2014 International Joint Conference on Neural Networks (IJCNN)
July 6-11, 2014, Beijing, China

Efficient Implementation of STDP Rules on SpiNNaker Neuromorphic Hardware

Peter U. Diehl and Matthew Cook

Abstract— **Recent development of neuromorphic hardware offers great potential to speed up simulations of neural networks. SpiNNaker is a neuromorphic hardware and software system designed to be scalable and flexible enough to implement a variety of different types of simulations of neural systems, including spiking simulations with plasticity and learning. Spike-timing dependent plasticity (STDP) rules are the most common form of learning used in spiking networks. However, to date very few such rules have been implemented on SpiNNaker, in part because implementations must be designed to fit the specialized nature of the hardware. Here we explain how general STDP rules can be efficiently implemented in the SpiNNaker system. We give two examples of applications of the implemented rule: learning of a temporal sequence, and balancing inhibition and excitation of a neural network. Comparing the results from the SpiNNaker system to a conventional double-precision simulation, we find that the network behavior is comparable, and the final weights differ by less than 3% between the two simulations, while the SpiNNaker simulation runs much faster, since it runs in real time, independent of network size.**

I. Introduction

SIMULATIONS of spiking neural networks (SNN) are an important tool for neuroscientists to test hypotheses of how the brain functions. In order to speed up simulations, different types of neuromorphic hardware and neuro-inspired hardware have been developed [9], [14], [26], one of them being SpiNNaker [16]. SpiNNaker is based on a large number of conventional digital processors, interconnected by a network optimized for quickly transmitting small spike-like packets between the processors. In a SpiNNaker simulation, the neuron model is implemented by conventional code running on the processors, in contrast with other neuromorphic hardware systems which typically implement a specific neuron model directly in the hardware. This architecture thus offers the promise of being able to run simulations with arbitrary neuron models.

SpiNNaker has only recently started to become available for use and so far only a few types of STDP rules have been implemented on SpiNNaker. Three STDP rules are currently provided by the SpiNNaker package [7], [15], [24]. One is a rule where weight updates depend only on presynaptic spike times, using the postsynaptic membrane potential as a surrogate for postsynaptic spike time information. Another is a standard "nearest neighbor" spike-pair rule, where weight changes occur whenever either the presynaptic or postsynaptic neuron spikes, with the change depending on the time since the other neuron spiked. The third rule implements an all-to-all spike-pair rule, but it is relatively inefficient. These rules are suitable for some STDP simulations, but for larger networks more efficient implementations are needed which can handle more general types of spike pair rules [20], or even more complex STDP learning rules that depend on other factors, such as synaptic traces [21], spacing of spike-triplets [22], or even spike-quadruplets [31]. In this paper we present an approach for efficiently implementing these classes of STDP rules. We will illustrate our approach specifically for synaptic trace-based rules, but the same method can be applied to any of these types of rules.

One challenge of the SpiNNaker simulation framework is that the synaptic data is easily accessible only at the times when a presynaptic spike event arrives [24], shown in figure 1 by grey vertical bars. However, due to axonal delay, the arrival of the biological spike at the simulated synapse will only happen after the delay has passed. This means that when the synapse code is triggered, it is still too early to process the spike that triggered it. For example, looking at the fourth presynaptic spike in figure 1, we can see that *at the synapse*, this presynaptic spike (dotted red line) occurs just *after* a postsynaptic spike (dotted blue line), and this timing can be critical for the STDP rule. However, at the time of processing, this postsynaptic spike *has not yet occurred*, even at the soma, and so it is clear that any processing of the presynaptic spike at this time would be premature.

Therefore on SpiNNaker it is not possible to implement STDP rules in a naive manner, *i.e.* by performing the weight updates for presynaptic spikes at the time the presynaptic spike event is received. Instead an approach similar to the deferred-event model presented in [24] has to be used, in which spike times are stored for deferred processing at the time of the next event following them. The implementation here uses the deferred-event approach for both presynaptic and postsynaptic spikes, allowing the incorporation of both axonal and dendritic delays. Compared with the three STDP rules provided with the SpiNNaker package, our approach is much more flexible, as well as being more efficient than the two rules which use postsynaptic spike times.

In the next sections we will explain in detail how our approach works in the context of the SpiNNaker system. We also evaluate both the speed and accuracy of our approach as compared with previous approaches. To demonstrate the type of simulation enabled by our method, we present two applications: using STDP to quickly balance excitation and inhibition, and using STDP to learn a temporal sequence.

Peter U. Diehl and Matthew Cook are with the Institute of Neuroinformatics, Swiss Federal Institute of Technology Zurich and University Zurich, Zurich, Switzerland (email: peter.u.diehl@gmail.com).

This work was supported by the SNF Grant 200021-143337 "Adaptive Relational Networks".

Our software is open source and is publicly available.

978-1-4799-1484-5/14/$31.00 ©2014 IEEE

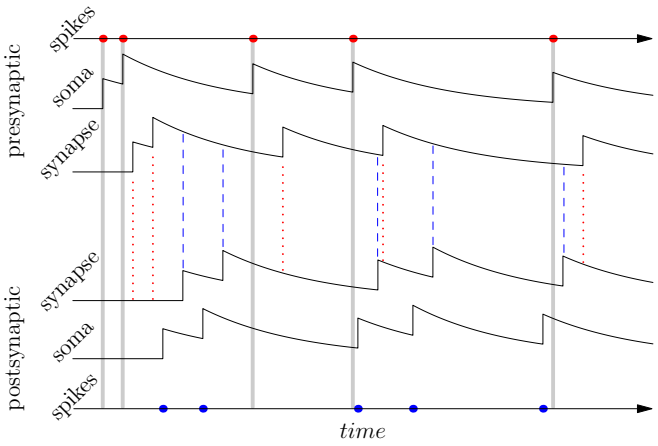

Fig. 1. For one synapse, this shows the relation between pre and postsynaptic somatic spike times (red and blue dots), spike traces at the somata (outer jagged lines), spike traces at the synapse (inner jagged lines), times relevant for computing changes in the synapse weight in STDP theory (vertical red dotted lines and blue dashed lines), and the times at which the synapse can be updated in the SpiNNaker framework (vertical gray lines). The spike trace is a function used in many STDP rules, which decays exponentially but increases whenever there is a spike. The presynaptic somatic trace (topmost jagged line) can be seen to increase with every presynaptic spike (red dot), and decays at all other times. The same trace can also be computed at the synapse (second jagged line), where it is the same but shifted by the "axonal delay," the time it takes the spike to go from the neuron to the synapse. The lower three graphs similarly show, from the bottom up, the postsynaptic spikes, somatic trace, and synaptic trace. STDP rules update the weight at the synapse according to the timing of the pre and postsynaptic spikes, shown in the middle as dotted red and dashed blue vertical lines. The SpiNNaker routing system delivers the spike within microseconds, orders of magnitude faster than biological axonal or dendritic delays, which are generally on the order of 0-10ms [17]. This means that the spike arrives at the postsynaptic cell essentially at the time of the presynaptic somatic spike (vertical gray lines). Since the arrival of the spike is what triggers the brief availability of synaptic data for processing, the code must then update the synapse for all the red and blue lines that have occurred in the interval between the previous and current gray lines.

II. STDP

Long-term synaptic plasticity (LTP) mechanisms, *i.e.* mechanisms that change the synaptic weight in a way that lasts for more than a few seconds, have been found to be necessary for memory and learning in biology [19]. One of the most studied LTP models is spike-timing dependent plasticity (STDP) [2], which has been demonstrated in simulation to be capable of many useful types of learning [1], [23], [27], [29]. The main idea of STDP is that changes to the weight of a synapse occur based on the relative timing of pre and postsynaptic spikes. How exactly the weight is changed depends on the specific STDP rule being used, and many such rules have been found in neuroscience experiments [5], [10], [18] and modeled in the literature [4], [12], [21].

The most general form of an STDP model is that there is some set of state information stored at the synapse, which is affected by the occurrance of a pre or postsynaptic spike, and which can also change passively with time. This state information suffices for calculating the EPSP produced by each incoming spike. Our implementation approach supports any rule of this form.

One broad class of STDP rules fitting this general form is the class of *trace-based* rules [20], [21]. These can be modeled by the application of a weight change function

$$w \mathrel{+}= \eta \cdot f_{pre}(T_{post}, w) \tag{1}$$

whenever there is a presynaptic spike, and

$$w \mathrel{+}= \eta \cdot f_{post}(T_{pre}, w) \tag{2}$$

whenever there is a postsynaptic spike, where η is the learning rate, and T_{pre} and T_{post} are the *traces*, *i.e.* filtered versions of the pre and postsynaptic spike trains (see fig. 1). The functions f_{pre} and f_{post} are typically mathematically fairly simple [21].

The synaptic traces decay exponentially with time constants τ_{pre} and τ_{post}, and are increased at the time of a pre or postsynaptic spike either *by* a constant value, allowing efficient implementation (due to linear summation of effect) of an "all-to-all" rule, or *to* a constant value, yielding a "nearest pair" rule [20].

We will describe our SpiNNaker implementation for a trace-based rule, which has a relatively simple state (consisting of a scalar weight and two traces) and simple dynamics (consisting of the spike-induced changes to the traces and weight, and the exponential decay of the traces). More complicated state and dynamics can easily be substituted if desired.

III. SpiNNaker

In recent years neuromorphic systems have gained interest as a tool for simulating SNNs. Different projects include both real-time [14] and accelerated-time ([32], [3]) analog VLSI neuromorphic circuits, including the Neurogrid [26] and the BrainScaleS project [9]. In contrast to these, the SpiNNaker hardware does not take the pure neuromorphic approach of using analog circuits to emulate ionic neural currents, but rather uses a large number of conventional digital processors, interconnected with a unique communication system optimized for very fast transmission of small packets (5 or 9 bytes, including routing information and control bits), intended to be used like spikes in biological neural networks.

A SpiNNaker system consists of one or more SpiNNaker boards, each containing up to 48 SpiNNaker chips, and each chip has 16 ARM9 processors running at 200MHz available for use in a neural simulation [16]. Each core is equipped with 32KB of instruction memory and 64KB of fast-access data memory. Additionally, as shown in figure 3, each SpiNNaker chip contains 128MB of SDRAM which is shared by the 16 simulation cores on the chip. This SDRAM is slow compared to the local core memory, using an interrupt-based protocol to allow cores to store and retrieve memory blocks asynchronously.

The SpiNNaker software system includes a convenient interface for implementing neural networks [11]. First, the end-user describes the network in a high-level language like PyNN [8] or Nengo [28]. Algorithmic parts of the

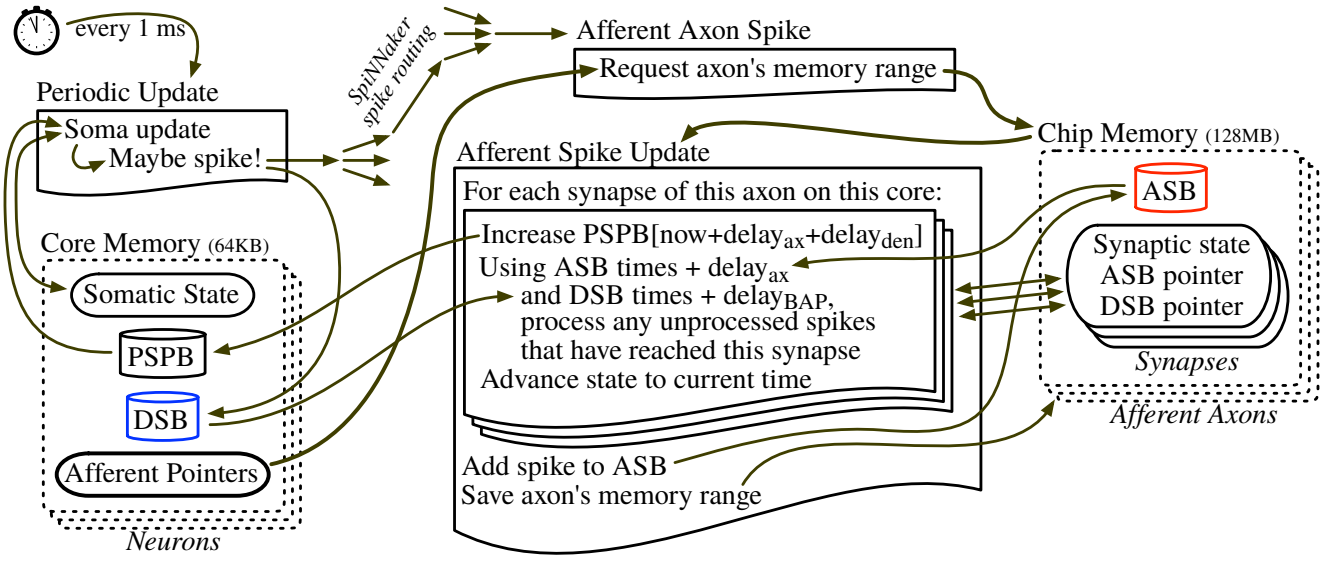

Fig. 2. Flow of information in a SpiNNaker simulation including the generalized STDP implementation. SpiNNaker is an event-driven system with two main types of events: a timer tick and a presynaptic event. The timer tick (which is asynchronous among different processors) triggers an update of the state of the neuron. This update can elicit the firing of a spike. The spike is then routed to the destination chips. For every incoming spike the synaptic data is requested using DMA. Subsequently the corresponding data is fetched from the SDRAM and is then processed by the receiving core. This processing includes calculating and adding the postsynaptic potential and performing STDP-related updates. The dendritic spike buffer (DSB, blue) and axonal spike buffer (ASB, red) and the parts of the code relating to them have been introduced to enable the use of the trace-based STDP rule (see also figure 3).

model, such as a leaky integrate-and-fire algorithm or STDP algorithm, are written in C and can be referred to from the high-level network description via a binding. Several common neural models are provided in the SpiNNaker package, and more can be added by the user. Then, using the PArtition and Configuration MANager (PACMAN), the high-level description and C code are used to automatically generate assignments of neurons to cores, routing tables for sending spikes according to the connectivity of the network, and code to run on the SpiNNaker cores.

Neural simulations are processed in an event-driven manner as shown in figure 2. Each core receives a timer event once per millisecond, triggering an update of all the neurons on that core. In a typical neural model, excitatory and inhibitory post-synaptic potentials (EPSPs and IPSPs) are scheduled by being placed into the PSP buffer (PSPB, shown on the left in figure 2), in which they are summed and eventually added to the membrane voltage, which also decays due to the leak. If the spiking threshold is reached, the membrane voltage is reset and a spike is sent out to the SpiNNaker routing system. The only information carried by the spike is the ID of its source neuron, and this is used by the routing tables to send the spike to all cores containing at least one of its postsynaptic target neurons. As the spike travels from chip to chip, the routing tables can make it branch and travel in multiple directions to multiple targets, similar to the branching in a biological axon.

When a spike arrives to a core containing a synaptic target, the routing system sends an interrupt to that core. The interrupt routine (shown at the top center of figure 2) does nothing beyond making a request to the memory system to load all synapses from that axon. This is done with the aid of the "afferent pointers" also shown towards the bottom left of the data structures shown in figure 3. In fact, the structure and use of the afferent pointer data is more complicated than shown in the figures, since this pointer information needs to have a small memory footprint even when there are tens of thousands of axons projecting to the core. However, for our purposes we can ignore the internal complexity and simply treat it as an opaque data structure from which it is possible to extract the correct pointer.

When the chip's memory system has loaded the requested afferent axon data structure (shown on the center right of figure 3) from the SDRAM into the core memory, it sends a signal to that core, triggering an interrupt handler which just enqueues the processing of this data structure. This processing is shown in the large center box of figure 2. Among other actions, based on the synapse weight it adds an EPSP or IPSP into the PSP buffer (PSPB), so that it will be added to the soma's membrane potential after the correct delay. This is also the place where the STDP rule must be implemented.

IV. Implementing an STDP Rule

The limited availability of the synaptic data on SpiNNaker imposes constraints on the implementation of the STDP rule, *i.e.* the postsynaptic event has to be processed at the moment a presynaptic spike arrives. Therefore instead of performing the weight update the moment a postsynaptic spike is fired, the postsynaptic spike time is stored. Since it is possible that multiple postsynaptic spikes occur before the next presynaptic spike arrives, it might be necessary to store

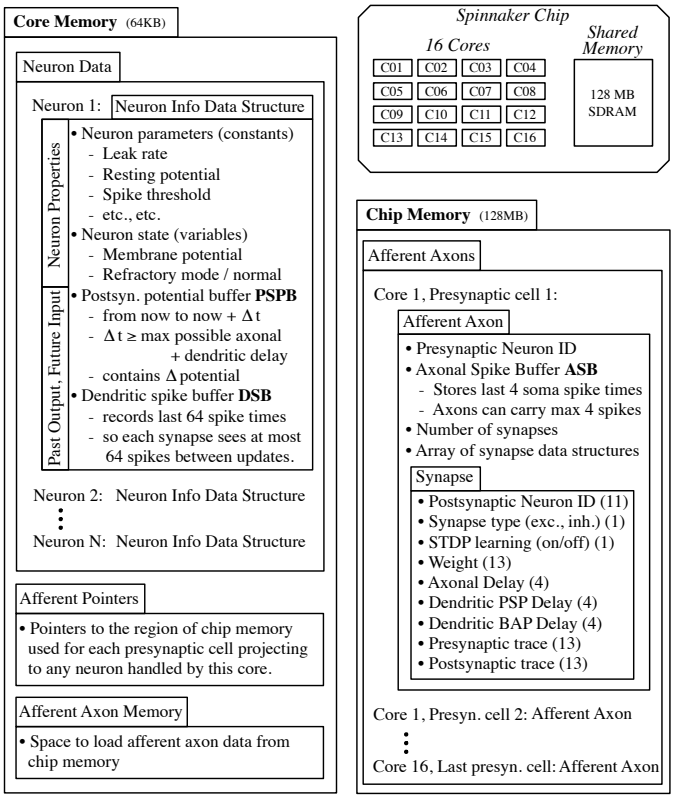

Fig. 3. Data structures used in SpiNNaker simulations, including structures used by the trace-based STDP implementation. Although a SpiNNaker chip contains 18 cores, only 16 are shown since only 16 are available for neural simulations. The chip memory (SDRAM) is shared across all cores, whereas the core memory (fast local memory) is only available on the corresponding core. In order to implement trace-based STDP rules, we added the DSB and ASB and expanded the state of the synapse to include the traces themselves. We also included dendritic delays, both for postsynaptic potentials (PSP delay) and for backpropagating action potentials (BAP delay), to allow more detailed simulations. The numbers in the synapse data structure in the lower right represent the number of bits used to store the given data.

the time of more than one postsynaptic spike. To achieve this, we employ a circular buffer for each neuron on the core to store postsynaptic spike times, which is stored on the local data memory. Note that the size of postsynaptic data is independent of the number of synapses, *i.e.* it depends only on the number of neurons on one core and on the buffer size. In contrast, the synaptic data (the state of the synapse and the time of the last presynaptic spikes) is stored on the SDRAM since this could take too much space on the local data memory, depending on the number of synapses.

The actual weight update takes place when a presynaptic spike arrives, see algorithm 1. The main idea of the algorithm is to process the pre-/postsynaptic events in chronological order for each synapse of the afferent axon. At each time a spike arrives at the synapse, the synaptic state (including the weight) is updated. This way we also can take axonal and dendritic delays into account. Note that dendritic delays are in the order of tens of milliseconds from the soma to the dendrites [17] and therefore maybe important to consider. For implementing a trace-based rule (see section 2) it suffices

Algorithm 1: executed at presyn. spike arrival at the core

```
1  for each synapse on core with corr. presyn. neuron do
2      find next presynaptic spike (incl. axonal delay);
3      find next postsynaptic spike (incl. dendritic delay);
4      while next spike is before current time do
5          if next spike is presynaptic then
6              update synaptic state (including weight);
7              find next presyn. spike (incl. axonal delay);
8          end
9          if next spike is postsynaptic then
10             update synaptic state (including weight);
11             find next postsyn. spike (incl. dendr. delay);
12         end
13     end
14     update synaptic state to current time;
15 end
16 store current time in presynaptic spike time buffer;
```

that the synaptic state consists of a presynaptic trace, a postsynaptic trace and the synaptic weight.

The for-loop of the algorithm iterates over all synapses on the core which correspond to the afferent axon of the arriving spike. It is important to keep in mind that the stored presynaptic spike time (without the synapse-specific axonal delay) is referring to the time the spike was fired at the presynaptic soma, not to the time the spike arrived at the synapse, see figure 1. In order to calculate the time a presynaptic spike arrives at the synapse, we have to add the axonal delay to the presynaptic spike time and to calculate the time a postsynaptic spike arrives at the synapse we have to add the dendritic delay (lines 2–3 in algorithm 1). The pre-/postsynaptic spike times at the soma are fetched from the SDRAM/local memory, see figure 2. In the presented implementation the spike time buffers are searched backwards. Since on average only one pre-/postsynaptic spike happened since the last presynaptic spike, this backward search is very efficient. However, if the network contains some highly active and some almost inactive neurons, it might be more desirable to have constant buffer access times, which can be achieved by storing an additional pointer for each spike time buffer to indicate the last processed spike. The while loop in lines 4–13 is executed as long as there are spikes which arrived at the synapse before the current time. Depending if the next spike is a presynaptic or a postsynaptic one, the corresponding update function is called. In both cases, the update of synaptic state (including the synaptic weight) is based on the time passed since the last update of the synaptic state. Those updates in lines 6, 10 and 14 are the only parts of the code which change for different STDP rules. If a pre-/postsynaptic spike is being processed, the time of the next pre-/postsynaptic spike is fetched from the corresponding spike time buffer and the axonal/dendritic delay is added. As the last step inside the loop, the state of the synapse is updated to the current time. This update of the synaptic state

is done to account for the changes of synaptic state between the last spike which arrived at the synapse and the current time. Finally, the current time is stored in the presynaptic spike time buffer, since the current time is the presynaptic spike time at the soma.

V. Evaluation

A. Speed

Chapter 6 in [6] shows an analysis of the complexity of previously implemented STDP mechanisms on SpiNNaker. One of the results is that the STDP mechanism requires at least an order of a magnitude more clock cycles than updating the membrane potential of a neuron. Therefore it is highly desirable to keep the computational complexity low since it is a significant proportion of the total required computation.

In this section we determine the costs of the implemented STDP mechanism for one core theoretically and empirically. The number of operations of pair-based STDP mechanisms depends on the number of presynaptic neurons per core n_{pre}, the number of postsynaptic neurons per core n_{post}, the firing rate of the presynaptic neuron f_{pre} and the firing rate of the postsynaptic neuron f_{post}, *i.e.* the number of presynaptic and postsynaptic events. Algorithm 1 is called once for each presynaptic event (f_{pre} times n_{pre}). The outer for-loop of algorithm 1 is executed once per postsynaptic neuron and the inner while-loop is executed once for each postsynaptic and presynaptic spike. Note that the number of presynaptic events which have to be processed per algorithm call is on average only one per presynaptic event and therefore the dependence on the number of presynaptic events is already accounted for by the number of calls of the algorithm. This means that the algorithm only needs a constant time to process every synaptic event. Additionally the algorithm uses only cost efficient operations such as addition, multiplication and bit operations.

We tested the performance of the implementation empirically by comparing the number of processed presynaptic and postsynaptic events per second as suggested in [25]. The resulting numbers are then compared to the number of pre- and postsynaptic events using the closest-pair and the all-to-all STDP implementation and to the results without applying any STDP rule, see figure 4. For this test we simulated 50 leaky integrate-and-fire neurons on one SpiNNaker core. Those neurons receive input from a varying number of presynaptic input neurons with a connection probability of 20%, *i.e.* each input neuron is connected to about 10 postsynaptic neurons. The input neurons were simulated on a different core than the postsynaptic neurons. The update rules we used are trace-based as explained in section 2, hence we will use the name trace rule in the following. We chose the update functions of the traces and the weight such that a conventional all-to-all STDP rule is simulated. For all simulations presented in this work, we used a postsynaptic spike time buffer size of 10. In figure 4 the number of synaptic events is plotted as a function of the number of input neurons, each with a

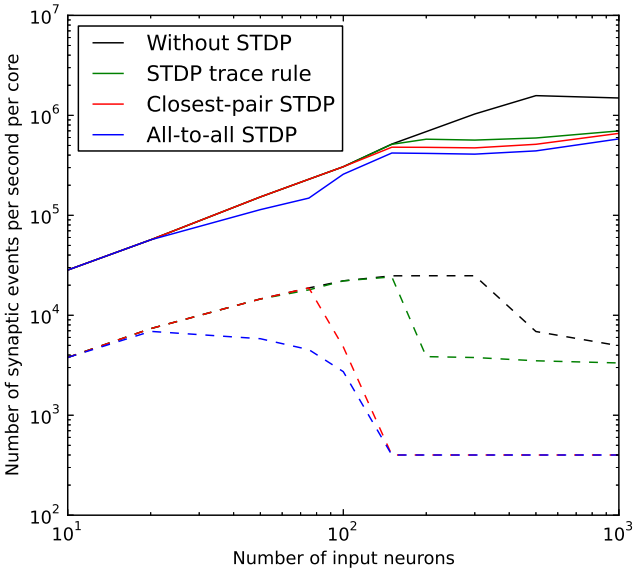

Fig. 4. Comparison of the performance of the STDP algorithm as a function of the number of the input neurons. Each input neuron has a probability of 20% to connect to a postsynaptic neuron. For the same number of input neurons the same connection pattern was used. Black lines denote the results without applying a STDP rule, green lines the results for applying the STDP trace rule presented here, red/blue lines show results of the performance of the SpiNNaker package closest-pair/all-to-all STDP rule. Solid lines represent the number of presynaptic events per second and dashed lines the number of postsynaptic events per second.

firing rate of $\approx$ 250 Hz. The number of presynaptic events is depicted by solid lines and the number of postsynaptic events by dashed lines. Colors indicate the applied STDP algorithm: blue for the SpiNNaker package all-to-all rule, red for the SpiNNaker package closest-pair rule, green for the presented trace rule and black for simulations without applying any STDP rule. Measurements without applying STDP serve as baseline measurement to know the upper limit if the cost of the STDP algorithm would be zero. The closest-pair STDP rule shows a significant decrease of postsynaptic events for more than 75 input neurons compared to the baseline. If the all-to-all rule is used, the number of events decreases for more than 20 input neurons. For the trace rule this decrease occurs if more than 150 input neurons are used. Similar differences in performance are also observed for different firing rates (data not shown). The reason for this drop is that the processing power limit of the core is reached and that we would need to distribute the simulation on more cores. Note that even without using STDP, many postsynaptic events are lost for more than 300 input neurons (this follows from the fact that the number of postsynaptic events should increase monotonically given the monotonically increasing input).

The limit to processing in the SpiNNaker system is typically not the number of neurons or even the number of synapses but rather the number of incoming spikes that a core must process within the one-millisecond window. With our method of STDP processing for trace-based rules, it

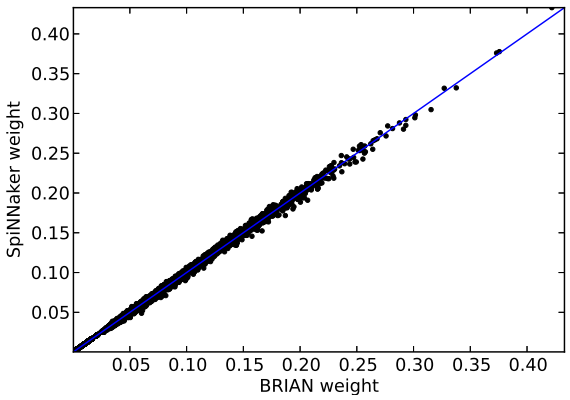

Fig. 5. Relation between the application of the STDP rule using BRIAN simulator and the implementation on SpiNNaker. Each black dot shows one weight pair. The blue line shows the identity, which represents the ideal result.

is possible to process around 500,000 incoming spikes per second per core (s/sc). For comparison, the nearest-pair rule provided with the SpiNNaker package can process around 200,000 s/sc, the all-to-all rule can process around 50,000 s/sc, and a plain leaky-integrate-and-fire simulation without STDP can process around 1,500,000 s/sc. If a simulation exceeds these bounds (for example by placing too many neurons on each core), then the simulation won't remain real-time, rendering the results meaningless, since in the SpiNNaker system the simulated timepoint of a spike is represented implicitly by the actual time it is produced rather than by an explicit timestamp.

B. Accuracy

There are a number of reasons why the simulations on SpiNNaker can differ from conventional simulations on usual computers. First of all, it is not guaranteed that a transmitted spike will be delivered, consequently no weight update will happen due to this spike. Furthermore other updates which would depend on the arrival of the spike are changed. A second reason is the asynchronous nature of SpiNNaker. The asynchronous update of the neurons can lead to a situation where a neuron receives a spike but has not yet stored the postsynaptic spike time. Therefore the postsynaptic trace would not be updated correctly and a wrong weight update would be performed. Additionally, it can happen that the implementation details (like fixed-point numbers) lead to differences. For example, typically it is necessary to compute an exponential function to update the synaptic state. To minimize computation, we precomputed an exponential function and stored it in the local memory. Since for the examples here this function is stored with little resolution (8 bit), there will be difference to traditional simulations with 32 or 64 bit precision. Another issue is if the firing rate of the pre-/postsynaptic neuron times the pre-/postsynaptic spike time buffer size is higher than the firing rate of the post-/presynaptic neuron. In this case the pre-/postsynaptic spike buffer will be full and pre-/postsynaptic spike times which have not been processed yet, will be forgotten. However, in the simulation presented in this section we did not use firing rates which would lead to such a forgetting of spikes. If the processor load is very high (see figure 4 for high number of input neurons and the explanation in section 3), the time to process incoming spikes can exceed 1 ms and thereby drastically change the results of the simulation, since the delay between the desired update time and the actual update increases as the simulation progresses. Such a high processor load should generally be avoided (also for simulations without STDP) by distributing the simulation across more cores.

For those reasons we compare the STDP trace rule implementation on SpiNNaker to a conventional neural network simulation in this section. In order to achieve this, we first simulate a neural network on SpiNNaker and afterwards simulate the same spiking pattern using the BRIAN simulator [13] with the same STDP trace rule. Figure 5 shows the final weights of the BRIAN simulation on the x-axis and the weights of the SpiNNaker simulation on the y-axis. The blue line depicts the ideal result, *i.e.* that the weights of both simulations are exactly the same. The mean error is ≈ 0.0013 with a mean weight of ≈ 0.057, which gives a mean error of $\approx 2.23\%$ of the weight. Over the whole range of weights, there is a good match of BRIAN and SpiNNaker simulation results, especially considering the asynchronous nature of SpiNNaker. The asynchronous update of the neurons will lead to a situation where a neuron receives a spike but has not yet updated the postsynaptic trace. Therefore a wrong weight update will be performed. In most cases this will just be the slight difference of the decay which results in a slight increase of the weight change, since for the update the function $F_{pre} = \eta(y_i - \alpha)$ was chosen (F_{pre} is explained in more detail in the application section). However it can also happen that a postsynaptic spike is not yet included in the trace, which will lead to a much bigger error, *i.e.* a decreased weight change for the chosen F_{pre}. Another reason for differences in the BRIAN and the SpiNNaker weights is the deferred-event like processing of the postsynaptic spikes on SpiNNaker. The presented STDP mechanism performs weight updates due to postsynaptic spikes at the moment the next presynaptic spike arrives at the corresponding synapse. This means that if no presynptic spike arrives before the end of the simulation, the postsynaptic spikes since the last presynaptic spike will not be processed. However, this unreliability of the timing is also apparent in biological systems and the possible neglect of the last postsynaptic spikes should not be important for a small learning speed η. A range of additional comparisons (data not shown) indicates that most of the differences arise from the fact that the exponential function is precomputed and stored on the SpiNNaker board with 8 bit precision. The comparison shown in figure 5 uses the usual clock-driven update of BRIAN to update the synaptic state with 64 bit precision. Even changing the precision of the BRIAN updates to 8 bit too does not give the same synaptic state updates since the

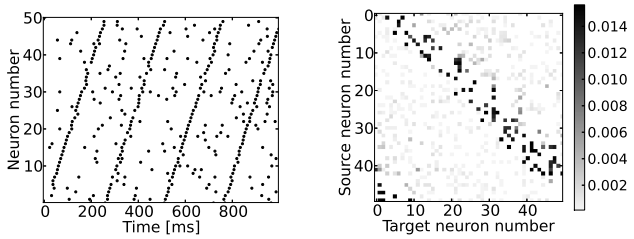

Fig. 6. Learning of excitatory weights. The left plot shows the spiking pattern of the excitatory neurons during training of the recurrent excitatory weights. The right plot shows the recurrent excitatory weights of the network after learning. Non-existing connections are shown in white. Existing connections range from light grey (weak connection) to black (strong connection). The initial weight was 0.005 for each synapse.

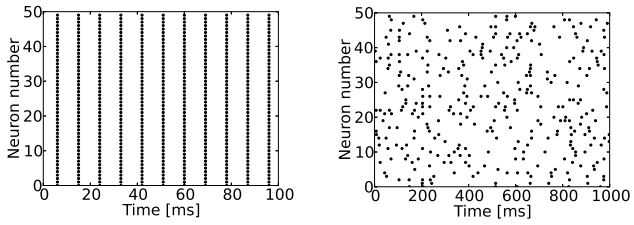

Fig. 7. Spiking pattern of the excitatory neurons. The left plot shows the initial state of the untrained network. The right plots shows the spiking pattern after one minute of training of the the inhibitory to excitatory weights.

SpiNNaker STDP implementation updates in a event-driven manner and is therefore more precise (even though it uses 8 bit precision, too). In case it is necessary to have a better precision of the updates, it would be possible to store a more precise precomputed function.

VI. Applications

To show some examples, we demonstrate that the presented approach can be used to learn timing patterns. For this example we use one presynaptic trace, one postsynaptic trace and the synaptic weight to represent the state of a synapse. The traces are exponentially decaying over time and are increased by a constant if a spike occurs (see section 2). This models an all-to-all spike timing dependence within the timing window of the STDP rule. For a weight update due to a presynaptic spike we used the function

$$F_{pre}(y_i, w_{ij}) = -\eta \cdot y_i,$$

where y_i is the trace of the postsynaptic neuron i. For a weight update due to a postsynaptic spike we used

$$F_{post}(x_j, w_{ij}) = \eta \cdot x_j,$$

where x_i is the trace of the presynaptic neuron j. This models a typical timing dependency as presented in [2]. The firing pattern is the following: neuron $(n+1)$ modulo 50 (the total number of neurons) receives excitatory input 5 ms after neuron n, see upper plot in figure 6. Additionally each neurons receives some noise in form of Poisson spiking neurons with a firing rate of 15 Hz. Using those functions for the STDP trace rule, the network learns that if n'th neuron fires the next neuron $n+1 \ mod \ 50$ is likely to fire soon. This can be seen in the recurrent excitatory weights of the network, shown in figure 6, *i.e.* all neurons have strong excitatory connections to their "successors". Note that the connections between neurons which are not firing closely in time are weakened, *i.e.* they are close to zero although all synapses were initialized with a weight of 0.005.

As a second example we show the automatic balancing of excitation and inhibition. The network we use consists of 50 excitatory and 50 inhibitory neurons, which receive a constant current as excitatory input. Instead of learning recurrent excitatory weights, the weights from inhibitory to excitatory neurons are learned. For this we use the STDP rule presented in [30]

$$F_{pre}(y_i, w_{ij}) = \eta(y_i - \alpha),$$

where α is a parameter which controls the firing rate of the postsynaptic neuron and

$$F_{post}(x_j, w_{ij}) = \eta \cdot x_j.$$

The state of the synapse is again represented by the pre- and postsynaptic trace and the weight, using the same update rule as before. The initial firing pattern is shown in figure 7 (a), it is highly regular and synchronous. After 10 seconds of training the pattern changes drastically, see figure 7 (b). Note that the time scale in (a) and (b) is changed to improve the visibility of the pattern. The firing pattern after learning the inhibitory to excitatory weights is irregular and asynchronous due to the detailed balance of the inhibition [30].

VII. Discussion

We have presented an approach for implementing STDP rules on SpiNNaker in a more general and more efficient way than previous methods. Like the STDP rules which have been implemented previously in SpiNNaker, we shifted the processing of the pre and postsynaptic spikes to the moment of the arrival of the next presynaptic spike. Therefore the new state of the synapse is computed at each presynaptic spike which arrives at the synapse. Using the presented approach, any pair-based STDP rule can be modeled. To do this, it is only necessary to store one presynaptic trace and one postsynaptic trace per synapse and the recent spikes of each neuron. Also only a small constant time is needed to process each synaptic event. This way of implementing pair-based STDP rules leads to learning rules which are much more efficient than the all-to-all and the closest-pair STDP rule of the SpiNNaker package.

Unlike conventional BRIAN simulations, which are slower for larger networks, SpiNNaker simulates networks in real time, independent of the network size. This gives SpiNNaker a speed advantage for large networks (the total activity rate is the limiting factor, given by figure 4 multiplied by the number of cores: 768 times the number of SpiNNaker boards). However, due to the inherently non-deterministic, asynchronous nature of the system and the limited precision, SpiNNaker simulation results can differ slightly from

BRIAN results, and can even differ minorly from run to run. Nonetheless, it is reassuring that the final trained weights are still within 3% of those trained in a BRIAN simulation.

One of the main advantages of the approach presented here is its versatility. To implement a new STDP rule, the user only needs to define new synaptic state update rules and add the synaptic state data structure. For example, to implement a triplet rule [21], the only thing which would need to be changed compared to the basic trace rule is to keep track of a second postsynaptic trace (with a different time constant) and change F_{post} to be dependent on the second postsynaptic trace. It is also easy to introduce voltage dependence by making F_{pre} or F_{post} be a function of the postsynaptic membrane voltage. Generally it is possible to use the presented framework for all STDP rules which are mathematically tractable. The necessity for expressing the rule in a closed form is due to the calculation of the change between the last and the current spike. However, even if there is no closed form, then instead of "jumping" from spike to spike, it is still possible (at the cost of decreased performance) to just simulate every single time step.

VIII. Acknowledgments

We would like to thank the Advanced Processors Technologies Research Group at University of Manchester for providing the SpiNNaker hardware and the SpiNNaker Software. We are also thankful for the SpiNNaker Workshop at University of Manchester in August 2013, which was particularly helpful to get first hands-on experience with SpiNNaker. Especially we thank Sergio Davies, Steve Temple and Stephen Furber for their helpful tips and fruitful discussions.

References

[1] L.F. Abbott and S. Song, *Temporally asymmetric hebbian learning, spike timing and neuronal response variability*, Advances in neural information processing systems **11** (1999), 69–75.

[2] G.Q. Bi and M.M. Poo, *Synaptic modifications in cultured hippocampal neurons: dependence on spike timing, synaptic strength, and postsynaptic cell type*, The Journal of Neuroscience **18** (1998), no. 24, 10464–10472.

[3] A.S. Cassidy, P. Merolla, J.V. Arthur, S.K. Esser, B. Jackson, R. Alvarez-Icaza, P. Datta, J. Sawada, T.M. Wong, ... Feldman, V., and D.S. Modha, *Cognitive computing building block: A versatile and efficient digital neuron model for neurosynaptic cores*, Neural Networks (IJCNN), The 2013 International Joint Conference on, IEEE, 2013, pp. 1–10.

[4] C. Clopath, L. Büsing, E. Vasilaki, and W. Gerstner, *Connectivity reflects coding: a model of voltage-based stdp with homeostasis*, Nature neuroscience **13** (2010), no. 3, 344–352.

[5] Y. Dan and M.M. Poo, *Spike timing-dependent plasticity of neural circuits*, Neuron **44** (2004), no. 1, 23–30.

[6] S. Davies, *Learning in spiking neural networks*, Ph.D. thesis, University of Manchester, 2013.

[7] S. Davies, F. Galluppi, A.D. Rast, and S.B. Furber, *A forecast-based stdp rule suitable for neuromorphic implementation*, Neural Networks **32** (2012), 3–14.

[8] Andrew P Davison, Daniel Brüderle, Jochen Eppler, J. Kremkow, E. Muller, D. Pecevski, L. Perrinet, and P. Yger, *Pynn: a common interface for neuronal network simulators*, Frontiers in neuroinformatics **2** (2008).

[9] J. Fieres, J. Schemmel, and K. Meier, *Realizing biological spiking network models in a configurable wafer-scale hardware system*, Neural Networks, 2008. IEEE International Joint Conference on, IEEE, 2008, pp. 969–976.

[10] R.C. Froemke and Y. Dan, *Spike-timing-dependent synaptic modification induced by natural spike trains*, Nature **416** (2002), no. 6879, 433–438.

[11] F. Galluppi, S. Davies, A. Rast, T. Sharp, L.A. Plana, and S.B. Furber, *A hierachical configuration system for a massively parallel neural hardware platform*, Proceedings of the 9th conference on Computing Frontiers, ACM, 2012, pp. 183–192.

[12] W. Gerstner, R. Kempter, J.L. van Hemmen, and H. Wagner, *A neuronal learning rule for sub-millisecond temporal coding*, Nature **383** (1996), no. 6595, 76–78.

[13] D. Goodman and R. Brette, *Brian: A simulator for spiking neural networks in python*, Frontiers in neuroinformatics **2** (2008), 5.

[14] G. Indiveri, E. Chicca, and R. Douglas, *A vlsi reconfigurable network of integrate-and-fire neurons with spike-based learning synapses*, (2004).

[15] X. Jin, A. Rast, F. Galluppi, S. Davies, and S.B. Furber, *Implementing spike-timing-dependent plasticity on spinnaker neuromorphic hardware*, Neural Networks (IJCNN), The 2010 International Joint Conference on, IEEE, 2010, pp. 1–8.

[16] M. Khan, D.R. Lester, L.A Plana, A. Rast, X. Jin, E. Painkras, and S.B. Furber, *Spinnaker: mapping neural networks onto a massively-parallel chip multiprocessor*, Neural Networks, 2008. IJCNN 2008.(IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on, IEEE, 2008, pp. 2849–2856.

[17] C. Koch, *Biophysics of computation: information processing in single neurons*, Oxford university press, 2004.

[18] J. Lisman and N. Spruston, *Postsynaptic depolarization requirements for ltp and ltd: a critique of spike timing-dependent plasticity*, Nature neuroscience **8** (2005), no. 7, 839–841.

[19] S.J. Martin, P.D. Grimwood, and R.G.M. Morris, *Synaptic plasticity and memory: an evaluation of the hypothesis*, Annual review of neuroscience **23** (2000), no. 1, 649–711.

[20] A. Morrison, A. Aertsen, and M. Diesmann, *Spike-timing-dependent plasticity in balanced random networks*, Neural Computation **19** (2007), no. 6, 1437–1467.

[21] A. Morrison, M. Diesmann, and W. Gerstner, *Phenomenological models of synaptic plasticity based on spike timing*, Biological Cybernetics **98** (2008), 459–478.

[22] J.-P. Pfister and W. Gerstner, *Triplets of spikes in a model of spike timing-dependent plasticity*, The Journal of neuroscience **26** (2006), no. 38, 9673–9682.

[23] R.P. Rao and T.J. Sejnowski, *Spike-timing-dependent hebbian plasticity as temporal difference learning*, Neural computation **13** (2001), no. 10, 2221–2237.

[24] A. Rast, X. Jin, M. Khan, and S.B. Furber, *The deferred event model for hardware-oriented spiking neural networks*, Advances in Neuro-Information Processing, Springer, 2009, pp. 1057–1064.

[25] T. Sharp and S.B. Furber, *Correctness and performance of the spinnaker architecture*, Neural Networks (IJCNN), The 2013 International Joint Conference on, IEEE, 2013.

[26] R. Silver, K. Boahen, S. Grillner, N. Kopell, and K.L. Olsen, *Neurotech for neuroscience: unifying concepts, organizing principles, and emerging tools*, The Journal of Neuroscience **27** (2007), no. 44, 11807–11819.

[27] S. Song, K.D. Miller, and L.F. Abbott, *Competitive hebbian learning through spike-timing-dependent synaptic plasticity*, Nature neuroscience **3** (2000), no. 9, 919–926.

[28] T.C. Stewart, B. Tripp, and C. Eliasmith, *Python scripting in the nengo simulator*, Frontiers in neuroinformatics **3** (2009).

[29] M.C.W. Van Rossum, G.Q. Bi, and G.G. Turrigiano, *Stable hebbian learning from spike timing-dependent plasticity*, The Journal of Neuroscience **20** (2000), no. 23, 8812–8821.

[30] T.P. Vogels, H. Sprekeler, F. Zenke, C. Clopath, and W. Gerstner, *Inhibitory plasticity balances excitation and inhibition in sensory pathways and memory networks.*, Science (New York, N.Y.) **334** (2011), no. 6062, 1569–573.

[31] H.X. Wang, R.C. Gerkin, D.W. Nauen, and G.Q. Bi, *Coactivation and timing-dependent integration of synaptic potentiation and depression*, Nature neuroscience **8** (2005), no. 2, 187–193.

[32] J.H.B. Wijekoon and P. Dudek, *Compact silicon neuron circuit with spiking and bursting behaviour*, Neural Networks **21** (2008), no. 2, 524–534.