-517, 2012.

[9] J. Pérez-Carrasco, B. Zhao, C. Serrano, B. Acha, T. Serrano-Gotarredona, "Mapping from Frame-Driven to Frame-Free Event-Driven Vision Systems by Low-Rate Rate Coding and Coincidence Processing Application to Feedforward ConvNets", *IEEE Trans. Pattern analysis and Machine inteligence*, vol. 35, no. 11, November 2013.

[10] R. Berner, T. Delbruck, A. Civit-Balcells, and A. Linares-Barranco, "A 5 Meps $100 USB2.0 address-event monitor-sequencer interface," in *Proc. IEEE Int. Symp. Circuits Syst.*, 2007, pp. 2451–2454.

[11] C. Zamarreo-Ramos, A. Linares-Barranco, T. Serrano-Gotarredona, and B. Linares-Barranco, "Multicasting Mesh AER: A Scalable Assembly Approach for Reconfigurable Neuromorphic Structured AER Systems. Application to ConvNets", *IEEE Trans. Biomedical Circuits and Systems*, vol. 7, no. 1, Feb 2013.

[12] Xilinx's documents, "Spartan-6 FPGA Block RAM Resources", UG383

[13] R. Serrano-Gotarredona, M. Oster, P. Lichtsteiner, A. Linares-Barranco R. Paz-Vicente, "CAVIAR: A 45k-Neuron, 5MSynapse, 12G-Connects/Sec AER Hardware Sensory-Processing-Learning-Actuating System for High Speed Visual Object Recognition and Tracking," *IEEE Trans. Neural Networks*, vol. 20, no. 9, pp. 1417-1438, Sept. 2009.

[14] R. Serrano-Gotarredona, T. Serrano-Gotarredona, A. Acosta-Jiménez, and B. Linares-Barranco, "A Neuromorphic Cortical-Layer Microchip for Spike-Based Event Processing Vision Systems", *IEEE Trans. Circuits Syst*. vol. 53, no. 12, Dec. 2006.

[15] G. Orchard, C. Meyer, R. Etienne-Cummings, C. Posch, N. Thakor, R. Benosman, "H-First: A Temporal Approach to Object Recognition," *IEEE Trans. Pattern Analysis an dMachine Intelligence*, DOI: 10.1109/TPAMI.2015.2392947, 2015.

[16] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-Based Learning Applied to Document Recognition," *Proc. IEEE,* vol. 86, no. 11, pp. 2278-2324, Nov. 1998.

[17] T. Iakymchuk, A. Rosado, T. Serrano-Gotarredona, B. Linares-Barranco, A. Jimenez-Fernandez, A. Linares-Barranco, and G. Jimenez-Moreno, "An AER handshake-less modular infrastructure PCB with x82.5Gbps LVDS serial links." *Proc. of the IEEE Int. Symp. Circ. and Syst.* (ISCAS), pp. 1556–1559, June 2014.

[18] A. Linares-Barranco, R. Paz-Vicente, F. Gómez-Rodriguez, A. Jiménez, M. R. G. Jiménez, and A. Civit, "On the AER convolution Processors for FPGA," in *Proc. IEEE Int. Symp. Circuits and Systems*, pp. 4237–4240, May 2010.