Implementation of Signal Processing Tasks on Neuromorphic Hardware

Olivier Temam and Rodolphe Héliot

Abstract— **Because of power and reliability issues, computer architects are forced to explore new types of architectures, such as heterogeneous systems embedding hardware accelerators. Neuromorphic systems are good candidate accelerators that can perform efficient and robust computing for certain classes of applications. We propose a spiking neurons based accelerator, with its hardware and software, that can be easily programmed to execute a wide range of signal processing applications. A library of operators is built to facilitate implementation of various types of applications. Automated placement and routing software tools are used to map these applications onto the hardware. Altogether, this system aims at providing to the user a simple way to implement signal processing tasks on neuromorphic hardware.**

I. Introduction

THE domain of processor architecture is currently experiencing a set of dramatic and rapid changes due to increasingly severe technology constraints. Silicon technology scaling has been fueling the VLSI circuits performance increase for decades, thanks to larger integration, higher speed, and lower power consumption. Power constraints have arisen with recent sub-micrometer processes, and clock frequencies have stabilized. Increasing the performance now implies to increase shift to multi-cores [1], and more specifically to heterogeneous multi-cores [2]: a combination of cores and specialized circuits, called accelerators. Using specialized circuits, it is possible to drastically reduce the power and time necessary to perform an operation, and hence of an algorithm. A key point in accelerator design is to find the best tradeoff between flexibility -i.e. an accelerator that spans a broad set of algorithms- and power efficiency.

Moreover, another technology constraint strongly influences the choice of accelerators: robustness. As transistor size decreases, process variability [3] (the varying latency and power dissipation of processor components, the variability of transistor properties,...), transient faults and permanent defects are becoming more and more common. Hence, there is a need for robust accelerators that can cope with hardware variability and defects. In this context, hardware neural networks are excellent candidate accelerators. With respect to robustness, they can compensate for hardware variations or defects thanks to learning and to the distribution of information across multiple neurons and synapses. With respect to application scope and efficiency, as we will later see, neurons considered as continuous-time operators can efficiently implement a broad range of signal processing tasks (radio, audio, image, video,...)

Hardware implementations of spiking neural networks have received considerable attention since the pioneering work of C. Mead [4]. Multiple implementations of silicon neurons have been proposed [5], [6], and interconnecting large ensemble of spiking neurons has been made possible thanks to the introduction of Address-Event Representation (AER) [7]. These neuromorphic architectures have created a paradigm shift is the way the brain is studied, since they allow for real-time simulation of large-scale networks [8], [9]. Although these architectures were primarily designed to model and understand the brain, a growing number of studies report the use of neuromorphic systems for computing purposes [10], [11], [12]. To be able to use a neuromorphic system for computing, there is a need for a computation model. A good computation model should allow the user to easily program the neuromorphic hardware, while being able to target a broad set of applications. To build a computation model, a possible solution is to replicate biological observations such as learning rules (e.g. spike-timing dependent plasticity [13]) or neural circuits (e.g. neural pathways for auditory [14] or visual processing [15]) on neuromorphic hardware. This approach is however limited in scope by our incomplete knowledge of brain functions. Another solution is to engineer a computing framework that is not necessarily biologically relevant, but which computational properties can easily be understood and synthesized.

In this paper, we propose a spiking neural networks based accelerator, with its hardware and software, that can be programmed to execute a wide range of signal processing applications. We show below that spiking neural networks can be programmed in an explicit way to perform signal processing tasks (section II). To this aim, we develop and a library of operators that can be used as building blocks for applications. The library specifies the synaptic weights that are assigned to a group of neurons in order to map a desired function. These functions can be then be composed in a modular approach to build applications. Applications are then mapped onto neuromorphic hardware; to this aim, we develop automated place and route software tools (section III).

II. Computing principles

A. Neuron Model

In the past two decades, a significant amount of work has been done in the field of Spiking Neural Networks

Olivier Temam is with INRIA Saclay Ile-de-France, rue Jean Rostand, 91893 Orsay Cedex, France (email: olivier.temam@inria.fr). Rodolphe Héliot is with CEA-LETI, Minatec Campus, 17 rue des Martyrs, 38054 Grenoble Cedex, France (email: rodolphe.heliot@cea.fr).

This work was supported by a "Agence Nationale de la Recherche" grant ANR-09-RPDOC-002-01.

978-1-4244-9637-2/11/$26.00 ©2011 IEEE

(SNN), or pulsed neural networks [16]. Spiking neurons models differ from their continuous[1] counterparts in that they exhibit a dynamic, continuous-discrete hybrid behavior. Spiking neurons integrate inputs over time and space -i.e. integrating inputs from different neurons. A simple spiking neuron model is the Leaky Integrate and Fire (LIF) model, that can be written as follows:

$$\dot{V}_i = \frac{-V_i}{\tau_i} + \sum_{n}^{j=1} W_{ij}\ s_j(t - \Delta_{ij}) \tag{1}$$

with the additional output and reset equation:

$$\begin{array}{ll} \text{if } V_i < V_{th}, & \text{then } s_i = 0 \\ \text{if } V_i \geq V_{th}, & \text{then } \left\{ \begin{array}{l} s_i = 1 \\ V_i = 0 \end{array} \right. \end{array} \tag{2}$$

where V_i is the internal potential for neuron i, V_{th} is the neuron threshold, s_j is the output of neuron j, τ_i is a leakage time constant, W_{ij} is the synaptic weight from neuron j to neuron i, and Δ_{ij} is the synaptic delay. In the following, all synaptic weights are normalized with respect to V_{th} (i.e. synaptic weights belong to $[-1; 1]$).

B. An operators-based approach

We show here how networks of spiking neurons can be configured in an explicit way to perform signal processing tasks. The behavior of a neural network is conditioned by its connections set, together with the neurons' internal parameters. Hence, to have a neural network perform a specific function, one may assign to it specific connectivity and parameters sets. Within such a framework, a wide set of operators -a library- can be implemented by networks of LIF spiking neurons. Signals are classically encoded using the instantaneous firing rate of neurons. We detail below the neural connectivity and parameters that allow a neural network to perform two example functions: addition and temporal differentiation. The spiking neural networks behaviors are then simulated to validate their functionality.

1) Application example: In this paragraph, we illustrate the library and operators concept with a simple yet realistic [17] video processing example: an intensity change detection algorithm. We assume the algorithm input is a grid of 3x3 pixels, for instance acquired by a digital camera, and provided as a set of 9 analog time-varying gray scale values; the input can be viewed as a 9-dimensional time function continuously fed to the accelerator. An intensity change is deemed detected if the average of the absolute values of the temporal derivatives of the 9 inputs exceed a constant threshold. A dataflow diagram of this algorithm is shown in Figure 1.

All operators shown in Figure 1 can be implemented using one or a few neurons. Neuron A in Fig. 2 is used as a delay neuron: for a given input signal $u(t)$, its output is $u(t - \delta t)$. Neuron B is then used as a difference operator, with inputs weights being $+1/\delta t$ and $-1/\delta t$. As a result,

[1]we refer here to neuron models classically used in Artificial Neural Networks

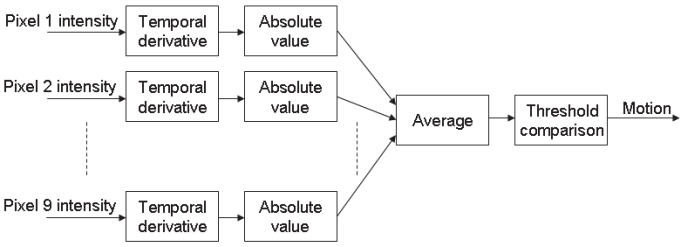

Fig. 1. Flowchart for a simple 3×3 pixels intensity change detection algorithm.

its output is $\frac{u(t)-u(t-\delta t)}{\delta t}$, which is equivalent to a time-derivative operator. This operator is circled in red in Fig. 2. Given the nature of a spiking neuron, it can only output positive signals (since we assumed signals are encoded using spike rates). Hence, neuron B can only output the positive values of the time-derivative: $max(\frac{u(t)-u(t-\delta t)}{\delta t}, 0)$. The same difference operation is performed by neuron B', swapping positive and negative weights. As a result, its output is $max(\frac{u(t-\delta t)-u(t)}{\delta t}, 0)$. Neuron C trivially sums (weights set to 1) the two outputs from B and B' to compute $max(\frac{u(t)-u(t-\delta t)}{\delta t}, 0) + max(\frac{u(t-\delta t)-u(t)}{\delta t}, 0) = |\frac{u(t)-u(t-\delta t)}{\delta t}|$, thus performing the absolute value operator, circled in green in Fig. 2. Then, the average of the absolute values is implemented using a 9-input neuron where all weights are set to $1/9$. And finally, the comparison with the threshold is implemented as a difference between the 9-input average and a constant threshold using a difference operator (neuron E).

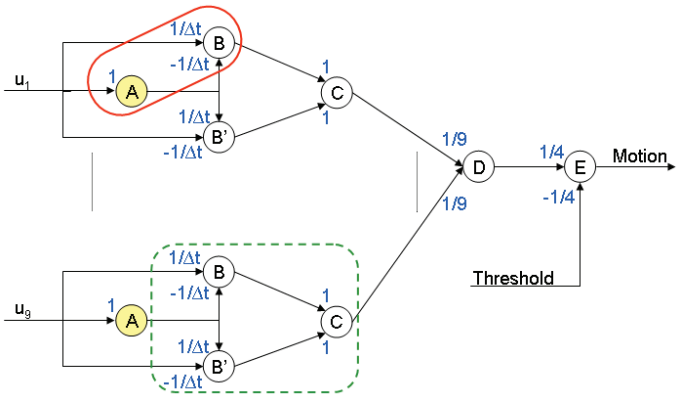

Fig. 2. Neural implementation of the detection algorithm, using basic neural operators: temporal derivation (circled in red) using a delay neuron (yellow neurons labeled A) and a difference operator (neuron B); absolute value (neurons B, B' and C, circled in dashed green line); averaging and threshold comparison (neuron D).

Note that, in fact, the difference operator (neuron E) does not use 1 and -1 as input weights: indeed, this set of weights would force the neuron to output a spike each time there is an incoming spike from the positive ($w = 1$) input, and do nothing but to reset to its resting potential when there is an incoming spike on the negative input ($w = -1$). To obtain a correct behavior, weights need to be set < 1 in

absolute value, which induces an output frequency decrease for this operator. This has to be factored in the subsequent chain of operators. In our example, the threshold input used in the difference operator (neuron E) must be adjusted correspondingly.

Altogether, the algorithm has been implemented with a set of simple operators, all mapped to analog spiking neurons. Since most signal processing applications can be built out of a restricted set of elementary operators, the same approach has a broad application span. For instance, a correlation requires the sum, difference, average and multiplication [18] operators; image filtering, e.g., convolution, essentially requires multiplication with a constant, image gradient is based on spatial derivation and thus on weighted sum and difference operators; control applications, e.g., proportional-integral-derivative (PID), requires the sum, difference, delay, derivative and integration operators,. . . .

From a theoretical point of view, the computational power of spiking neural networks has been assessed [19], [16]; it is known that spiking neural networks can implement any static or dynamic function [20], as well as state machines [21]. As a consequence, the potential application field of the proposed approach extends beyond signal processing tasks.

2) Simulation results: To validate our approach, we simulated the spiking neural network represented in Fig. 2 with a custom-made simulator developed under Scilab[2]. Input frames were composed of 9 pixels featuring a constant background intensity on top of which noise is added, the constant background component experiencing a sudden yet subtle change at time 0. The network behaves as expected, and Figure 3 shows the network output (neuron E in Fig. 2) together with 3 input frames at time $-10ms$, $30ms$, and $70ms$ from image intensity change. As expected, the output is positive just after the intensity changes, and then resets as the image stabilizes.

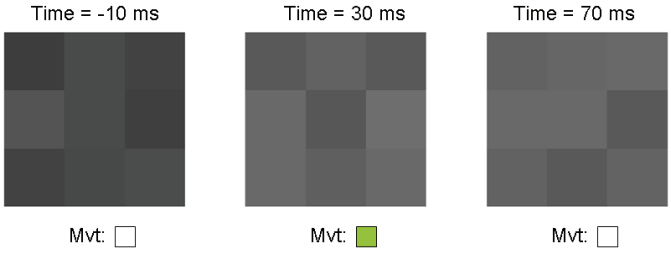

Fig. 3. Simulation of the intensity change detection application, using spiking neurons. Above figure shows 3 frames separated by $40ms$ each. Background illumination changes between frames 1 and 2; change is detected as expected.

Beyond the intensity change detector, we implemented and successfully simulated 3 other applications: image filtering, correlation and cross-correlation.

C. Programming: a Modular Software Library

A benefit of our approach is to make the system easy to program, thanks to a hierarchical and modular implementation of signal processing tasks. A user decomposes the analog signal processing task as a set of connected elementary tasks, in the same spirit as the intensity change detection algorithm presented in section II-B. Similarly, a more complex algorithm, e.g., body tracking can itself be implemented based on more elaborate algorithms such as motion detection. As a result, a full application is expressed as a combination of hierarchical modules decomposed into or combined with leaf modules (e.g., differentiation, delay, add, . . .). The leaf modules contain a description of their hardware-level implementation on the grid of spiking neurons: number of neurons, parameterization of their synapses, connections among neurons. This modular construction enables a library-based development, where each operator, leaf or hierarchical, is viewed as a library module. This modular software approach largely shields the user from the complexity of developing and implementing signal processing tasks on neuromorphic hardware. This composition of elementary functions results in an application graph that has to be mapped onto the hardware; this is developed in the next section.

III. Software to Hardware Mapping

A. Hardware description

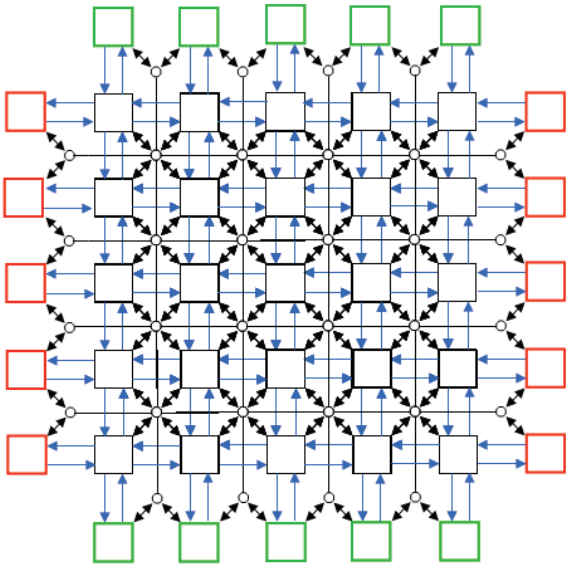

Fig. 4. Schematized presentation of the considered neuromorphic hardware. The array includes neurons (black squares), local (blue lines) and long-distance (black lines) interconnections, inputs (red), and outputs (green).

The hardware is implemented as a regular template of spiking neurons, as shown in Figure 4. This template is a regular grid of tiles, each tile being composed of spiking neurons and synapses. Neurons are implemented with analog silicon hardware; synaptic weights are stored with SRAM cells. The detail of the tile is shown in Figure 5. For each neuron, the input port on each orientation is actually composed of N_{tile} inputs, each associated with a 1-bit wire and a synapse. Similarly, there are N_{long} parallel global (long-distance) 1-bit wires horizontally and vertically crossing each tile, with

[2]An open-source platform for numerical computation; www.scilab.org

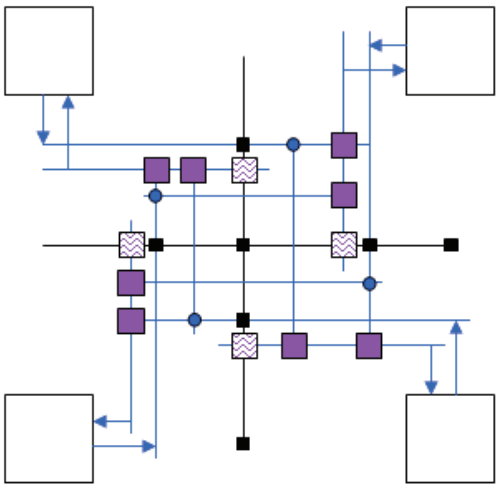

Fig. 5. Description of a tile (group of four neurons with synapses). Blue lines: local connections; Black lines: long-distance programmable interconnections; Black squares: programmable switch; Purple squares: synapses.

one synapse per horizontal or vertical wire for every other neuron on each side of the wire (see Figure 5).

Neurons are connected through direct local connections to their closest neighbors (blue links in Figures 4, 5), and can also establish long-distance connections with distant neurons through a dedicated, programmable interconnect (black links in Figures 4, 5). Input blocks (Figure 4, red) injects spikes into the array of spiking neurons, and readout blocks (Figure 4, green) collect output spikes from the array; these are the I/O mechanisms that allow the hardware accelerator to communicate with other components of the heterogeneous system in which it is embedded. Figure 4 represents an array of 5×5 neurons, but a real hardware would include a much larger number of spiking neurons ($N > 1000$).

B. Placement

For energy consumption issues, as well as congestion of long-distance communications infrastructure issues, it is best to use local connections as much as possible, i.e. to gather together groups of neurons that are densely interconnected. Keeping this in mind, we developed the placement algorithm described below.

1) Methods: In the same spirit as [22], we use a simulated annealing algorithm to place the software neurons from the application graph onto the hardware array of spiking neurons.

Simulated annealing is a heuristic for optimization problems, that is well adapted to large, discrete search spaces. A cost function to be minimized is needed, but no analytical computation of its gradient is required. Instead, at each step of the simulated annealing, a random move is performed; if the move decreases the cost function, it is accepted. If the move increases the cost function, it can be accepted with a probability that depends on the annealing "temperature", that is gradually decreased. At the beginning of the annealing procedure, temperature is high and virtually every move is accepted; towards the end, temperature is low and almost no move is accepted. This technique avoids the optimization procedure to get stuck in a local minimum. We use a wirelength Manhattan cost function that classically offers a good tradeoff between computation time and results quality. The cost function can be written as:

$$Cost = \sum_{i=1}^{N} \sum_{j=1}^{B_i} |x_C i,j - x_i| + |y_C i,j - y_i| \quad (3)$$

where N is the total number of neurons in the considered application graph; B_i is the number of neurons to which neuron i is connected; Ci,j is the j^{th} neuron connected to neuron i, x_i and y_i are the horizontal and vertical locations of neuron i, respectively.

Simulated annealing requires a good annealing schedule in order to provide with good solutions in an acceptable computation time. We computed the initial temperature in a similar manner as [22]. A random placement of the neurons is first performed; then $N^{4/3}$ moves (pairwise swaps between two neurons) are performed, and the standard deviation of the cost for each configuration is computed. The initial temperature is set to 10 times this standard deviation, so that virtually any move is accepted at the beginning of the annealing procedure. The temperature update schedule we use is again similar as the one proposed in [22], that increases the time spent at temperatures where a significant fraction of moves -but not all of them- are accepted. The simulated annealing is stopped when $Temperature < 0.005 \times Cost/N$. Since a move always affects at least one neuron, when the temperature is less than a small fraction of the average cost of a neuron, it is unlikely that any move that results in a cost increase will be accepted, so the simulated annealing is stopped.

2) Results: We placed the software graph presented in Figure 2 onto a physical hardware embedding an array of 16×16 neurons. The result is shown in Figure 6. Blue circle represent neurons; red circles are inputs, and green are outputs. Black lines represent the connections between neurons. Several observations can be made: first, neurons that are connected are placed close to each other; second, the network of neurons is placed close the Inputs and Outputs, in a corner of the array. Overall this behavior is exactly the one we desired for the placement procedure, since it minimizes the total length of connections. The results we obtain are thus very satisfactory. For clarity purpose, we only show here the placement of a small application (42 neurons total, 65 connections). We tested our placement procedure with applications involving a much larger number of neurons ($N > 700$); we obtained high-quality results as well.

C. Routing

1) Methods: Once neurons have been mapped, we attempt to map connections between pairs of neurons (routing). Since each connection must include one hardware synapse, the

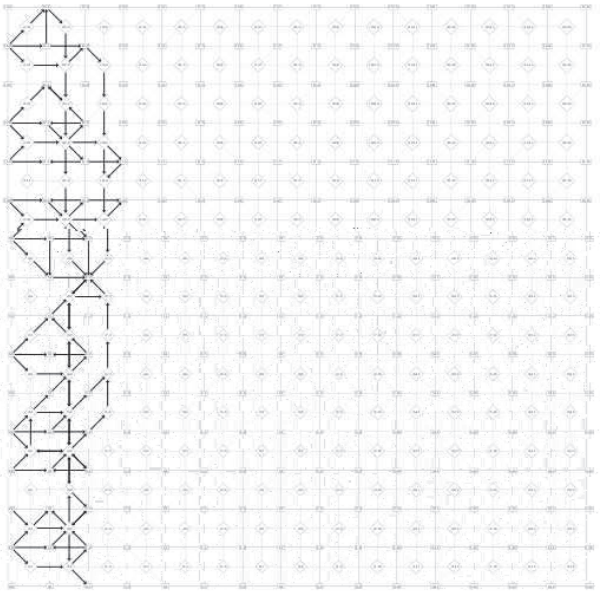

Fig. 8. Example of mapping for the `intensity change detection` application; the bold lines correspond to software connections; parallel wires are not shown for the sake of clarity.

2) Results: Until now, we could successfully place all four applications considered so far: `intensity change detection` with 42 software neurons and 65 software connections, `correlation` with 93 software neurons and 297 software connections, `image processing` with 329 software neurons and 718 software connections, and `cross-correlation` with 725 software neurons and 2298 software connections. The maximum mapping time, obtained with `cross-correlation`, was 30 seconds on PC with an iCore7 processor (one core used only). In Figure 8, we show the mapped routing for `intensity change detection` corresponding to the placement of Figure 6; the underlying hardware has the following characteristics: a grid of 16×16 neurons, with $N_{tile} = 4$ (4 parallel 1-bit wires between neighbor neurons), $N_{long} = 4$ (4 parallel 1-bit wires for global connections, horizontally and vertically). Notice the seemingly irregular disposition of connections due to the randomized mapping process. Over 10 mappings, the average number of bumped software paths was 0.2 in this example.

IV. Conclusions

In this paper, we proposed a spiking neurons based accelerator, with its hardware and software, that aims at providing to the user a simple way to implement signal processing tasks on neuromorphic hardware. The system is easy to program, thanks to a hierarchical and modular implementation of signal processing tasks; we were able to validate through simulation the functionality of an example application - intensity change detection on a 3×3 pixels video source. Application mapping onto the hardware is performed by automated software tools; we showed placement and routing results that behave as expected, minimizing communication lengths and congestion. We are currently designing and will tape out a circuit implementing the architecture we propose using ST CMOS 65 nm technology; applications will then be tested on actual hardware, and power efficiency evaluated.

References

[1] D. Mallik, K. Radhakrishnan, J. He, C. Chiu, T. Kamgaing, D. Searls, and J. Jackson, "Advanced package technologies for high-performance systems," *Intel Technology Journal*, vol. 09, no. 04, May 2005.

[2] M. Muller, "Dark silicon and the internet," in *Designing with ARM, EE Times Virtual Conference*, March 2010.

[3] S. Borkar, "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation," *IEEE Micro*, vol. 25, no. 6, pp. 10–16, 2005.

[4] C. Mead, *Analog VLSI and Neural Systems*. Addison-Wesley, 1989.

[5] M. Mahowald and R. Douglas, "A silicon neuron," *Nature*, vol. 354, no. 6354, pp. 515–8, 1991. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/1661852

[6] A. van Schaik, "Building blocks for electronic spiking neural networks." *Neural networks*, vol. 14, no. 6-7, pp. 617–28, 2001.

[7] K. Boahen, "Point-to-point connectivity between neuromorphic chips using address events," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 47, no. 5, pp. 416–434, mai 2000.

[8] ——, "Neuromorphic Chips," *Scientific American*, vol. 292, no. 5, 2005.

[9] R. Vogelstein, U. Mallik, J. Vogelstein, and G. Cauwenberghs, "Dynamically reconfigurable silicon array of spiking neurons with conductance-based synapses." *IEEE Transactions on Neural Networks*, vol. 18, no. 1, pp. 253–265, 2007.

[10] J. Buhmann, T. Lange, and U. Ramacher, "Image segmentation by networks of spiking neurons." *Neural computation*, vol. 17, no. 5, pp. 1010–31, mai 2005.

[11] R. Vogelstein, U. Mallik, E. Culurciello, G. Cauwenberghs, and R. Etienne-Cummings, "A multichip neuromorphic system for spike-based visual information processing." *Neural computation*, vol. 19, no. 9, pp. 2281–300, septembre 2007.

[12] S. Mitra, S. Fusi, and G. Indiveri, "Real-Time Classification of Complex Patterns Using Spike-Based Learning in Neuromorphic VLSI," *IEEE transactions on biomedical circuits and systems*, vol. 3, no. 1, pp. 32–42, 2009.

[13] "Unsupervised learning of visual features through spike timing dependent plasticity." *PLoS computational biology*, vol. 3, no. 2, p. e31, Feb. 2007.

[14] R. Lyon and C. Mead, "An analog electronic cochlea," *IEEE Trans. on Acoustics, Speech, and Signal Processing*, vol. 36, pp. 1119–1134, 1988.

[15] M. Riesenhuber and T. Poggio, "Hierarchical models of object recognition in cortex," *Nature Neuroscience*, vol. 2, pp. 1019–1025, 1999.

[16] W. Maass and C. Bishop, Eds., *Pulsed Neural Networks*. MIT Press, 1999.

[17] T. MIT, "Motion Detection and Segmentation," http://www.ai.mit.edu/projects/cog/VisionSystem/motion_detection.html.

[18] M. Srinivasan and G. Bernard, "A proposed mechanism for multiplication of neural signals," *Biological Cybernetics*, vol. 21, no. 4, p. 227236, 1976.

[19] J. Sima and P. Orponen, "General-purpose computation with neural networks: A survey of complexity theoretic results," *Neural Computation*, vol. 15, no. 12, pp. 2727–2778, 2003.

[20] C. Eliasmith and C. Anderson, *Neural Engineering: Computation, Representation and Dynamics in Neurobiological Systems*. MIT Press, 2003.

[21] U. Rutishauser and R. Douglas, "State-dependent computation using coupled recurrent networks," *Neural computation*, vol. 21, no. 2, p. 478509, 2009.

[22] V. Betz, J. Rose, and A. Marquardt, *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, 1999.

[23] W. Kocay and D. L. Kreher, *Graphs, Algorithms, and Optimization (Discrete Mathematics and Its Applications)*. Chapman and Hall/CRC, 2004.