frontiers in
NEUROINFORMATICS

ORIGINAL RESEARCH ARTICLE
published: 05 June 2009
doi: 10.3389/neuro.11.017.2009

Establishing a novel modeling tool: a python-based interface for a neuromorphic hardware system

Daniel Brüderle[1]*†, Eric Müller[1†], Andrew Davison[2], Eilif Muller[3], Johannes Schemmel[1] and Karlheinz Meier[1]

[1] Kirchhoff Institute for Physics, University of Heidelberg, Heidelberg, Germany
[2] Unité de Neurosciences Intégratives et Computationnelles, CNRS, Gif sur Yvette, France
[3] Laboratory of Computational Neuroscience, EPFL, Lausanne, Switzerland

Edited by:
Rolf Kötter, Radboud University Nijmegen, The Netherlands

Reviewed by:
Bernabe Linares-Barranco, Instituto de Microelectrónica de Sevilla, Spain
Adrian Whatley, University of Zurich, Switzerland

***Correspondence:**
Daniel Brüderle, Kirchhoff Institute for Physics, Im Neuenheimer Feld 227, 69120 Heidelberg, Germany.
e-mail: bruederle@kip.uni-heidelberg.de

†*Daniel Brüderle and Eric Müller have contributed equally to this work.*

Neuromorphic hardware systems provide new possibilities for the neuroscience modeling community. Due to the intrinsic parallelism of the micro-electronic emulation of neural computation, such models are highly scalable without a loss of speed. However, the communities of software simulator users and neuromorphic engineering in neuroscience are rather disjoint. We present a software concept that provides the possibility to establish such hardware devices as valuable modeling tools. It is based on the integration of the hardware interface into a simulator-independent language which allows for unified experiment descriptions that can be run on various simulation platforms without modification, implying experiment portability and a huge simplification of the quantitative comparison of hardware and simulator results. We introduce an accelerated neuromorphic hardware device and describe the implementation of the proposed concept for this system. An example setup and results acquired by utilizing both the hardware system and a software simulator are demonstrated.

Keywords: neuromorphic, VLSI, hardware, software, modeling, computational neuroscience, Python, PyNN

INTRODUCTION

Models of spiking neurons are normally formulated as sets of differential equations for an analytical treatment or for numerical simulation. So-called "neuromorphic" hardware systems represent an alternative approach. In a physical, typically silicon, form they mimic the structure and emulate the function of biological neural networks. Neuromorphic hardware engineering has a tradition going back to the 1980s (Mead, 1989; Mead and Mahowald, 1988), and today an active community is developing analog or mixed-signal VLSI models of neural systems (Ehrlich et al., 2007; Häfliger, 2007; Merolla and Boahen, 2006; Renaud et al., 2007; Schemmel et al., 2007, 2008; Serrano-Gotarredona et al., 2006; Vogelstein et al., 2007).

The main advantage of the physical emulation of neural network models, compared to their numerical simulation, arises from the locally analog and massively parallel nature of the computations. This leads to neuromorphic network models being typically highly scalable and being able to emulate neural networks in real time or much faster, independent of the underlying network size. Often, the inter-chip event-communication bandwidth sets a practical limit on the scaling of network sizes by inter-connecting multiple neural network modules (Berge and Häfliger, 2007; Costas-Santos et al., 2007; Schemmel et al., 2008). Compared to numerical solvers of differential equations which require Von-Neumann-like computer environments, neuromorphic models have much more potential for being realized as miniature embedded systems with low power consumption.

A clear disadvantage is the limited flexibility of the implemented models. Typically, neuron and synapse parameters and the network connectivity can be programmed to a certain degree within limited ranges by controlling software. However, changes to the implemented model itself usually require a hardware re-design, followed by production and testing phases. This process normally takes several months. Further fundamental differences between hardware and software models will be discussed in the Section "Neuromorphic Hardware".

Except for the system utilized in this work, all cited neuromorphic hardware projects currently work with circuits operating in biological real-time. This allows interfacing real-world devices such as sensors (Serrano-Gotarredona et al., 2006) or motor controls for robotics, as well as setting up hybrid systems with *in vitro* neural networks (Bontorin et al., 2007). The neuromorphic hardware systems we consider in this article, as described in Schemmel et al. (2007, 2008), possess a crucial feature: they operate at a highly accelerated rate. The device which is currently in operation (Schemmel et al., 2007) (see "The Accelerated Hardware System" for a detailed description) exhibits a speedup factor of 10^5 compared to the emulated biological real time. This opens up new prospects and possibilities, which will be discussed in the Section "Neuromorphic Hardware".

This computation speed, together with an implementation path towards architectures with low power consumption and very large scale networks (Fieres et al., 2008; Schemmel et al., 2008), makes neuromorphic hardware systems a potentially valuable research tool for the modeling community, where software simulators are more commonplace (Brette et al., 2006; Morrison et al., 2005, 2007). To establish neuromorphic hardware as a useful component of the neural network modelers' toolbox requires a proof of the hardware system's biological relevance and its operability by non-hardware-experts.

An approach which can help to fulfil both of these conditions is to interface the hardware system with the simulator-independent language PyNN (Davison et al., 2008) (see "PyNN and NeuroTools"). The PyNN meta-language allows for a unified description of neural

network experiments, which can then be run on all supported backends, e.g. various software simulators or the presented hardware system, without modifying the description itself. Experiment portability, data exchange and unified analysis environments are only some of PyNN's important implications. For neuromorphic devices, this provides the possibility to calibrate and verify the implemented models by comparing any emulated data with the corresponding results generated by established software simulators. Every scientist, who has already used such a simulator with scripting support or with an interpreter interface, will easily learn how to use PyNN. And every PyNN user can operate the presented hardware system without a deeper knowledge of technical device details.

In the Section "Simulator-like Setup, Operation and Analysis", the architecture of a Python (Rossum, 2000) interface to the hardware system, which is the basis for integration into PyNN, will be described in detail. The advantages and problems of the PyNN approach for the hardware system will also be discussed. In the Section "The Interface in Practice", an example of PyNN code for the direct comparison of an experiment run on both the hardware system and a software simulator, including the corresponding results, will be presented.

NEUROMORPHIC HARDWARE

Unlike most numerical simulations of neural network models, analog VLSI circuits operate in the continuous time regime. This avoids possible discretization artifacts, but also makes it impossible to interrupt an experiment at an arbitrary point in time and restart from an identical, frozen network state. Furthermore, it is not possible to perfectly reproduce an experiment because the device is subject to noise, to cross-talk from internal or external signals, and to temperature dependencies (Dally and Poulton, 1998). These phenomena often have a counterpart in the biological specimen, but it is highly desirable to control them as much as possible.

Another major difference between software and hardware models is the finiteness of any silicon substrate. This in principle also limits the software model size, as it utilizes standard computers with limited memory and processor resources, but for neuromorphic hardware the constraints are much more immediate: the number of available neurons and the number of synapses per neuron have strict upper limits; the number of manipulable parameters and the ranges of available values are fixed.

Still, neuromorphic network models are highly scalable at constant speed due to the intrinsic parallelism of their circuit operation. This scalability results in a relative speedup compared to software simulations, which gets more and more relevant the larger the simulated networks become, and provides new experimental possibilities. An experiment can be repeated many times within a short period, allowing the common problem of a lack of statistics, due to a lack of computational power, to be overcome. Large parameter spaces can be swept to find an optimal working point for a specific network architecture, possibly narrowing the space down to an interesting region which can then be investigated using a software simulator with higher precision. One might also think of longer experiments than have so far been attempted, especially long-term learning tasks which exploit synaptic plasticity mechanisms (Schemmel et al., 2007).

THE ACCELERATED HARDWARE SYSTEM

Within the FACETS research project (FACETS, 2009), an interdisciplinary consortium investigating novel computing paradigms by observing and modeling biological neural systems, an accelerated neuromorphic hardware system has been developed. It will be described in this section.

Neuron, Synapse and Connectivity model

The FACETS neuromorphic mixed-signal VLSI system has been described in detail in recent publications (Schemmel et al., 2006, 2007). Implemented is a leaky integrate-and-fire neuron model with conductance-based synapses, designed to exhibit a linear correspondence with existing conductance-based modeling approaches (Destexhe et al., 1998). The chip was built on a single 25 mm^2 die using a standard 180 nm CMOS process. It models networks of up to 384 neurons and the temporal evolution of the weights of 10^5 synapses. The system can be operated with an acceleration factor of up to 10^5 while recording the neural action potentials with a temporal resolution of approximately 0.3 ns, which corresponds to 30 µs in biological time.

The neuron circuits are designed such that the emulated membrane potential $V(t)$ is determined by the following differential equation for a conductance-based integrate-and-fire neuron:

$$-C_{\mathrm{m}}\frac{\mathrm{d}V}{\mathrm{d}t} = g_{\mathrm{m}}(V-E_{\mathrm{l}}) + \sum_j p_j(t)g_j(t)(V-E_{\mathrm{e}}) + \sum_k p_k(t)g_k(t)(V-E_{\mathrm{i}}) \quad (1)$$

where C_{m} represents the total membrane capacitance. The first term on the right hand side, the so-called leak current, models the contribution of the different ion channels that determine the potential E_{l} the membrane will eventually reach if no other currents are present. The synapses use different reversal potentials, E_{i} and E_{e}, to model inhibitory and excitatory ion channels. The index j in the first sum runs over all excitatory synapses while the index k in the second sum covers the inhibitory ones. The activation of individual synapses is controlled by the synaptic opening probability $p_{j,k}(t)$ (Dayan and Abott, 2001). The synaptic conductance $g_{j,k}$ is modeled as a product of the synaptic weight $\omega_{j,k}(t)$ and a maximum conductance $g^{\max}_{j,k}(t)$. The neuron emits a spike if a threshold voltage V_{th} is exceeded, after which the membrane potential is forced to a reset voltage V_{reset} and then released back into the influence of excitatory, inhibitory and leakage mechanisms. The weights are modified by a long-term plasticity algorithm (Schemmel et al., 2007) and thus can vary slowly with time. **Table 1** summarizes the most important hardware parameters, with their counterparts in the biological model, their available ranges and uncertainties.

Each chip is divided into two network blocks of 192 neurons each, and each block can receive 256 different input channels. Each input channel into a block can be configured to receive either a feedback signal from one specific neuron within the same block, a feedback signal from the opposite block, or an externally generated signal, for example from some controlling software. Every neuron within the block can be connected to every input channel via a configurable synapse. Synaptic time constants and the values for $g^{\max}$ are shared for every input channel, while the connection weights

Table 1 | The most important hardware model parameters, the type of physical quantity used for their implementation, their configurability and an estimation of uncertainty. The first four columns show their typical biological interpretation and the resulting value ranges. The translation between both domains depends on the chosen speedup and the desired biological parameter value ranges. The given estimations (some being educated guesses) of configuration uncertainty reflect the current state of available methods to measure, to adjust or to calibrate the values, and may not necessarily reflect hardware limitations. The uncertainty of E_e is load-dependent, the relation is not yet sufficiently analyzed.

Biological Interpretation				Hardware parameter implementation		
Param	**Unit**	**Min**	**Max**	**Physical quantity**	**Configurable**	**Estimation of uncertainty (%)**
C_m	nF	0.2	0.2	Capacitance	No	10
G_l	nS	20	40	Current	Yes	10
E_l	mV	–80	–55	Voltage	Yes	2
E_i	mV	–80	–55	Voltage	Yes	2
E_e	mV	–80	20	Voltage	Yes	Unknown
V_{th}	mV	–80	–55	Voltage	Yes	5
V_{reset}	mV	–80	–55	Voltage	Yes	10
τ_{syn}	ms	30	50	Current	Yes	25
g^{max}	nS	1	100	Current	Yes	25

can be set between 0 nS and g^{max} with a four bit resolution for each individual connection.

Although the free parameter space is already large, the model flexibility is clearly limited, especially in terms of its inter-neuron connectivity. Based on the experience acquired with the prototype chip described above, a wafer-scale integration[1] system (Fieres et al., 2008; Schemmel et al., 2008) with up to 1.8×10^5 neurons and 4×10^7 synapses per wafer is currently under development. It will be operated with a speedup factor of up to 10^4 and will provide a much more flexible and powerful connectivity infrastructure.

Support framework

In order to give life to such a piece of manufactured neuromorphic silicon, an intricate framework of various pieces of custom-made support hardware and software layers has to be deployed, which has previously been reported on. The chip is mounted on a carrier board called Nathan (Fieres et al., 2004; Grübl, 2007, Chapter 3) which also holds, among other components, an FPGA for direct communication control and some RAM memory modules for storing input and output data. Up to 16 of these carrier boards can be placed on a so-called backplane (Philipp et al., 2007), which itself is connected to a host PC via a PCI-based FPGA card (Schürmann et al., 2002).

The connection from chip to computer via the PCI card allows the configuration of the hardware, the definition and application of spike stimuli and the recording of spiking activity from within the network. Analog sub-threshold data can only be acquired via an oscilloscope[2], which is connected to pins that can output selectable membrane potentials. Via a network connection, the information from this oscilloscope can be read and integrated into the software running on the host computer (see **Figure 1** for a setup schematic).

Both an FPGA on the backplane and those on the carrier boards are programmed and configured with dedicated code. Communication with the PCI board utilizes a specific device driver and a custom-made protocol (Philipp, 2008, Chapter 2.2.4). Multi-user access is realized via userspace daemon multiplexing connections to different chips while encapsulating control commands and data from multiple users in POSIX Message Queues (IEEE, 2004). Data transfer from and to the oscilloscope is based on TCP/IP sockets (Braden, 1989; LeCroy, 2005). Interconnecting multiple chips in order to set up larger networks will be possible soon (Philipp et al., 2007).

SIMULATOR-LIKE SETUP, OPERATION AND ANALYSIS

As proposed in the introduction, attracting neuroscience experts into the field of neuromorphic engineering is essential for the establishment of hardware devices as modeling tools. Neuroscience expertise has to be consulted not only during the design process, but also, and especially, after manufacturing, when it comes to verifying the device's biological relevance. This implies a whole set of requirements for the software which provides the user interface to the hardware.

If the system is to be operated by scientists from fields other than neuromorphic engineering, the software must hide as many hardware-specific details as possible. We propose that it should provide basic control mechanisms similar to typical interfaces of pure software simulators, i.e. an interpreter for interactive operation and scripting. Parameters and observables should be given in biological dimensions and follow a biological nomenclature. Moreover, drawing the attention of the neuroscience community to neuromorphic hardware can be strongly facilitated by the possibility of porting existing software simulation setups to the hardware with little effort.

Multiple projects and initiatives provide databases and techniques for sharing or unifying neuroscientific modeling code, see for example the NeuralEnsemble initiative (Neural Ensemble, 2009), the databases of Yale's SenseLab (Hines et al., 2004) or the software database of the International Neuroinformatics Coordination Facility (INCF Software Database, 2009). Creating a bridge from the hardware interface to these pools of modeling experience will provide the important possibility of formulating transparent tests,

[1] A silicon wafer which will not be cut into single chips as is usual, but left in one piece. Further post-processing steps will interconnect the disjoint reticles on the wafer, resulting in a highly configurable silicon neural network model of unique dimensions.

[2] Currently: LeCroy WaveRunner 44Xi.

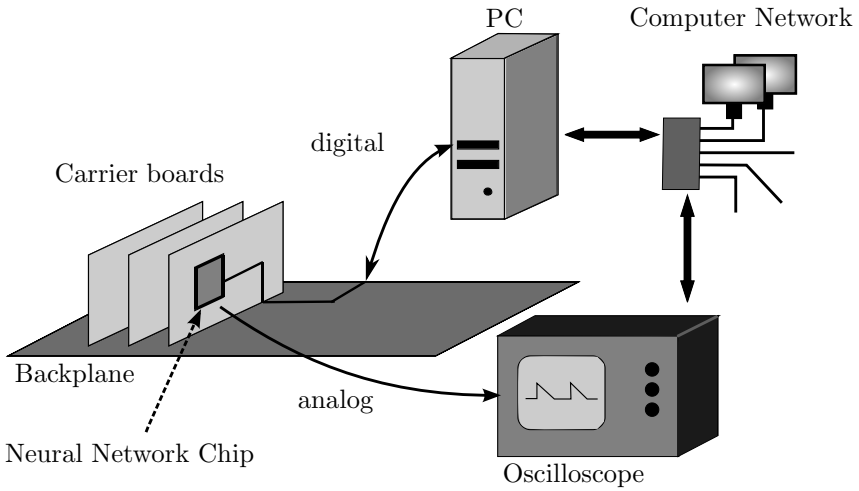

FIGURE 1 | Schematic of the accelerated FACETS hardware system framework. Via a digital connection, software running on the host computer can control the parameters of any neural network chip mounted on a carrier board on the communication backplane. It can stimulate the network with externally generated spikes and can record spikes generated on the chip. Analog sub-threshold information acquired with an oscilloscope can be integrated into the software via a network connection.

benchmarks and requests that will boost further hardware development and its establishment as a modeling tool.

Most software simulators for spiking neuron models come with an interpreter interface for programming, experiment setup and control. For example, NEURON (Hines and Carnevale, 2006; Hines et al., 2009) provides an interpreter called Hoc, NEST (Diesmann and Gewaltig, 2002; Eppler et al., 2008; Gewaltig and Diesmann, 2007) comes with a stack-based interface called SLI, and GENESIS (Bower and Beeman, 1998) has a different custom script language interpreter also called SLI. Both NEURON and NEST also provide Python (Rossum, 2000) interfaces, as do the PCSIM (PCSIM, 2009; Pecevski et al., 2009), Brian (Goodman and Brette, 2008) and MOOSE (Ray and Bhalla, 2008) simulators. Facilitating the usage of neuromorphic hardware for modelers means providing them with an interface similar to these existing ones. But there are further requirements arising from hardware specific issues.

TECHNICAL REQUIREMENTS

As shown in the Section "Support Framework", operating the presented neuromorphic hardware system involves multiple devices and mechanisms, e.g. Message Queue communication with a user-space daemon accessing a PCI board, TCP/IP socket connection to an oscilloscope, software models that control the operation of the backplane, the carrier board and the VLSI chip itself, and high-level software layers for experiment definition. On the software side, this multi-module system utilizes C, C++ and Python, and multiple developers from different institutions are involved, applying various development styles such as object-oriented programming, reflective programming or sequential driver code. The software has to follow the ongoing system development, including changing and improving FPGA controller code and hardware revisions with new features.

This complexity and diversity argues strongly for a top-level software framework, which has to be capable of efficiently gluing all modules together, supporting object-oriented and reflective structures, and providing the possibility of rapid prototyping in order to quickly adapt to technical developments at lower levels.

One further requirement arises: the speedup of the hardware system can be exploited by an interactive, possibly intuition-guided work flow which allows the exploration of parameters with immediate feedback of the resulting changes. This implies the wish to have the option of a graphical interface on top of an arbitrary experiment description.

EXISTING INTERFACES

Descriptions in the literature of existing software interfaces to neuromorphic hardware are very rare. In Merolla and Boahen (2006), the existence and main features of a GUI for the interactive operation of a specific neuromorphic hardware device are mentioned.

Much more detailed software interface reports are found in Dante et al. (2005). They describe a framework which allows exchange of AER[3] data between hardware and software while experiments are running. The framework includes a dedicated PCI board which is connected to the neuromorphic hardware module and which can be interfaced to Linux systems by means of a device driver. A C-library layered on top of this driver is available. Using this, a client-server architecture has been implemented which allows the on-line operation of the hardware from within the program MATLAB. The use of MATLAB implies interpreter-based usage, scripting support, the possible integration of C and C++ code, optional graphical front-end programming and strong numerical support for data analysis. Hence, most of the requirements listed so far are satisfied. Nevertheless, the framework is somewhat standalone and does not facilitate the transfer of existing software models to the hardware.

In Oster et al. (2005), an automatically generated graphical front-end for the manual tuning of hardware parameters is presented, including the convenient storing and loading of configurations.

[3]Address Event Representation.

Originally, a similar approach was developed for the hardware system utilized here, too (Brüderle et al., 2007). Manually defining parts of the enormous parameter space provided by such a chip via sliders and check-boxes can be useful for intuition-guided hardware exploration and circuit testing, but it turns out to be rather impractical for setting up large network experiments as usually performed by computational neuroscientists.

CHOOSING A PROGRAMMING LANGUAGE

Except for the convenient portability of existing experiment setups, an interface to the neuromorphic hardware system based on the programming language Python solves all of the requirements stated in the Sections "Importance of the Software Interface" and "Technical Requirements", especially the hardware-specific ones. Python is an interpreter-based language with scripting support, thus it is able to provide a software-simulator-like interface. It can be efficiently connected to C and C++, for example via the package Boost.Python (Abrahams and Grosse-Kunstleve, 2003). Python supports sequential, object-oriented and reflective programming and it is widely praised for its rapid prototyping. Due to the possibility for modular code structure and embedded documentation, it has a high maintainability, which is essential in the context of a quickly evolving project with a high number of developers.

In addition to its strengths for controlling and interconnecting lower-level software layers, it can be used to write efficient post-processing tools for data analysis and visualization, since a wide range of available third-party packages offers a strong foundation for scientific computing (Jones et al., 2001; Langtangen, 2008; Oliphant, 2007), plotting (Hunter, 2007) and graphics (Lutz, 2001, Chapter 8; Summerfield, 2008). Hence, a Python interface to the hardware system would already greatly facilitate modeler adoption.

Still, the possibility of directly transferring existing experiments to the hardware is even more desirable; a unified meta-language usable for both software simulators and the hardware could achieve that. Thus, the existence of the Python-based, simulator-independent modeling language PyNN (see PyNN and NeuroTools) was the strongest argument for utilizing Python as a hardware interface, because the subsequent integration of this interface into PyNN depended on the possibility of accessing and controlling the hardware via Python.

Possible alternatives to Python as the top layer language for the hardware interface have been considered and dropped for different reasons. For example, C++ requires a good understanding of memory management, it has a complex syntax, and, compared to interpreted languages, has slower development cycles. Interpreter-based languages such as Perl or Ruby also provide plotting functionality, numerical packages (Berglihn, 2006; Glazebrook and Economou, 1997) and techniques to wrap C/C++ code, but eventually Python was chosen because it is considered to be easy to learn and to have a clean syntax.

PYNN AND NEUROTOOLS

The advantages of Python as an interface and programming language are not limited to hardware back-ends. For the software simulators NEURON, NEST, PCSIM, MOOSE and Brian, Python interfaces exist. This provides the possibility of creating a Python-based, simulator-independent meta-language on top of all these back-ends. In the context of the FACETS project, the open-source Python module PyNN has been developed which implements such a unified front-end (see Davison, 2009; Davison et al., 2008).

PyNN offers the possibility of porting existing experiments between the supported software simulators and the FACETS hardware and thus to benchmark and verify the hardware model. Furthermore, on top of PyNN, a library of analysis tools called NeuroTools (2009) is under development, exploiting the possibility of a unified work flow within the scope of Python. Experiment description, execution, result storage, analysis and plotting can be all done from within the PyNN and NeuroTools framework. Independent of the used back-end, all these steps have to be written only once and can then be run on each platform without further modifications.

Especially since the operation of the accelerated hardware generates large amounts of data at high iteration rates, a sophisticated analysis tool chain is necessary. For the authors, as well as for every possible PyNN user, making use of the unified analysis libraries based on the PyNN standards (e.g. NeuroTools) avoids redundant development and debugging efforts. This benefit is further enhanced by other third-party Python modules, like numerical or visualization packages.

INTERFACE ARCHITECTURE

The complete software framework for interfacing the FACETS hardware is structured as follows: Various C++ classes encapsulate the functionality of the neural network chip itself, of its configuration parameter set, of the controller implemented on the carrier board FPGA, and of the communication protocol between the host software and this controller. There is a stand-alone daemon written in C++ which provides the transport of data via the PCI card. It utilizes a device-driver which is available for Linux systems. Furthermore, there is a C++ class which encapsulates the TCP/IP Socket communication with the oscilloscope.

The Boost.Python library (Boost.Python, 2003) is used to bind C++ classes and functions to Python. An instructive outline of the wrapping technique used can be found in Abrahams and Grosse-Kunstleve (2003).

On top of these Python bindings, a pure Python framework called PyHAL[4] (Brüderle et al., 2007) provides classes for neurons, synapses and networks. All these classes have model parameters in biological terminology and dimensions, and their constructors impose no hardware specific constraints.

The main functionality of PyHAL is encapsulated by a hardware access class which implements the exchange layer between these higher-level objects and the low-level C++ classes exposed to Python via Boost. The hardware access layer performs the translation from biological parameters like reversal potentials, leakages, synaptic time constants and weights to the available set of hardware configuration parameters. This set consists of discrete integers, for example for the synaptic weights, and of analog values for currents and voltages. Some of these parameters do have a direct biological counterpart, some do not. For example, neuron voltage parameters like reversal potentials are mapped linearly to the available hardware membrane potential range of approximately 0.6–1.4 V,

[4]Python Hardware Abstraction Layer.

while membrane leakage conductances and synaptic time constants have to be translated into currents.

The translation layer also performs the transformation from biological to hardware time domain and back. Furthermore, all hardware-specific constraints, like the limited number of possible neurons or connections, the finite parameter ranges and the synaptic weight discretization, are incorporated in this hardware access class, generating instructive warnings or error messages in case of constraint violations.

Since the PyHAL framework is all Python code, it provides the desired interpreter-based interface to the hardware, corresponding to comparable Python interfaces to, for example, NEST or PCSIM. Also, as for these software simulators, a module for the integration of this interface into the meta-language PyNN has been implemented. **Figure 2** shows a schematic of the complete software framework with its most important components.

Thanks to this integration, all higher-level PyNN concepts like populations and inter-population projections plus the analysis and visualization tools developed on top of PyNN are now available for the hardware system.

Still, the integration of the hardware interface into PyNN also raises problems. Some of the PyNN API function arguments are specific to software simulators. In the hardware context, they have to be either ignored or be given a hardware-specific interpretation. For example, the PyNN function `setup` has an argument called `timestep`, which for pure software back-ends determines the numerical integration time step. In the PyNN module for the continuously operating hardware, this argument defines the temporal resolution of the oscilloscope for membrane potential recordings. Furthermore, the strict constraints regarding neuron number, connectivity and possible parameter values require an additional software effort, i.e. checking for violations and providing the messages mentioned above. PyNN does not yet sufficiently support fast and statistics-intensive parameter space searches with differential formulations of the changes from step to step, which will be needed to optimize the exploitation of hardware specific advantages.

Without having access to the real hardware system, it is of course not possible to use the PyNN hardware module, hence it is not available for download. Still, it is planned to publicly provide a modified module on the PyNN website (Davison, 2009) which allows testing of PyNN scripts intended to be run on the hardware, i.e. to get back all warnings or error messages which might occur with the real system. With such a mapping test module, scripts can be prepared offline for a later, optimized hardware run.

THE INTERFACE IN PRACTICE

To demonstrate the usage and functionality of the PyNN interface, a simple example setup is given in the following. **Listing 1** shows the experiment described in PyNN, which is then executed both on the hardware system and using the software simulator NEST. A network consisting of 80 excitatory and 20 inhibitory neurons is created. The inhibitory sub-population is fed back into the network randomly with a probability of 0.5 for each possible inhibitory-to-excitatory connection. 160 excitatory and 40 inhibitory Poisson spike trains are randomly connected to the network with the same probability of 0.5 for each possible train-to-neuron connection.

Figure 3 shows a schematic of the implemented network architecture.

The maximum synaptic conductance g^{max} is 0.5 nS for excitatory and 1.6 nS for inhibitory connections. The output spikes of eight neurons are recorded, and the average firing rate of these eight neurons over a period of 5 s of biological time is determined.

In line 1, the PyNN back-end NEST is chosen. In order to utilize the hardware system, the only necessary change within this script is to replace line 1 by `from pyNN.hardware.stage1 import *`, all the rest remains the same. From lines 4 to 9, the population sizes, the numbers of external stimuli, and the synaptic weights are set. In lines 11–17, the neuron parameters are defined. Lines 19

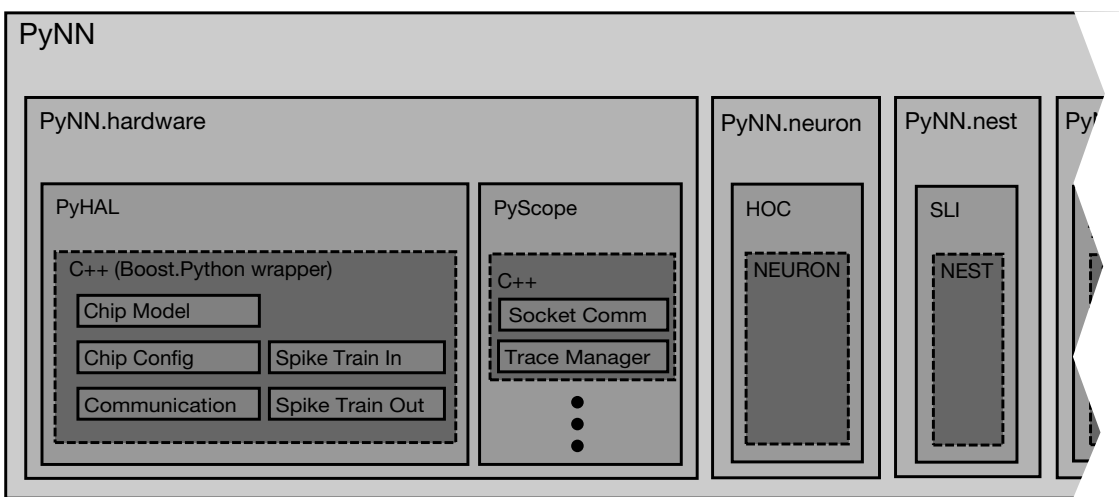

FIGURE 2 | Schematic of the software framework for the operation of the hardware system. It is integrated into the Python-based, simulator-independent language PyNN, which also supports back-ends like NEURON, NEST and more. The module for the hardware back-end consists of Python-based sub-modules for the digital and analog access to the chip. Each of those wrap the functionality of lower-level C++ layers, which are described in more detail in the text.

```
 1  from pyNN.nest2 import *
 2  # OR: from pyNN.hardware.stage1 import *
 3
 4  numInhNeurons =  20
 5  numExcNeurons =  80
 6  numInhInputs  =  40
 7  numExcInputs  = 160
 8  w_exc = 0.0005  # uS
 9  w_inh = 0.0016  # uS
10
11  neuronParams = {    'v_reset'    : -80.0,    # mV
12                      'e_rev_I'    : -75.0,    # mV
13                      'v_rest'     : -70.0,    # mV
14                      'v_thresh'   : -57.0,    # mV
15                      'g_leak'     :  20.0,    # nS
16                      'tau_syn_E'  :  30.0,    # ms
17                      'tau_syn_I'  :  30.0   } # ms
18
19  inputParameters = { 'rate'       : 5.0,      # Hz
20                      'duration'   : 5000    } # ms
21
22  setup(timestep=0.1)
23
24  n_inh = create(IF_facets_hardware1,neuronParams,n=numInhNeurons)
25  n_exc = create(IF_facets_hardware1,neuronParams,n=numExcNeurons)
26  net   = n_exc + n_inh
27
28  i_exc = create(SpikeSourcePoisson,inputParameters,n=numExcInputs)
29  i_inh = create(SpikeSourcePoisson,inputParameters,n=numInhInputs)
30
31  connect(i_exc,net,weight=w_exc,synapse_type='excitatory',p=0.5)
32  connect(i_inh,net,weight=w_inh,synapse_type='inhibitory',p=0.5)
33
34  connect(n_inh,net,weight=w_inh,synapse_type='inhibitory',p=0.5)
35
36  record(net[0:8], 'spikes.dat')
37  record_v(net[0], 'membrane.dat')
38
39  run(5000)  # duration in ms
40  end()
```

LISTING 1 | PyNN Example Script. For detailed explanation see text.

and 20 determine the rate and duration of the Poisson spike train stimuli. In line 22, PyNN is initialized, the numerical integration step size of 0.1 ms is passed. If the hardware back-end is chosen, no discrete step size is utilized due to the time continuous dynamics in its analog network core, and the function argument is used instead to determine the time resolution of the oscilloscope, if connected. In lines 24 and 25, the excitatory and inhibitory neurons are created, with the neuron parameters and the size of the populations as the second and the third arguments.

The first argument, `IF_facets_hardware1`, specifies the neuron type to be created. For the hardware system, no other neuron type is possible. For the NEST back-end, the neuron type determines parameter values for e.g. C_m, which are fixed to resemble the hardware. Line 26 concatenates the two populations. In lines 28 and 29, the Poisson spike sources are generated, passing the type of source, the previously defined parameters and the desired number. From lines 31 to 34, the neurons and spike generators are interconnected. The arguments of the connect command specify first a list of sources, then a list of targets, followed by the synaptic weights, the synapse types and finally by the probability with which each possible pairing of source and target objects is actually connected. The recording of the spikes of eight neurons and of one membrane

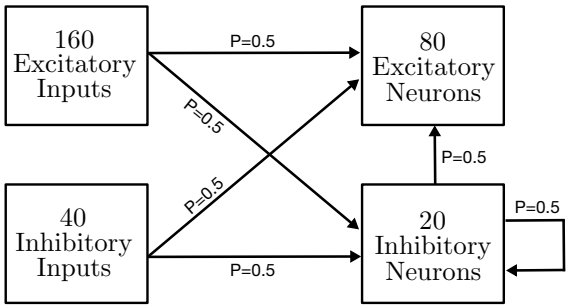

FIGURE 3 | Connectivity schematic of the implemented network. An excitatory and an inhibitory population of Poisson spike train generators stimulate an excitatory and an inhibitory population of neurons. The inhibitory population is fed back into itself and into the excitatory one. All inter-population projections have a unit-to-unit connection probability of 0.5.

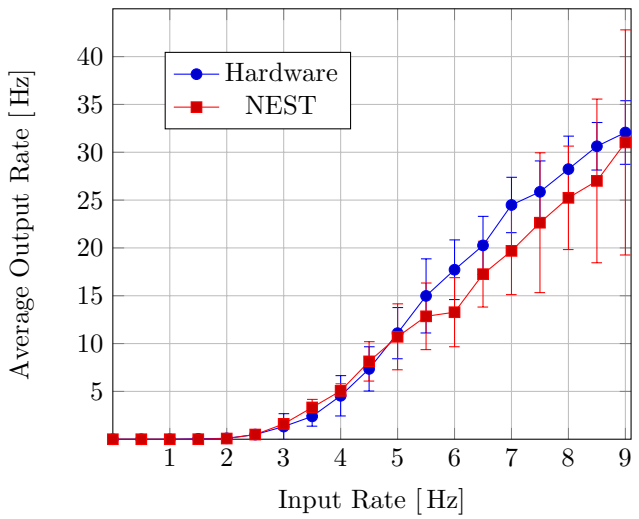

FIGURE 4 | Average output firing rate of the example network neurons as a function of input rate. The script shown in Listing 1 has been executed with various stimulation rates on both the hardware system (blue circles) and the software simulator NEST (red squares). Each data point represents the mean over 20 runs, the error bars denote the corresponding standard deviations.

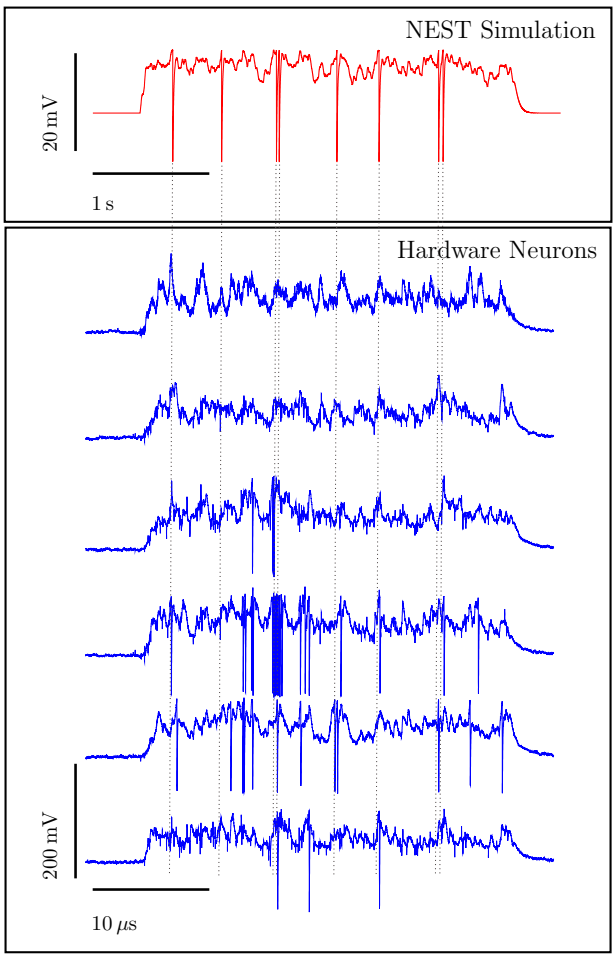

FIGURE 5 | Membrane potentials of a neuron under Poisson stimulation. Input spike times are identical for all traces. The uppermost trace (red) represents a NEST simulation. Spike times determined by NEST are marked with dashed vertical lines in light gray. The lower six traces (blue) represent measurements from adjacent hardware neurons recorded in separate runs. For the hardware traces, the given time and voltage scales indicate the real physical dimensions of the emulation.

potential is prepared in lines 36 and 37 (not all neurons, due to a bug in the current hardware revision). In line 39, the experiment is executed for a duration of 5000 ms. Line 40 defines the end of the script, and deals with writing recorded values to file.

The experiment was run both on the FACETS hardware system and using the software simulator NEST. The firing rate of the stimulating Poisson spike trains was varied from 0 to 9 Hz in steps of 0.5 Hz, and for each rate the experiment was repeated 20 times with different random number generator seeds. **Figure 4** shows the resulting average output firing rates.

The firing rates measured on both back-ends exhibit a qualitative and, within the observed fluctuations, quantitative correspondence. For both NEST and the hardware system, the onset of firing activity occurs at the same level of synaptic stimulation. The small but seemingly systematic discrepancy for higher output rates indicates that for the NEST simulation the inhibitory feedback has a slightly stronger impact on the network activity than on the hardware platform. The firing rate does not reflect dynamic properties like firing regularity or synchrony, which might be interesting for the estimation of possible differences in network dynamics due to the limited precision of hardware parameter determination or due to electronic noise. With PyNN, studies like these have now become possible, but go beyond the scope of this paper.

To give an impression of the inhomogeneities of a hardware substrate and of the noise a typical hardware membrane is exposed to, a second measurement is shown. A single neuron receives 80 excitatory and 20 inhibitory Poisson spike trains with 2.5 Hz each. It is connected to these stimuli with the same synaptic weights as in the setup described above, but gets no feedback from other neurons. The spike sources fire for 4 s, with a silent phase of 0.5 s before and after. Using a single PyNN description, the identical setup with identical spike times and identical connectivity can be deployed for both NEST and the hardware system. **Figure 5** shows

the resulting membrane potential trace simulated by NEST and the membrane potentials acquired from six adjacent neurons on the neuromorphic hardware. For the hardware traces, the unprocessed time and voltage scales are given as measured on the chip in order to illustrate the accelerated and physical nature of the neuromorphic model. The PyHAL framework automatically performs a translation of these dimensions into their biological equivalents.

The constant noise level in the hardware traces can be best observed during the phases with no external stimulation. This noise is a superposition of the noise actually occurring within the neuron circuits and the noise being added by the recording devices. The differences from hardware neuron to hardware neuron represent mainly device fluctuations on the transistor level, which strongly dominate time-dependent influences like temperature-dependent leakages or an unstable power supply. Counterbalancing these fixed-pattern effects with calibration methods is work in progress.

DISCUSSION

Today, the communities of computational neuroscientists and neuromorphic engineers work rather in parallel instead of benefitting from each other. We believe that closing this gap will boost the development, the usability and the number of application fields of neuromorphic systems, including the establishment of such devices as valuable modeling tools that will contribute to the understanding of neural information processing. Based on this motivation, we have described a set of requirements that a software interface for a neuromorphic system should fulfill.

Following these guidelines, we have implemented a Python-based interface to an existing accelerated neuromorphic hardware system developed within the research project FACETS, and we have integrated it into the common neural network simulator interface PyNN, proving the potential of PyNN to also serve as a hardware interface. This approach provides the novel possibility of porting existing experiments from the software simulator to the hardware domain and vice versa with a minimum of effort. In order to illustrate the unification and portability aspects, we have presented an example PyNN code sequence for a simple experiment. The correspondence between the results acquired with both a software simulator and the hardware system demonstrate the functionality of the framework.

With a neuromorphic device accessible and controllable via PyNN, its advantages can be exploited by non-hardware-experts from all fields. Hardware and software co-simulations based on PyNN descriptions can be used to test, to tune and to benchmark neuromorphic devices. Furthermore, the integration of hardware interfaces into the PyNN framework can avoid parts of the often redundant effort that has to be invested into creating a new individual software layer stack on top of any new neuromorphic system, since high-level tools, e.g. for analysis and plotting, are already available and maintained by an active community.

ACKNOWLEDGMENTS

This work is supported by the European Union under grant no. IST-2005-15879 (FACETS).

REFERENCES

Abrahams, D., and Grosse-Kunstleve, R. W. (2003). Building Hybrid Systems with Boost.Python. Available at: http://www.boostpro.com/writing/bpl.pdf.

Berge, H. K. O., and Häfliger, P. (2007). High-speed serial AER on FPGA. In ISCAS (IEEE), pp. 857–860.

Berglihn, O. T. (2006). RNUM Website. Available at: http://rnum.rubyforge.org.

Bontorin, G., Renaud S., Garenne, A., Alvado, L., Le Masson, G., and Tomas, J. (2007). A real-time closed-loop setup for hybrid neural networks. In Proceedings of the 29th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBS2007).

Boost.Python. (2003). Version 1.34.1 Website. Available at: http://www.boost.org/doc/libs/1_34_1/libs/python.

Bower, J. M., and Beeman D. (1998). The Book of GENESIS: Exploring Realistic Neural Models with the GEneral NEural SImulation System, 2nd Edn. New York, Springer-Verlag. ISBN 0387949380.

Braden, R. T. (1989). RFC 1122: Requirements for Internet Hosts–Communication Layers. Available at: ftp://ftp.internic.net/rfc/rfc1122.txt.

Brette, R., Rudolph, M., Carnevale, T., Hines, M., Beeman, D., Bower, J. M., Diesmann, M., Morrison, A., Goodman, P. H., Harris, F. C., Jr., Zirpe, M., Natschlager, T., Pecevski, D., Ermentrout, B., Djurfeldt, M., Lansner, A., Rochel, O., Vieville, T., Muller, E., Davison, A. P., El Boustani, S., and Destexhe, A. (2006). Simulation of Networks of Spiking Neurons: A Review of Tools and Strategies. Available at: http://arxiv.org/abs/q-bio.NC/0611089.

Brüderle, D., Grübl, A., Meier, K., Mueller, E., and Schemmel, J. (2007). A software framework for tuning the dynamics of neuromorphic silicon towards biology. In Proceedings of the 2007 International Work-Conference on Artificial Neural Networks, Vol. LNCS 4507 (Berlin, Springer Verlag), pp. 479–486.

Costas-Santos, J., Serrano-Gotarredona, T., Serrano-Gotarredona, R., and Linares-Barranco, B. (2007). A spatial contrast retina with on-chip calibration for neuromorphic spike-based AER vision systems. *IEEE Trans. Circuits Syst.* 54, 1444–1458.

Dally, W. J., and Poulton, J. W. (1998). Digital Systems Engineering. Cambridge, Cambridge University Press. ISBN 0-521-59292-5.

Dante, V., Del Giudice, P., and Whatley, A. M. (2005). Hardware and software for interfacing to address-event based neuromorphic systems. *Neuromorphic Eng.* 2, 5–6.

Davison, A. (2009). PyNN – A Python Package for Simulator-Independent Specification of Neuronal Network Models. Available at: http://www.neuralensemble.org/PyNN.

Davison, A. P., Brüderle, D., Eppler, J., Kremkow, J., Muller, E., Pecevski, D., Perrinet, L., and Yger, P. (2008). PyNN: a common interface for neuronal network simulators. *Front. Neuroinform.* 2, 11. doi: 10.3389/neuro.11.011.2008.

Dayan, P., and Abott, L. F. (2001). Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems. Cambridge, The MIT Press. ISBN 0-262-04199-5.

Destexhe, A., Contreras, D., and Steriade, M. (1998). Mechanisms underlying the synchronizing action of corticothalamic feedback through inhibition of thalamic relay cells. *J. Neurophysiol.* 79, 999–1016.

Diesmann, M., and Gewaltig, M.-O. (2002). NEST: an environment for neural systems simulations. In Forschung und wisschenschaftliches Rechnen, Beiträge zum Heinz-Billing-Preis 2001, Vol. 58, GWDG-Bericht, Theo Plesser and Volker Macho, eds (Göttingen, Ges. für Wiss. Datenverarbeitung), pp. 43–70.

Ehrlich, M., Mayr, C., Eisenreich, H., Henker, S., Srowig, A., Grübl, A., Schemmel, J., and Schüffny, R. (2007). Wafer-scale VLSI implementations of pulse coupled neural networks. In Proceedings of the International Conference on Sensors, Circuits and Instrumentation Systems.

Eppler, J. M., Helias, M., Muller, E., Diesmann, M., and Gewaltig, M.-O. (2008). PyNEST: a convenient interface to the NEST simulator. *Front. Neuroinform.* 2, 12. doi: 10.3389/neuro.11.012.2008.

FACETS (2009). Fast Analog Computing with Emergent Transient States, Project Homepage. Available at: http://www.facets-project.org.

Fieres, J., Grübl, A., Philipp, S., Meier, K., Schemmel, J., and Schürmann, F. (2004). A platform for parallel operation of VLSI neural networks. In Proceedings of the 2004 Brain Inspired Cognitive Systems Conference, University of Stirling, Scotland.

Fieres, J., Schemmel, J., and Meier, K. (2008). Realizing biological spiking network models in a configurable wafer-scale hardware system. In Proceedings of the 2008 International Joint Conference on Neural Networks.

Gewaltig, M.-O., and Diesmann, M. (2007). NEST (NEural Simulation Tool). *Scholarpedia* 2, 1430.

Glazebrook, K., and Economou, F. (1997). PDL: The Perl Data Language. Dr. Dobb's Journal. Available at: http://www.ddj.com/184410442.

Goodman, D., and Brette, R. (2008). Brian: a simulator for spiking neural networks in Python. *Front. Neuroinform.* 2, 5. doi: 10.3389/neuro.11.005.2008.

Grübl, A. (2007). VLSI Implementation of a Spiking Neural Network. PhD Thesis, Heidelberg, Ruprecht-Karls-University. Available at: http://www.kip.uni-heidelberg.de/Veroeffentlichungen/details.php?id = 1788. Document No. HD-KIP 07-10.

Häfliger, P. (2007). Adaptive WTA with an analog VLSI neuromorphic learning chip. *IEEE Trans. Neural Netw.* 18, 551–572.

Hines, M. L., and Carnevale, N. T. (2006). The NEURON Book. Cambridge, Cambridge University Press. ISBN 978-0521843218.

Hines, M. L., Davison, A. P., and Muller, E. (2009). NEURON and Python. *Front. Neuroinform.* 3, 1. doi: 10.3389/neuro.11.001.2009.

Hines, M. L., Morse, T., Migliore, M., Carnevale, N. T., and Shepherd, G. M. (2004). ModelDB: a database to support computational neuroscience. *J. Comput. Neurosci.* 17, 7–11.

Hunter, J. D. (2007). Matplotlib: a 2D graphics environment. *IEEE Comput. Sci. Eng.* 9, 90–95.

IEEE (2004). Standard for Information Technology – Portable Operating System Interface (POSIX). Shell and Utilities. Technical Report, IEEE. Available at: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber = 1309816.

INCF Software Database (2009). Website. Available at: http://software.incf.net.

Jones, E., Oliphant, T., Peterson, P. et al. (2001). SciPy: Open Source Scientific Tools for Python. Available at: http://www.scipy.org/.

Langtangen, H. P. (2008). Python Scripting for Computational Science, 3rd Edn. (Berlin, Springer). ISBN 978-3-540-73915-9.

LeCroy (2005). X-Stream Oscilloscopes–Remote Control Manual. Technical Report Revision D, New York, LeCroy Corporation. Available at: http://lecroygmbh.com.

Lutz, M. (2001). Programming Python: Object-Oriented Scripting. Sebastopol, O'Reilly & Associates, Inc. ISBN 0596000855.

Mead, C. A. (1989). Analog VLSI and Neural Systems. Reading, Addison Wesley.

Mead, C. A., and Mahowald, M. A. (1988). A silicon model of early visual processing. *Neural Netw.* 1, 91–97.

Merolla, P. A., and Boahen, K. (2006). Dynamic computation in a recurrent network of heterogeneous silicon neurons. In Proceedings of the 2006 IEEE International Symposium on Circuits and Systems.

Morrison, A., Aertsen, A., and Diesmann, M. (2007). Spike-timing-dependent plasticity in balanced random networks. *Neural Comput.* 19, 1437–1467.

Morrison, A., Mehring, C., Geisel, T., Aertsen, A., and Diesmann, M. (2005). Advancing the boundaries of high connectivity network simulation with distributed computing. *Neural Comput.* 17, 1776–1801.

Neural Ensemble (2009). Website. Available at: http://neuralensemble.org.

NeuroTools (2009). Website. Available at: http://neuralensemble.org/trac/NeuroTools.

Oliphant, T. E. (2007). Python for scientific computing. *IEEE Comput. Sci. Eng.* 9, 10–20.

Oster, M., Whatley, A. M. Liu, S.-C., and Douglas, R. J. (2005). A hardware/software framework for real-time spiking systems. In Proceedings of the 2005 International Conference on Artificial Neural Networks.

PCSIM (2009). Website. Available at: http://www.lsm.tugraz.at/pcsim/.

Pecevski, D. A., Natschläger, T., and Schuch, K. N. (2009). PCSIM: a parallel simulation environment for neural circuits fully integrated with python. *Front. Neuroinform.* 3, 11. doi: 10.3389/neuro.11.011.2009.

Philipp, S. (2008). Design and Implementation of a Multi-Class Network Architecture for Hardware Neural Networks. PhD Thesis, Heidelberg, Ruprecht-Karls Universität.

Philipp, S., Grübl, A., Meier, K., and Schemmel, J. (2007). Interconnecting VLSI Spiking Neural Networks Using Isochronous Connections. In Proceedings of the 9th International Work-Conference on Artificial Neural Networks, Vol. LNCS 4507 (Berlin, Springer Verlag), pp. 471–478.

Ray, S., and Bhalla, U. S. (2008). PyMOOSE: interoperable scripting in Python for MOOSE. *Front. Neuroinform.* 2, 6. doi: 10.3389/neuro.11.006.2008.

Renaud, S., Tomas, J., Bornat, Y., Daouzli, A., and Saighi, S. (2007). Neuromimetic ICs with analog cores: an alternative for simulating spiking neural networks. In Proceedings of the 2007 IEEE Symposium on Circuits and Systems.

Rossum, G. V. (2000). Python Reference Manual: February 19, 1999, Release 1.5.2. iUniverse, Incorporated. ISBN 1583483748.

Schemmel, J., Brüderle, D., Meier, K., and Ostendorf, B. (2007). Modeling synaptic plasticity within networks of highly accelerated I&F neurons. In Proceedings of the 2007 IEEE International Symposium on Circuits and Systems, IEEE Press.

Schemmel, J., Fieres, J., and Meier, K. (2008). Wafer-scale integration of analog neural networks. In Proceedings of the 2008 International Joint Conference on Neural Networks.

Schemmel, J., Grübl, A., Meier, K., and Mueller, E. (2006). Implementing synaptic plasticity in a VLSI spiking neural network model. In Proceedings of the 2006 International Joint Conference on Neural Networks. IEEE Press.

Schürmann, F., Hohmann, S., Schemmel, J., and Meier, K. (2002). Towards an artificial neural network framework. In Proceedings of the 2002 NASA/DoD Conference on Evolvable Hardware, A. Stoica, J. Lohn, R. Katz, D. Keymeulen, and R.S. Zebulum, eds (Los Alamitos, CA, IEEE Computer Society), pp. 266–273.

Serrano-Gotarredona, R., Oster, M., Lichtsteiner, P., Linares-Barranco, A., Paz-Vicente, R., Gómez-Rodríguez, F., Riis, H. K., Delbrück, T., Liu, S. C., Zahnd, S., Whatley, A. M., Douglas, R. J., Häfliger, P., Jimenez-Moreno, G., Civit, A., Serrano-Gotarredona, T., Acosta-Jiménez, A., and Linares-Barranco, B. (2006). AER building blocks for multi-layer multi-chip neuromorphic vision systems. In Advances in Neural Information Processing Systems 18, Y. Weiss, B. Schölkopf, and J. Platt, eds (Cambridge, MIT Press), pp. 1217–1224.

Summerfield, M. (2008). Rapid GUI Programming with Python and Qt. Prentice Hall, Upper Saddle River, NJ, ISBN 0132354187.

Vogelstein, R. J., Mallik, U., Vogelstein, J. T., and Cauwenberghs, G. (2007). Dynamically reconfigurable silicon array of spiking neuron with conductance-based synapses. *IEEE Trans. Neural Netw.* 18, 253–265.

Conflict of Interest Statement: The authors declare that the research presented in this paper was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Received: 14 September 2008; paper pending published: 23 December 2008; accepted: 09 May 2009; published online: 05 June 2009.

*Citation: Brüderle D, Müller E, Davison A, Muller E, Schemmel J and Meier K (2009) Establishing a novel modeling tool: a python-based interface for a neuromorphic hardware system. Front. Neuroinform. (2009) **3**:17. doi:10.3389/neuro.11.017.2009*