Biologically Sound Neural Networks for Embedded Systems Using OpenCL

István Fehérvári[1], Anita Sobe[2] and Wilfried Elmenreich[1,3]

[1] Mobile Systems Group, Lakeside Labs, Institute for Networked and Embedded Systems, Alpen-Adria-Universität Klagenfurt, Austria
{istvan.fehervari,wilfried.elmenreich}@aau.at
[2] Computer Science Department, University of Neuchâtel, Switzerland
anita.sobe@unine.ch
[3] Complex Systems Engineering, University of Passau, Germany
wilfried.elmenreich@uni-passau.de

Abstract In this paper, we present an OpenCL implementation of a biologically sound spiking neural network with two goals in mind: First, applied neural dynamics should be accurate enough for bio-inspired training methods, thus resultant network data is reproducible in "*in vitro*" experiments. The second is that the implementation produces code that runs adequately on up-to-date embedded graphical chips for fast on-board classification applications, e.g., video image processing. We describe the necessary steps required to implement an efficient algorithm using the OpenCL framework and present evaluation results of the execution time compared to traditional serial CPU code. We show that an optimized GPU kernel code can perform sufficiently fast to be used for future embedded neural processing.

1 Introduction

Artificial neural networks (ANNs) are general function approximators and are noise resistant, and are therefore popular in many classification applications. Spiking neural networks (SNNs) are a type of ANN where communication between neurons occurs by means of time-stamped events (spikes). Researchers in the field of computational intelligence have shown that biologically sound spiking neural networks (SNNs) are comparable, but more powerful than traditional artificial neural networks [1], [2]. SNNs have been applied to many areas like epilepsy and seizure detection [3], robot navigation [4] and image processing [5]. Such neural networks are usually computationally complex and often require high performance computers (or even supercomputers) to run. They are, however, inherently parallel processes and therefore implementations on cheap and easy available GPUs are advantageous. In [6], the authors showed the feasibility of running different simple spiking neural network models on GPUs, but concentrated on the architecture comparison. The authors of [7] implemented a SNN using CUDA [8] and showed its real-time capabilities. However, this implementation and others (e.g., [9], [10], [11]) rely on the most basic integrate-and-fire SNN models or their variations. The model is easy to implement, however it is neither accurate nor biologically-sound.

The goal of applying more complex models of neural dynamics is to provide a testbed that is suitable for evaluating advanced bio-inspired training algorithms that can also be verified by *in vitro* experiments. Such a neural testbed running on novel embedded graphical chips could open up new possibilities, not only for researchers with limited access to high-speed computer clusters, but also for the development of smart devices performing real-time classification tasks.

In this paper, we present the first steps of the implementation of a very large, biologically plausible, SNN, based on the spike response model (SRM)[12], which is accurate and trainable. Our implementations rely on OpenCL for embedded chips that support OpenCL specifications [13], which allows for portability among a wide range of GPU types from different vendors. Results show that an OpenCL application running on a GPU can easily outperform single and multi-core CPUs. One challenge is that the biologically sound SNN models are memory-intense and GPUs provide a limited memory bandwidth, which is usually circumvented by simplified implementations [14]. We are facing this challenge by introducing a local connection scheme with a constant number of synapses per neuron. This fact can also be exploited to optimize the memory management of the OpenCL application, which resulted in a further significant increase in speed.

The paper is structured as follows: Section 2 describes the spiking neural model we used in our simulations, while Section 3 gives a brief introduction on general-purpose computing on GPUs with OpenCL. Afterwards, Section 4 presents the particular steps we took to implement and optimize the algorithm to achieve best performance, which is explored in Section 5. Finally, we conclude the paper in Section 6.

2 Spiking Neural Network Model

Spiking neural networks offer various advantages over traditional sigmoidal artificial neural networks. They are not only biologically plausible but they can potentially reproduce the computational dynamics observed in a biological brain. SNNs are designed to be phenomenological models of biological neurons; reflecting natural action-potential generation, post-synaptic potential shaping, and refractory periods.

In the case of ANNs information processing happens in a much simpler manner: an iterative series of calculations based on neuron outputs and layers. In an SNN, neurons emit pulses or *spikes* through their synapses whenever their membrane potential ϑ_i reaches its threshold value. This happens due to incoming spikes from other presynaptic neurons. Each spike has an amplitude of 100 mV and lasts approximately 1-2 ms. Since all the spikes in the network have the exact same form, information is encoded in the chronological order of the spikes, or the so-called *spike train*. In our model, spikes are represented by the Dirac delta function, $\delta(t - t_i^k)$, which is a singularity function occurring at time $t = t_i^k$, where k is the spike index in the train and i is the neuron index.

2.1 Spike Response Model

Related work indicates that there are numerous ways to model a spiking neural network, but most of them are overly simple, neglecting several aspects of neural dynamics (bursting, inhibitory rebound and shunting inhibition [15]). The model presented in this paper is based on the Spike Response Model (SRM) model described in [12]. Both models provide networks that are mathematically tractable and trainable. Moreover, they both approximate the Hogkin-Huxley model with a high degree of accuracy [16].

The major difference between earlier models and the SRM is that neurons are not binary: they keep track of past spikes as exponentially decaying functions. Although the SRM is also considered a generalization of the leaky integrate-and-fire model [16] it expresses the membrane potential u_i of neuron i with the following integral instead of a differential equation:

$$\begin{aligned} u_i(t) &= \eta(t-\hat{t_i}) + \sum_j w_{ij} \sum_k \epsilon_{ij}(t-\hat{t_i}, t-t_k^k) \\ &+ \int_0^\infty \lambda(t-\hat{t_i},\tau) I(t-\tau) d\tau \end{aligned} \tag{1}$$

where $t-\hat{t_i}$ represents the time that has passed since last firing of neuron i, while the $\eta()$, $\epsilon()$ and $\lambda()$ functions describe the dynamic behavior of this neuron. w_{ij} is the weight of the synapse between neuron i and j. The function $I()$ determines the resting potential expressed by the integral (1) and is set to a small random signal.

The *refractory response* function, $\eta()$, describes the positive pulse and afterpotential subsequent to each firing of a neuron. This is intended to model the hyperpolarization phase and is expressed as an exponential decay function:

$$\eta(x) = -\eta_0 e^{-x/\tau_f} H(x) \tag{2}$$

where η_0 and τ_f are scaling constants and $H(x)$ is the Heaviside function.

The *postsynaptic potential* function, $\epsilon()$, formulates the change of membrane potential evoked by the reception of a pulse from a presynaptic neuron. It is obtained by mapping integrate-and-fire neurons to the SRM and modeled with exponential decay:

$$\epsilon(x,y) = \frac{e^{-t/\tau_s}}{\tau_s} \int_0^x e^{-\tau(1/\tau_m - 1/\tau_s)} H(t-\tau) d\tau \tag{3}$$

where τ_m and τ_s are scaling constants of the exponential function.

The *linear response* function $\lambda()$ captures the change of the membrane potential to an impulse current input, which decays exponentially after the firing of a neuron. It is also expressed as a decaying exponential:

$$\lambda(x,\tau) = \frac{R}{\tau_m}[1-e^{-x/\tau_r}] e^{-\tau/\tau_m} H(\tau) H(x-\tau) \tag{4}$$

where τ_m, $\tau_r >> \tau_f$ are the recovery time constants and R is the recovery factor that scales the magnitude of the postsynaptic potential in accordance to the amount of time elapsed since the neuron's last firing event.

In traditional neural networks and also in the integrate-and-fire models, each neuron has the same constant threshold value. However, in the SRM each neuron is characterized by its own dynamic threshold value ϑ_i. Whenever a neuron enters the refractory period, T_f the normal threshold jumps to approximately 100mV, preventing the neuron firing again. After T_f the threshold quickly decreases back to its normal value (eq. 5).

$$\vartheta(t-\hat{t_i}) = \begin{cases} \vartheta_f & 0 < t-\hat{t_i} \\ \vartheta_0[1-e^{-(t-\hat{t_i}-T_f)/\tau_\vartheta}] & t-\hat{t_i} \geq T_f \end{cases} \tag{5}$$

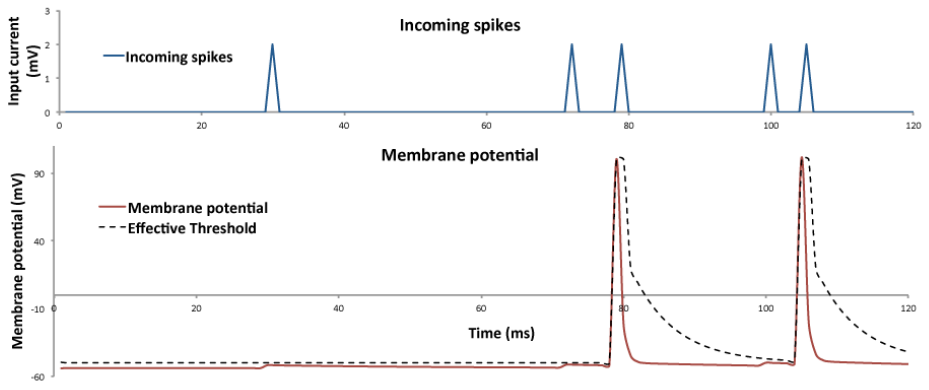

Figure 1: Changes to the membrane potential and threshold due to incoming spikes

Figure 1 illustrates the changes to the neuron's membrane potential and threshold due to incoming pulses. Individual pulses increase the membrane potential for a short time period. Once the threshold has been reached, the neuron fires by emitting a near instantaneous pulse and enters a refractory period. This effect also increases the threshold to prevent the neuron from firing in the next short time period.

3 OpenCL Architecture

GPUs were originally built to enhance graphical game experience, but were soon adopted by researchers to exploit their performance for parallel tasks in other domains. OpenCL is a framework developed by the Khronos Group[4] for writing programs that can be executed across heterogeneous platforms, including CPUs, GPUs or dedicated hardware accelerators. In particular, we have chosen OpenCL because of its support for different operating systems and graphics card vendors. OpenCL applications can be also executed hardware without GPU support. It is further expected to be possible to use OpenCL on a wide range existing and future embedded devices such as smartphones, tablets, and embedded control systems.

In comparison with CUDA, which is more directly connected to the execution platform (which needs to support Nvidia's PTX instruction set to run CUDA), OpenCL shows comparable performance given that the same optimizations are implemented as in the CUDA implementation [17,18]. While CUDA is limited to Nvidia GPUs, it is notable that a CUDA kernel can be converted to an OpenCL kernel with minimal modifications [17].

OpenCL includes a language that is based on the C99 standard [19] and a set of application programming interfaces for writing individual processing threads (kernels) especially meant for data-parallel and task-parallel applications. Tasks are executed on a host that forwards the application to the execution devices (i.e., CPU or GPU). It divides the program into so called work-items that will be executed in parallel. If the number

[4] http://khronos.org/opencl

of work-items is higher than the processing capability of the platform, then they will be grouped into several work-groups, where synchronization between them will take place during runtime.

An important part of OpenCL is the memory model as shown in Figure 2. It adopts the device's model, where the global memory refers to the main memory of the device but it can also be shared with the host. The global memory is the largest but slowest memory, especially if shared with the host system. Work-items belonging to the same work-group can access a shared local memory that provides much faster read/write operations. This memory is mainly used to synchronize work-items in a group and has a maximum size of 16kB. Finally, each of the work-items has a private memory space, which is a register-type memory. Simply put, the local memory is faster than the global memory, and the private memory is faster than the local memory. Although implementing a parallel algorithm is in general not complicated with OpenCL, understanding the memory model could help in improving the execution time of the program as shown in the next section.

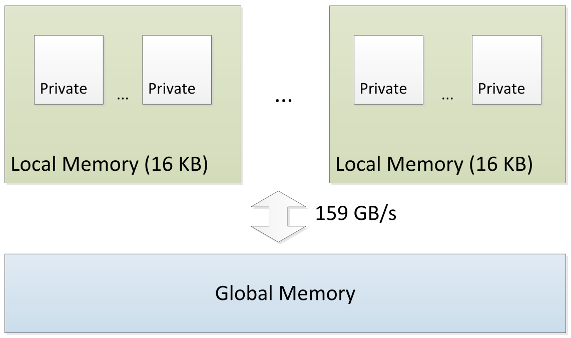

Figure 2: OpenCL memory model

4 SNN Implementation with OpenCL

Related implementations of spiking neural networks meant for parallel executions are mostly event-based simulations [10], however, the integrating nature of the spike response model requires a simulation of discrete time steps, thus we choose a step length of 1 ms. In order to calculate the membrane potential of a neuron at any time step we need to keep track of the time steps elapsed since its last firing, its threshold and the weights of all connecting synapses. Moreover, the membrane potential of every neuron also has to be stored for analytical purposes. The time steps were stored as 32-bit integer values, while the membrane potential, the threshold and the weights were defined as 32-bit floating point values.

For best performance, it is advised to store all state variables and weights directly on the OpenCL device for fast read/write access. Therefore, before starting the simulation the initial values of these variables and the weights must be allocated, initialized and transferred to the device's global memory which can be then accessed in runtime. As explained in Section 3, the device's global memory has a high access time and a very limited size.

A fully-connected neural network requires to store one floating point value for every $n(n-1)/2$ connections, which is impracticable for larger network sizes in terms of memory usage, training and execution times. To make the neural network scalable for high numbers of nodes, we will use a partially-connected neural network where each neuron has a fixed number of connections to its immediate neighbors (e.g., for a neighborhood radius of 5 this yields 124 connections per neuron). This way, the memory requirements grow only linearly with the number neurons (instead of quadratically).

Initially, we created two different kernel programs: An *update kernel* and a *threshold kernel*. The first is used to update the neuron's membrane potential according to Equation 1, while the second handles the firing if the membrane potential is higher than the neuron's threshold. Thus, one time step of the simulation consists of the execution of the first kernel on each neuron, followed by the second kernel executed on every neuron. The reason for this divided architecture is to provide a general execution model that is not influenced by the connection topology of the neural network.

5 Simulation and Experiences

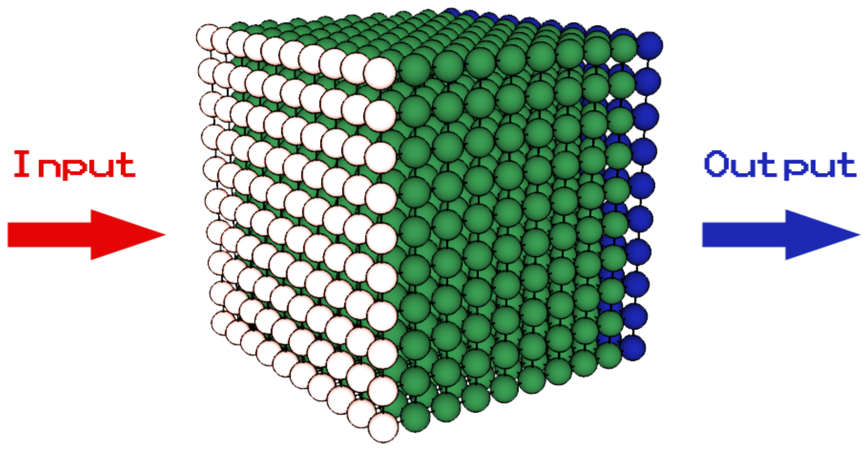

Figure 3: Example network structure with $N \times N \times N$ neurons

With an example application idea in mind to process sequences of images we created an $N \times N \times N$ grid network structure where each neuron is only connected to the local Moore neighborhood of range 2, thus having maximum $5^3 - 1 = 124$ synapses based on how far it is situated from the edges. The computational effort depends mainly on the total number of neurons and the connections per neuron. Our approach is not limited to cubic dimensions. Depending on the type of application it is also possible to have a larger image with fewer layers (for example 640 pixels x 480 pixels x 3 layers) with the same performance regarding frames per second. Neurons on the frontal face of the grid were used as inputs and selected neurons on the backface were used as outputs (see Figure 3). For the sample input data we used 8-bit grayscale images obtained for the built-in camera in order to have the source already on the graphics card's memory in an OpenGL context that can be directly accessed by OpenCL kernels.

For the evaluation of the presented approach, we tested two different algorithms with a structure size being between $N = 10...100$ by measuring the execution time required to perform one step of simulation. All of our experiments were performed using an AMD ATI Mobility Radeon HD 5750 graphics card that is in performance comparable to an up-to-date embedded GPU with OpenCL support.

5.1 Initial naive algorithm

Based on the implementation described above, we created a naive implementation with the state variables and the two different kernels programmed in OpenCL. For comparison, we also implemented a native C++ version of the same code that was executed on a single thread.

We executed the program 100 times with each setup and measured the time required to compute one time step of the simulation. As shown in Figure 5, the execution time for the CPU code increased rapidly with size of the network.

We compared the performance of the OpenCL implementation to the same application running on a single core CPU. This served as a measure to tell us how many times faster the approach is than the single CPU solution. In practice, an implementation on a state-of-the-art microprocessor might also provide parallel processing via multiple cores. However, for an embedded system, the parallelization would be still limited to a reatively low number of cores. In the best case, such a multi-core system would be faster by a factor equal to the number of cores.

We measured a significant speed performance improvement of a factor of at least 50 over a single-core CPU for all input sets with more than $40 \times 40 \times 40$ neurons. Realistic input sets are expected to have at least this size or be even larger. Thus, in comparison to CPU code, our OpenCL implementation performed exceptionally fast, even with one million neurons one step took only 126 ms on average.

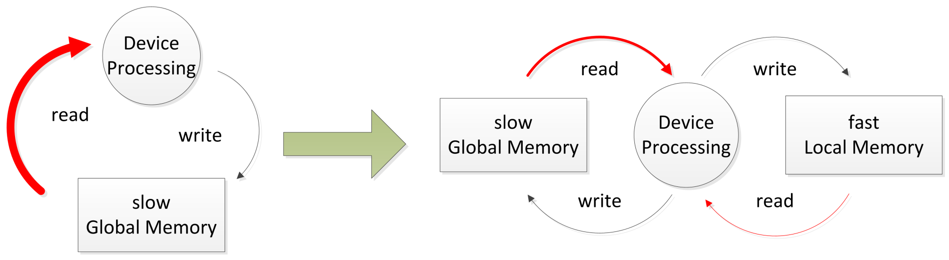

Figure 4: Data-flow model of the first algorithm

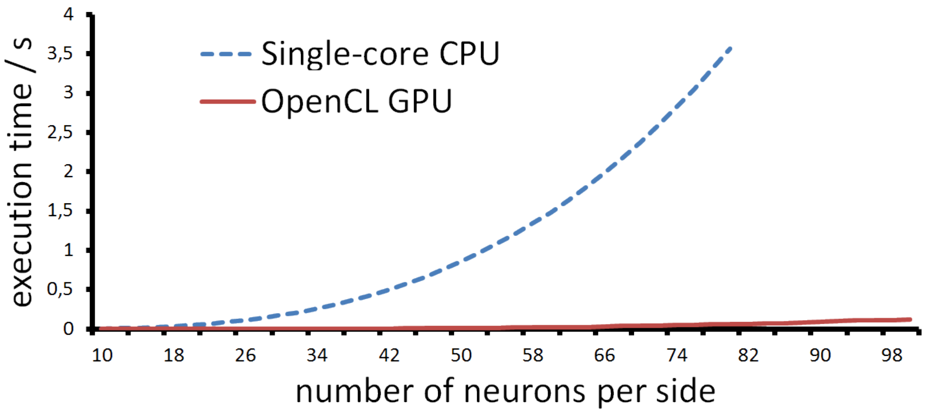

Figure 5: Execution time of one time step of serial CPU and OpenCL implementation with different network size

5.2 Optimized algorithm

If we analyze the data flow of the previous algorithm, we observe that in each time step both kernels were required to read and write data directly from the global memory. In general, this operation is time-wise the most expensive. A more effective approach would be to utilize the local memory block available for kernels being in the same work-group that provides significantly faster read/write access.
As explained in Section 3, OpenCL instructs the target device to execute the parallel work items in groups, which in our case form a $K \times K \times K$ grid layout. By exploiting the fact, that neurons are only connected in a local neighborhood, the program of each neuron could share its own data, which is already loaded from the global memory, via the shared local memory. By making this transition from exclusive read/write on global data to a locally shared architecture we could theoretically improve the performance of the algorithm (see Figure 4).
Furthermore, since the membrane potential of each neuron only depends on locally connected neurons there is no need to wait a whole execution cycle of the *update kernel* before running the *threshold kernel*. Instead the two kernels can be merged saving additional time on memory access. By running the same simulation as written above, we obtained lower execution times (depicted in Figure 6). We achieved an increase of speed of approximately 25 %, which further increases with increased network size. Our results also compare to similar implementations of spiking neural networks with different neural dynamics [10].

6 Conclusions and Future Work

In this paper, we presented an OpenCL implementation of a biologically plausible spiking neural network. We based our neural dynamics on the spike response model that is evaluated in a discrete-time simulation. We compared our naive OpenCL implementation to a single-thread CPU code, as well as to an optimized version of the same code.

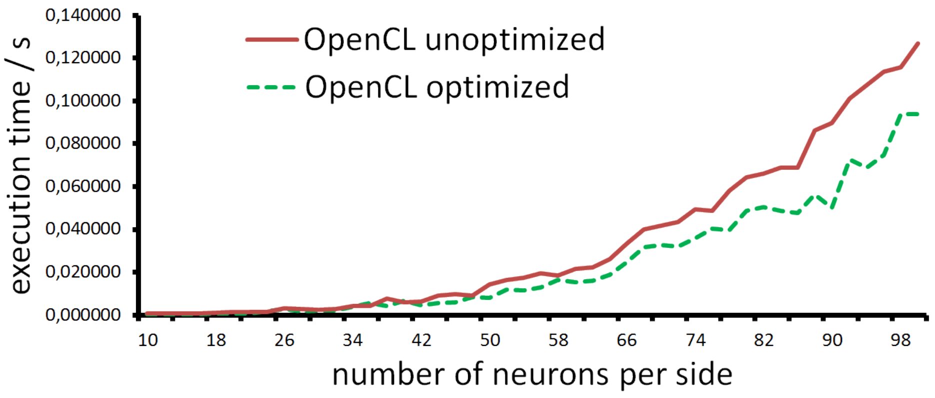

Figure 6: Execution time of one time step of the naive and the optimized code with different network size

Our results show that OpenCL provides much faster execution time, thus being a valid platform for future embedded on-board neural processing.

In our test scenario, a network with 1000 000 neurons and approximately 124 000 000 synapses could be calculated within 93 ms with GPU support. This is sufficient for typical low-framerate image processing scenarios. In comparison, a CPU-based implementation was slower by several orders of magnitude.

In our future work, we plan to test and validate bio-inspired training algorithms (e.g., Hebbian learning) directly on the embedded hardware. Furthermore, the methodology will be applied and verified in video image processing (e.g., activity recognition) in smart cameras.

Acknowledgment

We thank Lizzie Dawes and John NA Brown for proofreading and constructive comments. This work was supported by the Austrian Science Fund grant FFG 2305537, Lakeside Labs via funding from the EU Regional Development Fund and the KWF under grant KWF 20214|21532|32604, and funding from the ParaDIME Project (FP7/2007-2013 grant 318693).

References

1. W. Maass, “Noisy spiking neurons with temporal coding have more computational power than sigmoidal neurons,” *Advances in Neural Information Processing Systems*, vol. 9, 1997.
2. S. Ghosh-Dastidar and H. Adeli, “Spiking neural networks,” *International Journal of Neural Systems*, vol. 19, no. 4, pp. 295–308, 2009.
3. ——, “A new supervised learning algorithm for multiple spiking neural networks with application in epilepsy and seizure detection,” *Neural Networks*, vol. 22, no. 10, pp. 1419–1431, 2009.
4. E. Nichols, L. McDaid, and N. Siddique, “Case study on a self-organizing spiking neural network for robot navigation,” *International Journal of Neural Systems*, vol. 20, no. 06, pp. 501–508, 2010.

5. W. Zou, Z. Chi, and K. Lo, “Improvement of image classification using wavelet coefficients with structured-based neural network,” *International Journal of Neural Systems*, vol. 18, no. 03, pp. 195–205, 2008.
6. V. Pallipuram, M. Bhuiyan, and M. Smith, “Evaluation of GPU architectures using spiking neural networks,” in *2011 Symposium on Application Accelerators in High-Performance Computing (SAAHPC)*, 2011, pp. 93 –102.
7. D. Yudanov, M. Shaaban, R. Melton, and L. Reznik, “GPU-based simulation of spiking neural networks with real-time performance amp; high accuracy,” in *International Joint Conference on Neural Networks (IJCNN)*, 2010, pp. 1 –8.
8. D. Kirk, “NVIDIA CUDA software and GPU parallel computing architecture,” in *International Symposium on Memory Management: Proceedings of the 6 th international symposium on Memory management*, vol. 21, no. 22, 2007, pp. 103–104.
9. J. Nageswaran, N. Dutt, Y. Wang, and T. Delbrueck, “Computing spike-based convolutions on GPUs,” in *IEEE International Symposium on Circuits and Systems (ISCAS 2009)*. IEEE, 2009, pp. 1917–1920.
10. D. Yudanov and R. L., “Scalable multi-precision simulation of spiking neural networks,” in *2012 IEEE World Congress on Computational Intelligence (WCCI 2012)*, 2012.
11. F. Bernhard and R. Keriven, “Spiking neurons on GPUs,” *Computational Science–ICCS 2006*, pp. 236–243, 2006.
12. S. Ferrari, B. Mehta, G. Di Muro, A. VanDongen, and C. Henriquez, “Biologically realizable reward-modulated hebbian training for spiking neural networks,” in *IEEE International Joint Conference on Neural Networks, 2008. IJCNN 2008. (IEEE World Congress on Computational Intelligence)*, 2008, pp. 1780 –1786.
13. “AMD Radeon E6760 embedded GPU specification sheet,” 2012. [Online]. Available: http://www.amd.com/us/Documents/AMD-Radeon-E6760-Discrete-GPU-product-brief.pdf
14. J. Nageswaran, N. Dutt, J. Krichmar, A. Nicolau, and A. Veidenbaum, “A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors,” *Neural Networks*, vol. 22, no. 5-6, pp. 791–800, 2009.
15. W. Maass, “Networks of spiking neurons: The third generation of neural network models,” *Neural Networks*, vol. 10, no. 9, pp. 1659 – 1671, 1997.
16. W. Gerstner and W. M. Kistler, *Spiking Neuron Models: Single Neurons, Populations, Plasticity*, 1st ed. Cambridge University Press, 2002.
17. K. Karimi, N. G. Dickson, and F. Hamze, “A performance comparison of CUDA and OpenCL,” *CoRR*, vol. abs/1005.2581, 2010. [Online]. Available: http://arxiv.org/abs/1005.2581
18. J. Fang, A. L. Varbanescu, and H. Sips, “A comprehensive performance comparison of CUDA and OpenCL,” in *40th International Conference on Parallel Processing (ICPP’11)*, 2011, pp. 216–225.
19. JTC 1/SC 22/WG 14, “ISO/IEC 9899:1999: Programming languages – C,” International Organization for Standards, Tech. Rep., 1999.