Systematic configuration and automatic tuning of neuromorphic systems

Sadique Sheik, Fabio Stefanini, Emre Neftci, Elisabetta Chicca, Giacomo Indiveri
Institute of Neuroinformatics
University of Zurich and ETH Zurich, Switzerland
Email: sadique@ini.phys.ethz.ch

***Abstract*— In the past recent years several research groups have proposed neuromorphic Very Large Scale Integration (VLSI) devices that implement event-based sensors or biophysically realistic networks of spiking neurons. It has been argued that these devices can be used to build event-based systems, for solving real-world applications in real-time, with efficiencies and robustness that cannot be achieved with conventional computing technologies.**

In order to implement complex event-based neuromorphic systems it is necessary to interface the neuromorphic VLSI sensors and devices among each other, to robotic platforms, and to workstations (*e.g.* for data-logging and analysis). This apparently simple goal requires painstaking work that spans multiple levels of complexity and disciplines: from the custom layout of microelectronic circuits and asynchronous printed circuit boards, to the development of object oriented classes and methods in software; from electrical engineering and physics for analog/digital circuit design to neuroscience and computer science for neural computation and spike-based learning methods.

Within this context, we present a framework we developed to simplify the configuration of multi-chip neuromorphic VLSI systems, and automate the mapping of neural network model parameters to neuromorphic circuit bias values.

I. Introduction

While computational neuroscience models simulate neurons and synapses using parameters directly related to their biological characteristics (such as leak conductance, time constants, *etc.*), neuromorphic VLSI systems emulate them using circuits that can be configured by setting bias voltages and currents. The biases in these circuits are often only indirectly related to the parameters of computational neuroscience models. More generally, the relationship between parameters in theoretical models, software simulations, and hardware emulations of spiking neural networks is highly non-linear, and no systematic methodology exists for establishing it automatically.

Current automated methods for mapping the VLSI circuits bias voltages to neural network type parameters are based on heuristics and result in ad-hoc custom made calibration routines. For example, in [1] the authors perform an exhaustive search of the parameter space to calibrate their hardware neural networks, using the simulator-independent description language "PyNN" [2]. This type of brute-force approach is possible because of the accelerated nature of the hardware used, but it becomes intractable for real-time hardware or for very large systems, due to the massive amount of data that must be measured and analyzed to carry out the calibration procedure. An alternative model-based approach is proposed in [3], where the authors fit data from experimental measurements with equations from transistors, circuit models, and computational models to map the bias voltages of VLSI spiking neuron circuits to the parameters of the corresponding software neural network. This approach does not require the extensive parameter space search techniques, but new models and mappings need to be formulated every time a new circuit or chip is used, making it's application quite laborious.

In this article, we propose a systematic and modular framework for the tuning of parameters on multi-chip neuromorphic systems that combines and extends the approaches described above. On the one hand the modularity of the framework allows the definition of a wide range of generic (network, neural, synapse, circuit) models that can be used in the parameter translation routines; on the other hand, the framework does not require detailed knowledge of the hardware/circuit properties, and can optimize the search and evaluate the effectiveness of the parameter translations by measuring experimentally the behavior of the hardware neural network. We implemented this framework using the Python programming language, and making strong use of its object-oriented features. Indeed, Python's recent popularity in the neuroscience community [4], combined with its platform independence and the ease of extending it with other programming languages, makes it the natural choice for this framework.

The framework consists of two software modules: pyNCS and pyTune (see Fig. 1). The pyNCS tool-set allows the user to interface the hardware to a workstation, to access and modify the VLSI chip bias settings, and to define the functional circuit blocks of the hardware system as abstract software modules. The abstracted components represent computational neuroscience relevant entities (*e.g.* synapses, neurons, populations of neurons, *etc.*) which do not depend directly on the chip's specific circuit details, and provide a framework that is independent of the hardware used. The pyTune tool-set allows users to define abstract high-level parameters of these computational neuroscience relevant entities, as functions of other high- or low-level parameters (such as circuit bias settings). This tool-set can then be used to automatically calibrate the properties of the corresponding hardware components (neurons, synapses, conductances, *etc.*), or to determine the optimal set of high- and low-level parameters that minimize arbitrary defined cost-functions.

Using this framework, neuromorphic hardware systems can be automatically configured to reach a desired configuration or state, and parameters can be tuned to maintain the system in the optimal state.

978-1-4244-9474-3/11/$26.00

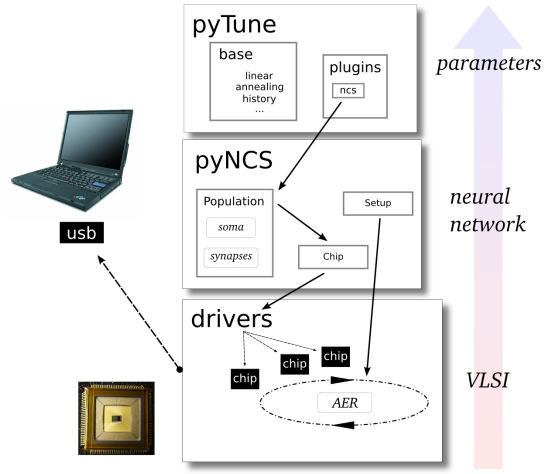

Fig. 1. Neuromorphic system configuration framework. Low-level drivers interface custom chips to workstations (*e.g.*, using USB connections). The pyNCS tool-set abstracts the chip specific characteristics, defines the setup, and the chip's functional blocks (*e.g.*, populations of neurons). The pyTune tool-set performs the calibration of high-level parameters and optimization of cost functions, using the optimization algorithms in the base sub-module.

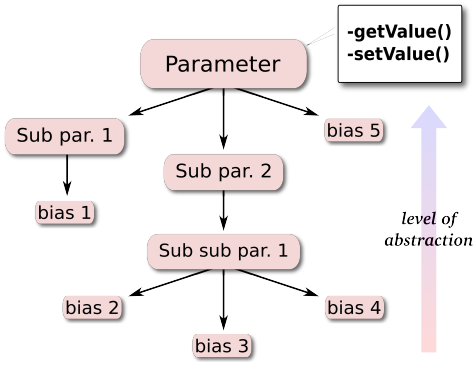

Fig. 2. Example of parameter definition. A hierarchical dependency tree specifies which lower-level parameters affect the parameter being defined. The `bias` parameters directly change the biase voltages on the chip(s). Each parameter contains a definition of a `getValue` and (optionally) a `setValue` function. These functions specify how to measure and set the chip signals that define the parameter. If the `setValue` function is not defined, the system travels the tree following the specified optimization algorithm until it finds the lowest-level parameters that affect the chip biases.

II. The pyNCS tool-set

The pyNCS tool-set acts on neuromorphic chips interfaced to workstations. At the lowest level dedicated drivers are required to interface custom neuromorphic chips to computers. Although custom drivers must be developed for each specific hardware, they can be cast as Python modules and integrated as plug-ins in the pyNCS tool-set. Once the drivers are implemented, pyNCS creates an abstraction layer to simplify the configuration of the hardware and its integration with other software modules. The experimental setup is then defined using information provided by the designer on the circuit functional blocks, their configuration biases, and the chip's analog and digital input/output channels. The setup, the circuits, and their biases are encapsulated into abstract components controllable via a Graphical User Interface (GUI) or an Application Programming Interface (API).

Experiments (equivalent to software simulation runs) can be defined, set-up, and carried out, using methods and commands analogous to those present in software modern neural simulators such as Brian [5] or PCSIM [6].

pyNCS uses a server-client architecture, thereby allowing multi-client support, load sharing, and remote access to the multi-chip setups. Thanks to this server-client architecture multiple clients can control the hardware remotely, regardless of the operating system used.

III. The pyTune tool-set

The pyTune tool-set is a Python module which automatically calibrates user defined high-level parameters, and optimizes user defined cost-functions. The parameters are defined using a dependency tree that specifies lower-level sub-parameters in a recursive hearachical way. An example of such type of dependency tree is shown in Fig. 2.

This hierarchical scheme allows the definition of arbitrarily complex parameters and related cost-functions. For example, synaptic efficacies in neural network models can be related to the bias voltages in neuromorphic chips which control the gain of synaptic circuits. In the DPI synapse [7], there are three bias voltages that simultaneously affect the synaptic gain in a non-linear fashion. Using the pyTune tool-set it is possible to automatically search the space of these bias voltages and set a desired synaptic efficacy by measuring the neuron's response properties (*e.g.* mean output rate) from the chip.

The automated parameter search can be applied to more complex scenarios to optimize high-level parameters related to network properties. For example, the user can specify the mapping between low-level parameters and the gain of a winner-take-all network [8], or the error of a learning algorithm [9]. Furthermore, the pyTune tool-set is not restricted to neuromorphic chip setups: it can be used in pure software simulation scenarios in which it is necessary to optimize cost-functions that involve complex or abstract parameters, as a function of direct or low-level model parameters. In every case, the mappings from low-level parameters to high-level parameters can be arbitrarily defined.

IV. Methods

Here we describe the steps required to integrate, configure, and calibrate a custom neuromorphic system using the pyNCS and pyTune tool-sets. They involve the creation of files describing the chip's functional blocks (excitatory synapses, inhibitory synapses, neurons, *etc.*), the experimental setup (*e.g.*, how many chips and what measurement instruments are used), and the topology of the neural network to be emulated. In addition it is necessary to define low- and high-level parameters that characterize the network, using the hierarchical scheme described in Fig. 2.

A. Chip functional blocks

All the details of the chip are collected in a single file in which its functional blocks are defined and the chip's pins are related to their parameters. The file specifies how

to change the biases of the functional blocks (*e.g.* via Digital-to-Analog (DAC)s). In addition, input and output signals are defined according to the Address Event Representation (AER) protocol [10].

The chip is represented in pyNCS as a `Chip` object. Once this object is instantiated, chip bias voltages are treated as ordinary variables and any operation that uses these variables actually reads the signals from the pins, while any operation that modifies these variables actually sets the corresponding DAC value.

B. Multi-chip experimental setup

A full experimental setup, typically comprising several chips, different types of measurement instruments, and different boards for interfacing the chips to workstations, is described in a separate specification file.

The chips send and receive spikes to/from other chips using the AER protocol. The setup specification file contains information about the correspondence between chips and their relative AER address space.

The experimental setup specification file provides a means for creating populations and networks as easily as it is done on modern software simulators, sparing the user from dealing directly with the hardware details. This abstraction is done by the `NeuroSetup` class in pyNCS (see Listing 1).

C. Neural network topology

The topology of the neural-network is described in terms of populations of neurons (`Population`) and inter-connections thereof (`Mapping`). The properties of the connections are defined by the synapses used to make these connections (*e.g.* excitatory, inhibitory or plastic). pyNCS provides convenient methods to build such topologies on the neuromorphic setups. Listing 1 demonstrates how this works.

```
#Initialize setup
setup = pyNCS.NeuroSetup('setup.xml')
# Populate 5 LIF neurons from 'chip1'
pop1 = pyNCS.Population(id='pop', description='Pop 1')
pop1.populate_by_number(setup, chip='chip1', type='IF_leaky
    ', N=5)
# Populate 10 LIF neurons from 'chip2'
pop2 = pyNCS.Population(id='pop', description='Pop 2')
pop2.populate_by_number(setup, chip='chip2', type='IF_leaky
    ', N=10)
# Connect pop1 to pop2
mapping = pyNCS.Mapping('network')
mapping.connect_one2one(pop1, pop2, type='excitatory')
```

Listing 1. A code snippet demonstrating how to use the pyNCS tool-set. In this example we define two populations of neurons on two chips ('chip1' and 'chip2') that are part of `setup`. Both `pop1` and `pop2` are first defined as neural populations and then populated with I&F neurons on the two chips. This operation sets the actual communication between the software abstraction layer and the hardware. Finally, a `mapping` instance is created and finally `pop1` is connected to `pop2` in a one-to-one fashion (excitatory connection).

D. Automatically tuning system parameters

The pyTune tool-set relies on the translation of the problem into parameter dependencies. The user defines each parameter by its measurement routine (`getValue` function) and its sub-parameters dependencies. At the lowest level, the parameters are defined only by their interaction with the hardware, *i.e.* they represent biases of the circuits. The user can choose a minimization algorithm from those available in the package or can define custom methods, to do the optimization that sets the parameters value. Optionally one can also define a specific cost function, that needs to be minimized. By default, the cost function is computed as $(p - p_{desired})^2$ where p is the current measured value of the parameter and $p_{desired}$ is the desired value. Explicit options (such as maximum tolerance for the desired value, maximum number of iteration steps, *etc.*) can also be passed as arguments to the optimization function.

Finally, the sub-parameters' methods are mapped by the appropriate plug-in onto the corresponding driver-calls, in the case of an hardware system, or onto method calls and variables in the case of a system simulated in software. Each mapping specific to a system has to be separately implemented and included in pyTune as a plug-in.

V. Results

In this section we demonstrate how pyTune is used in conjunction with pyNCS to set the mean firing rate of a population of neurons on a multi-neuron chip.

The setup comprises a multi-neuron chip which is a $10\ mm^2$ prototype VLSI device implemented in a standard 0.35μm CMOS technology. The chip comprises an array of 128 integrate-and-fire neurons and 4096 adaptive synapses with biologically plausible temporal dynamics [7]. Each of the 128 neurons receive input from 32 synaptic circuits which are subdivided into sets of excitatory non-plastic synapses, inhibitory non-plastic synapses, and excitatory plastic ones, with on-chip learning capabilities [11]. The multi-neuron chip biases can be modified by a board comprising a series of DACs and interfaced via USB to the workstation. Input and output spikes are sent to/from the chip using the AER protocol.

In this example, we created a population of 5 neurons and defined a `Rate` parameter corresponding to the mean firing rate of the population. The parameter uses built-in functions of pyNCS to stimulate the neurons, monitor their output spiking activity and compute the population mean firing rate.

In principle the population's mean firing rate can depend on several parameters (*e.g.* injected currents, recurrent connectivity, external inputs, time-constants). In order to simply illustrate how pyTune handles the dependencies we define the parameter as dependent on two sub-parameters, the `Injection` current to the neurons and the `Leak` current of the membrane. These are in fact biases of the chip, i.e. voltages to the gate transistors generating the injection current (p-type transistor) and the leak current (n-type) [12].

To visualize the dependence of `Rate` on its two sub-parameters, we carry out a two-dimensional sweep across the parameter space. The points in the 3D plot of Fig. 3 represent the values of `Rate`, which lie on a non-linear surface because of the exponential relationship between the biases and their respective currents. The default cost function is $(r - r_{desired})^2$, where r is the value of `Rate` measured from the hardware system and $r_{desired} = 50$ Hz is the target value. In this example, pyTune minimizes the cost function using an implementation of the Truncated Newton (TN) algorithm [13] provided by SciPy's optimization module [14]. The blue path shown in Fig. 3 connects the points measured by pyTune while setting the `Rate`.

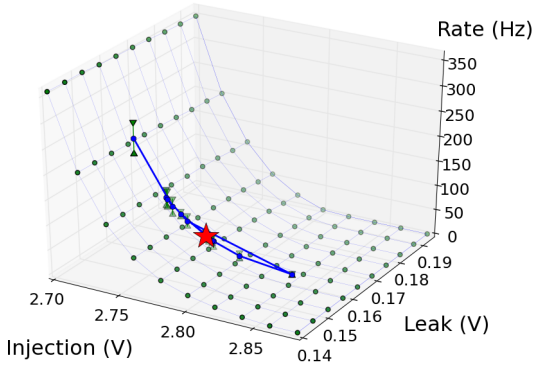

Fig. 3. A 3D wire-frame plot of the parameter space for the experiment described in Sec. V. Each dot on the wire-frame nodes is a measure of the `Rate` value. The surface represents the non-linear dependence of the parameter to be optimized (`Rate`) on the two sub-parameters (`Leak` and `Injection`). The sub-parameters are voltages on the gate of the n/p-transistor which controls the current of the leak/injection to the neuron circuit [12]. The blue line shows the algorithm's path during the optimization process (the red star represents the final point at 48.3 Hz). The error-bars on each point show the standard deviation on the rate measurement computed over the population, mainly due to transistors mismatch.

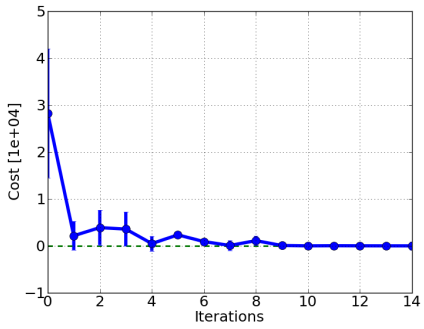

Fig. 4. The cost function is minimized to set the `Rate` parameter to a desired value of 50Hz on the example experiment of Sec. V. Error bars represent standard deviations. The final value of the population rate is 48.3Hz, with a deviation from the desired value below the tolerance we passed to the algorithm. See text for details.

Fig. 4 shows the progress of the optimization algorithm (cost versus iteration step). After 10 iterations, the algorithm converges to a mean rate of 48.3 Hz, which is compatible with the target rate of 50 Hz and the tolerance of 3 Hz declared as argument of the algorithm.

The code used to produce this data is available online (`http://ncs.ethz.ch/publications/examples-iscas-2011`).

VI. Conclusions

We presented a modular and expandable platform-independent Python-based framework to control, calibrate and tune custom neuromorphic systems. We showed how these Python software tools can be used to automatically configure neuromorphic hardware for emulating spiking neural networks. The framework is open, and modular in order to easily integrate a wide range of additional modules. These modules range from drivers (for interfacing the tools to custom VLSI chip), to programs for controlling measurement instruments (and acquiring data from the hardware setups), and optimization routines (for finding the optimal set of parameters that produce a desired behavior in the hardware setup). In this respect, the strengths of the work proposed are not in the specific methods used to solve the parameter mapping and calibration problems, but in providing an easy-to-extend, modular interface. It allows a high-level formal description and automation of the parameter optimization problem in which existing methods can be easily integrated and adapted, for use in conjunction with the hardware. The package is, in principle, completely compatible with other existing Python tool-sets such as pyNN [2]. It extends their capabilities in terms of systems that can be controlled, and can be considered as an additional extremely useful tool that can be included in the increasing number of Python applications developed for the neuroscience and neuromorphic engineering community.

Acknowledgment

This work was supported by the European FP7 grant #231168 – "SCANDLE" and the Swiss National Science Foundation grants #119973 – "SoundRec". The authors would like to thank the NCS group (`http://ncs.ethz.ch/`) for contributing to the development of the AER and multi-chip experimental setups.

References

[1] D. Brüderle, E. Müller, A. Davison, E. Muller, J. Schemmel, and K. Meier, "Establishing a novel modeling tool: a python-based interface for a neuromorphic hardware system," *Frontiers in Neuroinformatics*, vol. 4, 2009.

[2] A. Davison, D. Brüderle, J. Eppler, J. Kremkow, E. Muller, D. Pecevski, L. Perrinet, and P. Yger, "Pynn: a common interface for neuronal network simulators." *Frontiers in Neuroinformatics*, vol. 2, p. 11, 2008.

[3] E. Neftci, E. Chicca, G. Indiveri, and R. J. Douglas, "A systematic method for configuring vlsi networks of spiking neurons," *Neural Computation (submitted)*, 2010.

[4] "Special topic: Python in neuroscience," 2009. [Online]. Available: http://www.frontiersin.org/neuroinformatics/specialtopics/python_in_neuroscience/8

[5] D. Goodman and R. Brette, "Brian: a simulator for spiking neural networks in Python," *Frontiers in Neuroinformatics*, vol. 2, 2008.

[6] D. Pecevski, T. Natschläger, and K. Schuch, "PCSIM: a parallel simulation environment for neural circuits fully integrated with python," *Frontiers in Neuroinformatics*, vol. 3, no. 11, 2009.

[7] C. Bartolozzi and G. Indiveri, "Synaptic dynamics in analog VLSI," *Neural Computation*, vol. 19, no. 10, pp. 2581–2603, Oct 2007.

[8] A. L. Yuille and D. Geiger, *Winner-Take-All Networks*. The MIT Press, Cambridge, Massachussets, 2003, ch. Part III: Articles, pp. 1228–1231.

[9] J. Hertz, A. Krogh, and R. G. Palmer, *Introduction to the Theory of Neural Computation*. Reading, MA: Addison-Wesley, 1991.

[10] K. Boahen, "Communicating neuronal ensembles between neuromorphic chips," in *Neuromorphic Systems Engineering*, T. S. Lande, Ed. Norwell, MA: Kluwer Academic, 1998, pp. 229–259.

[11] 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, Feb. 2009.

[12] G. Indiveri, E. Chicca, and R. Douglas, "A VLSI array of low-power spiking neurons and bistable synapses with spike–timing dependent plasticity," *IEEE Transactions on Neural Networks*, vol. 17, no. 1, pp. 211–221, Jan 2006.

[13] S. G. Nash, "A survey of truncated-newton methods," *Journal of Computational and Applied Mathematics*, vol. 124, no. 1-2, pp. 45 – 59, 2000. [Online]. Available: http://www.sciencedirect.com/science/article/B6TYH-41MJ0RK-4/2/e9e5a91a3219fd5d4bdce9ac15e92dd5

[14] E. Jones, T. Oliphant, P. Peterson, *et al.*, "SciPy: Open source scientific tools for Python," 2001–. [Online]. Available: http://www.scipy.org/