FPGA Accelerated Simulation of Biologically Plausible Spiking Neural Networks

David B. Thomas and Wayne Luk
Department of Computing
Imperial College
London, England
{dt10,wl}@doc.ic.ac.uk

***Abstract*—Artificial neural networks are a key tool for researchers attempting to understand and replicate the behaviour and intelligence found in biological neural networks. Software simulations offer great flexibility and the ability to select which aspects of biological networks to model, but are slow when operating on more complex biologically plausible models; while dedicated hardware solutions can be very fast, they are restricted to fixed models. This paper uses FPGAs to achieve a compromise between model complexity and simulation speed, such that a fully-connected network of 1024 neurons, based on the biologically plausible Izhikevich spiking model, can be simulated at 100 times real-time speed. The simulator is based on a re-usable interconnection architecture for storing synapse weights and calculating thalamic input, which makes use of the large number of available block-RAMs and huge amounts of fine-grain parallelism. The simulator achieves a sustained throughput of 2.26 GFlops in double-precision, and a single Virtex-5 xc5vlx330t without off-chip storage running at 133MHz is 16 times faster than a 3GHz Core2 CPU, and 1.1 times faster than a single-precision 1.2GHz 30-core GPU.**

I. INTRODUCTION

Artificial neural networks are of wide interest and applicability, both as a practical tool in areas such as pattern recognition, and as a means of developing an understanding of the brains of animals and humans and ultimately increasing our understanding of intelligence and consciousness. There have been many successes in practical applications, using simple neuron models and training rules to achieve specific tasks, but very little success in the use of artificial network to plausibly model the high-level behaviours of brains - we are still a long way from artificial intelligence!

Part of the problem is that we still do not know exactly what it is about biological neural networks that gives them high-level capabilities, such as the ability to learn. Although the low-level electrical and chemical properties of individual neurons and synapses are well understood, it is not clear which of these properties are required, and which are just "implementation details". Is it the number of neurons? The number of synapses? Axonal delay? Neuron transition function?

Software can be used to model neural networks with a given subset of potentially interesting properties, but is often too slow to allow researchers to observe the changes in networks over long simulations, or to observe the statistical properties of many different networks. Hardware implementations of neural networks are an attractive proposition, as they can be designed to implement the selected properties of the network exactly. However, this still raises the question: which properties are the important ones that should be modelled in the dedicated hardware?

FPGAs offer an interesting alternative to software and custom hardware, as different biological features can be added or removed according to the requirements of the researcher, while also still offering at least some of the speed-up provided by hardware. In this paper we examine the acceleration of densely connected spiking neural networks, using a biologically plausible neuron model. Our key contributions are:

- An architecture for simulating fully-connected spiking neural networks in FPGAs, which uses their fine-grain parallelism and high bandwidth local memories to provide an efficient and flexible simulator, providing a constant speed-up over real-time, independent of firing activity.
- An evaluation of the FPGA implementation for fully-connected networks with 1024 neurons, showing that a Virtex-5 xc5vlx330t at 133MHz provides a sustained 2.26 GFlops in double-precision, which is 16 times faster than a 3GHz Core2 CPU, and 1.1 times faster than a 1.2GHz 30-core GPU operating in single-precision.

II. BACKGROUND AND GOALS

Figure 1 provides a very abstract overview of a neural network. On the left are shown a set of *neurons*, each of which has an electrical output which occasionally produces voltage spikes, or *spike trains*. Each neuron has a set of *synapses* (inputs), which are connected to the output of other neurons via *axons* (wires). Each synapse has a weight, which it uses to scale the incoming spike train. The scaled spike-trains from the synapses are then summed to provide the overall *thalamic input* for the neuron. The neuron then combines the thalamic input and its current state to decide

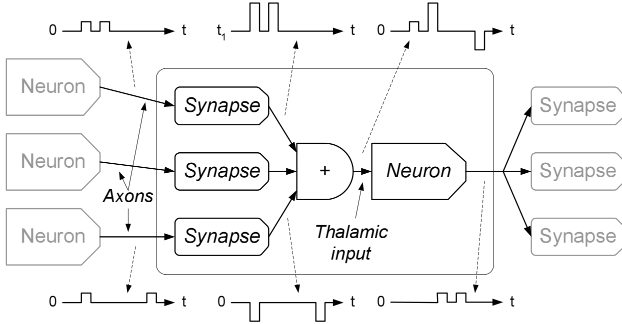

Figure 1. Overview of a spiking neural network.

how to modify its state, and what kind of spike train to produce.

Depending on the type of neuron, the state of the neuron, and the input the neuron has just received, the characteristics of the spike train will change: spiking modes include phasic spiking, where a single isolated spike is fired; tonic spiking, where single spikes are fired at regular intervals; and tonic bursting, where small bursts of spikes are fired at regular intervals [2].

The current prevailing theory is that it is the type and timing of these spike trains which encodes communications between neurons within the brain, so it is critical that any artificial network should be able to replicate these different types of spiking. Another point of broad agreement is that at least some of the power of biological networks comes from the sheer level of connectivity: typically each neuron has around 1000 synapses, i.e. each neuron is connected to the output of 1000 other neurons. The synapses also seem to be related to memory and learning: over time the weight of each synapse changes in response to firings within the network, reinforcing some connections, and reducing others.

In this paper we focus on the development of systems which can replicate these three phenomena:

- Biologically plausible spiking behaviour.
- Thousands of neurons, millions of synapses.
- Run-time modifiable per-synapse weights.

In particular we are interested in providing end-users with hyper real-time simulations of neural networks. This allows researchers to investigate statistical properties of networks, by simulating the same network many times and identifying common behaviour between the runs. It also allows for iterative modification of networks, where local network features are adjusted to explore the effects on local and global behaviour.

What we are *not* attempting to capture are the effects of large networks: our aim is to enable the investigation of small-scale phenomenon related to spiking and bursting behaviour, minimising the restrictions on the user. Supporting large networks involves restricting either the types of networks that can be supported, the accuracy of the simulation, the simulation speed, or all three, and is beyond the scope of this paper.

III. Neural Network Model

We will now provide a basic framework for describing the types of network in which we are interested, before describing in more detail the specific biologically plausible spiking network we implement.

Our starting assumption is that we are dealing with relatively small (around 1000 neurons), dense networks. In particular we assume that the network is fully connected: every neuron has a synapse for every other neuron in the network. This means that a network with n neurons has n^2 synapses.

Because the network is fully connected, the synapses can be described using a matrix $\mathbf{W}$, which is an $n \times n$ matrix of real numbers, where $\mathbf{W}[i,j]$ describes the scaling applied to the output of neuron j before feeding it to the input of neuron i. So the row $\mathbf{W}[i,:]$ describes the synapse weights at the inputs of neuron i, while the column $\mathbf{W}[:,j]$ describes the weights that will be applied to the output of neuron j whenever it produces a spike.

The neurons are described in terms of two features: fixed neuron parameters, which are neuron properties which do not change, or at least change over a large time-scale; and neuron state, which contains properties which vary rapidly in response to the incoming thalamic input. For the moment we will use an abstract model of neuron dynamics, using the set $\mathbb{P}$ to represent the set of possible neuron parameters, and $\mathbb{S}$ for the set of neuron states. The n element column vector $\vec{\mathbf{c}}$ (where $\vec{\mathbf{c}}_i \in \mathbb{P}$) describes the fixed parameters of the neurons.

There are two components needed to describe the dynamic state of the network. One is the previous state of each neuron, an n element column vector $\vec{\mathbf{s}}$, with $\vec{\mathbf{s}}_i \in \mathbb{S}$. The other is the set of spikes just produced by the neurons, a boolean column vector $\vec{\mathbf{f}}$, where $\vec{\mathbf{f}}_i = 1$ means that neuron i has just produced a spike. From the combination of $\vec{\mathbf{f}}$ and the neuron weights $\mathbf{W}$ we can calculate the current vector of thalamic stimuli:

$$\vec{\mathbf{i}} = \mathbf{W} \times \vec{\mathbf{f}} \tag{1}$$

The final piece needed is the neuron step function f, which takes as input the neuron parameters, previous neuron state and thalamic input, and produces as output the next neuron state and whether the neuron fired:

$$f : (\mathbb{P}, \mathbb{S}, \Re) \mapsto (\mathbb{S}, [0,1]) \tag{2}$$

We can now advance the network by one time-step:

$$\left(\vec{\mathbf{s}}, \vec{\mathbf{f}}\right) \leftarrow f\left(\vec{\mathbf{c}}, \vec{\mathbf{s}}, \mathbf{W} \times \vec{\mathbf{f}}\right) \tag{3}$$

Repeatedly stepping the network forward allows us to simulate the evolution of the network over time. A typical

time-step is 1ms, so 1000 steps are required to simulate 1 second of network operation; or, conversely, we must be able to perform 1000 steps per second to achieve a real-time simulation.

IV. Hardware Synapse Interconnect

The general update process shown in Equation 3 is expressed in terms of matrix-vector multiplication, but in practice will be implemented using specialised routines, rather than linear algebra libraries. A key optimisation is to observe that in practice the average number of neurons producing spikes in each update (time-step) will be relatively low (although there may be isolated periods of very high activity where many neurons fire synchronously). To optimise for this low average activity, software implementations use an intermediate vector to accumulate stimuli, accumulating weights into it as each neuron fires.

This algorithm is shown in Algorithm 1, using a temporary vector called $\vec{\mathbf{t}}$. Although this method improves performance in the average case, it has the unfortunate side-effect of introducing dependencies between the update of individual neurons: if neuron i fires, then the vector summation in statement 5 must be executed. This summation must be atomic, so if another neuron j is evaluated in parallel and also fires, then it must wait for neuron i to complete before it is able to perform it's own summation.

Algorithm 1 Fan-out (software) update algorithm.

1: $\vec{\mathrm{t}} \leftarrow 0_n$
2: **for** $i = 1..n$ **do**
3: $\quad (\vec{\mathrm{s}}_i, \vec{\mathrm{f}}_i) \leftarrow f(\vec{\mathrm{c}}_i, \vec{\mathrm{s}}_i, \vec{\mathrm{i}}_i)$
4: $\quad$ **if** $\vec{\mathrm{f}}_i$ **then**
5: $\quad\quad \vec{\mathrm{t}} \leftarrow \vec{\mathrm{t}} + \mathrm{W}[i, :]$
6: $\quad$ **end if**
7: **end for**
8: $\vec{\mathrm{i}} \leftarrow \vec{\mathrm{t}}$

Algorithm 2 Fan-in (hardware) update algorithm.

1: $\vec{\mathrm{t}} \leftarrow 0_n$
2: **for** $i = 1..n$ **do**
3: $\quad \vec{\mathrm{i}}_i \leftarrow \mathrm{W}[:, i] \times \vec{\mathrm{f}}$ {Vector dot-product.}
4: $\quad (\vec{\mathrm{s}}_i, \vec{\mathrm{t}}_i) \leftarrow f(\vec{\mathrm{c}}_i, \vec{\mathrm{s}}_i, \vec{\mathrm{i}}_i)$
5: **end for**
6: $\vec{\mathrm{f}} \leftarrow \vec{\mathrm{t}}$

In software this contention is not a problem, as a single-threaded implementation is inherently safe, and even a multi-threaded implementation can simply create a private copy of t for each thread, before combining the (small number) of private copies into one overall vector in the final statement. However, in hardware this summation presents more of a problem.

If the summation is implemented using a shared sequential vector adder then the simulator will be severely reduced in speed. Updating the n neurons will take $n+K$ cycles, where K is the number of pipeline stages needed to implement function f. In the cases we consider, where $n \sim 1000$, then K will be less than n, and so the number of cycles needed to update the neuron state will be at most $2n$. However, if a single shared sequential vector adder is used, then the minimum update time is increased to pn^2 cycles, where p is the proportion of neurons which fire in the time-step. Even in quiet networks it is likely that more than two neurons will fire in each update, so the summation becomes the bottleneck.

Instantiating multiple vector summation units is one possible solution, but the problem becomes where to store the synapse weights used within the summation. In software the weights are trivially shareable between threads, but in hardware each extra thread requires its own RAM port, and so using more than two summation instances will require multiple copies of the weight matrix. Given that the weight matrix consists of $n^2 \sim 10^6$ words, we quickly run out of RAMs in which to store it. Possible solutions include partially or fully parallel summation units, but these are more complicated, and also present problems with how to store the weights.

Although all these problems can be solved with some engineering effort, we prefer to re-organise the algorithm in a way that both simplifies the architecture and allows us to provide a fixed throughput of $n + K$ cycles per update. The key is to rearrange the update algorithm into the form shown in Algorithm 2, which is much closer to the original vector-matrix version given in Equation 3. Now instead of maintaining the vector of stimuli ($\vec{\mathbf{i}}$) between steps, we retain the vector of firings ($\vec{\mathbf{f}}$), as in the original formulation.

The reason for doing this is that the cost of operations in hardware is different from that in software. In software the cost of an operation is dependent on whether it was executed or not, while in hardware the cost is related to the area required to implement the operation. So if we can fit a hardware structure that is able to implement the vector dot-product $\mathbf{W}[:, i] \times \vec{\mathbf{f}}$ into the device then we might as well use it, regardless of how many neurons actually fire. Fortunately, because $\vec{\mathbf{f}}$ is a boolean vector, the multipliers in the dot-product are actually and-gates, so even with $n \sim 1000$ the whole structure can be fit into larger contemporary FPGAs, providing a throughput of one dot-product per cycle.

Figure 2 shows how this algorithm can be implemented in hardware. On the left-hand side is shown the structure of a single synapse. Each synapse is dedicated to the spike-train of a single neuron i, with the last spike value held in the (1-bit) register $\mathbf{f}_i$. Over n successive cycles the synapse iterates through the column of weights $\mathbf{W}[i, :]$ held in the RAM, and outputs the successive elements of the vector $\mathbf{f}_i \times \mathbf{W}[i, :]$.

The right-hand side of the figure shows the overall ar-

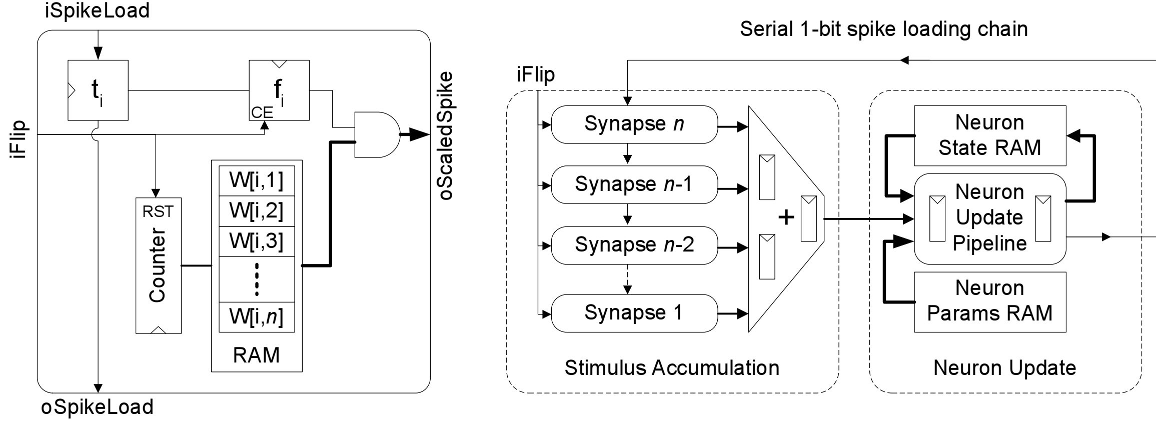

Figure 2. Architecture of a synapse (left) and a neural network (right).

chitecture. Each of the n synapse units outputs one scaled spike per cycle, so in each cycle all the scaled inputs for one neuron are produced. These n scaled spikes are then combined using a pipelined adder tree to produce the overall stimulus value. The stimulus can now be passed onto a heavily pipelined implementation of the neuron update function f. The neuron state is stored in a dual-port RAM, and can be updated in place.

The other output of the neuron update pipeline is the 1-bit value indicating whether each neuron fired. This needs to be propagated back into the synapses so that it can be loaded into the $\mathbf{f}_i$ registers at the start of the next update. This feedback is achieved using a 1-bit n-stage shift-register running through the synapses, shown as $\mathbf{t}_i$. The shift register starts at the synapse dedicated to neuron n, then moves back to neuron 1. Because the values of $\vec{\mathbf{f}}$ are produced in order, after n shifts the spike values are all in the correct location. After $n + K$ cycles the whole network has been updated, and the synapses can be flipped, moving the values of the $\mathbf{t}_i$ register into the associated $\mathbf{f}_i$, and allowing the next update step to start.

V. Izhikevich neuron model

As well as the interconnection method, another requirement for biologically plausible spiking behaviour is the selection of an appropriate neuron update function f. This is responsible for taking the incoming thalamic stimulus, combining it with and updating the neuron state, and producing the resulting spike train. The difficulty lies in choosing a model that is sufficiently complex to produce the different types of spiking behaviour, while simple enough to fit in an FPGA.

At the simple end of the spectrum is the leaky integrate and fire model, with three parameters a, b, c, and v_t, and one state variable v:

$v \leftarrow v + i + bv$
if $v > v_t$ **then**
  $v \leftarrow c$
end if

The model is very efficient, requiring only four floating-point operations per 1ms time-step, but unfortunately is too simple to exhibit complex behaviour such as intermittent bursting.

At the other end of the scale is the much more complicated Hodgkin-Huxley model. This consists of four update equations, and tens of neuron constants, which provide detailed models of the chemical and electrical processes within the neuron, and is able to replicate all kinds of spiking phenomena. However, to achieve this the model must operate using very small time-steps of 0.1ms, rather than the more normal 1ms steps. Because each small time-step requires 120 floating-point operations, this means that around 1200 floating-point operations are required to advance one neuron by 1ms.

The model that we have found provides the best balance between efficiency and biological plausibility is the relatively recent Izhikevich spiking model [1]. This uses two neuron state variables v and u, and five neuron parameters a, b, c, d, and s, and can exhibit all known types of spiking modes, such as intermittent bursting and chaotic spiking. The basic update algorithm uses 13 floating-point operations per time-step, but we choose to adopt the more stable version given in the matlab source code in [1], which steps variable v in two steps for stability. For efficiency we transform some of the constants and rearrange the equations into the form shown in Algorithm 3, without affecting the model or the flexibility presented to the user. Note that the expressions ab and $(1-a)$ on line 4 only involve neuron parameters, and so can be pre-calculated and stored instead of the underlying parameters a and b.

The function $\mathrm{N}()$ provides a random normal variable, which perturbs the thalamic input, so simulations of networks starting from the same starting state will diverge over time. Note that parameter s is encoded implicitly using numeric constants in [1], but we have treated it as an explicit

	Floating-point Ops.				Parameters	
	Total	$\times$	$+$	N()	Global	Neuron
Calc v	15	5	7	1	3	0
Calc u	4	2	2	0	0	2
Firing	1	0	1	0	1	2
Total	**20**	9	10	1	4	4

Table I
COST OF THE IZHEKIVICH NEURON MODEL IN FLOATING-POINT OPERATIONS AND CONSTANTS.

parameter

It is worth noting that we do *not* transform lines 2 and 3 to Horner form, i.e. $(0.02v + 3.5)v + t$. The reason for this is that the polynomial form has a latency of two multiplies and an add, while the Horner form has two multiplies and two adds. Any decrease in update latency reduces the K part of the $n + K$ cycles required per network update, so it is worth spending a little extra area to get a small performance increase.

Algorithm 3 Izhikevich Neuron Update Algorithm

1: $t \leftarrow (i + s\ \mathrm{N}() - u + 140)/2$
2: $v \leftarrow 0.02v^2 + 3.5v + t$
3: $v \leftarrow 0.02v^2 + 3.5v + t$
4: $u \leftarrow (ab)v + (1 - a)u$
5: **if** $v > 30$ **then**
6: $\quad v \leftarrow c$
7: $\quad u \leftarrow u + d$
8: **end if**

The costs of the neuron model are summarised in Table I, in terms of floating-point operations, and number of parameters (numeric constants). The divide by two on line 1 and the comparison on line 5 are not counted, as they are relatively cheap operations. The random number generator is counted as 1 floating-point operation - this is actually an underestimate when considering software, where it takes multiple instructions (floating-point and integer) to generate one random number, but is a fair comparison in hardware where random number generation is comparatively cheap.

In total each neuron update requires 20 floating-point operations, so with a time-step of 1ms this means that updating one neuron over a one second time period requires 20KFLOPs. With the target network size of 1024 neurons, this means that at least 20MFLOPs are required to simulate a neural network at real-time speed - however, our aim is to achieve significantly higher throughput to provide hyper-real-time speeds, for the reasons given in Section 2.

VI. IMPLEMENTATION

To explore the performance of the proposed architecture and neuron model outlined in the previous sections, we implemented a 1024 neuron fully-connected network using the Virtex-5 xc5vlx330t device. This is a comparatively large device, with an emphasis on basic logic elements, containing 324 36Kbit RAMs, 192 DSP blocks, and just over 200,000 pairs of Logic Elements (LEs), each containing a 6-LUT and a Flip-Flop. It has previously been used in investigations into High-Performance Computing using FPGAs, for example in the 16-FPGA Maxwell system, and so is reasonably typical of the kind of FPGA that can easily be bought and used by neural network researchers.

The interconnection architecture outlined in Section IV uses RAMs to store the weights assigned to each neuron, and a tree of adders to sum the scaled spike-trains to form the stimulus. In software these constants and summation operations would all be floating-point, but this is infeasible in hardware: a fully connected network requires n^2 weights, so even in 32-bit single-precision we would need 33Mbits of storage, far more than the 12Mbits available in the target device. Even in a device with more RAM blocks, we would still not be able to fit the $n - 1$ adders required for the summation tree, as even the largest devices cannot accommodate 1023 single-precision adders.

To fit the stimulus calculation into a contemporary FPGA, we must use fixed-point for both the weights and the summation tree. Although this may appear to introduce a significant limitation to the system in terms of what networks can be represented, and in how accurately the networks are simulated, in reality it is less of an issue. Weights below a certain level simply do not matter, as they will be swamped by the thalamic noise term, and the weights are also limited to a relatively small range, as they are modelled on physical processes with fixed limits. This means that only the absolute accuracy of the weights is relevant, rather than the relative accuracy, so floating-point is unnecessary, and fixed-point weights will not limit the types of networks that can be described.

Assuming that a network can be described using fixed-point weights, then there is consequentially no effect on the accuracy of the simulation. The weights can be accumulated using a tree of adders that increases in width from the leaves to the root, so the stimulus will be calculated exactly. In certain situations this may actually be *more* accurate than single-precision stimulus accumulation, as inhibitory and excitatory synapses (those with negative and positive weights) may cause cancellation and loss of precision.

Figure 3 shows the basic cell used within our concrete implementation of the abstract architecture shown in Figure 2. Each cell is based around a single Virtex-5 RAMB36 primitive, configured as a 36-bit wide by 1024 element RAM. Each RAM holds the weights for multiple synapses, in this case four. The maximum width (w_w) of the weights is determined by the number of synapses per RAM, and the width of the RAM ports. For four synapses we must have $4w_w < 36$, so the maximum width is $w_w = 9$ bits.

On the left of the RAM is shown a $4w_w$-bit wide register chain that passes through all the cells. This is used to load

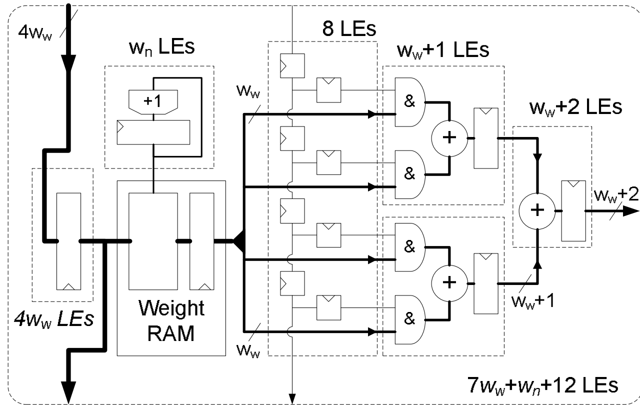

Figure 3. Cell implementing four synapses.

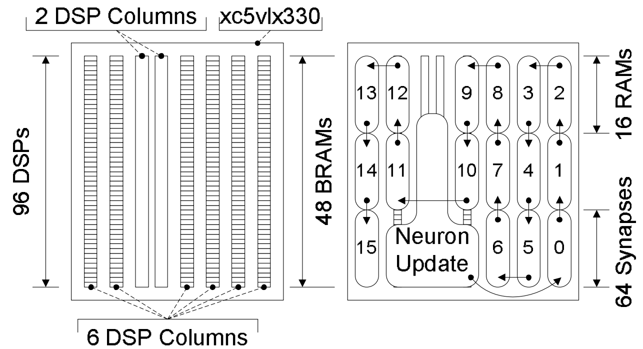

Figure 4. Arrangement of Block-RAM and DSP columns in the Virtex-5 xc5vlx330t device, and placement of synapse clusters and neuron update logic within device.

synapse weights into the RAM, without requiring a large fan-out address and data distribution network. Loading the synapse weights takes $n^2/4$ cycles: shifting new weights through the column takes $n/4$ cycles, and this must be repeated n times to load the synapses for each neuron.

In the centre of the cell is the spike distribution and storage shift chain, equivalent to registers $\mathbf{t}_i$ and $\mathbf{f}_i$ in Figure 2. During each neuron update pass, the RAM steps through the synapse weights for each neuron, which are multiplied with the 1-bit spike value, then added together. The logic for the multipliers and the first-stage of adders can be combined into one stage of logic elements, followed by another stage to produce the overall stimulus from all four synapses.

In order to support n neurons we require $n/4$ of these basic cells, so 256 cells are required to support all 1024 neurons. Because this basic cell is repeated so many times it is important to use as little logic as possible. Figure 3 shows the cost of each four-synapse cell in LEs, where w_w is the width of each weight, and $w_n = \lceil \log_2 n \rceil$. Our implementation uses $w_w = 8$ and $n = 1024$, so the total cost of each cell is 78 LEs, or just over 20,000 LEs in total (10% of the device total). Combining the 256 partial sums produced by these cells requires another 3000 LEs, so a total of just over 23000 LEs are required for the synapses and stimulus accumulation.

The neuron update function is a straightforward pipeline, scheduled using an ASAP (As-Soon-As-Possible) approach. The floating-point units are generated using Xilinx CoreGen 9.2, using maximum latency versions for maximum speed, and choosing the default (balanced) option for DSP48E usage. The neuron state and parameters are stored in block-RAMs, and are retrieved individually where needed within the pipeline, rather than reading all the inputs at the beginning of the pipeline - this rather trivial optimisation significantly reduced area when using double-precision, as otherwise large numbers of SRL32 shift-registers are needed to pipeline the parameters until the point in the pipeline where they are used. The random number generator is implemented using the alias method [6], with one central-limit accumulation to improve the distribution.

Because the stimulus interconnect is so regular, it is possible to manually place large amounts of the logic. The left side of Figure 4 shows the arrangement of RAM and DSP columns in the xc5vlx330. [1] To target this arrangement, relatively placed clusters containing 16 cells (i.e. 64 synapses) are created, then placed in a chain that snakes from the bottom right over to the bottom left (numbered 0 to 15 on the figure). The bottom third of the RAM columns on either side of the DSP columns are deliberately left clear, providing an empty space for the neuron update pipeline containing DSPs, RAMs, and logic. Both relative and absolute placement were achieved using VHDL level constraints (RLOC and LOC), and placing the logic allows us to perform the full tool-flow in under an hour for all designs.

Table II summarises the resource-usage and clock-rate of the different components, and for the whole simulation architecture. The individual components are placed and routed with no specific timing constraints, while the overall simulators are constrained to 133MHz. In principle the simulators could achieve a higher clock rate, as the individual components can run much faster, but 133MHz is both convenient for the Alpha-Data FPGA cards that we use, and balances performance against place-and-route time for the combined design.

The first column shows the total number of LEs used, either fully or partially occupied. The number of LEs used for the synapse interconnect is 24226, very close to the predicted 23000. The extra usage is due to FFs replicated by the tools for timing reasons, and due to small amounts of shared control and timing logic. Overall logic usage is just over 20% for the double precision version, which is very promising for future extensions, as it suggests that there is

[1] There is also an additional irregular column on the far right containing a combination of RAMs and a PCI-Express interface.

Module	Format	LEs	(%)	LUTs	FFs	Slices	(%)	DSPs (%)		RAMs (%)		MHz
Synapses	8-bit	24226	(11.7)	12816	24226	6844	(13.2)	0	(0.0)	256	(79.0)	311
Neuron update	Single	8727	(4.2)	6261	7781	2859	(5.5)	16	(8.3)	8	(2.5)	307
	Double	18147	(8.8)	12943	16293	5489	(10.6)	96	(50.0)	15	(4.6)	214
Entire simulator	Single	33452	(16.1)	19397	32420	9976	(19.2)	16	(8.3)	264	(81.5)	133
	Double	43263	(20.9)	26286	40919	13204	(25.5)	96	(50.0)	271	(83.6)	133

Table II
RESOURCE-USAGE AND PERFORMANCE FOR NEURAL NETWORK SIMULATOR AND ITS COMPONENTS, USING SINGLE-PRECISION AND DOUBLE-PRECISION FLOATING-POINT.

Implementation		Firing Activity		
Platform	Precision	Standard	Heavy	None
xc5vlx330t	Single	118.75x		
xc5vlx330t	Double	113.58x		
Core2/3GHz	Single	8.41x	3.98x	28.8x
Core2/3GHz	Double	6.84x	3.83x	12.9x
GT200/1.2GHz	Single	101.11x	32.02x	183.77x

Table III
SPEED-UP OF PLATFORMS OVER REAL-TIME, FOR DIFFERENT LEVELS OF FIRING ACTIVITY.

plenty of room left to add new features, even on a per-synapse level.

The RAM utilisation is much higher, with both single and double precision versions using over 80% of available resources. This is an unavoidable consequence of using a fully-connected network and storing the synapses on-chip. Larger networks would require the use of off-chip RAM resources, and probably could not use the fan-in approach adopted here.

VII. EVALUATION

The performance of the simulator is determined by the number of neurons (n), the latency of the neuron update pipeline (K), and the clock-rate of the circuit (F). Each pass through the network takes $n + K$ cycles, so the number of time-steps achieved per second is $F/(n + K)$. For both the single and double precision implementations F=133MHz, and $K = 96$ for the single-precision or $K = 147$ for double-precision, resulting in a peak performance at $n = 1024$ of 118.75 and 113.58 KSteps/s. Using a time-step of 1 ms, 1 KStep corresponds to one seconds worth of simulation time, so the speed-up of the simulators over real-time is also 118.75 and 113.58.

Table III compares the performance of the hardware implementation against a software version running in an Intel Core2 3GHz CPU, and a GPU version using an NVidia 1.2GHz GT200 containing 30 SIMD processors. The software implementation is not explicitly vectorised, but is written in a way that allows the Intel C Compiler to automatically vectorise the neuron update procedure and the accumulation of stimulus. Performance is shown in multiples of real-time - i.e. how much faster the simulated network runs than the "real" network. The GPU implementation is described using CUDA, and great care is taken to ensure that all memory accesses can correctly coalesce, that the correct number of thread blocks are launched, and that fast maths intrinsics are used wherever possible.

Because the software version uses the fan-out model shown in Algorithm 1, the performance is dependent on the level of spiking activity occurring in the network. For this reason we consider three different spiking activity levels: *standard*, as produced by the initialisation code in [1], where about 7% of neurons fire per time-step; *heavy*, where the network has been tuned so that 20% fire per step; and *none*, where no spikes at all occur. Because the FPGA implementation uses the fan-in model, the simulation rate is independent of spiking activity, while in software we see that the *heavy* activity level approximately halves the performance of the simulator.

The GPU provides slightly lower performance than the two FPGA versions under the standard load, and is only faster under the unrealistic (and uninteresting) situation where no neurons fire. When very heavy firing occurs the GPU performance degrades significantly, as even the high-bandwidth memory architecture of a GPU cannot supply the synapse weights quickly enough. Even with standard firing activity, where performance is broadly comparable, we would observe that the FPGA is able to provide double-precision, while the GPU only operates in single-precision.

VIII. RELATED WORK

There is a huge body of existing work on the implementation of neural networks using FPGAs – a contemporary survey of the various approaches can be found in [4]. Our focus is on hyper-real-time simulation of fully connected networks with around 1000 neurons using a complex biologically plausible neuron model, unlike many approaches which use a simpler neuron model while attempting to support much larger networks at slower speeds.

One approach [5] that provides a similar level of performance and functionality to our approach is to use a CPU and FPGA in parallel. Here the synapse weights and interconnection are handled in software, while the computationally intensive neuron update is performed in hardware. This provides more flexibility and scalability when constructing the neural network, but significantly reduces performance: a 1024 neuron network can be simulated in real-time, using

a 14-bit fixed-point neuron model. This is 100 times slower than our double-precision model, although we have the advantage of a much more recent FPGA platform.

The Izhikevich model has been previously implemented in FPGAs [3], but only as a standalone neuron model using 24-bit fixed point. The only reported experiments are for two directly coupled neurons operating at 1MHz in a unspecified FPGA device.

IX. Conclusion

This paper has demonstrated the feasibility of using FPGAs for the simulation of biologically plausible spiking neural networks. Using a single Virtex-5 xc5vlx330 device at 133MHz we are able to simulate fully-connected 1024 neuron networks at over 100 times real-time speeds, with a simulation rate independent of the spiking activity level. This guaranteed throughput is achieved using a method that distributes synapse weights through hundreds of block RAMs, dedicating small groups of synapses to each RAM, and using a shifting mechanism to distributed spike-trains.

The network uses the Izhikevich neuron model, which is capable of replicating all spiking patterns observed in real neurons, using a process requiring 20 floating-point operations. Using this model the FPGA simulator achieves a sustained processing rate of 2.26 GFlops in double-precision, over 16 times the speed of a 3GHz Core2 CPU implementation.

Current and future work includes scaling up our approach to cover larger neural networks with more complex properties, such as axonal delay and spike-timing dependent plasticity. Automating design optimisations, such as those involved in transforming Algorithm 1 to Algorithm 2 in Section 4, would also be investigated with the view of extending them to support a wide variety of applications.

References

[1] Eugene M. Izhikevich. Simple model of spiking neurons. *IEEE Trans. on Neural Networks*, 14(6), 2003.

[2] Eugene M. Izhikevich. Which model to use for cortical spiking neurons? *IEEE Transactions on Neural Networks*, 15(5):1063–1070, 2004.

[3] M. La Rosa, E. Caruso, L. Fortuna, M. Frasca, L. Occhipinti, and F. Rivoli. Neuronal dynamics on FPGA: Izhikevich's model. *Proceedings of the SPIE*, 5839:87–94, 2005.

[4] L.P. Maguire, T.M. McGinnity, B. Glackin, A. Ghani, A. Belatreche, and J. Harkin. Challenges for large-scale implementations of spiking neural networks on fpgas. *Neurocomputing*, 71:13–29, 2007.

[5] Eduardo Ros, Eva M. Ortigosa, Rodrigo Agís, Richard Carrillo, and Michael Arnold. Real-time computing platform for spiking neurons (rt-spike). *IEEE Trans. on Neural Networks*, 17(4), 2006.

[6] David B. Thomas and Wayne Luk. Non-uniform random number generation through piecewise linear approximations. *IET Computers and Digital Techniques*, 1(4):312–321, 2007.