EFFICIENT HANDLING OF SYNAPTIC UPDATES IN FPGA-BASED LARGE-SCALE NEURAL NETWORK SIMULATIONS

Paul J Fox, Simon W Moore

Computer Laboratory
University of Cambridge
Cambridge, United Kingdom
email: {paul.fox, simon.moore}@cl.cam.ac.uk

ABSTRACT

We present novel methods for handling synaptic updates in FPGA-based neural network simulations that are designed to makes efficient use available memory and to allow for real-time simulation of large numbers of neurons and synapses. Our methods allows simulations to scale using networks of FPGAs, with little overhead and retaining real-time performance. An implementation based on our methods has been built on our Bluehive multi-FPGA system, and simulates 64k neurons with 64M synapses per FPGA in real-time.

1. INTRODUCTION

We wish to simulate networks of many thousands (and ultimately millions and even billions) of neurons, with a fan-out of around a thousand. Section 2 introduces the biologically plausible Izhikevich neural simulation algorithm [1], which is suited to simulations of large numbers of neurons, and analyses its communication and resource requirements, showing that the majority of the complexity of the algorithm (and neural simulation in general) results from the need to communicate and apply synaptic updates rather than calculating the effects of each neuron in isolation. Hence it is critical that a neural network simulator handles synaptic updates efficiently if it is to scale to the size that we require.

Our analysis from Section 2 leads us to consider appropriate implementation platforms in Section 3. We select an implementation platform based on a network of FPGA boards for a number of reasons, particularly the availability of high-speed serial and memory interfaces. However, using FPGAs does bring some limitations compared to an ASIC implementation, particularly limited on-chip memory.

If a simulation is to scale while remaining in real-time then synaptic updates must introduce little latency and make efficient use of both inter-FPGA communication and memory. As fan-out is high and we expect significant locality (As discussed in Section 2), this would suggest using a multicast algorithm, but this leads to inefficient use of both on- and off-chip memories. In Section 4 we propose an alternative algorithm which aims to provide a real-time, scalable simulation using a FPGA implementation. We evaluate this algorithm against unicast and multicast in Section 5 and show that it consistently makes more efficient use of communication and memory bandwidth.

We then present a FPGA implementation of a neural network simulator which uses our algorithm in Section 6, focussing particularly on how synaptic updates are applied to make maximal use of memory bandwidth.

Section 7 presents the results of simulating a network of 256k neurons with 256M synapses in real-time using four FPGAs from our Bluehive multi-FPGA system. We discuss how these results and our goals compare to other work in Section 8 and provide conclusions in Section 9.

2. COMMUNICATION AND RESOURCE REQUIREMENTS OF SPIKING NEURAL NETWORK SIMULATION

We have selected the Izhikevich spiking-neuron algorithm [1] for our neural network simulations as we believe that it offers a good compromise between biological accuracy and computational efficiency [2]. The next subsection briefly introduces the algorithm from a computational point of view before we analyse its communication and resource requirements. At this stage we identify optimisations that will assist in achieving our goals of creating a large-scale, real-time simulation, but without focussing on any particular implementation technology. We will assume that 10^5 neurons are simulated per discrete device to allow us to provide resource usage examples.

2.1. The Izhikevich spiking-neuron algorithm

The Izhikevich algorithm uses Equation (1) to simulate the spiking behaviour of a neuron. This equation is designed to be evaluated in continuous-time using floating-point arithmetic, however it is possible to derive suitable discrete-time, fixed-point alternatives, which will be evaluated every 1 ms, matching prior work [3].

$$v' = \begin{cases} 0.04v^2 + 5v + 140 - u + I & v < 30\,\mathrm{mV} \\ c & v \geq 30\,\mathrm{mV} \end{cases}$$
$$u' = \begin{cases} a(bv - u) & v < 30\,\mathrm{mV} \\ u + d & v \geq 30\,\mathrm{mV} \end{cases} \quad (1)$$

The variable v represents the membrane voltage of the neuron and u the refractory voltage, with a to d being parameters which control the behaviour of the neuron. The variable I represents the sum of the magnitudes of all spikes arriving via the neuron's dendritic inputs. The synapses, which connect neurons together, are represented by tuples of source neuron, target neuron, delay and weight.

The range of these variables and parameters is bounded, allowing us to use 16-bit fixed-point arithmetic, avoiding complex floating-point units in our implementation. As described in [4], Equation (1) is easily laid out as a pipelined structure that can be clocked at 200 MHz. So if every neuron is evaluated once every 1 ms, we can easily evaluate 10^5 neurons using just one copy of the evaluation pipeline. This makes it clear that the problem is not compute bound, so it surprises us that many people focus on optimising computation even for the comparatively simple Izhikevich model.

2.2. Communication requirements

In addition to evaluating Equation (1) for each neuron it is necessary to communicate synaptic updates between neurons, delay them and then sum their weights to provide the I-value for every neuron at each time step. As the fan-in of a neuron increases, the computational cost of summing I-values dominates that of evaluating Equation (1), becoming the critical inner loop of the algorithm. But the computation is just addition, so the real challenge is streaming the weights through addition units.

Typically neurons have a fan-out and fan-in of around 1000. Fan-in mirrors the fan-out so let us focus on fan-out. The mean firing rate for neurons is 10 Hz [5], so the fan-out bandwidth for 10^5 neurons is $1000 \times 10 \times 10^5 = 10^9$ events/s. In our model, each fan-out message consists of a 32-bit value to index the receiving neuron, 12-bits for the weight and 4-bits for the delay giving 48-bits or 6-bytes per event. So the mean fan-out bandwidth for 10^5 neurons is 6 GB/s.

Inter-device communication is governed by the locality and overall size of the neural network. It has been shown that interconnect in mammalian brains can be analysed using a variant of Rent's rule which is often used to analyse communication requirements in VLSI chips [6]. This analysis indicates that there is a great deal of locality. So, for very large networks spanning many devices, we see a great deal of communication between neighbouring devices and very little communication travelling any distance provided the communication topology has at least three dimensions. To get an upper bound on the communication requirements, we take the pathological case that all 10^5 neurons fan-out to neurons off-device, with all target neurons being on different devices. In this case all 6 GB/s of bandwidth is needed (calculated in the previous paragraph), giving us an upper bound on the bandwidth that an implementation of this simulation will need between devices.

Communication latency is also an important consideration. For real-time simulation we must deliver spike events in well under a 1 ms simulation time-step.

2.3. Resource requirements

We now look at the data storage requirements of our example of 10^5 neurons per device. The parameters of equation (1) take for a single neuron take less than 16 bytes. So 10^5 neurons requires around 1.6 MB of storage.

With 10^5 neurons per node, a fan-out of 10^3 and 6 bytes per event, we need 6×10^8 bytes of storage (0.6 GB) to store the synaptic parameters.

As the memory bandwidth required is proportional to both the number of neurons being simulated and (more significantly) their fan-out, it is clear that memory bandwidth provides a bound on the number of neurons we can simulate in real-time per device.

We must also consider the memory requirements of applying synaptic updates. Since any incoming synaptic update could target any neuron on the device and summing the update with the current I-value is the critical inner loop of the algorithm, the current I-value must be readily available. This means that the current I-value for each neuron (16 bit) must be stored in on-chip memory, providing another limit to the number of neurons we can simulate in real-time per device.

3. IMPLEMENTATION PLATFORMS FOR SPIKING NEURAL NETWORK SIMULATION

Our choice implementation technology is governed by many factors, including ability to implement the chosen simulation at the chosen scale, cost and usability. It also affects the design of parts of the implementation, particularly how synaptic updates are routed and applied. The principle choice is between using ASICs or FPGAs.

Given the volume of communication that we envisage between devices, high-speed communication will be needed, using serial transceivers. While it is possible to implement high-speed serial transceivers in ASICs (remembering that FPGAs are themselves a type of ASIC), this can only be achieved using recent nanometre-scale processes, leading to massively increased costs for small production volumes. In comparison these transceivers are readily available in high-end commercial FPGAs and so are available to us at significantly less cost. FPGA evaluation boards which provide access to high-speed communication links as well as other resources such as DDR2 SDRAM are readily available off-the-shelf, which avoids the need to create complex PCBs to route high-speed signals, further lowering costs.

ASICs also give no scope for reprogramming, and hence limited scope for fixing errors or altering many aspects of system behaviour, meaning that massive effort needed for design and testing and hardware cannot be altered to suit future design requirements. In comparison FPGAs can be reprogrammed with little cost beyond the time taken to resynthesise a design. This significantly lowers design and testing effort.

Therefore we have chosen to implement our system using FPGAs. This means that our design must consider the very limited amount of on-chip memory (Block RAM or BRAM) available (2 MB for a Stratix IV 230). As discussed in Section 2.3, the current I-value for each neuron must be stored in BRAM to avoid a performance bottleneck. To ensure that the value being fed to Equation (1) is not affected by race conditions we will actually store two copies of each I-value, one to be fed to the equation and one which is being updated ready for the next time step. With two copies of the I-values at 16 bit per neuron the absolute maximum number of neurons that could be simulated on each FPGA is $\frac{2\,\mathrm{MB}}{16\,\mathrm{bit}\times 2} = 256\mathrm{k}$. However in practice FPGA designs cannot make use of every available resource, and BRAM will also be needed for many other purposes such as pipeline FIFOs and DRAM controllers.

In view of this we propose to keep all data other than I-values in off-chip RAM, giving the challenge of routing massive volumes of synaptic update messages without multicast routing reliant on large amounts of BRAM. As will be shown in Section 5, unicast routing is unsuitable as it uses communication resources in direct proportion to fan-out, and so we must design a routing algorithm for synaptic updates that approximates multicast while using only off-chip memory. This means that off-chip memory bandwidth becomes the limiting factor to the volume of synaptic updates that can be routed in real-time, and by extension the size and scalability of the simulation as a whole. Our routing algorithm must be designed to make maximum use of off-chip memory bandwidth, and this means making efficient use of burst read transactions.

4. HANDLING SYNAPTIC UPDATES IN A FPGA-BASED SIMULATION

Our synaptic update handling algorithm is designed to approximate multicast routing to allow efficient routing of synaptic update messages with high fan-outs, while keeping all data (other than current I-values and data being processed) in off-chip memory to fit the resource constraints of a FPGA implementation. If we are to meet our goal of a large, real-time simulation the we must do everything we can to maximise use of memory bandwidth, and for DRAM this means using burst reads.

Our implementation platform is the Altera DE4 230 development board, which provides two independent DDR2 memory channels. Each channel supports burst reads of up to 8 256 bit words at a local clock frequency of 200 MHz. The latency between a burst read request and the first response is around 10 clock cycles, after which the remaining data is delivered on adjacent clock cycles. Requests for burst reads can be queued in the memory controller for future action even if all of the data from a previous burst read has not been delivered.

4.1. Our synaptic update handling algorithm

Our algorithm applies the complete set of synaptic updates that result from a neural spike by following a tree of pointers through regions of off-chip memory. It proceeds as follows:

1. A pointer is found alongside the parameters of Equation (1) for the neuron that has spiked.
2. The pointer is used to burst read a set of fan-out tuples. These tuples consist of either:
   (a) destination FPGA, delay and a pointer to a set of update tuples. *or*
   (b) destination FPGA and a pointer to a further set of fan-out tuples.
3. All tuples are transmitted to their destination FPGA.
4. Tuples of type 2a are delayed.
5. Tuples of type 2b repeat step 2.
6. After being delayed each fan-out tuple is used to burst read a set of update tuples. These tuples consist of target neuron number and weight. Each weight is applied to the target neuron to perform the synaptic update.

Repetition of step 2 is conditional on the location of target neurons, and is suited to cases where a number of target neurons are clustered on groups of FPGAs remote from the source. A single message can be sent to one of the group, with further fan-out then sending messages to the other FPGAs in the group. The path through memory for a single synaptic update is shown in Figure 1.

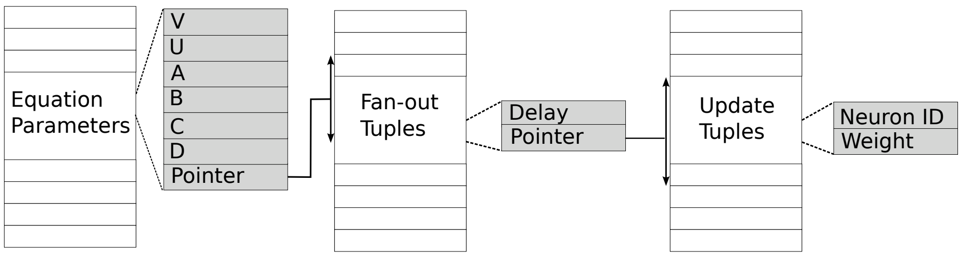

Fig. 1. Path of synaptic update handling algorithm through off-chip memory

4.2. Benefits of our algorithm

Our algorithm minimises use of on-chip BRAM, with all data describing the network being simulated being stored in off-chip memory. This facilitates its implementation using FPGAs. We optimise the accesses to off-chip memory (and hence maximise the number of neurons that can be simulated per FPGA in real-time) in a number of ways:

- The pointer in step 1 is stored alongside the parameters of Equation (1) and so an additional memory access is not required to fetch it.
- Separate tuples from the fan-out stage for each pair of destination FPGA and delay avoid an additional memory access at the destination FPGA before applying delays.
- Our assumption of locality makes it likely that the majority of synaptic updates will be applied to neurons which are either on the same FPGA as the source neuron or on nearby FPGAs. Combined with our assumption of high fan-out this means that there are a large number of synaptic updates will be fetched by each pointer from the fan-out stage, which makes efficient use of burst memory accesses.
- The pointer needed by step 6 is supplied by the incoming message, unlike multicast algorithms such as that proposed in [3] that require memory accesses to determine the pointer based on the identity of the source neuron.
- If the size of an off-chip memory word is greater than the size of a fan-out tuple (either type in step 2) then multiple tuples can fit in a word.
- If the size of an off-chip memory word is greater than the size of an update tuple then multiple updates can be applied in parallel, making efficient use of both time and off-chip memory bandwidth.

While the algorithm requires that three inter-dependent regions of memory are populated before a simulation can be run, this is a similar situation to populating the routing tables used by multicast algorithms.

Finally, since all data required by a simulation is stored in off-chip memory it is simpler to load a simulation since it is not necessary to transfer routing data into on-chip memories before the simulation is run. Nor is it necessary to resynthesise the FPGA design, as is the case with many previous FPGA implementations [7].

5. EVALUATION OF OUR ALGORITHM

To show the performance of our algorithm we compare it to unicast and multicast algorithms using a mathematical model of the number of clock cycles needed to handle the synaptic updates produced by a single neural spike with a fan-out of 1000. We count the cycles needed to transmit inter-FPGA messages and to perform off-chip memory accesses. We use a number of distributions of target neurons relative to the source neuron to provide several points of comparison with varying degrees of locality.

5.1. Target neuron distribution

We assume that a network of FPGAs is arranged in a 2-D mesh, with each FPGA having 4 neighbours. We distribute a varying percentage of target neurons on FPGAs a given distance from the source FPGA using the distributions in Table 1. Table 2 shows the number of FPGAs at each distance that hold target neurons and the number of target neurons on each FPGA for distribution 3 from Table 1 as an example.

5.2. Off-chip memory accesses

We assume that off-chip memory has a word size of 256 bits, a burst size of 8 and a delay of 5 clock cycles between a burst (or single access) being requested and the first word being returned. A new read request can be made after the last word of a burst has returned.

For our algorithm we assume that 4 fan-out tuples or 4 update tuples are stored in a word, and that these tuples are accessed using burst reads. We assume that unicast performs burst reads at the source FPGA to retrieve tuples of target

Distance	0	1	2	3	4	5
Id	Percentage of targets					
1	100	0	0	0	0	0
2	80	20	0	0	0	0
3	70	16	8	6	0	0
4	50	20	15	10	5	0
5	30	30	15	15	5	5

Table 1. Target distributions

Distance	Percentage	FPGAs	Targets per FPGA
0	70	1	700
1	16	4	40
2	8	8	10
3	6	12	5

Table 2. Distribution of neurons for distribution 3 from Table 1

FPGA, target neuron, delay and weight (again with 4 tuples in a word), with no further memory accesses required. Multicast performs no memory accesses until each target FPGA, where a lookup table needs to be scanned to find a pointer to a region of memory containing tuples of target neuron, delay and weight [3]. It is assumed that the lookup table contains 1000 entries (equal to the fan-in), and so traversing it takes $\log_2 1000 \approx 10$ memory cycles. Fetching tuples then takes the same number of cycles as our algorithm.

5.3. Inter-FPGA messages

We count the number of times that a message traverses a link between two FPGAs, and assume that each message takes 10 clock cycles to traverse each link. Unicast sends one message per target neuron, with many traversing more than one link. Multicast sends messages in a minimum spanning tree between the source and target FPGAs. Our algorithm sends one message per target FPGA, with some traversing more than one link.

5.4. Analysis

Figure 2 shows the total number of clock cycles needed to apply synaptic updates for each algorithm, grouped by the distributions of target neurons from Table 1.

It can be seen that our algorithm consistently uses fewer clock cycles than either unicast or multicast, particularly as locality decreases (higher-numbered distributions). This means that it uses less communication and memory bandwidth per set of synaptic updates, and makes it suited to implementing a real-time, scaleable, multi-FPGA simulation.

The number of memory cycles for multicast is a result of scanning lookup tables to find a pointer to the correct set

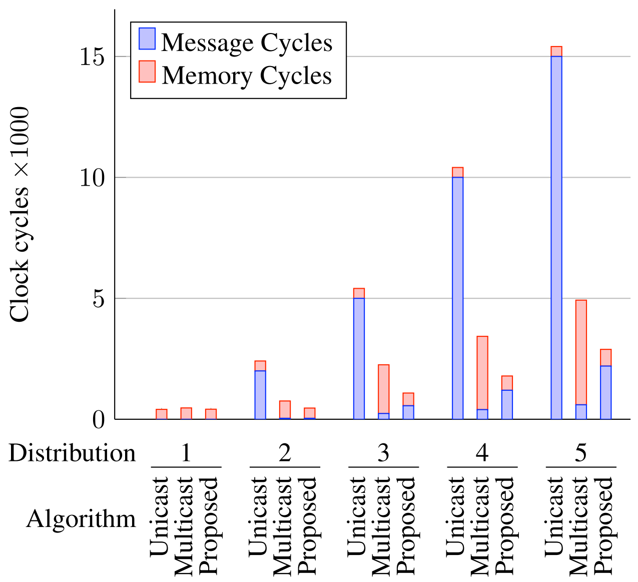

Fig. 2. Total system clock cycles needed to apply synaptic updates. Distribution refers to the distributions of target neurons in Table 1

of update tuples. This takes 60 cycles per target FPGA, and cannot take advantage of burst accesses.

6. FPGA IMPLEMENTATION

The implementation of our multi-FPGA system is presented in our previous work [4]. As shown in Figure 3 there are separate hardware blocks for fan-out, delay and accumulation of synaptic updates. These handle steps 2, 4 and 6 of the algorithm in Section 4 respectively. Computation of Equation (1) is performed by the equation block which also performs step 1 of the routing algorithm. There are also hardware blocks to record spike activity (spike auditor) and to inject spikes (spike injector), the latter being used particularly when the simulation starts.

The fan-out block is a simple implementation of step 2 of the algorithm. The delay block uses 16 FIFOs (one for each permitted delay size) and a circular pointer to delay each input tuple for the required time. We will focus on the implementation of the accumulator block, as it is the most interesting with regard to implementation of our algorithm on FPGA

The accumulator block implements step 6 of our algorithm. Since it occurs at the leaves of a tree with a high fan-out it is the inner loop of the algorithm, which makes it the most performance critical. Its input is a stream of pointers to sets of update tuples from the delay block. A burst read is used to fetch the tuples, which are packed in to 256 bit words with up to 4 tuples per word. The tuples are fed into the core of the accumulator block as shown in Figure 4.

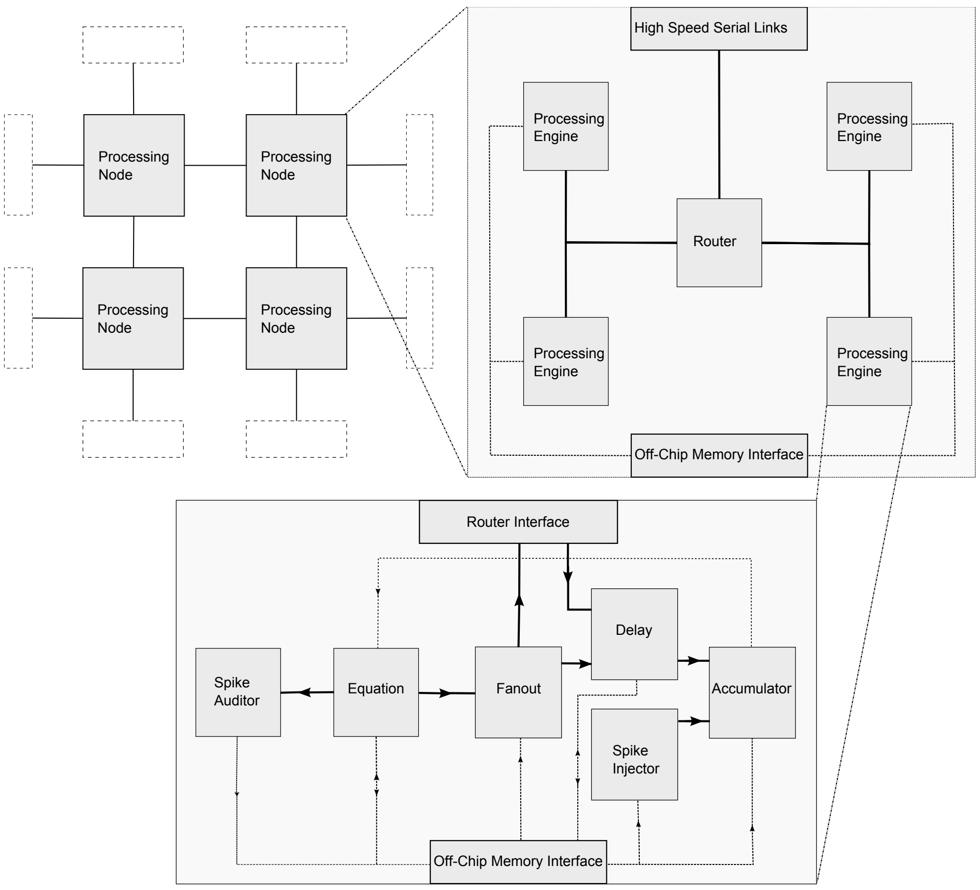

Fig. 3. Block diagram of the complete multi-FPGA system

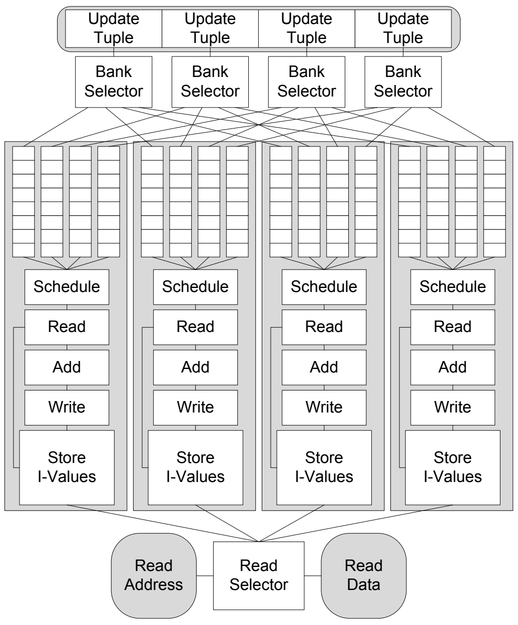

Fig. 4. Layout of the accumulator block

The core of the accumulator consists of four blocks, each containing the I-values for one quarter of the neurons hosted by the FPGA in a BRAM, and associated logic to allow the values to be accessed and updated. Neurons that are frequently updated together (as a result of locality) are allocated to different blocks so that they can be updated in parallel. Each input word causes synaptic updates to be applied by:

1. Each of the four update tuples is fed to a block selector. This determines which of the four blocks hosts the neuron which is updated by that tuple.
2. The tuple is routed to a FIFO queue in its target block. There is one FIFO per block per position in the input word to allow update tuples to appear in any position in the input word. Without these FIFOs two update tuples targeting the same block would cause a stall.
3. Update tuples are dequeued from the set of FIFOs in each block in a round-robin fashion.
4. The current I-value for the target neuron is fetched from BRAM.

5. The weight in the update tuple is added to the current value.

6. The new value is stored to BRAM.

The accumulator block helps us to maximise off-chip memory bandwidth efficiency and hence maximise the number of neurons that can be simulated per FPGA in real-time.

7. RESULTS

Given the absence of widely used neural netlist benchmarks, we created our own networks to test our system. Initially netlists were created using the PyNN [8] tool created by the neuroscience community. Whilst PyNN can produce a range of complete neural netlists, it appears not to scale much beyond 8k neurons, which is insufficient to demonstrate a 4-FPGA system with 64k neurons per FPGA. Therefore we created our own generator tool to produce a neural netlist with biologically-plausible parameters – an average neuron firing rate of 10 Hz and fan-out of 1000. Care had to be taken to generate an appropriate network which neither extinguishes itself or explodes in activity. Chunks of 1000 neurons were grouped into populations. This helped to achieve our network activity goal by biasing synaptic connections towards the next adjacent population whilst keeping the fan-out constant. This results in a network where around 1% of the neurons fire in any 1 ms time step, though this varies slightly over time as shown in Figure 5.

Figure 5 presents a scatter plot (in fine red dots) showing neuron firing events, and the total number of clock cycles needed to complete each 1ms time step (black line). Figure 6 is a larger-scale copy of a small section of Figure 5 which more clearly shows the neuron firing pattern.

Our aim is to build a real-time system (no faster, no slower), and we can run our design at 200 MHz. Since the maximum workload for any 1ms period is completed within 2×10^5 clock cycles, we have met our target.

We also compared the performance of our FPGA-based system to a CPU-based system using the same network. Our single-threaded neural network simulator written in C required 48.8 s to calculate 300 ms of simulation time on a single thread of a 16-thread, 4-core Xeon X5560 2.80 GHz server with 48 GB RAM. So the four-FPGA version is 162 times faster than the software simulator, which has similar performance to other reported software simulators [9].

8. RELATED WORK

Given the requirements in Section 2, current GPGPUs and multicore CPUs do not have the communication bandwidth needed for scalable massively-parallel spiking neuron simulation (e.g. thousands of CPUs or GPUs). The custom SpiNNaker machine [3] scales to 10^6 ARM processors using a

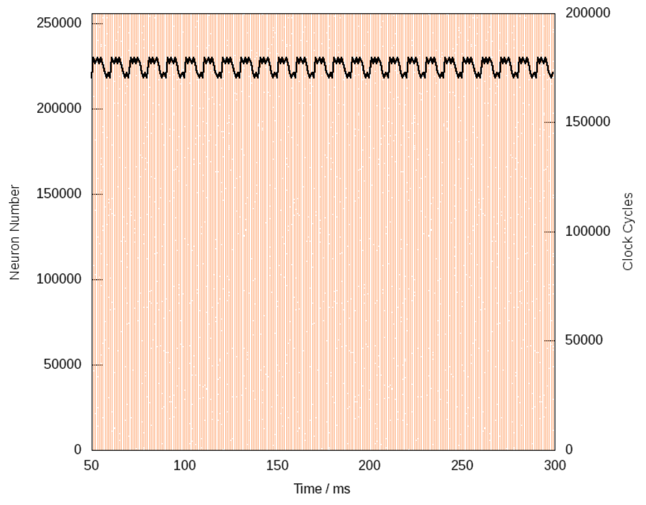

Fig. 5. Graph showing per-neuron activity (fine red dots) and the total number of cycles needed to complete every 1 ms step (black line). Real-time achieved if the total number of cycles per 1ms never exceeds 2×10^5, i.e. 200 MHz operating rate.

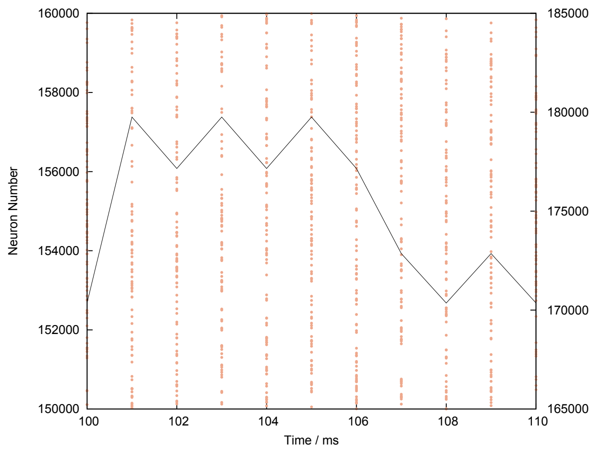

Fig. 6. Graph showing a small section of Figure 5 to more clearly show the neuron firing pattern.

custom ASIC with custom interconnect providing a reprogrammable platform suited to neural simulation. This is an alternative approach to our proposed FPGA system but the custom ASICs are likely to be moderately expensive for a few thousand parts and will be on an implementation technology which is several generations behind FPGAs.

FPGAs pay a significant area and performance penalty for being reconfigurable, however they can be produced cost effectively using small feature-size processes (40 nm for Stratix IV parts), which allows integrated high-speed memory interfaces and serial transceivers that are not possible using older implementation technologies. Since large-scale neuron simulation is communication-bound, FPGAs have an advantage. On the other hand, current FPGAs are more power hungry than SpiNNaker chips. Given these advantages and disadvantages it remains to be seen whether the SpiNNaker approach is more competitive than FPGAs in this space for large machines.

Much research has been undertaken on FPGA based artificial neural-network simulators, often for multi-layer perceptron models [10]. In contrast, our work is focused on spiking neuron models [2]. Often research focuses on single FPGA implementations [7] where we are interested in parallel FPGA machines, for example Thomas and Luk [9] present an implementation of 1k Izhikevich neurons running $100\times$ faster than real-time whereas we have focuses on real-time simulation and can easily manage 64k neurons per FPGA. We achieve comparable performance but with a design scalable to far more neurons per FPGA and many FPGAs.

In common with [7, 11], we time-multiplex the hardware and stream neuron parameters from external memory but we have a multi-FPGA implementation allowing more neurons to be simulated in real-time (for the same complexity of neuronal algorithm, numerical precision used and fan-in:neuron ratio).

9. CONCLUSION

Three contributions are made in this paper:

Firstly we characterise large-scale, real-time neural network simulation and provide reasons why FPGAs are an appropriate implementation platform, along with the challenges that must be faced to make such an implementation a reality, particularly the available memory hierarchy.

Secondly we propose a synaptic update handling algorithm which makes efficient use of the resources available to FPGAs and compare it to multicast and unicast algorithms. We find that our algorithm is suited to implementing large-scale, real-time simulations on FPGA as it makes more efficient use of communication and memory bandwidth than either of the other algorithms.

Thirdly we present details of the implementation of our algorithm on FPGA, focussing on our accumulator block, which optimises the critical inner loop of the algorithm. We present results from a simulation of a network of 256k neurons and 256M synapses in real-time which utilises a network of 4 FPGAs.

10. REFERENCES

[1] E. Izhikevich, “Simple model of spiking neurons,” *Neural Networks, IEEE Transactions on*, vol. 14, no. 6, pp. 1569–1572, Nov. 2003.

[2] ——, “Which model to use for cortical spiking neurons?” *Neural Networks, IEEE Transactions on*, vol. 15, no. 5, pp. 1063–1070, Sept. 2004.

[3] X. Jin, S. Furber, and J. Woods, “Efficient modelling of spiking neural networks on a scalable chip multiprocessor,” in *Neural Networks, 2008. IJCNN 2008. (IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on*, 1-8 2008, pp. 2812–2819.

[4] S. W. Moore, P. J. Fox, S. J. Marsh, A. T. Markettos, and A. Mujumdar, “Bluehive - a field-programable custom computing machine for extreme-scale real-time neural network simulation,” in *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, 29 2012-may 1 2012, pp. 133 –140.

[5] C. Mead, “Neuromorphic electronic systems,” *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1629–1636, Oct. 1990.

[6] D. S. Bassett, D. L. Greenfield, A. Meyer-Lindenberg, D. R. Weinberger, S. W. Moore, and E. T. Bullmore, “Efficient physical embedding of topologically complex information processing networks in brains and computer circuits,” *PLoS Comput Biol*, vol. 6, no. 4, p. e1000748, 04 2010.

[7] 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,” *Neurocomput.*, vol. 71, no. 1-3, pp. 13–29, 2007.

[8] A. P. Davison, D. Bruderle, J. M. Eppler, J. Kremkow, E. Muller, D. Pecevski, L. Perrinet, and P. Yger, “PyNN: a common interface for neuronal network simulators,” *Frontiers in Neuroinformatics*, vol. 2, no. 11, 2009.

[9] D. Thomas and W. Luk, “FPGA accelerated simulation of biologically plausible spiking neural networks,” in *Field Programmable Custom Computing Machines, 2009. FCCM ’09. 17th IEEE Symposium on*, 2009, pp. 45–52.

[10] A. R. Omondi and J. C. Rajapakse, *FPGA Implementations of Neural Networks*. Springer, 2006.

[11] J. Martinez-Alvarez, F. Toledo-Moreo, and J. Ferrandez-Vicente, “Discrete-time cellular neural networks in FPGA,” in *Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on*, 2007, pp. 293–294.