Visual Spike Processing based on Cellular Automaton

M. Rivas-Pérez, A. Linares-Barranco and G. Jiménez, A. Civit
Department of Computer Architecture and Technology. University of Seville
Spain

1. Introduction

Cellular organization in biology has been an inspiration in several research fields, such as the description and definition of Cellular Automaton, Neural Networks, Spiking Systems, etc. Cellular Automaton (CA) is a bio-inspired processing model for problem solving, initially proposed by Von Neumann. This approach modularizes the processing by dividing the solution into synchronous cells that change their states at the same time in order to get the solution. The communication between them and the operations performed by each cell are crucial to achieve the correct solution. On the other hand, Spiking Systems (SS) are composed by spiking neurons that receive the information codified into spikes, also called action potential, from several inputs with different weights, and after an internal processing, they produce new spikes in the output that are sent to other spiking neurons. The number and the connectivity between spiking neurons will determine the Spiking System. Weights are programmable and usually learned in order to make effective the result of the network. These weights usually depend on the spike rate of the inputs. The learning process for these weights is called Spike-Timing-Dependent Plasticity. One example of spiking neuron is the Integrate and Fire neuron (Farabet et al., 2009). This neuron can receive weighted spikes from different inputs. It performs internally an addition operation. When the internal value is greater than a configurable threshold, an output spike is produced and the neuron is reset. Researches and engineers try to integrate into a chip several thousand of these IF neurons, but a connectivity problem arises when they try to extract the output of all cells from the chip. The Address-Event-Representation (AER) is a possible solution. AER is a neuro-morphic communication protocol for transferring asynchronous events between VLSI chips that implement a spike-based process typically by using spiking neurons. These neuro-inspired implementations have been used to design sensor chips, like retinas (Lichtsteiner et al., 2008) and cochleas, processing chips (convolutions, filters) and learning chips, which makes it possible to develop complex, multilayer, multichip neuro-morphic systems (Mahowald, 1992).

Both CA and SS have important similarities and they also complement each other. CA is a processing model for problem solving and SS with AER gives a solution for implementing a grid of neurons in hardware. In this chapter our goal is to join both of them and study the viability of this new field for visual processing. Most of the complex visual processing mechanisms are based on solving a set of convolutions for edge detection, contrast adjustment, blur, noise reduction ... But it may also be possible to detect or recognize simple

shapes using large enough convolution filters. In SS the information is codified into spikes, so special sensors must be used or when using conventional digital sensors, special converters must be implemented. A camcorder, for example, offers a frame-based video that consists of a sequence of frames with a normal frame rate of 25-30 frames per second. In contrast a spike-based visual sensor will offer the information codified in spikes, in a continuous way, without the need of frames. These spikes can represent gray levels or any other characteristic, like temporal luminosity changes, contrast changes, etc. There are several published works that demonstrate the
efficiency of spike-based visual processing, e.g. results of the European project CAVIAR, which Spiking System developed was able to sense, detect, filter and learn the trajectory of an object in a totally spike-based way, with a low latency time (<1ms), without frames and performing convolutions for object detection.
Thanks to the CA architecture and its characteristics, it is possible to perform more complex object detection and categorization. Cells in a CA can communicate with their neighbours in a more complex way than IF neurons. Therefore, while a complex visual processing needs several layers of IF neurons in a SS, it is possible to join several layers in a unique CA. A software simulator of spike-based CA is very useful and necessary when trying to implement this processing in hardware.

2. What is Address-Event Representation?

Address-Event Representation (AER) is a spike-based representation technique for communicating asynchronous spikes between layers of neurons in different VLSI neuro-inspired chips, that allows the construction of multilayered, hierarchical, and scalable processing systems. The spikes in AER are carried as addresses of sending or receiving cells on a digital bus. Time represents itself as the asynchronous occurrence of the event. An arbitration circuit ensures that neurons do not access the bus simultaneously. This AER circuit is usually built using self-timed asynchronous logic (Boahen, 1998).
Every time a cell generates a spike, a digital word (address) which identifies the cell, is placed on an external bus. A receiver chip connected to the external bus receives the event and sends a spike to the corresponding cell. In this way, each cell from a sender chip is virtually connected to the respective cell in the receiver chip through a single time division multiplexed bus.

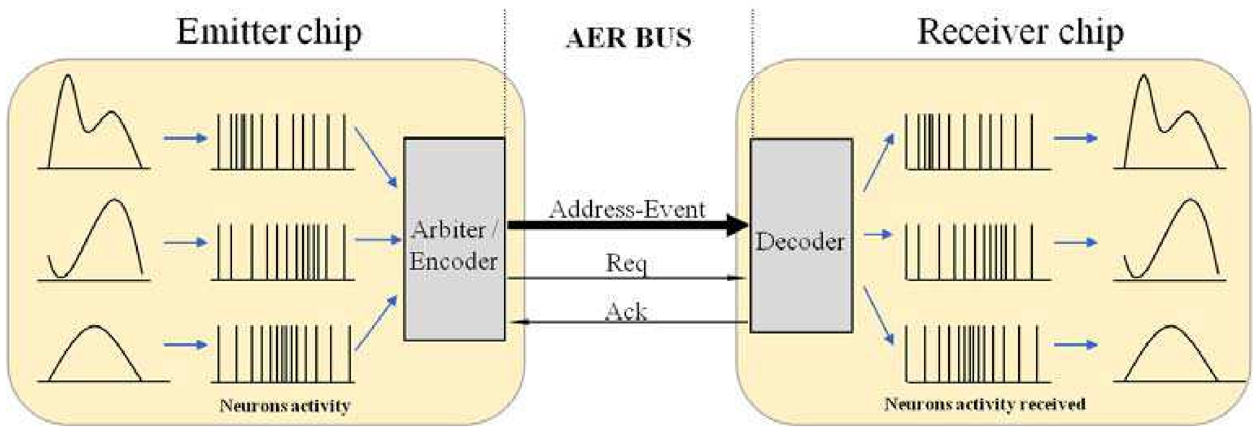


Fig. 1. AER inter-chip communication scheme

The most active cells access the bus more frequently than less active ones, for example, the AER information transmitted by a visual AER sensor is usually coded in gray levels, so the number of events transmitted by a pixel through the bus identifies the gray level of that pixel.

3. Cellular automaton and AER processing

One of the first processing layers in the cortex consists of applying different kinds of convolution filters with different orientations and kernel sizes. These spike-based convolution filters may be developed through an AER system based on Cellular Automaton (AER-CA)

The philosophy of AER systems is lightly different from CA but also similar in a certain sense, so an AER-CA only uses some characteristics of each one. For example, the state of a cell is a function of the current state of the cell and its neighborhood like CA, but a cell only modifies its state when it receives a stimulus.

An AER-CA consists of a 2D grid of cells and every cell is connected to its neighbors. The state of a cell is defined by a set of bits that varies longitudinally when a stimulus arrives. The state of each cell is defined as a function of current state of the cell and its neighbors.

When an AER event arrives, its address is decoded and a spike is sent to the corresponding cell. When a cell receives a spike, its state with an increment of the centre of the KxK kernel coefficient and it sends a spike to its neighbors which increments its state by the corresponding KxK kernel coefficient, i.e. the convolution kernel is copied in the neighborhood of the targeted cell. Therefore, for each spike received, several increment operations are carried out in the neighborhood of the target cell. When a cell reaches its threshold, a spike is produced and the state of the cell is reset.

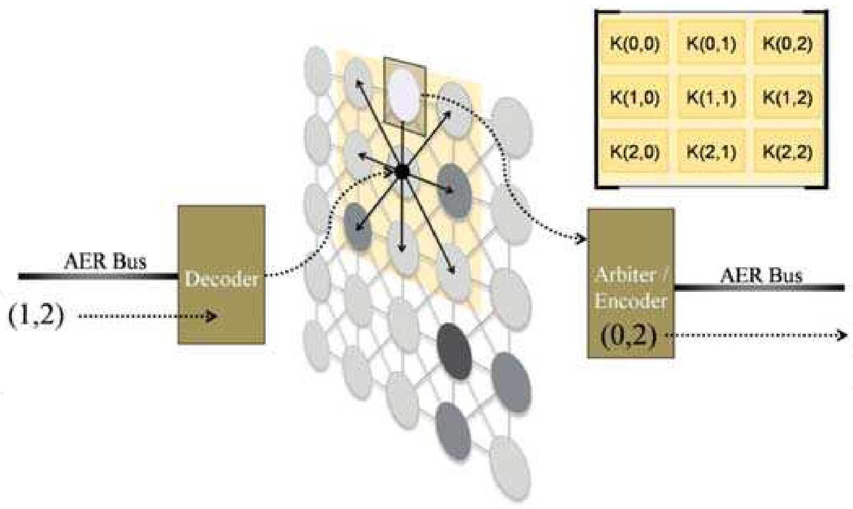


Fig. 2. Behavior of an AER-CA

I4n a Bi-dimensional image convolution, each cell represents a pixel from the image. So, each cell receives a number of events depending on the gray level of the corresponding pixel, so neighborhood increments its state as many times as the gray level and thus the multiplication of the convolution operation is implemented.

An acknowledge signal is sent to the AER emitter when the cell and its neighbors are updated. If the cell state achieves a threshold, such cell generates an event and resets itself.

This behavior corresponds to IF neuron model. Coefficients and threshold are available for all cells.

4. Software simulation of an AER-CA

A first approach to demonstrate the AER-CA behavior consists of using a simulation AER tool. In this section, we describe a self-made simulator to test AER systems whether CA-based or non-CA-based.
AER simulator is an application in C# that offers a friendly interface for simulating AER systems. This application contains a set of basic AER tools which are showed like boxes. These boxes are interconnected and configured to build the AER system to test. Once our design is completed, the simulation starts and results are showed or logged.

4.1 Description of AER simulation software

AER simulator is divided into 2 areas: the control-space and work-space. Control-space shows a set of AER tools that can be used and several buttons to start/stop the simulation, load/save an AER design, etc. Work-space is the area where the AER system design is built.

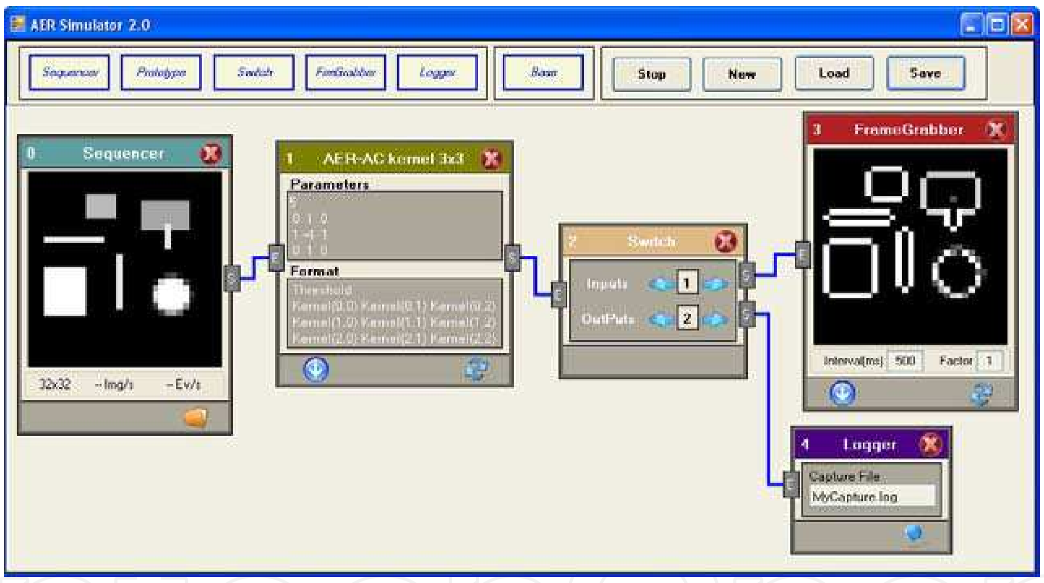


Fig. 3. Snapshot of AER simulation software

During AER system design, AER tools are dragged from control-space to work-space and they are interconnected through their input and output ports to create one or more AER chains. An AER design can contain several tools of the same type.
Every AER tool performs a concrete function in the AER chain. These tools are executed as process that communicate to others process. They can be classified into three categories: sender, actuator (modifier) and receiver device.
Sender devices are tools which generate a new AER event stream and they are used as input to the simulator. AER simulator implements 2 sender devices: Sequencer and DataPlayer. Sequencer generates an AER event stream from a bitmap image that is transmitted cyclically and Dataplayer sends repeatedly an AER event stream specified by a plain text file.
Receiver devices capture received events to be displayed or saved. They are used at the end of the chain to obtain results of the simulation. There are 2 receiver devices implemented in

the simulator: Framegrabber and Datalogger. Framegrabber captures events for a specified integration time and rebuilds an image from these events to be displayed and Datalogger saves received events in a plain text file. Dataplayer can use directly this file.
Actuators are those devices which modify AER event stream received in some way. There are 2 actuator implemented: Switch and Prototype.
Switches can perform 2 different functions simultaneously. They allow distributing received events to several outputs to create new streams from the same source. On the other hand, they can also join several event streams, e.g. they may be used to introduce simulated noise into the design.
Prototypes collect every algorithm to test registered in the simulator and display a list of them. User uses this list to select the prototype to test, so a prototype acts as the selected algorithm from this list by user. This tool also contains a textbox to specify the configuration parameters of the algorithm.
Every algorithm must be implemented in C# and it basically consists of implementing the interface 'Interface Prototype' through a template. Later, this new interface is registered in the simulator to be detected by prototype tools.
The implementation of this interface consists basically of:

a. Establishing in the class the name properties of the device, number of setting parameters, parameters format and example of the format that the user of the application may visualize using the prototype tool.
b. Defining the behavior of the prototype through the implementation of the class method 'Behavior'. The access to input and output ports is carried out using the 'Send' and 'Receive' methods.

Once the interface is implemented, it is added to the simulator in order to be detected by the prototype tool.

4.2 Software simulation of an AER-CA for AER filtering

A first step for the study of the AER-CA is to verify its behavior in the software simulator through the steps described in the previous section.
As it was previously indicated, the prototype behavior is established in the class method 'behavior'. Figure 4 shows the body of this method for a AER-CA for a 3x3 convolution kernel.
Each device of the simulator is executed as a process that is executed indefinitely and communicates and synchronizes with the rest of the processes through the buffers. In the code of figure 4, the prototype receives the events through the input buffer 'MyInBuf' and sends the new events to the next device through the output buffer 'MyOutBuf'. In this algorithm the cell net is represented by a bidimensional matrix, in which each element of the matrix keeps the internal state of a cell. Every time an event arrives, the algorithm executes a double loop in order to modify the state of neighbor cells. If the state of one cell reaches the threshold, they send an event to the exit and this cell resets its state.
In figure 3, window 1 labeled as AER-CA Kernel 3x3 represents the AER-CA prototype implemented for a 3x3 kernel. This prototype uses the release threshold of neurons and the coefficients of the 3x3 convolution matrix as input parameters, as shown in figure 3. The prototype uses a release threshold of 5 and a convolution kernel of [0 1 0 ; 1 -4 1 ; 0 1 0] for edge detection. The device FrameGrabber shows the edge of the image generated by the sequencer after applying the convolution to the image.

```
...
while (true)
{   EvReceived = MyInBuf.Receive();
    for (int i=EvReceived.i-1; i<=EvReceived.i+1; i++)
    {   for (int j=EvReceived.j-1; j<=EvReceived.j+1; j++) // for each i,j
        {   // when i,j are not out of the matrix.
            if ((i>=0 && i<TamMatrix.Y) && (j>=0 && j< TamMatrix.X))
            {   // Adds the corresponding kernel coefficient.
                NewState=State[i,j]+Kernel[i-EvReceived.i+1,j-EvReceived.j+1];
                if (NewState < Threshold)
                    State[i, j] = NewState;
                else
                {   State[i, j] = 0;
                    NewOutEvent.i = i;
                    NewOutEvent.j = j;
                    MyOutBuf.Send(NewOutEvent);
                }
            }
        }
    }
}
...
```

Fig. 4. C# code for simulating the AER-CA

5. Hardware design of an AER-CA

Once the behaviour of the AER-CA is checked in the simulator, we are going to explain a first hardware implementation of the prototype. As seen in figure 5, this hardware will consist basically of an input stage that controls the input AER bus and sends a signal to the corresponding cell of the cellular automaton from the input AER event received. The output stage moderates the signals received from the cells of the cellular automaton and translates them into AER events, which are sent through the output bus. The setting parameters that have been sent by USB are recorded in registries by the USB Controller. The cells of the cellular automaton modify their state from the setting parameters, Kernel Coefficients and Threshold, and the pulses received from the input stage.

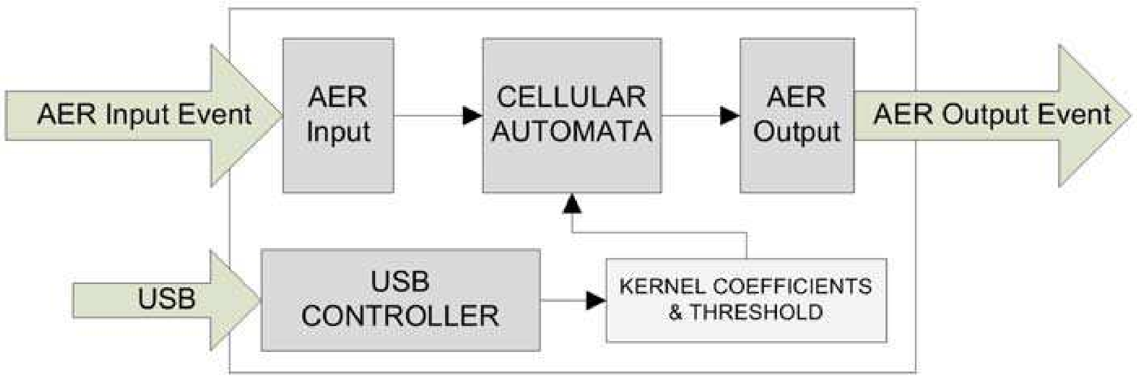


Fig. 5. USB-AER boards and MultiLoadFPGA application

One possible implementation of this system for a 3x3 kernel without the USB Controller is shown in figure 6. AER input is connected to the grid through two-level decoders. When an event comes, e.g. from a vision sensor, its address is divided into row and column. The first level decodes the row by Input Event X and the second level decodes the column by Input Event Y in order to select the targeted cell. Input Req signal notifies when the new event arrived.

In the AER Output, each cell is also connected to a two-level arbiter. The first level consists of a row arbiter and OR gates to encode the output event row address (Output Event X). The second level consists of a column arbiter and an array of multiplexors. This array selects a row from the grid and it is controlled by the first level. The column arbiter encodes output event column address (Output Event Y) from the array of multiplexer. The array of multiplexors and a multiplexor connects the cell to the output. 3x3 kernel elements and threshold are available for all cells of the grid as figure 6 shows. Each cell is connected to its eight neighbors through a single wire, only request signal, to minimize the number of connections.

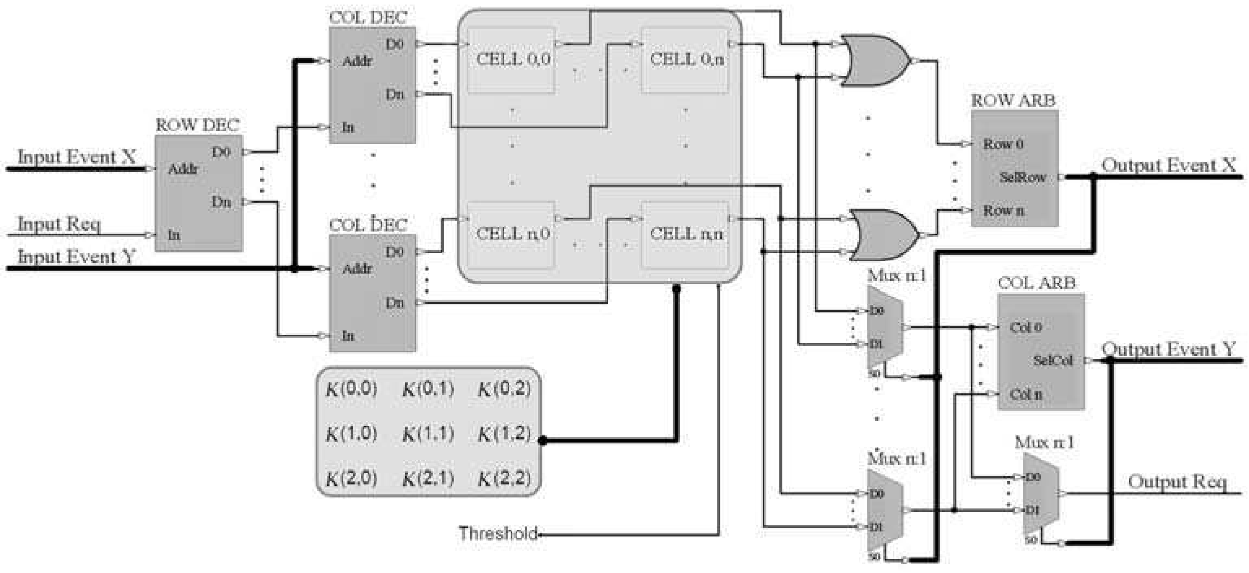


Fig. 6. Block diagram of an AER-CA

Digital logic of each cell in the grid for a 3x3 kernel and 8-wide bus is illustrated in Figure 7. A cell consists basically of two multiplexers to select the kernel coefficient to add (Mux8:1 and Mux2:1), an adder ADD8 to update the cell internal state, a register FD8CE to save the state, a comparator COMP to calculate when the cell must fire, D Flip-Flop FDSR to communicate the cell with the arbiters by handshaking process and some logic gates to control the overflow. The lower overflow is also controlled because some kernel coefficients may be negative.

When a cell receives a spike, it increments its internal state according to kernel center by ADD8 and sends a spike to its eight neighbors simultaneously. Each one of these neighbors adds a kernel coefficient to its internal state depending on where the spike arrives, e.g. when a spike comes from the bottom right cell, it adds the bottom right coefficient of the kernel to its internal state. That is, the cell that receives the spike and its eight neighbors (nine cells in total) modify their states by the corresponding kernel element.

When the state of a cell reaches the threshold, the comparator COMP in the figure 7 resets its state and saves a request in the FDSR. The output arbiter (Figure 6) processes the fired cells by a fixed priority. This arbiter generates the output event address according to the cell attended. When this output event is acknowledged, the arbiter clears the request stored in the FDSR of the cell.

When all fired cells are attended, an acknowledge signal is sent to the emitter AER chip and therefore processing of the incoming event concludes. In this way, no new input events can

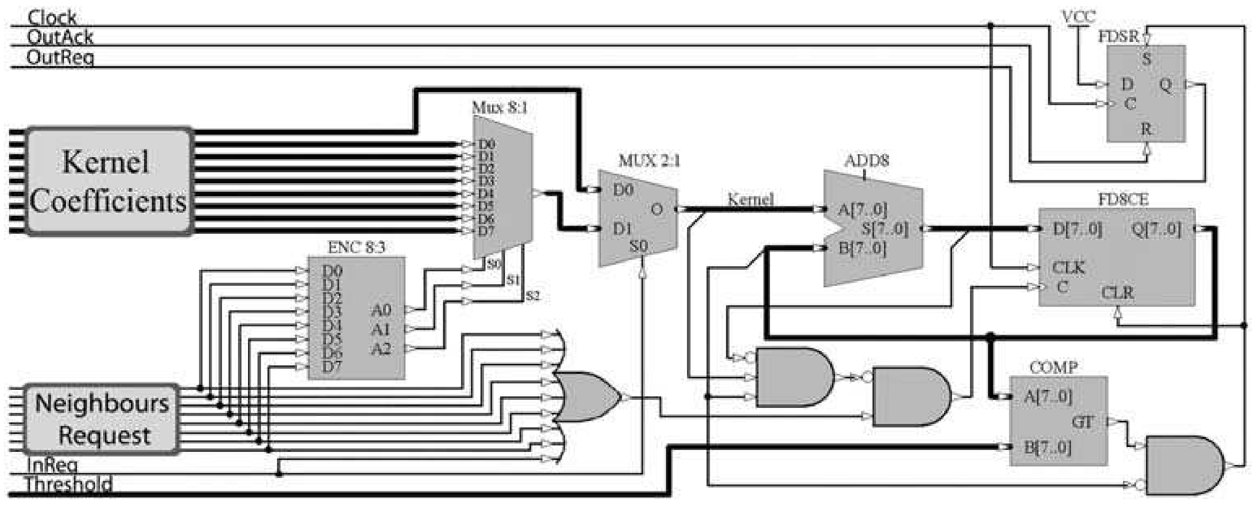


Fig. 7. Block diagram of a cell

arrive before the current event has finished. This constrain simplifies this design because each cell never has more than one pending request so only a Flip-Flop is needed (instead of Input FIFO) to store all new events that arrive while the current event is being processed. Additionally, output arbiter may be fixed priority, which is the simplest implementation, because there is no risk that lower priority cells starve.

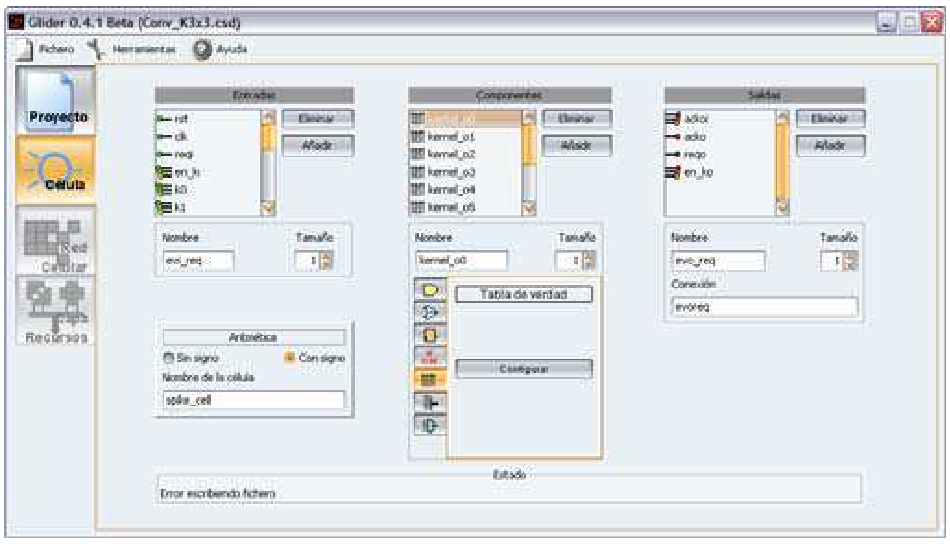
5.1 Implementation on a FPGA using glider

The design proposed will be implemented in hardware through a FPGA. The use of FPGAs has spread widely within the last few years both in prototype development research as for commercial implementations thanks to the multiple advantages they involve, such as resetting ease, shorter development time, they provide cheaper and more flexible solutions than those provided by application-specific integrated circuits (ASIC).
The hardware description languages (HDL) were first developed in order to describe the behavior of ASICs and they are used these days to describe the hardware design in FPGAs. For the design of the AER-CA in the FPGA we will use the VHDL language, which is, along with Verilog, one of the most spread hardware description languages.
The description of a cellular automaton through a description language may be difficult because it requires the definition of a great number of cell connections, especially in big cellular automatons. Glider, a graphic application for the description of cellular automatons in VHDL developed by the department of Electronic Engineering of the Technical University of Valencia, will be used in order to simplify the process

5.2 Glider

Glider is a graphic interface that allows: (a) describing a cellular automaton composed of cells and a cellular network, (b) translating the graphical definition into a Cellular Specification Description Language (CSDL) and (c) translating CSDL into VHDL for hardware implementation of the Cellular Automaton (Cerdá et al., 2003 ; Cerdá, 2004).
Glider has been implemented in Java 1.5, which makes it compatible with any platform and operating system.

There are three main descriptions that define Cellular Automaton: cells, cellular network and resource layer. Figure 8 shows the Glider interface for the description of a Cellular Automaton.
Cell: The cell definition consists of defining the set of inputs and outputs, that could come from or go to other cells or could be common for all cells; and defining the operations that cell has to implement with the inputs to generate the outputs. As the cellular automaton depends on the time for the state definition, both sequential and combinatorial logic must be used for a cell. A Cellular Automaton could be formed by several cell definitions.
Cellular Network: The connectivity of cells, the morphology of the network and type of cells used are the parameters that define the Cellular Network.
Resource Layer: This layer defines how a cellular network is connected to another cellular network. It also defines the global inputs and outputs, and the initialization of cells.



Nombre: evi_req Tamaño: 1

Nombre: kernel_o0 Tamaño: 1

Nombre: evo_req Tamaño: 1 Conexión: evoreq

Nombre de la célula: spike_cell

Estado: Error escribiendo fichero

Fig. 8. Snapshot of Glider Java application

Figure 9 shows a block diagram of the Glider internal operation. The graphics interface generates the CSDL files of the Cellular Automaton definition. The CSDL compiler is invoked automatically from this Java application and VHDL files are generated. The console messages are redirected to the application.

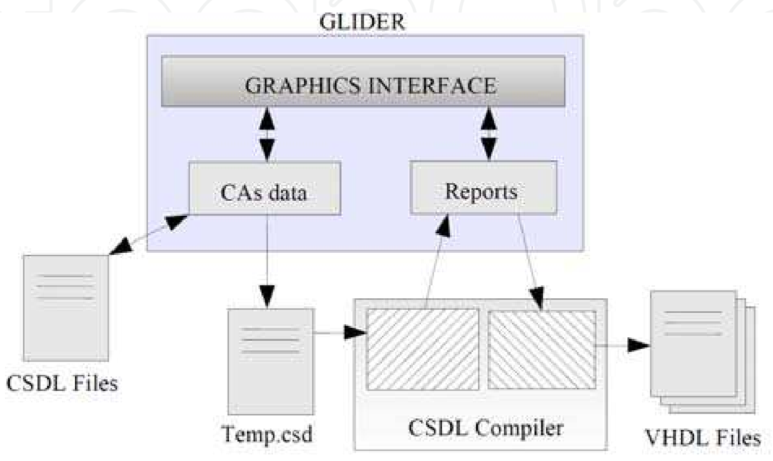


Fig. 9. Internal block diagram of Glider Java application

An interesting capability of Glider is that it keeps continuously checking the consistency of the CSDL generated from the graphics developed by the user. The error or warning messages are reported to the user automatically and in real time.

6. Testing scenario

In order to test this prototype the USB-AER board will be used. This card was developed by the department of Computer Architecture and Technology of the University of Seville for the development of AER devices and it is based on a FPGA Spartan-II 200 of Xilinx.

6.1 AER tools for hardware implementation of an AER-CA

The USB-AER board includes two AER ports (Gomez-Rodriguez et al., 2006), an input AER bus and an output AER bus, connected directly to the FPGA that allows implementing any hardware for manipulating or processing AER information. The Spartan-II 200 FPGA that can be loaded from MMC/SD or USB through the C8051F320 micro-controller. It also contains a large SRAM bank (512Kx32 12ns).

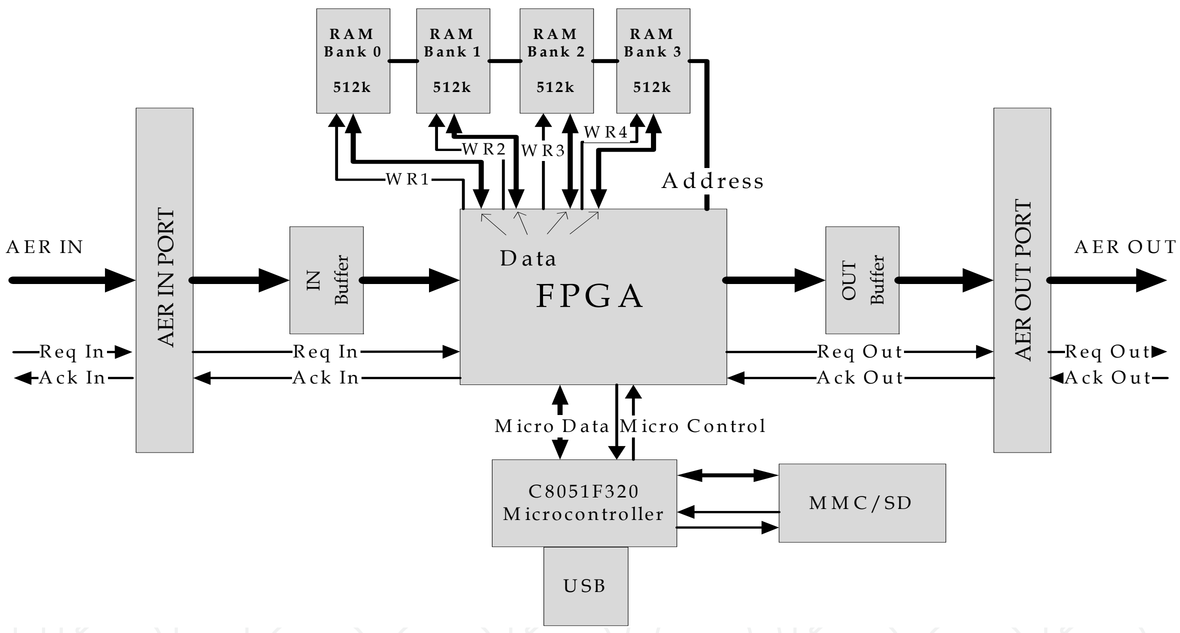


Fig. 10a. USB-AER Board block diagram

The USB-AER functionality depends on the module that is loaded in the FPGA. For example, it may act as a sequencer, monitor, mapping, event processor, data-logger, etc. Most of such functionalities may be performed in a standalone manner. This standalone operating mode requires loading the FPGA and RAM banks from some type of non-volatile storage, e.g. MMC/SD cards. USB input is also provided for development stages.

6.2 Testing AER-CA

Three USB-AER boards have been connected in line, as figure 10b shows, in order to test and measure the performance of these processors. Sequencer firmware is loaded in the first board to send a stream of events which depends on the image previously loaded. The convolution processor under test is loaded in the second USB-AER board. This board receives spikes from the sequencer and carries out the filtering operation according to 3x3 configurable kernel loaded. The third USB-AER board is loaded with a data-logger firmware to store, in real-time, the time-stamped address of the convolution output events.

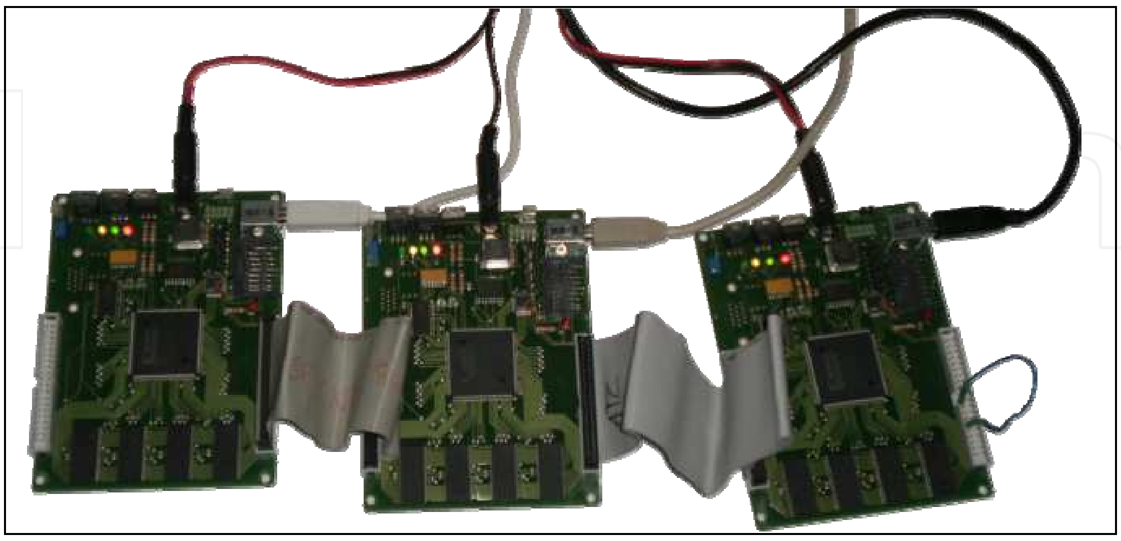
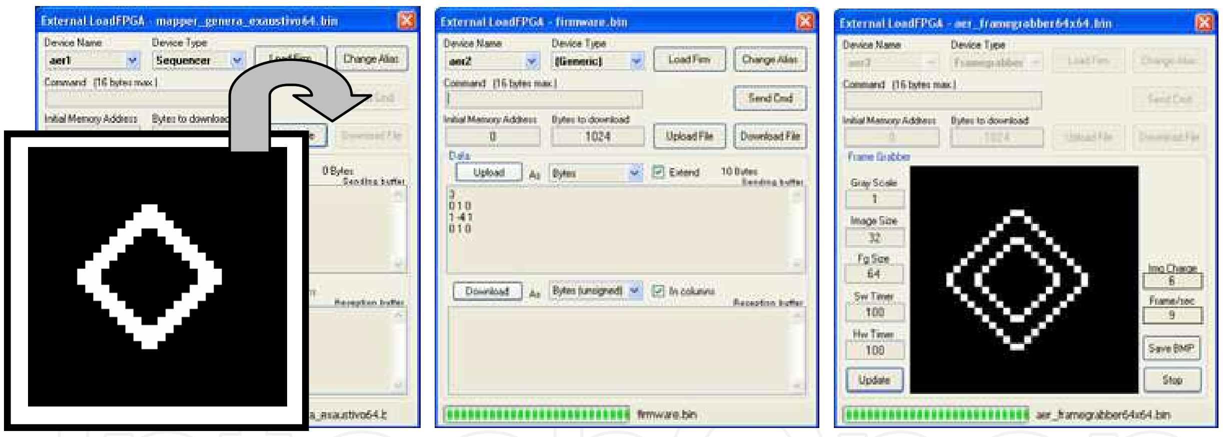


Fig. 10b. USB-AER boards for testing AER-CAs.


External LoadFPGA - mapper_genera_exaustivo64.bin
Device Name: aer1
Device Type: Sequencer

External LoadFPGA - firmware.bin
Device Name: aer2
Device Type: (Generic)
Initial Memory Address: 0
Bytes to download: 1024
Data: Upload As Bytes ✓ Extend 10 Bytes
```
3
0 1 0
1 -4 1
0 1 0
```
Download As Bytes (unsigned) ✓ In columns

External LoadFPGA - aer_framegrabber64x64.bin
Gray Scale: 1
Image Size: 32
Fg Size: 64
Sw Timer: 100
Hw Timer: 100
Img Change: 6
Frame/sec: 9

Fig. 11. Snapshot of MultiloadFPGA application

A Windows XP application called MultiloadFPGA (figure 11) controls and configures every USB-AER via USB. Captured spikes can be processed off-line by MATLAB to test the processor and measure performance.

6.3 Performance

The implementation developed requires three clock cycles for every event received: a synchronization cycle, a cycle for edge detection and another one to calculate cell states. Furthermore, this version requires two cycles for every new event generated: one cycle to send an event and another one to wait for the acknowledgement. This implementation yields up to 16.6 mega-events per second when no event is generated. In addition, nine ADD operations are computed every three cycles, thus the system yields up to 150 MOPS when a 50 MHz clock is used.

The event rate achieved is determined not only by convolution processor delay but also by input and output event rate. The input event rate depends on the image loaded into the sequencer and the output event rate is related to the kernel coefficients and threshold.
Figure 12 shows the percentage of the resources used in the Spartan-II 200 of the USB-AER board for different net sizes. Each cell records its state in a 8 bit registry and the kernel coefficients are 4 bit.

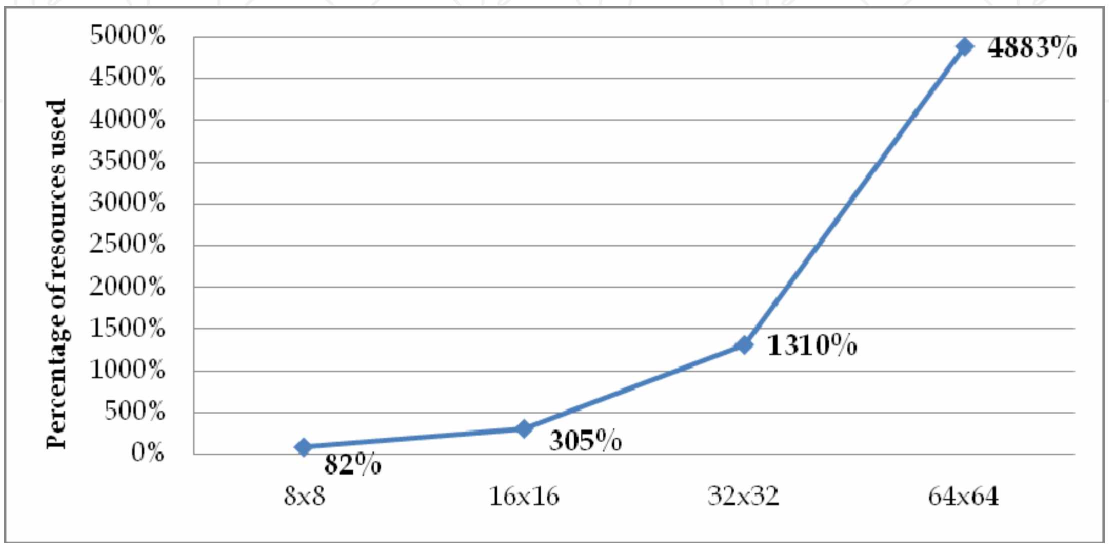


Fig. 12. Percentage of used resources respect to Spartan-II 200

An 8x8 grid spends 84% of all resources in a Spartan-II 200 and it is the largest grid that may be loaded. The resources needed for the AER-CA grow fast with the increase of the net size, e.g. a 32x32 grid requires a FPGA thirteen times larger. This growth is due to the requirement for individual resources of every cell implemented.

7. Resource optimization

The previous implementation requires many resources, so a new implementation is proposed that optimizes resources.

7.1 AER-CA in memory

This new implementation exploits a quirk of the AER-CA: when an input spike arrives, only a subset of cells work simultaneously, the targeted cell and its neighbors. So, it may be possible to maintain the state of every cell in memory and to implement only a shared subset of cells (computational units) that perform the operations needed when a new spike arrives. This new implementation requires less logic but uses memory banks to save the cell states.
Each one of the implemented cells always works with the same kernel coefficient and a different cell but it is the same neighbor regarding to the target cell. Additionally, cell states are strategically distributed to several memory banks so that each implemented cell accesses to a different RAM bank simultaneously.

An example for a 3x3 Kernel and 5x5 grid size is illustrated in figure 13. Each location in the grid indentifies a cell and it is labeled by Bx where x is the number of bank associated to that cell. When the event (2, 3) arrives, the implemented cell (1, 1) uses kernel element K(1,1) and bank B6 to modify the state of cell C(3, 2), the implemented cell C(0,2) uses K(0,2) and B4 to modify the state of neighbor (1,4), and so on until eight neighbors, but each implemented cell accesses to a different bank.
The memory address and memory bank that every implemented cell uses is a function of the corresponding row and column cell. When row and column are divided by 3, quotient corresponds to memory addresses and remainder indicates memory banks, e.g. when the event (2, 3) arrives, neighbor (1, 4) uses B(1 mod 3 , 4 mod 3) = B(1,1) that belongs to B4 (because B(0,0) = B0, B(0,1) = B1, B(0,2) = B2, B(1,0)= B3, B(1,1) = B4, etc).

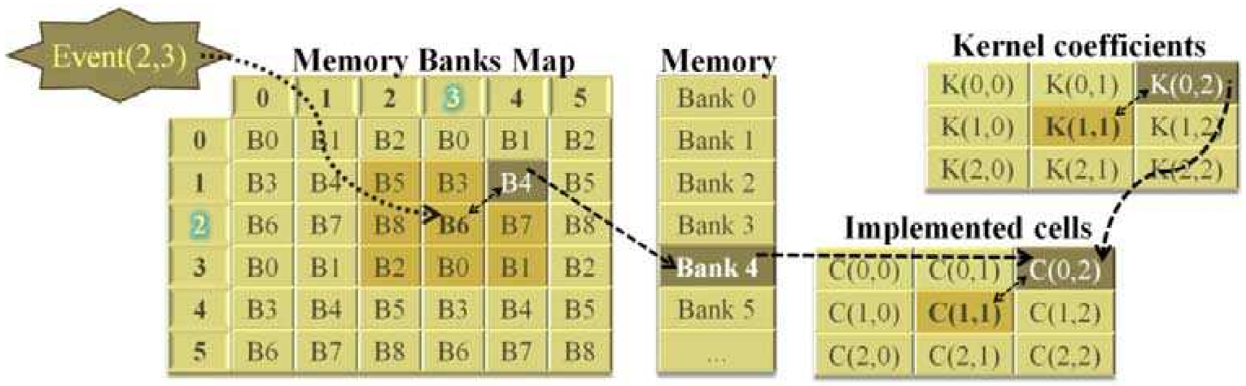
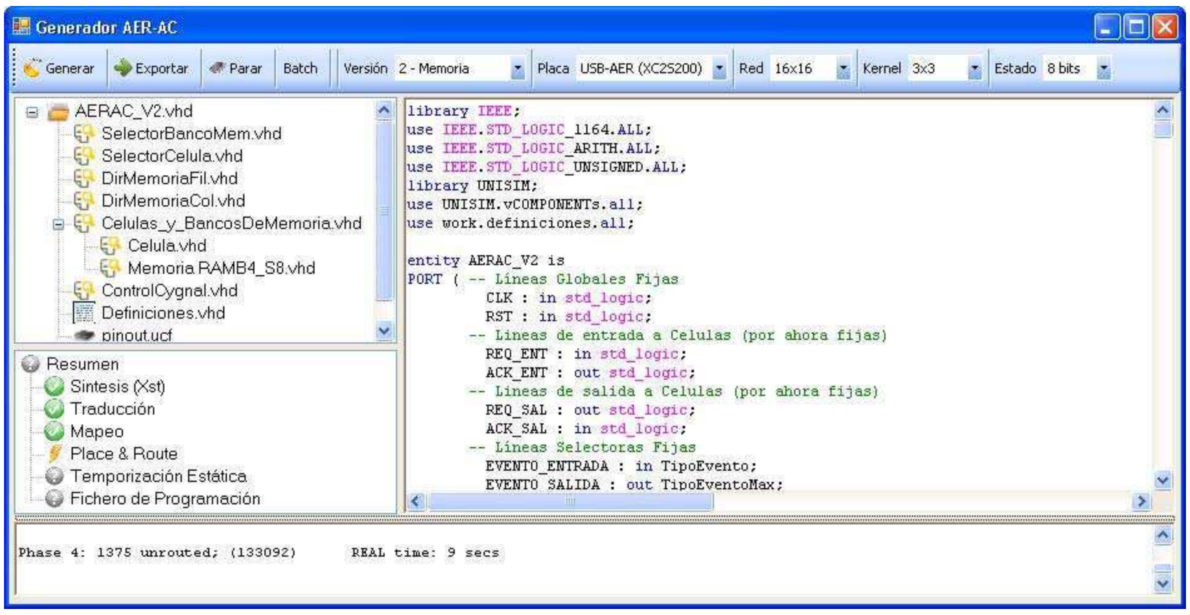


Fig. 13. Example of 3x3 Kernel AER-CA with cell states in memory

7.2 Implementing AER-CA in memory

As it would happen in the first version, the development of element nets in VHDL may be difficult, especially if we want to make several designs with different net sizes, kernel, etc. This is why an application in C# that generates the AER-CA automatically from the desired parameters has been designed. Besides, this application allows to synthesize the VHDL code generated for the FPGA wanted. Figure 14 shows a picture of the application running.
3 areas can be differentiated within this application. The control zone is located at the upper section and it is used to set the design wanted and to initiate the generation and synthesis process. The design parameters that will be used to set the development board are net size, convolution kernel size and the number of bits used to store the state of each cell. The bottom section shows a state window to report the evolution of the process. The left-mid section shows the tree of files generated and the state of the synthesis process. The right section shows the code of the selected file in the file tree. This application also allows to choose the AER-CA implementation desired to be generated, the original version or the memory-based one.



```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
library UNISIM;
use UNISIM.vCOMPONENTs.all;
use work.definiciones.all;

entity AERAC_V2 is
PORT ( -- Líneas Globales Fijas
         CLK : in std_logic;
         RST : in std_logic;
       -- Lineas de entrada a Celulas (por ahora fijas)
         REQ_ENT : in std_logic;
         ACK_ENT : out std_logic;
       -- Lineas de salida a Celulas (por ahora fijas)
         REQ_SAL : out std_logic;
         ACK_SAL : in std_logic;
       -- Líneas Selectoras Fijas
         EVENTO_ENTRADA : in TipoEvento;
         EVENTO_SALIDA : out TipoEventoMax;
```

```
Phase 4: 1375 unrouted; (133092)      REAL time: 9 secs
```

Fig. 14. Snapshot of AER generator application

7.3 Performance

This new implementation saves many resources in comparison to the former one. The diagram in figure 15 shows the percentage of the internal logic of the Spartan-II 200 needed to implement this new version for different sizes of kernel. For greater net sizes the percentage of usage of the Spartan-II 200 increases very slowly. This is because the number of implemented cells does not vary with the size of the net, unlike what occurred in the first version. The number of cells implemented is determined by the size of the kernel.

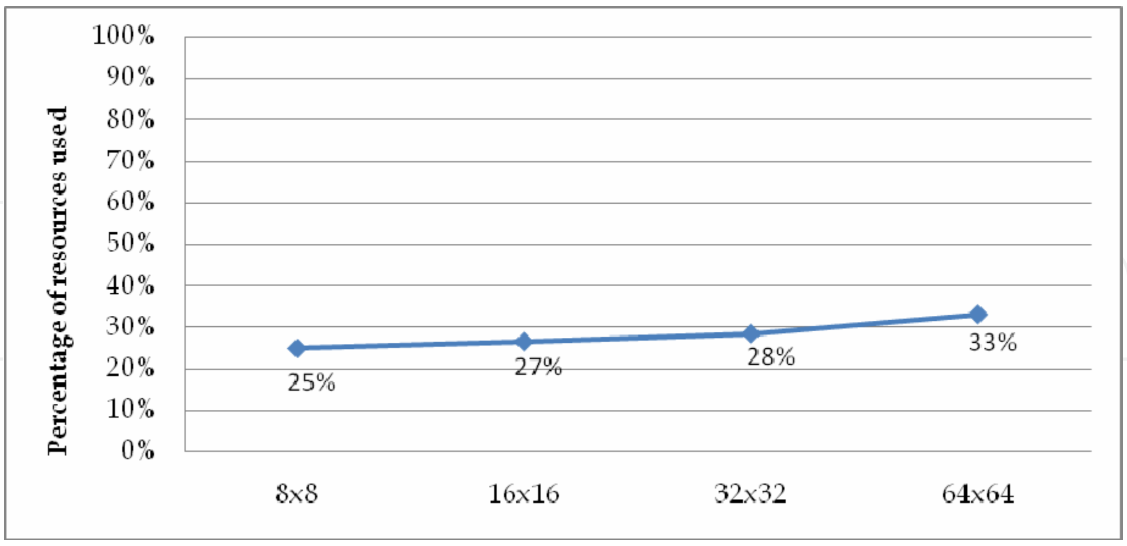


Fig. 15. Percentage of used resources for improved version respect to Spartan-II 200.

In this version, the memory of the FPGA is the one that determines the size of the net since the state of cells is stored in memory. The Spartan-II has 7 Kbytes distributed in 14 memory banks of 512 Bytes. An AER-CA with 3x3 kernel uses 3x3=9 from the 14 memory banks. The biggest net that can be obtained from an AER-CA with 3x3 kernel in a Spartan-II is 64x64

and it uses 4 Kbytes of the 7 Kbytes that the Spartan-II has. AER-CA with Kernel greater than 3x3 requires FPGA with a greater number of banks.
This new implementation requires six clock cycles per received event. Unlike the previous version, this implementation takes four cycles to calculate cell states: A cycle to calculate memory addresses and banks, a cycle to connect each unit to the appropriate bank, another one to read the current state and the last one to write the new state in memory. As the former implementation, this version also requires two cycles per event. It performs up to 8.3 mega-events per second and 75 MOPS.

7.4 Future improvements. A probabilistic AER-CA

Both presented versions require storing the state of every cell. A probabilistic version may be developed to avoid storing states, i.e. a random number and the corresponding probability determine when cells fire instead of adding kernel coefficients until threshold is achieved. This new version may increase processing speed and may be able to work with larger grids in the Spartan-II 200.

8. Conclusion

Cellular Automaton approach has been proposed to develop AER neuro-inspired filters for vision processing. These filters can implement two layers, one for the input processing and the second one thanks to the evolution rule.
AER filters based on 3x3 kernel convolutions have been implemented in VHDL using Cellular Automaton approach.
Two 3x3 kernel convolution AER processor for vision processing inspired in Cellular Automaton have been implemented for FPGA. The original implementation assigns resources to each cell. It performs up to 150 MOPS for a 3x3 kernel and yields up to 16.6 Mega-event per second in a Spartan-II 200. An improved version, that stores the cell states in memory to save resources by reducing number of implemented cells, has also been implemented. This second version achieves up to 75 MOPS and 8.3 Mega-event/s.
A real scenario consisting of three USB-AER tools has been used to prove these implementations and to carry out a performance analysis.
A probabilistic version is suggested to increase processing speed and to reduce resources.

9. References

Boahen, K.A. (1998). Communicating Neuronal Ensembles between Neuromorphic Chips. Neuromorphic Systems. Kluwer Academic Publishers, Boston 1998.

Cerdá, J.; Gadea, R.; Herrero, V.; Sebastià, A. (2003). On the Implementation of a Margolus Neighborhood Cellular Autómata on FPGA. Field-Programmable Logic and Applications. Lecture Notes in ComputerScience 2778, pp. 76-785.

Cerdá, J. (2004). Arquitecturas VLSI de autómatas celulares para modelado físico (in Spanish). PhD Thesis. Universidad Politécnica de Valencia, Valencia, Spain, 2004.

Farabet, C.; Poulet, C.; Han, J.Y.; LeCun, Y. (2009). CNP: An FPGA-based Processor for Convolutional Networks. International Conference on Field Programmable Logic and Applications, 2009. FPL 2009.

Gomez-Rodriguez, F.; Paz, R.; Linares-Barranco, A.; Rivas, M.; Miró, L.; Jimenez, G.; Civit, A. (2006). AER tools for Communications and Debugging. Proceedings of the IEEE ISCAS 2006, Kos, Greece. May 2006.

Lichtsteiner, P.; Posch, C.; Delbruck, T (2008). A 128×128 120 dB 15 μs Latency Asynchronous Temporal Contrast Vision Sensor. IEEE Journal of Solid-State Circuits, IEEE Journal, Vol 43, Issue 2, pp. 566-576, Feb. 2008.

Mahowald, M. (1992). VLSI Analogs of Neuronal Visual Processing: A Synthesis of Form and Function. Ph.D. Thesis. California Institute of Technology, Pasadena, California 1992.