Bluehive — A Field-Programable Custom Computing Machine for Extreme-Scale Real-Time Neural Network Simulation

Simon W Moore, Paul J Fox, Steven JT Marsh, A Theodore Markettos and Alan Mujumdar
Computer Laboratory
University of Cambridge
Cambridge, United Kingdom
{*simon.moore, paul.fox, steven.marsh, theo.markettos, alan.mujumdar*}*@cl.cam.ac.uk*

***Abstract*—Bluehive is a custom 64-FPGA machine targeted at scientific simulations with demanding communication requirements. Bluehive is designed to be extensible with a reconfigurable communication topology suited to algorithms with demanding high-bandwidth and low-latency communication, something which is unattainable with commodity GPGPUs and CPUs. We demonstrate that a spiking neuron algorithm can be efficiently mapped to Bluehive using Bluespec SystemVerilog by taking a communication-centric approach. This contrasts with many FPGA-based neural systems which are very focused on parallel computation, resulting in inefficient use of FPGA resources. Our design allows 64k neurons with 64M synapses per FPGA and is scalable to a large number of FPGAs.**

***Keywords*-simulation; neural network; FPGA;**

I. Introduction

Modern FPGAs offer large reconfigurable resources, and high-speed serial and memory interfaces. These FPGAs facilitate construction of highly parallel systems where each FPGA is programmed with the same image to describe a node in the system. This is in contrast to older approaches to using FPGAs to prototype large systems, where one design spans multiple FPGAs. Each node might be a more conventional CPU and coherent memory system (à la RAMP [1]), or an implementation of a highly parallel algorithm (e.g. neural network simulation). Both of these application spaces have high-bandwidth and low-latency communication requirements and benefit from communication in 3D or higher dimensions. They also require plenty of external memory (e.g. 4 GiB per FPGA).

We found it surprisingly difficult to find commodity FPGA platforms which were symmetric (the same design can be used on each FPGA), had plenty of external memory and a suitable communication topology. There are some more exotic boards (e.g. the BEE3 from BeeCube) but we were looking for a more cost-effective commodity solution. Section II describes the FPGA system we constructed using commodity DE4 boards from Terasic together with our own PCIe-to-SATA break-out board and parallel programming solution. This system exploits the many high-speed serial links found on Altera Stratix IV parts, which are becoming a widespread feature of modern FPGAs. Our engineering work has been placed into the public domain to help others build similar systems.

We present a detailed case study of mapping a spiking neural network simulation platform onto our FPGA system. In contrast to much research in this space, we identify that large-scale, real-time neural simulation is primarily communication-bound and not compute-bound (Section III).

Having completed the requirements analysis, we proceed to describe our custom spiking neural network simulator implemented using Bluespec SystemVerilog (BSV [2]) (Section IV). We include observations of the effectiveness of the BSV language to construct such systems.

The performance of our neural network simulator is presented in Section V and we compare our work with that of others in Section VI. Conclusions are drawn in Section VII.

II. Building a FPGA Custom Computer

Modern FPGAs with integrated high-speed serial links simplify the construction of cost-effective multi-FPGA systems. Previous generations of FPGAs had few links so it was common to use as many parallel FPGA-to-FPGA links as possible. Despite running at a lower clock rate, care still needed to be taken to ensure good signal integrity and low skew on these parallel interconnects. Consequently it was common to put several FPGAs onto one large multilayer PCB in order to meet these parallel electrical signalling requirements. Such large PCBs are expensive to design and build and tend to sell in low quantity, which makes their commercial unit cost high. In contrast, single FPGA boards sell in higher volume, thereby reducing unit cost, and modern boards can be interconnected using the many high-speed serial links. The next sections discuss the construction of a 64-FPGA system.

A. Choice of FPGA board

A combination of technical merit and prior experience of working with Terasic lead us to choose the DE4 FPGA board. Its key characteristics are an Altera Stratix IV 230 FPGA with two DDR2 memory channels (satisfying our memory requirements) and various serial links. Four bidirectional serial links are presented as SATA connectors which we can directly use. We also use the PCIe connector with eight bidirectional serial links (see Figure 2), giving us 12 in total, thus $12 \times 6 = 72\,\text{Gbit/s}$ of bidirectional communication bandwidth per FPGA board. A further four serial links presented as $1\,\text{Gbit/s}$ Ethernet were too slow for our uses. There are also two HSMC connectors, one with four and one with eight serial links which we are not using for the moment.

B. Interconnect

We ascertained that we could fit sixteen DE4 boards in an 8U 19" rack box, eight at the front and eight at the back (Figure 1). Serial links from the PCIe $8\times$ connector on each DE4 board are used for communication within the box. We could have designed one large motherboard to connect these together, but this would have been a complex and expensive PCB given the high-speed signalling. Instead we designed a small four-layer PCIe-to-SATA adapter board to break the DE4's PCIe channels out to SATA connectors (Figure 2). SATA links are normally used for hard disc communication since they are very low cost and yet work at multigigabit rates. Using SATA3 links we easily achieve $6\,\text{Gbit/s}$ of bandwidth each way per link with virtually no bit errors (less than 10^{-15}), a data rate much higher than that reported in [3] for FPGA-to-FPGA links. We use these transmission standards at an electrical level only and do not use SATA or PCIe protocols. SATA also comes in an external variety, eSATA, offering better electrical characteristics and a more robust plug, so we use these for for box-to-box communication. SATA and eSATA are capacitively decoupled which gives some board-to-board isolation. A common ground plane makes them suitable for rack scale systems but probably no further due to cable length and three-phase power issues.

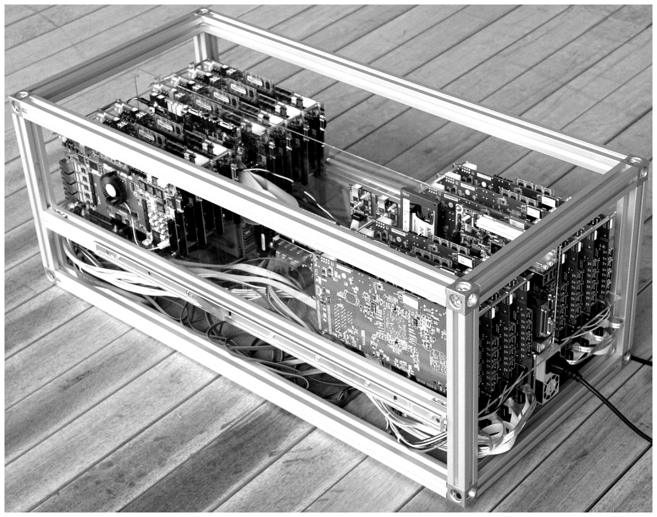

Figure 1. One of the Bluehive rack boxes with side panels removed showing 16 × DE4 boards at the top. There are 16 × PCIe-to-SATA adapters, SATA links and the power supplies at the bottom.

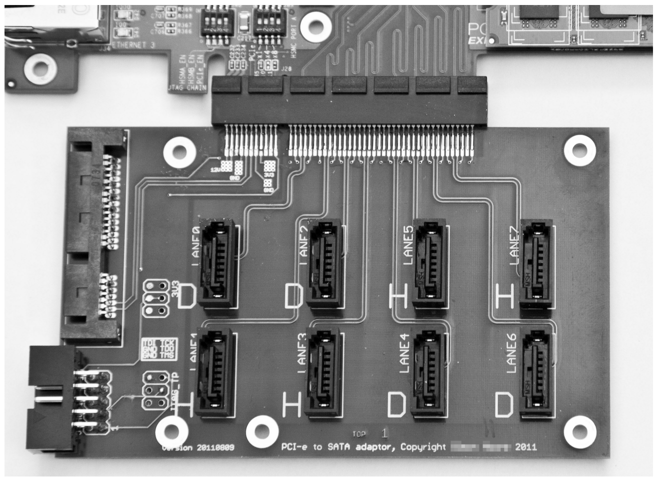

Figure 2. PCIe-to-SATA adapter board connected to a DE4 board providing $48\,\text{Gbit/s}$ of extra bidirectional bandwidth

The PCIe-to-SATA adapter board permits the intrabox communication to use a reconfigurable (repluggable) topology. We chose a PCIe edge connector socket, rather than the more conventional through-hole motherboard socket, to allow the adapter board to be in the same plane as the FPGA board (Figure 2), thereby allowing a 3D "motherboard" of adapter boards to be constructed. This adapter board also provides power, JTAG programming and SPI status monitoring (e.g. of FPGA temperature).

C. Programming and diagnostics

The DE4 board has on-board USB-to-JTAG and we could have connected them via USB hubs to a PC to program them. However Altera's `jtagd` is only capable of programming boards sequentially and, at 22 s per board, this becomes inconvenient for large arrays. Given that we often want to program all FPGAs with the same image, we developed a parallel broadcast programming mechanism using a DE2-115 (Cyclone IV) board to fan-out signals from a single USB-to-JTAG adapter. The DE2-115 board has a selectable GPIO output voltage allowing us to easily match the DE4's JTAG chain requirements. A small NIOS processor system on the DE2-115 board facilitates communication with the programming PC allowing the multicast configuration to be selected. This allows everything from broadcast of an image to all FPGAs through to programming just one FPGA. In broadcast mode, the PC only sees one FPGA but the image file is sent to many FPGAs. We can also make the FPGAs appear in one long JTAG chain, e.g. for communication with JTAG-UARTs on each FPGA board post-configuration.

For diagnostic purposes we also wanted to monitor the on-board temperature and power sensors on each DE4 via an SPI interface. We use a small MAX II CPLD to multiplex the SPI signals between the DE4 boards and the DE2-115 programming board. This MAX II part sits on a simple

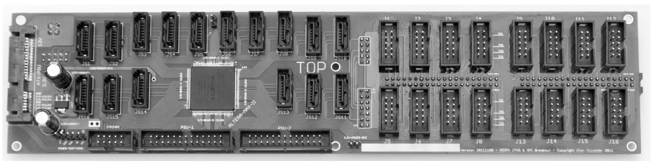

Figure 3. Parallel programming and status monitoring board

two-layer custom PCB that we designed (Figure 3). This PCB also fans out the JTAG programming signals using the standard Altera JTAG header so it could easily be used to program other Altera FPGA boards.

Inside each box we included a mini-ITX based Linux PC to act as a remote programming device. A small power supply powers this PC and the DE2-115 board. Via the DE2-115, the PC can power up a server-class power supply which powers the DE4 boards. This allows the box of FPGAs to be brought up to run a task and then be put to sleep afterwards, reducing standby power to under 10 W. The PC also monitors the DE4 temperatures/voltages and the server power supply status (via an I^2C link), and powers them down if they go outside of the operating range.

D. Open-sourcing reusable components

We have open-sourced the PCIe-to-SATA adapter board and the parallel programming board. The PCIe-to-SATA adapter board is suitable for use with other PCIe based FPGA boards, e.g. the NetFPGA10G. The parallel programming board can be directly used to program other Altera-based FPGA boards and with some adaptation would work with Xilinx-based boards.

III. Spiking neural networks — a communication requirements analysis

We have selected the Izhikevich spiking-neuron algorithm [4] for our neural network simulations as we believe that it offers a good compromise between biological accuracy and computational efficiency [5]. The next subsection briefly introduces the algorithm from a computational point of view before we embark on a communication analysis.

A. An computational view of Izhikevich spiking-neurons

The Izhikevich algorithm uses equation (1) to simulate the spiking behaviour of a neuron. This equation (1) 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 [6]. 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.

$$v' = \begin{cases} 0.04v^2 + 5v + 140 - u + I & v < 30\,\mathrm{mV} \\ c & v \geq 30\,\mathrm{mV} \end{cases}$$

$$u' = \begin{cases} a(bv - u) & v < 30\,\mathrm{mV} \\ u + d & v \geq 30\,\mathrm{mV} \end{cases} \qquad (1)$$

The range of these variables and parameters is bounded, allowing us to use 16-bit fixed-point arithmetic. We break time down into discrete steps of 1 ms, matching prior work [6]. 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, leaving plenty of space on the FPGA. Clearly the problem is not compute bound, so it surprises us that many neuroscientists focus on the computational problem even for the comparatively simple Izhikevich model.

We now look at the data storage requirements of our example of 10^5 neurons per FPGA. 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. This would just fit in the BRAM on the Stratix IV part that we are using but we have chosen to store this in off-chip memory since it does not use excessive bandwidth and results in all of the neuron parameters being held with the synaptic parameters (weights, delays, etc.). This makes it easy to change the neural netlist without changing the FPGA configuration.

B. A communication view of Izhikevich spiking-neurons

In addition to evaluating equation (1) for each neuron it is necessary to communicate spikes between neurons, apply appropriate delays and weights, and sum them 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). But the computation is just addition, so the real challenge is streaming the data through the 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 [7], 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 $6\,\mathrm{GB/s}$.

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), so these values have to be stored in off-chip memory. Fortunately, with two banks of DDR2 memory we have plenty of storage (4 GiB to 8 GiB) and plenty of memory bandwidth (peak $12.8\,\mathrm{GB/s}$ and we typically achieve 75% of the peak) so we can stream the fan-out values from memory. Nevertheless, it is clear that communication from memory to FPGA provides a bound on the number of neurons we can simulate in real-time per FPGA. However, there are two advantages of storing parameters in external memory:

1) The external memory is effectively being used as a giant switch, mapping neuron firings to lists of firing events. This takes care of a great deal of the communication complexity.
2) Neural netlists may be loaded without need to resynthesize the FPGA.

FPGA-to-FPGA communication is governed by the locality and overall size of the 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 [8]. This analysis indicates that there is a great deal of locality. So, for very large networks spanning many FPGAs, we see a great deal of communication between neighbouring FPGAs 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-FPGA, with all target neurons being on different FPGAs. In this case all 6 GB/s of bandwidth is needed (calculated in the previous paragraph). Our 12 SATA links per FPGA give us 72 Gbit/s or 9 GB/s of raw bandwidth so, even with protocol overheads, we can manage 6 GB/s of FPGA-to-FPGA bandwidth. In practice we exploit locality (grouping physically close neurons on the same FPGA) and also use multicast routing between FPGAs. For the results in Section V, we observe a mean bandwidth of 250 Mbit/s between each FPGA board.

Communication latency is also an important consideration. For real-time simulation we must deliver spiking events in well under a 1 ms simulation time-step. Fortunately, our FPGA-to-FPGA links have a mean latency of 10 clock cycles at 200 MHz, so just 50 ns per hop. We aim to keep the network lightly loaded to reduce the risk of congestion, and use low-latency routing, so communication latency across a large FPGA fabric is not a problem.

IV. Architecting a neural network simulator in Bluespec

Given the analysis of the Izhikevich spiking neuron model in Section III, we divided our design into the following functional components:

- **Equation Processor** — performs the neuron computation, i.e. calculating equation (1).
- **Fan-out Engine** — takes neuron firing events, looks up the work to be performed and farms it out.
- **Delay-Unit** — performs the first part of the fan-in phase. Messages are placed into one of sixteen 1ms bins, thereby delaying them until the right 1ms simulation time step.
- **Accumulator** — performs the second part of the fan-in phase, accumulating weights to produce an I-value for each neuron.
- **Router** — routes firing events destined for other processing nodes.
- **Spike auditor** — records spike events to output as the simulation results.
- **Spike injector** — allows external spike events to be injected into the simulated network. This is used to provide an initial stimulus. It could also be used to interface to external systems (e.g. sensors).

First we present our implementation method and then discuss how the above functional components were coded.

A. Implementation method and the BSV language

We chose to implement our system using Bluespec SystemVerilog (BSV) [2] since it is higher-level than Verilog or VHDL but still allows low-level design optimisations. In particular, channel communication can be concisely expressed both within and between modules. This allowed us to easily express the architecture in a 'communicating sequential processes' style. Initially we broke our architecture down into building blocks that could be implemented using small NIOS processors interconnected by channels. These NIOS processors could then be interchangeably replaced by Bluespec components, facilitating incremental refinement and unit testing.

B. Equation Processor

The Equation Processor needs to implement equation (1). We use fixed-point arithmetic and the equation only requires multiply, add, subtract and shift operations. We stream parameters for thousands of neurons every 1 ms (64k neurons per FPGA for the results presented).

I-value updates come from on-FPGA memory but the other parameters are burst-read from off-chip memory. The logic to do this is written in BSV and relies on the BSV compiled Verilog going through the Altera Quartus II synthesis system to map multiplication and addition functions onto the embedded DSP blocks (multipliers, etc.). The Quartus synthesis tool does a good job of this mapping allowing us to focus on the higher-level issues of pipelining and stream management. Note that the parameters a to d do not change, and are only written back to external memory since 256-bit writes are most efficient for our memory configuration.

Inputs:	parameters streamed from external memory (256-bit chunks every clock cycle): (a,b,c,d,u,v) parameter fetched from on-FPGA memory: I
Function:	equation (1)
Outputs:	written to external memory: (a,b,c,d,u,v) conditionally output neuron number to the Fan-out Engine on neuron firing ($v \geq 30\,\mathrm{mV}$)

C. Fan-out Engine

Firing events from the Equation Processors are sent to the Fan-out Engine which uses the source neuron number to read a list of destinations. One neuron firing event typically fans-out to 1000 target neurons, so we use custom engines to efficiently burst-read this information. The first stage of the fan-out groups the synapses by destination node and delay, with the second stage being performed by the Accumulator on the target node.

Input:	neuron number from Equation Processor
Function:	requests list of work to be performed from external memory
Output:	a stream of tuples: (destination_node,delay, pointer_to_neuron_updates,length)

D. Delay Unit

The Delay Unit receives tuples from Fan-out Engines on both the local node and other nodes in the system. It sorts them into one of 16 FIFOs, one for each permitted delay (in 1 ms discrete time steps). FIFOs are assigned to 1 ms delay periods in a cyclical fashion and every 1 ms step, the “2 ms” FIFO logically becomes the “1 ms” FIFO and the “1 ms” FIFO becomes the “0 ms” FIFO, and so on. Work in the current “0 ms” FIFO is drained and sent to the Accumulator.

The number of spike events that can be generated in a given time step is effectively unbounded (limited only by the characteristics of the network), so the FIFOs in the Delay Unit theoretically need to be able to store an unbounded number of tuples. We achieve this using a cached-FIFO design which spills contents to external memory when it fills up. Care is taken to preserve FIFO ordering and exploit burst reads and writes.

Input:	a stream of tuples: (delay,pointer_to_neuron_updates,length) burst-read FIFO data previously spilled to external memory
Function:	sort work into FIFO delay buckets
Output:	a stream of tuples for the current time period: (pointer_to_neuron_updates,length) burst-write FIFO data spilled to external memory

E. Accumulator

The Accumulator is the inner loop of the algorithm and as such it is the most performance-critical when the fan-in is biologically plausible (e.g. 1000 inputs). From the Delay Unit it receives a stream of tuples: (pointer_to_neuron_updates,length). For every tuple, the Accumulator burst-reads a stream of (neuron_number,weight) pairs and performs the I-value updates:

I[neuron_number]+=weight

However, for good performance, we wish to burst-read the update pairs four at a time per clock cycle and so four accumulations need to be performed in parallel. We partition the I-values across eight banks of BRAMs on-FPGA and route the four pairs of updates to the banks that currently represent the I-values that will be used in the next time step. Together with some buffering, this statistical multiplexing means that we will only stall when we get an unusually high number of updates to one bank.

We hold two copies of the I-values on-FPGA, with one copy holding the values that will be used in this time step, and the other copy being updated ready for the next time step.

Input:	a stream of tuples: (pointer_to_neuron_updates, length) a burst-read stream of pairs: (neuron_number, weight)
Function:	read a stream of neuron updates and perform I-value accumulation in parallel
Output:	I-value updates from the previous time-step held in BRAM for Equation Processor to read

F. Router

Firing events destined for Delay Units on other FPGAs are routed off-FPGA using a simple dimension-ordered routing scheme. The destination node number in the tuples produced by the Fan-out Engine is converted into a number of hops in each plane of the network. Events coming into the FPGA are routed to the Delay Unit along with other events originating from the Fan-out Engine on that FPGA.

Input:	a stream of tuples from the Fan-out Engine and SATA links: (destination_node,delay, pointer_to_neuron_updates,length)
Function:	route tuples destined for this node to the Delay Unit, otherwise to external SATA links
Output:	a stream of tuples to the Delay Unit: (delay,pointer_to_neuron_updates,length) and the SATA links: (destination_node,delay, pointer_to_neuron_updates,length)

For the board-to-board SATA links we use a reliable link overlay layer which uses conventional CRC protection and a replay mechanism to guarantee correct delivery. In practice the links are very reliable so the error detection and replay mechanism is rarely needed.

G. Spike Auditor and Spike Injector

The Spike Auditor records the spike events that are generated by the simulation. Each FPGA maintains a record of spike events generated by the neurons that it hosts, which are saved off-chip in simple tuples of simulation time and neuron number. The spike event records from each FPGA are downloaded by the host PC post-simulation and combined to form the record of spike events for the whole simulation.

The Spike Injector allows spike events from outside the simulation to be introduced. This is used to provide some initial stimulus to start the simulation, and could also be used to interface to external systems. The initial spikes are fetched from off-chip memory as tuples of simulation time, neuron number and injected I-value.

H. Parallel System Overview

Figure 6 illustrates the complete system and its hierarchy. The design scales to a large number of FPGAs. The only limiting factor is the FPGA-to-FPGA bandwidth but with our current 3D torus configuration, we achieve a massive 12 Gbit/s of bidirectional bandwidth per channel and require few hops, so the system is highly scalable.

V. 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 [9] tool created by the neuroscience community. Whilst PyNN can produce a range of complete neural netlists, it appears not to scale much beyond 8k neuron, 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 4.

Figure 4 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 shown (black line). Figure 5 is a larger-scale copy of a small section of Figure 4 which more clearly shows the neuron firing pattern.

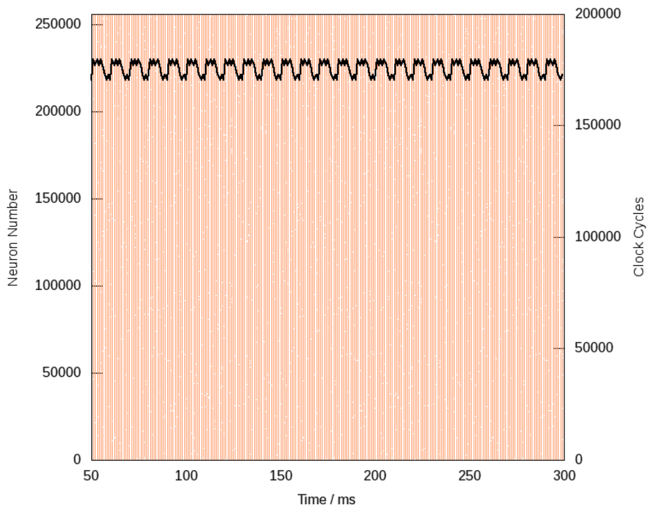

Figure 4. Graph showing per-neuron activity (fine red dots) and the total number of cycles needed to complete every 1 ms step (black line). Real-time achieved if the total number of cycles per 1ms never exceeds 2×10^5, i.e. 200 MHz operating rate.

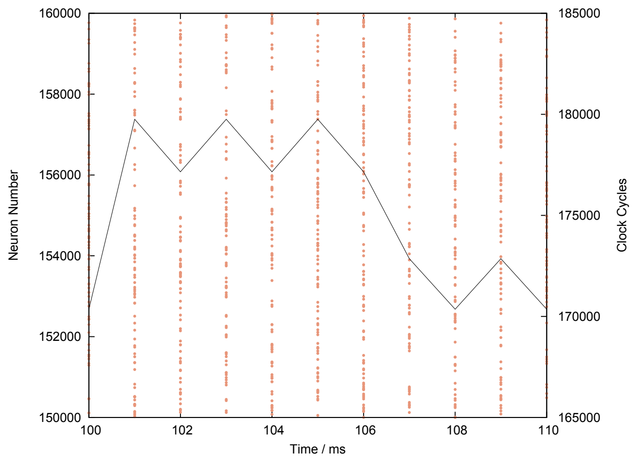

Figure 5. Graph showing a small section of Figure 4 to more clearly show the neuron firing pattern.

Our aim is to build a real-time system (no faster, no slower), and we can run our design at 200 MHz and since the maximum workload for any 1ms period is completed in 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 and our C simulator has similar performance to other reported software simulators [10].

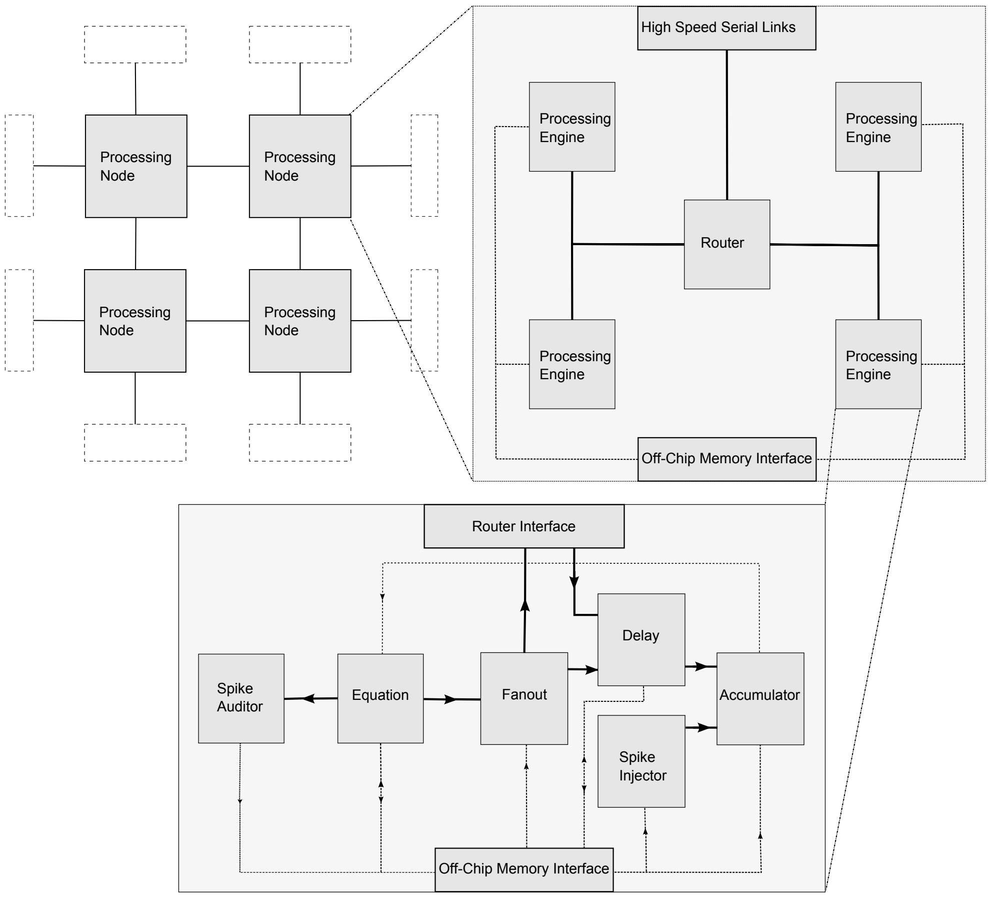

Figure 6. Block diagram of the complete multi-FPGA system

VI. Related Work

Given the requirements in Section III, 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 [6] scales to 10^6 ARM processors using a 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 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 [11]. In contrast, our work is focused on spiking neuron models [5]. Often research focuses on single FPGA implementations [12] where we are interested in parallel FPGA machines, for example Thomas and Luk [10] 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 [12], [13], 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).

VII. Conclusions

Three contributions are made in this paper:

Firstly, a report on engineering work to build a scalable FPGA custom computer called Bluehive. Unlike many other systems, this uses commodity evaluation cards which amortises the development costs over a larger market. We have created custom PCBs to break out serial links to pluggable SATA channels ($12 \times 6\,\mathrm{Gbit/s}$ links per FPGA) and to facilitate broadcast programming of multiple FPGAs. These designs have been placed into the public domain to help others build similar systems.

Secondly, a characterisation of large-scale real-time neural network simulation and an analysis of why FPGAs are much better than current GPGPUs and commodity CPUs for this problem space due to the low-latency and high-bandwidth communication needs.

Thirdly, details of a custom architecture to simulate spiking neural networks in real-time. We take a communication-centric approach which is in contrast to the many computation-centric designs in the literature. This design approach allows us to simulate large networks: 64k spiking neurons per FPGA with 64M synapses. Also, the design is highly scalable to large numbers of FPGAs and we have already demonstrated a four-FPGA system with 256k neurons and 256M synapses. Given the low utilisation of the inter-FPGA bandwidth, we predict linear scaling to at least 64 FPGAs. All of the neural network parameters are held in external memory so changing the netlist is simply a matter of downloading fresh data: no reconfiguration of the FPGAs is necessary. It also allows for run-time plasticity of the network (e.g. for learning). This is in stark contrast to many other FPGA designs where the neural netlist is an integral part of the design and a change to the neural netlist demands resynthesis of the design.

Further work is needed to bring FPGA-based tools to the neuroscience community. Currently the community focuses on smaller networks and produces tools (e.g. PyNN) that do not scale to larger networks. This presents both a challenge and opportunity. The challenge is that the neuroscience community does not provide large neural benchmarks, so we have no clear targets. But there is also an opportunity for FPGA designers to provide tools (hardware and software) to enable extreme-scale neural simulation. We have made a significant further step along this path.

Acknowledgements

Many thanks are due to Chuck Thacker (Microsoft Research) for giving us advice on the PCIe-to-SATA adapter board and to our colleagues Prof. S.B. Furber (University of Mancheter), Prof. A.D. Brown (University of Southampton) and Prof. D. Allerton (University of Sheffield) for collaborating on neural simulation. The UK research council, EPSRC, provided much of the funding through grant EP/G015783/1.

References

[1] Arvind, K. Asanovic, D. Chaiu, J. Hoe, C. Kozyrakis, S.-L. Lu, M. Oskin, D. Patterson, J. Rabaey, and J. Wawrzynek, “RAMP: Research accelerator for multiple processors — a community vision for a shared experimental parallel HW/SW platform,” Berkeley, Tech. Rep., 2005. [Online]. Available: http://ramp.eecs.berkeley.edu/Publications/ramp-nsf2005.pdf

[2] R. S. Nikhil and K. R. Czeck, *BSV by Example*. CreateSpace, Dec. 2010.

[3] A. Schmidt, W. Kritikos, S. Datta, and R. Sass, “Reconfigurable computing cluster project: Phase I brief,” in *Field-Programmable Custom Computing Machines, 2008. FCCM ’08. 16th International Symposium on*, Apr. 2008, pp. 300–301.

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

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

[6] 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.

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

[8] 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.

[9] 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.

[10] 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*, Apr. 2009, pp. 45–52.

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

[12] 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.

[13] 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*, Apr. 2007, pp. 293–294.