AER Spiking Neuron Computation on GPUs: The Frame-to-AER Generation

M.R. López-Torres, F. Diaz-del-Rio, M. Domínguez-Morales, G. Jimenez-Moreno, and A. Linares-Barranco

Department of Architecture and Technology of Computers, Av. Reina Mercedes s/n, 41012, University of Seville, Spain
rlopez@atc.us.es

Abstract. Neuro-inspired processing tries to imitate the nervous system and may resolve complex problems, such as visual recognition. The spike-based philosophy based on the Address-Event-Representation (AER) is a neuromorphic interchip communication protocol that allows for massive connectivity between neurons. Some of the AER-based systems can achieve very high performances in real-time applications. This philosophy is very different from standard image processing, which considers the visual information as a succession of frames. These frames need to be processed in order to extract a result. This usually requires very expensive operations and high computing resource consumption. Due to its relative youth, nowadays AER systems are short of cost-effective tools like emulators, simulators, testers, debuggers, etc. In this paper the first results of a CUDA-based tool focused on the functional processing of AER spikes is presented, with the aim of helping in the design and testing of filters and buses management of these systems.

Keywords: AER, neuromorphic, CUDA, GPUs, real-time vision, spiking systems.

1 Introduction

Standard digital vision systems process sequences of frames from video sources, like CCD cameras. For performing complex object recognition, sequences of computational operations must be performed for each frame. The computational power and speed required make it difficult to develop a real-time autonomous system. However, brains perform powerful and fast vision processing using millions of small and slow cells working in parallel in a totally different way. Vision sensing and object recognition in brains are not processed frame by frame; they are processed in a continuous way, spike by spike, in the brain-cortex. The visual cortex is composed of a set of layers [1], starting from the retina, which captures the information. In recent years, significant progress has been made in the study of the processing by the visual cortex. Many artificial systems that implement bio-inspired software models use biological-like processing that outperforms more conventionally engineered machines [2][3][4]. However, these systems generally run at extremely low speeds because the models are implemented as software programs in classical CPUs. Nowadays, a growing number of research groups are implementing some of these computational principles onto real-time spiking hardware through the so-called AER (Address Event Representation) technology, in order to achieve real-time processing.

AER was proposed by the Mead lab in 1991 [5][7] for communicating between neuromorphic chips with spikes. In AER, a sender device generates spikes. Each spike transmits a word representing a code or address for that pixel. In the receiver, spikes are processed and finally directed to the pixels whose code or address was on the bus. In this way, cells with the same address in the emitter and receiver chips are virtually connected by streams of spikes. Usually, these AER circuits are built using self-timed asynchronous logic [6]. Several works have implemented spike-based visual processing filters. Serrano et al. [10] presented a chip-processor that is able to implement image convolution filters based on spikes, which works with a very high performance (~3 GOPS for 32x32 kernel size) compared to traditional digital frame-based convolution processors [11]. Another approach for solving frame-based convolutions with very high performances are the ConvNets [12][13], based on cellular neural networks, that are able to achieve a theoretical sustained 4 GOPS for 7x7 kernel sizes.

One of the goals of the AER processing is a large multi-chip and multi-layer hierarchically structured system capable of performing complicated array data processing in real time. But this purpose strongly depends on the availability of robust and efficient AER interfaces and evaluation tools [9]. One such tool is a PCI-AER interface that allows not only reading an AER stream into a computer memory and displaying it on screen in real-time, but also the opposite: from images available in the computer's memory, generate a synthetic AER stream in a similar manner a dedicated VLSI AER emitter chip [8][9][4] would do. This PCI-AER interface is able to reach up to 10Mevents/sec bandwidth, which allows a frame-rate of 2.5 frames/s with an AER traffic load of 100% for 128x128 frames, and 25 frames/s with a typical 10% AER traffic load.

Nowadays computer-based systems are increasing their performances exploiting architectural concepts like multi-core and many-core. Multi-core is referred to those processors that can execute in parallel as many threads as cores are available on hardware. On the other hand, many-core computers consist of processors (that could be multi-core) plus a co-processing hardware composed of several processors, like the Graphical Processing Units (GPU). By joining a many-core system with the mentioned PCI-AER interface, a spike-based processor could be implemented.

In this work we focus on the evaluation of GPU to run in parallel an AER system, starting with the frame-to-AER software conversion and continuing with several usual AER streaming operations, and a first emulation of a silicon retina using CCD cameras. For this task monitoring internal GPU performance through the Compute Visual Profiler [17] for NVIDIA® technology was used. Compute Visual Profiler is a graphical user interface based profiling tool that can be used to measure performance. Through the use of this tool, we have found several key points to achieve maximum performance of AER processing emulation.

Next section briefly explains the pros and cons of Simulation and emulation of Spiking systems, when compared to real implementation. In section 3, a basic description of Nvidia CUDA (Compute Unified Device Architecture) architecture is shown, focusing on the relevant points that affect the performance of our tool. In section 4 the main parts of the proposed emulation/simulation tool are discussed. In section 5 the performance of the operations for different software organization is evaluated, and in section 6 the conclusions are presented.

2 Simulation and Emulation of Spiking Systems

Nowadays, there is a relatively high number of spiking processing systems that use FPGA or another dedicated integrated circuits. Needless to say that, while these implementations are time effective, they present many design difficulties: elevated time and cost of digital synthesis, tricky and cumbersome testing, possibility of wiring bugs (which often are difficult to detect), etc. Support tools are then very helpful and demanded by circuit designers. Usually there are several levels in the field of emulation and simulation tools: from the lower (electrical) level to the most functional one. There is no doubt that none of these tools can cover the whole "simulation spectrum". In this paper we present a tool focused on the functional processing of AER spikes, in order to help in the design and testing of filters and buses management of these systems.

CPU is the most simple and straightforward simulation, and there are already some of these simulators [25][26]. But simulation times are currently far from what one can wish for: a stream processing that lasts microseconds in a real system can extend for minutes on these simulators even when running in high performance clusters.

Another hurdle that is common in CPU emulators is their dependence on input (or traffic) load. In a previous work [14], the performance of several frame-to-AER software conversion methods for real-time video applications was evaluated, by measuring execution times in several processors. That work demonstrated that for low AER traffic loads any method in any multicore single CPU achieved real-time, but for high bandwidth AER traffics, it depends on which method and CPU are selected in order to obtain real-time. This problem can be mitigated when using GPGPUs (General Purpose GPUs).

A few years ago GPU software development was difficult and close to bizarre [23]. During the last years, with the expansion of GPGPUs, tools, function libraries, and hardware abstraction mechanisms that hide the GPU hardware from developers, have appeared. Nowadays there are reliable debugging and profiling tools like those from CUDA [18]. Two additional ones complement these advantages. First, the very low cost per processing unit, which is supposed to persist, since the PC graphics market subsidizes GPUs (only Nvidia has already sold 50 million CUDA-capable GPUs). Secondly, the annual growth performance ratio is predicted to stay very high: by 70% per year due to the continuing miniaturization. The same happens for Memory Bandwidth. For example, extant GPUs with 240 floating point arithmetic cores, realizing a performance of 1 TFLOPS with one chip. One of the consequences of this advantage to its competitors, is that the top leading supercomputer (November 2010) is a GPU based machine (www.top500.org).

Previous reasons have pushed the scientific community to incorporate GPUs in several disciplines. In video processing systems, GPUs application is obvious, therefore some very interesting AER emulation systems have begun to appear [15][16].

The last fact regards the need for spiking output cameras. These cameras (usually called silicon retinas) are currently expensive, rare and inaccurate (due to electrical mismatching between their cells [22]). Moreover, their actual resolution is very low, making it impossible to work with real elevated address streams at present. This hurdle is expected to be overcome in next years. However, current researchers cannot evaluate their spike processing systems for high resolutions. Therefore, a cheap CCD camera where a preprocessing module generates a spiking streaming is a very attractive alternative [20], even if the real silicon retina speed cannot be reached. And we believe GPUs may be the most well-placed platform to do it. Emulating a retina by a GPU has additional benefits: on the one hand, different address coding can be easily implemented [24]. On the other hand, different types of silicon retinas could be emulated over the same GPU platform, simply by changing the preprocessing filter executed before generating the spiking stream. Nowadays implemented retinas are mainly of three types [22]: gray level, spatial contrast (gradient) and dynamic vision sensor (diferential) retinas. The first one can be emulated by generating spikes for each pixel [8], the second option through a gradient and the last one, with a discrete time derivative.

In this work the balance between generality and efficiency of emulation is considered. Tuning simulator routines with a GPU to reach a good performance involves a loss of generality [21].

3 CUDA Architecture

A CUDA GPU [18] includes an array of streaming multiprocessors (SMs). Each SM consists of 8 floating-point Scalar Processors (SPs), a Special Function Unit, a multi-threaded instruction unit, and several pieces of memory. Each memory type is intended for a different use; in most cases they have to be managed by the programmer. This makes the programmer to be aware of their resources, achieving maximum performance. Each SM has a “warp scheduler” that selects at any cycle a group of threads for execution, in a round-robin fashion. A warp is simply a group of (currently) 32 hardware-managed threads. As the number and types of threads may be enormous, a five dimension organization is supported by CUDA, with two important levels: a grid contains blocks that must not be very coupled, while every block contains a relatively short number of threads, which can cooperate deeply.

If a thread in a warp issues a costly operation (like an external memory access), then the warp scheduler switches to a new warp, in order to hide the latency of the other thread. In order to use the GPU resources efficiently, each thread should operate on different scalar data, with a certain pattern. Due to these special CUDA features, some key points must be kept in mind to achieve a good performance of an AER system emulator. These are basically: It is a must to launch thousands or millions of very light threads; the pattern memory access is of vital importance (the CUDA manual [18] provides detailed algorithms to identify types of coalesced/uncoalesced memory accesses); there is a considerable amount of spatial locality in the image access pattern required to perform a convolution (GPU texture memory is used in this case). Any type of bifurcation (branch, loops, etc.) in the thread code should be avoided. The same for any type of thread synchronization, critical sections, barriers, atomic accesses, etc. (this means that each thread must be almost independent from the others); Nevertheless, because GPU do not usually have hardware-managed caches there will be no problem with false dependencies (as usual in multicore systems when every thread has to write in the same vector as the others).

4 Main Modules of the Emulation/Simulation Tool

An AER system emulation/simulation tool must contain at most the following parts:

- Images Input module. It can include a preprocessing filter to emulate different retinas.
- Synthetic AER spikes generation. It is an important part since the distribution of spikes throughout time must be similar to that produced by real retinas.
- Filters. Convolution kernels are the basic operation, but others like low pass filters, integrators, winner takes all, etc. may be necessary.
- Buses management. It includes buses splitters, merges, and so on.
- Result output module. It must collect the results in time order to send them to the CPU.
- AER Bus Probes. This module appears necessarily as discussed below.

The tool presented here is intended to emulate a spiking processing hardware system. As a result, the algorithms to be carried out in the tool are generally simple. In GPGPU terminology, this means that the "arithmetic intensity" [18] (which is defined as the number of operations performed per word of memory transferred) is going to be very low. Therefore, optimisation must focus on memory accesses and types. As a first consequence some restrictions on the number and size of the data objects were done. Besides, a second conclusion is presented: instead of simulating several AER filters in cascade (as usual in FPGA processing), it will be usually better to execute only a combined filter that fuses them, in order to save GPU DDRAM accesses or CPU-GPU transactions. This is to be discussed in next sections, according to the results. Finally the concept of AER Bus Probe is introduced here in order to only generate the intermediate AER values that are strictly necessary. Only when a probe is demanded by the user, a GPU to CPU transaction is inserted to collect AER spikes in the temporal order. Besides, some code adjustments have been introduced to avoid new data structures despite of adding more computation.

Synthetic AER generation is one of the key pieces of an AER tool. This is because it usually lasts a considerable time and because the spike distribution should have a considerable time uniformity to ensure that neuron information is correctly sent [8]. Besides, in this work two additional reasons have to be considered. Firstly, it has to be demonstrated that a high degree of parallelism can be obtained using CUDA, so that the more cores the GPU has, the less time the frame takes to be generated. And secondly, we have performed a comparison of execution times with those obtained for the multicore platforms used previously in [14].

In [14] these AER software methods were evaluated in several CPUs regarding the execution time. In all AER generation methods, results are saved in a shared AER spike vector. Actually, spike representation can be done in several forms (in [24] a wide codification spectrum is discussed). Taking into account the consideration of previous section, we have concluded that the AER spike vector (the one used in [14]) is very convenient when using GPUs.

One can think of many software algorithms to transform a bitmap image (stored in a computer's memory) into an AER stream of pixel addresses [8]. In all of them the frequency of appearance of the address of a given pixel must be proportional to the intensity of that pixel. Note that the precise location of the address pulses is not critical. The pulses can be slightly shifted from their nominal positions; the AER receivers will integrate them to recover the original pixel waveform.

Whatever algorithm is used, it will generate a vector of addresses that will be sent to an AER receiver chip via an AER bus. Let us call this vector the "*frame vector*". The *frame vector* has a fixed number of time slots to be filled with event addresses. The number of time slots depends on the time assigned to a frame (for example *Tframe*=40 ms) and the time required to transmit a single event (for example *Tpulse*=10 ns). If we have an image of $N \times M$ pixels and each pixel can have a grey level value from *0* to *K*, one possibility is to place each pixel address in the *frame vector* as many times as the value of its intensity, and distribute it with equidistant positions. In the worst case (all pixels with maximum value *K*), the *frame vector* would be filled with $N \times M \times K$ addresses. Note that this number should be less than the total number of time slots in the *frame vector*. Depending on the total intensity of the image there will be more or less empty slots in the *frame vector Tframe/Tpulse*.

Each algorithm would implement a particular way of distributing these address events, and will require a certain time. In [8] and [14] we discussed several algorithms that were convenient when using classical CPUs. But if GPUs are to be used, we have to discard those where the generation of each element of *frame vector* cannot be independent from the others. This clearly happens in those methods based on Linear Feedback Shift Registers (LFSR): as the method requires calling a random function that always depends on itself, the method cannot be divided in threads.

To sum up, the best-suited method for GPU processing is the so-called Exhaustive method. This algorithm divides the address event sequence into *K* slices of $N \times M$ positions for a frame of $N \times M$ pixels with a maximum gray level of *K*. For each slice (*k)*, an event of pixel (*i,j*) is sent on time *t* if the following condition is asserted:

$$(k \cdot P_{i,j}) \bmod K + P_{i,j} \geq K \qquad \text{and} \qquad N \cdot M \cdot (k-1) + (i-1) \cdot M + j = t$$

where $P_{i,j}$ is the intensity value of the pixel *(i,j)*.

The Exhaustive method tries distributing the events of each pixel in equidistant slices. In this method, there is a very important advantage when using CUDA: elements of *frame vector* can be sequentially processed, because the second condition above can be implemented using *t* as the counter of the frame vector (that is, the thread index in CUDA terminology). This means that several accesses (performed by different threads) can be coalesced to save DDRAM access time. The results section is based on this algorithm.

Nevertheless, other algorithms could be slightly transformed to adapt them to CUDA. The most favorable case is that of the Random-Square method. While this method requires the generation of pseudorandom numbers (which are generated by two LFSR of 8 and 14 bits), LFSR functions can be avoided if all the lists of numbers are stored in tables. Although this is possible, it will involve two additional accesses per element to long tables, which probably will reside in DDRAM. This will add a supplementary delay that is avoided with the exhaustive method.

The rest of methods are more difficult to fine-tune to CUDA (namely the Uniform, Random and Random-Hardware methods) because of the aforementioned reasons.

There are another group of software methods dedicated to manage the AER buses. Fortunately, these operations are intrinsically parallel, since in our case they basically consist of a processing of each of the frame vector elements (which plays the role of a complete AER stream).

The other operations involved in AER processing are those that play the role of image filtering and convolution kernels. Nowadays, the algorithms that execute them on AER based systems are intrinsically sequential: commonly for every spike that appears in the bus, the values of some counters related to this spike address are changed [15][10]. Therefore, these counters must be seen as critical sections when emulating this process through software. This makes impractical to emulate this operation in a GPU. On the contrary, the standard frame convolution operation can be easily parallelized [19] if the image output is placed in a memory zone different from that of image input. Execution of a standard convolution gives an enormous speedup when comparing to a CPU. Due to this and considering that the other group of operations does present a high degree of parallelism, in this paper convolutions are processed in a classical fashion. Nonetheless, this combination of AER-based and classical operations results in a good enough performance as seen in the following section.

5 Performance Study

In order to analyze the performance and scalability of CUDA simulation and emulation of AER systems, a series of operations have been coded and analyzed in two Nvidia GPUs. **Table 1** summarizes the main characteristics of the platforms tested. The second GPU have an important feature: it can concurrently copy and execute programs (while the first one cannot).

Table 1. Tested Nvidia GPUs

Characteristics	GeForce 9300 ION	GTX 285
Global memory	266010624 bytes	1073414144 bytes
Maximum number of threads per block	512 threads	512 threads
Multiprocessors x Cores/MP	2 x 8 = 16 Cores	30 x 8 = 240 Cores
Clock rate	1.10 GHz	1.48 GHz

Fig. 1 depicted a typical AER processing scenario. The first module ‘ImageToAER’ transforms a frame into an AER stream using the Exhaustive method. The ‘Splitter’ divides spikes into two buses according to an address mask that represents the little clear square in the almost black figure. The upper bus is then rotated 90 degrees simply by going through the ‘Mapper’ module, which changes each spike’s address into another. Finally a merge between the original image and the upper bus gives a new emulated AER bus, which can be observed with a convenient AERToImage module (which, in a few words, makes a temporal integration). A consequence of the use of a rigid size frame vector composed of time slots is that a merge operation between two full buses cannot fuse perfectly the two images represented in the initial buses. In our implementation, if both buses have a valid spike in a certain time slot, the corresponding output bus slot is going to be filled only by one of the input buses (which is decided with a simple circuit). This aspect appears also in hardware AER implementations, where an arbiter must decide which input bus "looses", and then its spike does not appear in the merged bus [8][10].

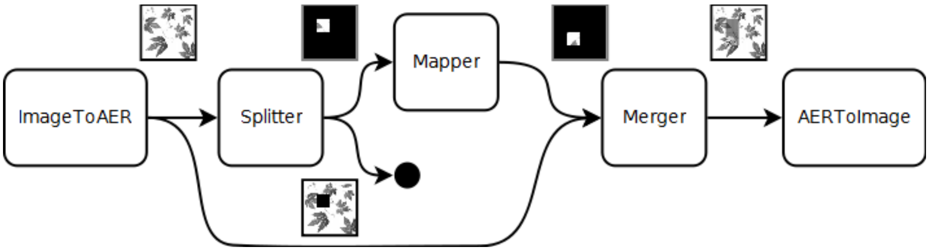

Fig. 1. Cascade AER operations benchmark

According to section 4, three kinds of benchmarks have been carried out. In a first group, an AER cascade operations were simulated: in the middle of two operations, (that is, in an AER bus) an intermediate result is collected by the CPU to check and verify the bus values. In the second group, operations are executed in cascade, but transitional values are not "downloaded" to the CPU, thus preserving GPU to CPU transactions. This implies that no AER Bus Probe modules are present, thus inhibiting inspection opportunities. Finally a third collection is not coded in previous modular fashion since the first four operations are grouped together in one CUDA kernel (the same thread executes all of them sequentially). This avoids several GPU DDRAM accesses, saving an enormous execution time in the end. The fifth operation that transforms an AER stream into an image frame cannot be easily parallelized for the same reasons described in previous section for convolution kernels. Timing comparison for these three groups is summarized in Table 2.

It is important to remark that the execution time for the third group almost coincides with the maximum execution time of all the operations in the first group. Another obvious but interesting fact is that transactional times are proportionally reduced when CPU-GPU transactions are eliminated. And speed-up between GTX285 and ION 9300 is also near to the ideal. One can conclude that scalability for our tool is good, which means for faster upcoming GPU a shorter execution time is expected.

Finally, contrast retina emulation has been carried out: for a frame, first a gradient convolution is done in order to extract image edges, and secondly, the AER frame vector is generated (in the same CUDA thread). A hopeful result is obtained: the mean execution time to process one frame is 313.3 μs, that is, almost 3200 frames per second. As the execution times are small we can suppose that these times could be overlapped with the transaction ones. The resulting fps ratio is very much higher (around 50x) than those obtained using multicore CPUs in previous studies of AER spikes generation methods [14].

Table 2. Benchmarking times for GPU GTX285 and for 9300 ION. Times in microseconds.

Measured Time	First Group GTX285	Second Group GTX285	Third Group GTX285	Third Group 9300
CPU to GPU transactions	8649.2	360.3	125.5	5815.5
CUDA kernel execution	2068.6	1982.7	781.6	15336.8
GPU to CPU transactions	11762.8	2508.9	2239.2	12740.8
Total Time	22480.5	4851.9	3146.3	33893.0

Through these experiments we have demonstrated that major time is spent in these types of AER tools in external DDRAM GPU accesses, since data sizes are necessarily big while algorithms can be implemented with a few operations per CUDA thread. This conclusion gives us an opportunity to develop a completely functional AER simulator. A second consequence derived from this is that the size of the image can introduce a considerable increment of time emulation. In this work, the chosen size (128×128 pixels) results in a frame vector of 8 MB (4 Mevents $\times 2$ bytes/event). However, a 512×512 pixel image will have 4x the size image, plus twice the bytes per address (if no compression is implemented). This means an 8x total size, and then an 8x access time. To sum up, eliminating the restrictions on the number and size of data objects can have an important impact on the tool performance.

6 Conclusions and Future Work

A CUDA-based tool focused on the functional processing of AER spikes and its first timing results are presented. It intends to emulate a spiking processing hardware system, using simple algorithms with a high level of parallelism. Through experiments, we demonstrated that major time is spent in DDRAM GPU accesses, so some restrictions on the number and size of the data objects have been done. A second result is presented: instead of simulating several AER filters in cascade (as usual in FPGA processing), it is better to execute only a combined filter that fuses them, in order to save GPU DDRAM accesses and CPU-GPU transactions. Due to the promising timing results, the immediate future work comprises a fully emulation of an AER retina using a classical video camera. Running our experiments on a multiGPU platform is another demanding extension because of the scalability of our tool.

Acknowledgments. This work was supported by the Spanish Science and Education Ministry Research Projects TEC2009-10639-C04-02 (VULCANO).

References

1. Drubach, D.: The Brain Explained. Prentice-Hall, New Jersey (2000)
2. Lee, J.: A Simple Speckle Smoothing Algorithm for Synthetic Aperture Radar Images. IEEE Trans. Systems, Man and Cybernetics SMC-13, 85–89 (1983)
3. Crimmins, T.: Geometric Filter for Speckle Reduction. Applied Optics 24 (1985)
4. Linares-Barranco, A., et al.: On the AER Convolution Processors for FPGA. In: ISCAS 2010, Paris, France (2010)

5. Sivilotti, M.: Wiring Considerations in analog VLSI Systems with Application to Field-Programmable Networks, Ph.D. Thesis, California Institute of Technology (1991)
6. Boahen, K.A.: Communicating Neuronal Ensembles between Neuromorphic Chips. In: Neuromorphic Systems. Kluwer Academic Publishers, Boston (1998)
7. Mahowald, M.: VLSI Analogs of Neuronal Visual Processing: A Synthesis of Form and Function. Ph.D. Thesis. California Institute of Technology Pasadena, California (1992)
8. Linares-Barranco, A., Jimenez-Moreno, G., Civit-Ballcels, A., Linares-Barranco, B.: On Algorithmic Rate-Coded AER Generation. IEEE Transaction on Neural Networks (2006)
9. Paz, R., Gomez-Rodriguez, F., Rodríguez, M.A., Linares-Barranco, A., Jimenez, G., Civit, A.: Test Infrastructure for Address-Event-Representation Communications. In: Cabestany, J., Prieto, A.G., Sandoval, F. (eds.) IWANN 2005. LNCS, vol. 3512, pp. 518–526. Springer, Heidelberg (2005)
10. Serrano, et al.: A Neuromorphic Cortical-Layer Microchip for Spike-Based Event Processing Vission Systems. IEEE Trans. on Circuits and Systems Part 1 53(12), 2548–2566 (2006)
11. Cope, B., Cheung, P.Y.K., Luk, W., Witt, S.: Have GPUs made FPGAs redundant in the field of video processing? In: IEEE International Conference on Field-Programmable Technology, pp. 111–118 (2005)
12. Farabet, C., et al.: CNP: An FPGA-based Processor for Convolutional Networks. In: International Conference on Field Programmable Logic and Applications (2009)
13. Farriga, N., et al.: Design of a Real-Time Face Detection Parallel Architecture Using High-Level Synthesis. EURASIP Journal on Embedded Systems (2008)
14. Domínguez-Morales, M., et al.: Performance study of synthetic AER generation on CPUs for Real-Time Video based on Spikes. In: SPECTS 2009, Istambul, Turkey (2009)
15. Nageswaran, J.M., Dutt, N., Wang, Y., Delbrueck, T.: Computing spike-based convolutions on GPUs. In: IEEE International Symposium on Circuits and Systems (ISCAS 2009), Taipei, Taiwan, pp. 1917–1920 (2009)
16. Goodman, D.: Code Generation: A Strategy for Neural Network Simulators. Neuroinformatics 8.3, 183–196 (2010), Issn: 1539-2791
17. Compute Visual Profiler User Guide, `http://developer.nvidia.com/`
18. NVIDIA CUDA Programming Guide, Version 2.1, `http://developer.nvidia.com/`
19. NVIDIA Corporation. CUDA SDK Software development kit, `http://developer.nvidia.com/`
20. Paz-Vicente, R., et al.: Synthetic retina for AER systems development. In: IEEE/ACS International Conference on Computer Systems and Applications, AICCSA, pp. 907–912 (2009)
21. Owens, J.D., Houston, M., Luebke, D., Green, S., Stone, J.E., Phillips, J.C.: GPU Computing. Proceedings of the IEEE 96(5) (May 2008)
22. Indiveri, G., et al.: Neuromorphic Silicon Neurons. Frontiers in Neuromorphic Engineering 5, 7 (2011)
23. Halfhill, T.R.: Parallel Processing With CUDA. Microprocessor The Insider's Guide To Microprocessor Hardware (2008)
24. Thorpe, S., et al.: Spike-based strategies for rapid processing. Neural Networks 14(6-7), 715–725 (2001)
25. Pérez-Carrasco, J.-A., Serrano-Gotarredona, C., Acha-Piñero, B., Serrano-Gotarredona, T., Linares-Barranco, B.: Advanced Vision Processing Systems: Spike-Based Simulation and Processing. In: Blanc-Talon, J., Philips, W., Popescu, D., Scheunders, P. (eds.) ACIVS 2009. LNCS, vol. 5807, pp. 640–651. Springer, Heidelberg (2009)
26. Montero-Gonzalez, R.J., Morgado-Estevez, A., Linares-Barranco, A., Linares-Barranco, B., Perez-Peña, F., Perez-Carrasco, J.A., Jimenez-Fernandez, A.: Performance Study of Software AER-Based Convolutions on a Parallel Supercomputer. In: Cabestany, J., Rojas, I., Joya, G. (eds.) IWANN 2011, Part I. LNCS, vol. 6691, pp. 141–148. Springer, Heidelberg (2011)