Massively Parallel Neural Computation

Paul James Fox

University of Cambridge
Computer Laboratory

Jesus College

October 2012

This dissertation is submitted for
the degree of Doctor of Philosophy

Declaration

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration except where specifically indicated in the text.

This dissertation does not exceed the regulation length of 60 000 words, including tables and footnotes.

Massively Parallel Neural Computation

Paul James Fox

Summary

Reverse-engineering the brain is one of the US National Academy of Engineering's "Grand Challenges." The structure of the brain can be examined at many different levels, spanning many disciplines from low-level biology through psychology and computer science. This thesis focusses on real-time computation of large neural networks using the Izhikevich spiking neuron model.

Neural computation has been described as "embarrassingly parallel" as each neuron can be thought of as an independent system, with behaviour described by a mathematical model. However, the real challenge lies in modelling neural communication. While the connectivity of neurons has some parallels with that of electrical systems, its high fan-out results in massive data processing and communication requirements when modelling neural communication, particularly for real-time computations.

It is shown that memory bandwidth is the most significant constraint to the scale of real-time neural computation, followed by communication bandwidth, which leads to a decision to implement a neural computation system on a platform based on a network of Field Programmable Gate Arrays (FPGAs), using commercial off-the-shelf components with some custom supporting infrastructure. This brings implementation challenges, particularly lack of on-chip memory, but also many advantages, particularly high-speed transceivers. An algorithm to model neural communication that makes efficient use of memory and communication resources is developed and then used to implement a neural computation system on the multi-FPGA platform.

Finding suitable benchmark neural networks for a massively parallel neural computation system proves to be a challenge. A synthetic benchmark that has biologically-plausible fan-out, spike frequency and spike volume is proposed and used to evaluate the system. It is shown to be capable of computing the activity of a network of 256k Izhikevich spiking neurons with a fan-out of 1k in real-time using a network of 4 FPGA boards. This compares favourably with previous work, with the added advantage of scalability to larger neural networks using more FPGAs.

It is concluded that communication must be considered as a first-class design constraint when implementing massively parallel neural computation systems.

For George and Reg, who couldn't be here to see this completed

Acknowledegments

Simon Moore gave many hours of supervision and advice. Theo Markettos proof-read this entire thesis and corrected many typos and other errors. Andrew Moore provided the coffee machine that allowed sufficient work to be put in to get this finished.

None of this would have been possible without the support of my friends and family. Particular thanks go to Cambridge University Yacht Club for providing many opportunities to escape from the pressures of writing this thesis. Hopefully future Commodores will not need to spend a whole week on the phone trying to fix an engine failure half-way across the world.

Contents

Chapter 1

Introduction

Reverse-engineering the brain is one of the US National Academy of Engineering's "Grand Challenges." While there are many ways to go about this, one method is to study the behaviour of real brains and attempt to emulate their behaviour using computer-based computation of an inferred neural network. This process will be referred to as "neural computation" throughout this work.

1.1 Motivation for neural computation

The structure and behaviour of individual neurons is well known, particularly as a result of experiments by Hodgkin and Huxley (1952), who studied the behaviour of giant squid neurons. We also have a good understanding of the high-level functional structure of the human brain based on studies using brain imaging technologies such as MRI scans. However, we currently know very little about how neurons in the human brain are connected to form neural systems capable of exhibiting the functions that we observe.

One way to explore the connectivity of neurons in the brain is to infer candidate neural networks based on observed functionality and then compute their behaviour. The results of these computations can be used to evaluate and refine the inferred neural networks and hence to gain a greater understanding of the low-level structure of the human brain.

Another application of neural computation is creating "brain-like" systems, particularly in the fields of computer vision (Rowley *et al.*, 1998) and pattern recognition (Rice *et al.*, 2009). By emulating the methods used by the human brain to perform visual processing, there is potential to create systems that have higher performance and lower power consumption than those created using conventional computer architectures. Such systems will need to be developed with the support of neural computation.

1.2 Fundamental processes of neural computation

All neural computation systems need to emulate the behaviour of biological neural networks. While they vary in the level of detail and accuracy that they provide (principally as a result of the algorithms used to model the behaviour of neurons and their synaptic connections), they all need to provide:

Algorithms to model the behaviour of neurons and their interaction via synaptic connections

Data structures to store the parameters of neurons and synaptic connections

Communication infrastructure to support modelling of the interaction between neurons

The majority of existing neural computation systems, including PC-bases systems such as NEST (Gewaltig and Diesmann, 2007) and many hardware-based systems such as those by Thomas and Luk (2009) and Fidjeland and Shanahan (2010) have been designed with little focus on either data structures or communication. As a result, the size of neural network that these systems can perform neural computation with is limited to whatever can be handled by a single device (such as a single PC or single custom hardware device) as they lack the ability to scale to handle larger neural networks using additional devices operating in parallel. The state-of-the-art is around 50k neurons and 50M synaptic connections on a single device as exhibited by Fidjeland and Shanahan.

The main exceptions to this are supercomputer-based neural computation systems such as that by Ananthanarayanan *et al.* (2009), which are able to take advantage of the custom communications infrastructure provided by supercomputers. This allows them to perform neural computation with neural networks which are orders of magnitude larger than those that can be handled by other systems (around 10^9 neurons and 10^{13} synaptic connections), but at a financial cost that puts these systems out of reach of the majority of researchers.

1.3 Goals and hypotheses

This work aims to design a neural computation architecture with communication and data structures as first-class design constraints alongside the algorithms used to model neurons and synaptic connections. This will lead to an architecture that can scale to use many devices operating in parallel and hence perform neural computation with neural networks consisting of millions of neurons and billions of synaptic connections in real-time.

Since I aim to perform neural computations at a scale that greatly exceeds the capacity of any single device, I propose the Scalability Hypothesis.

Scalability Hypothesis

The scale of neural network that can be handled by a neural computation system in real-time must be able to be increased by scaling the system to multiple devices

The Scalability Hypothesis indicates that a neural computation system must be a parallel processing system. There are two major types of resources in a parallel processing system:

Compute resources
: Perform a portion of the computation involved in a parallel processing task

Communication resources
: Pass data between compute resources and provide coordination between them so that the complete system produces the correct result for a parallel processing task

If a neural network (as well as the system used to perform neural computation of this network) is considered to be a parallel processing system, then neurons can be considered to be compute resources, while synaptic connections (see Section 2.2) can be considered to be communication resources. This leads to the Communication-Centric Hypothesis.

Communication-Centric Hypothesis

The scalability of a neural computation system is communication-bound, not compute-bound

The Communication-Centric Hypothesis means that the work involved in modelling communication in a neural computation system dominates the work involved in modelling the behaviour of neurons. It also means that the scale of neural network that can be handled by a neural computation system in real-time is bounded by the availability of communication resources in this system rather than the availability of compute resources.

Communication resources in a parallel processing system can be divided into those that provide inter-device communication and those that provide access to memory. This leads to the Bandwidth Hypothesis.

Bandwidth Hypothesis

The scale of neural network that can be handled by a neural computation system in real-time is bounded by inter-device communication bandwidth and memory bandwidth

The Bandwidth Hypothesis provides two important considerations when designing a massively parallel neural computation system:

1. Using an implementation platform that provides sufficient inter-device communication bandwidth and memory bandwidth
2. Using this bandwidth efficiently to maximise the scale of neural network that can be handled by the neural computation system in real-time

This thesis will provide justification for these hypotheses.

1.4 Overview

I begin by reviewing related work on neural computation systems in Chapter 2 to discover what methods have been used to model the behaviour of neurons and their communication, and how the scale of neural computation that can be handled by these systems has been increased, both on a single chip and by scaling to multiple chips. This provides justification for the Scalability Hypothesis and begins to provide justification for the Communication-Centric Hypothesis and Bandwidth Hypothesis.

Chapter 3 analyses the Izhikevich spiking neuron model to determine how its neuron modelling equations can be adapted from floating-point and continuous-time to create a fixed-point, discrete-time neuron model that is better suited to massively parallel neural computation. The communication properties of biological neural networks are also analysed, as they affect the design of both neural computation systems and the benchmark neural networks that are used to evaluate them.

Chapter 4 divides neural computation into separate tasks and analyses the volume of work involved in each task and their memory and communication bandwidth requirements. This justifies the Bandwidth Hypothesis and leads to discussion of appropriate implementation platforms for a massively parallel neural computation system, after which a platform based on a network of Field Programmable Gate Arrays (FPGAs) is selected. This platform is presented in Chapter 5, with discussion of how its features influence the design of a neural computation system implemented on it.

Chapter 6 develops and evaluates an algorithm for modelling communication between neurons that takes account of the Bandwidth Hypothesis and is optimised for a FPGA-based neural computation platform. This platform is implemented in Chapter 7 and evaluated in Chapter 8. Chapter 9 evaluates the three hypotheses proposed in this chapter, draws conclusions and considers future work.

1.5 Publications

The following publications were produced in the course of this work:

Simon W Moore, Paul J Fox, Steven JT Marsh, A Theodore Markettos and Alan Mujumdar; "Bluehive - A Field-Programable Custom Computing Machine for Extreme-Scale Real-Time Neural Network Simulation"; 20th Annual International Symposium on Field-Programmable Custom Computing Machines pp 133-140; April 29 - May 1 2012

Paul J Fox and Simon W Moore; "Efficient Handling of Synaptic Updates in FPGA-based Large-Scale Neural Network Simulations"; 2012 Workshop on Neural Engineering using Reconfigurable Hardware; September 1 2012; Held in Conjunction with FPL 2012

Paul J Fox and Simon W Moore; "Communication-focussed Approach for Real-time Neural Simulation"; 3rd International Workshop on Parallel Architectures and Bioinspired Algorithms pp 9-16; September 11 2010; Held in conjunction with PACT 2010

Chapter 2

Background

2.1 Introduction

To create a massively parallel neural computation system that can scale to computations of large neural networks in real-time, we must understand the tasks involved in neural computation, how they have been approached by other researchers and what limits the scale and speed of a neural computation system.

We begin with an overview of the fundamentals of neural computation, finding that there are two fundamental requirements – modelling neurons and modelling their communication. Modelling neural communication can be further divided into modelling the topology of a neural network and modelling the effect that neurons have on other neurons via synaptic connections. Having identified these fundamental tasks of neural computation, we review how other work has approached them, ranging from methods that closely mirror the processes and structure of biological neural networks to those that aim to emulate only their externally observable behaviour.

2.2 Fundamentals of neural computation

Neural computation is the process of replicating the behaviour of a biological neural network using a computer system. A diagram of a biological neuron is shown in Figure 2.1 on the facing page. It has four major components that are pertinent to neural computation:

Dendrites
: Receive input from other neurons via synaptic connections.

Soma
: The main body of a neuron. It receives input from the dendrites. The aggregated input has an effect on its membrane potential. If the membrane potential passes a threshold then the soma will emit an action potential or "spike," and then reset its membrane potential and become less responsive to input for a period.

Axon
: Carries spikes over some distance. Wrapped in a myelin sheaf to provide electrical insulation. Its end is fanned-out to connect to multiple target neurons.

Synapse

A gap between the axon endings of a source neuron and the dendrites of a target neuron. Introduces a delay to the communication of spikes between neurons and also provides a varying degree of effect of each spike on the target neuron, which results in each target neuron receiving a varying "synaptic update."

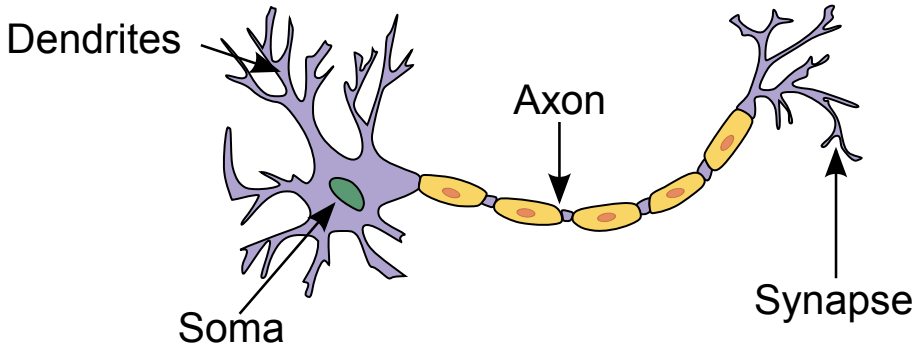


Figure 2.1: Diagram of neuron showing components that are pertinent to neural computation

2.3 Neural computation methods

All methods of neural computation will need to model each of the four major components of a neuron to some degree. Computation methods can be categorised based on how closely they aim to replicate biological processes, both at a chemical and an electrical level.

The soma can be thought of as the computing the state of a neuron, while the dendrites, axon and synaptic connections collectively communicate synaptic updates between neurons. While each neuron has a single soma and axon, there can be many dendrites and synaptic connections (around 10^3 per neuron in the human brain). As will be shown in Chapter 3, this means that the work involved in modelling neural communication greatly exceeds that involved in modelling the behaviour of the soma, particularly if the behaviour of each dendrite and synapse is modelled separately.

Since the axon, dendrites and synapses are all involved in neural communication rather than computing a neuron's state, the remainder of this work refers to modelling the behaviour of the soma as modelling the behaviour of a neuron.

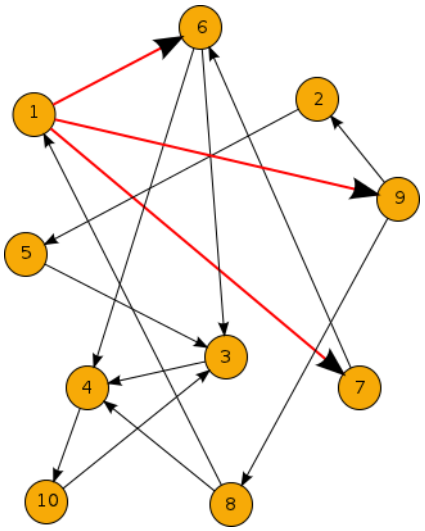


Figure 2.2: A neural network as a number of communicating processes

2.3.1 Mapping neural networks on to computation systems

Before neural computation can be performed with a neural network, it needs to be mapped on to a neural computation system. The first stage of this process involves selecting a neuron model and a communication model. The neural network can then be though of as a number of neuron processes communicating with each other via communication channels which represent axons, synapses and dendrites.

Figure 2.2 shows a neural network modelled as a number of communicating processes. The circles represent neuron processes and the lines communication channels between these processes. The red lines show the communication channels between neuron process 1 and neuron processes 6, 7 and 9. This communicating process model provides a layer of abstraction between neural networks (and in particular biological neural networks) and neural computation systems. Given a suitable model for each neuron process and communication channel, it will be possible to model the behaviour of a neural network with the desired level of biological plausibility and accuracy.

If a neural computation system conceptually consists of a number of connected devices, then each neuron process will need to be mapped on to one of these devices. The communication channels between the neuron processes will then need to be mapped on to the physical communication links between the devices. Since biological neural networks frequently have communication channels that form "non-uniform" patterns, with it being possible for arbitrary neurons in a network to be connected, this means that multiplexing and routing will be needed to overlay these communication channels on to physical communication links.

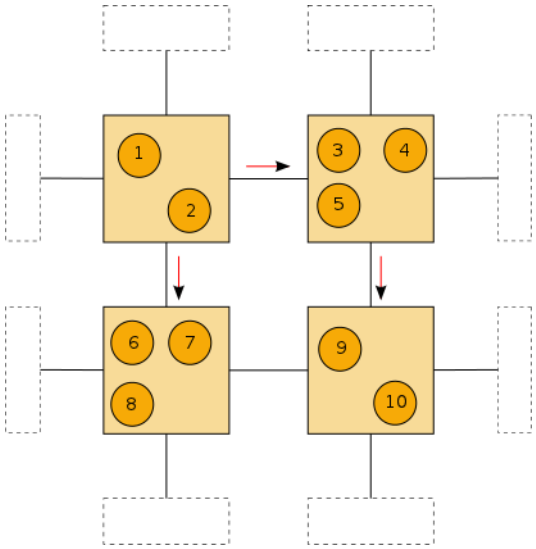


Figure 2.3: Communicating process network from Figure 2.2 mapped on to a neural computation system

For example, the communicating process network in Figure 2.2 could be mapped on to a neural computation system with a number of devices connected in a two-dimensional grid as shown in Figure 2.3. The channels between neuron process 1 and neuron processes 6 and 7 each use one physical link between two adjacent devices, while the channel between neuron process 1 and neuron process 9 uses two physical links and some form of routing on the intermediate device.

2.3.2 Neuron models

Neuron models range in complexity from detailed models that are claimed to be biologically accurate, to much simpler models that aim to model only the externally-observable spiking behaviour of a neuron, known as spiking neuron models. Which type of model is used in a neural computation system depends on the goals of individual researchers and the resources available to them, as many researchers are happy to accept the reduced accuracy of spiking neuron models or believe that the extra detail provided by more biologically accurate models is irrelevant. Using a simpler neuron model also allows neural computation to be performed for larger neural networks than more complex models using the same resources.

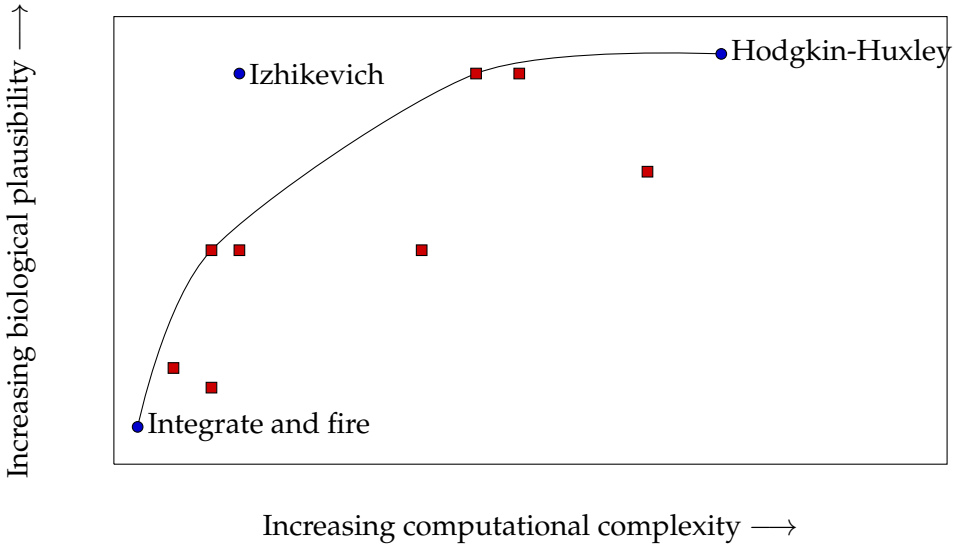


Figure 2.4: Biological plausibility and computational complexity of spiking neuron models. Redrawn from (Izhikevich, 2004). Unlabelled points are other neuron models considered by Izhikevich

One of the most commonly used biologically-accurate neuron models is the Hodgkin-Huxley model (1952) which aims to model the behaviour of a neuron based on observed chemical and electrical processes, so that both the internal and external behaviour of a modelled neuron follow biology. The model uses ten differential equations per neuron.

There are many spiking neuron models, with varying levels of complexity and biological accuracy of the spike patterns that they produce. Some of the simplest spiking neuron models are the integrate and fire (Abbott, 1999) and leaky integrate and fire (Stein, 1965) models, which both model the behaviour of a neuron using a single differential equation. Another, more complex, spiking neuron model is the Izhikevich model (2003), which aims to be able to reproduce many observed types of neuron spike patterns using two differential equations per neuron.

Izhikevich (2004) surveys a number of spiking neuron models, and classifies them based on their biological plausibility and implementation cost (analogous to computational complexity). The results of this analysis are shown in Figure 2.4, and show that (based on the metrics used by Izhikevich) the Izhikevich model has an attractive combination of biological plausibility and computational complexity.

Neurons can also be modelled using electrical circuits that replicate their behaviour, typically using analogue components such as capacitors to model features such as membrane potential. For example the Hodgkin-Huxley model can be specified in terms of an electrical circuit as well as a set of differential equations (Toumazou *et al.*, 1998).

2.3.3 Communication models

Spiking neuron models assume that axons and dendrites are essentially equivalent to electrical wiring, and hence their communication models are restricted to communicating neural spikes and modelling the effect of synapses on the transmission of these neural spikes from source to target neurons. Synapses are normally modelled using a simple model with each synaptic connection having a delay and a multiplicative weight, which can be either positive or negative, to model varying effect of a spike on a target neuron, and hence a varying synaptic update. The weights of each synaptic update targeted at a neuron are summed to produce a single value that is input to the neuron model.

More complex neuron models are often accompanied by complex communication models, with detailed models of the effect of axons, dendrites and synapses on the transmission of neural spikes. Axons and dendrites can be modelled using cable theory (Rall, 1959), while synapse models can have a range of complexities, which can model the effect of both electrical and chemical processes on the transmission of neural spikes.

This results in complex models of the behaviour of the soma, axon, dendrites and synapses such as that used by Traub *et al.* (2005). These models are often designed to model specific regions of the brain (particularly cortical regions), for example the model used by Ananthanarayanan *et al.* (2009) is specific to the cat cortex.

These detailed communication models add orders of magnitude of complexity to a neural computation as the number of dendrites and synaptic connections that need to be modelled is orders of magnitude greater (between $10^3\times$ and $10^4\times$) than the number of neurons that need to be modelled. Therefore such models result in neural computation systems that operate many times slower than real-time (Ananthanarayanan *et al.*, 2009).

Communication models may also implement some form of learning mechanism, such as Hebbian learning (Hebb, 1949). This is often based on altering the connectivity or parameters of synaptic connections (synaptic plasticity) in response to spike activity (Song *et al.*, 2000), which will add computational complexity to a neural computation but will not affect communication complexity.

2.3.4 Implementation methods

There is a major division in neural computation between software-based and hardware-based systems. Software-based neural computation systems support larger neural networks by using more compute resources, with communication

between these resources provided by some form of shared memory. Hardware-based neural computation systems show more variation, with different methods used to model neurons and to model their communication, with work by other researchers often combining different neural modelling methods with different communication modelling methods. Therefore we will survey methods of modelling neurons and of modelling their communication in hardware-based neural computation systems separately.

2.4 Software-based neural computation

In a software-based neural computation system, neurons and neural communication are both modelled in software. Communication resources typically use some form of shared memory. Scaling to larger neural networks is achieved primarily by increasing compute and communication resources (as suggested by the Scalability Hypothesis), but shared memory communication scales poorly without specialised interconnect, so neural computation of large neural networks either requires a supercomputer with a custom memory hierarchy (Migliore *et al.*, 2006) or many multiples of real-time on commodity PC-based systems with finite resources.

2.4.1 Supercomputer-based systems

A notable supercomputer-based neural computation system has been produced by the Blue Brain project (Markram, 2006), using an IBM BlueGene/L supercomputer. A similar system by Ananthanarayanan *et al.* (2009) can simulate a neural network based on the cat cortex with 0.9×10^9 neurons and 0.9×10^{13} synaptic connections around $83\times$ times slower than real-time per Hertz of mean spike frequency (which would be $830\times$ slower than real-time for a mean spike frequency of 10 Hz) with a sampling interval of 1 ms. This saturates the memory bandwidth in an IBM BlueGene/P supercomputer with 147456 CPUs and 144 TB of RAM. The number of CPUs in this system provides evidence for the Scalability Hypothesis.

Ananthanarayanan *et al.* demonstrate that doubling the available RAM in this system almost doubles the size of neural network that can be handled by this system in a given time-scale, which agrees with the Bandwidth Hypothesis. These results are impressive (a neural network around $25\times$ the size of this network would match the scale of the human brain), but the cost, size and power requirements of the supercomputer required by this system puts it out of reach of the vast majority of researchers.

2.4.2 PC-based systems

PC-based systems are more readily available to researchers than supercomputer-based systems, and can be used for smaller-scale neural computations. For example NEURON (Hines and Carnevale, 1997), Brian (Goodman and Brette, 2008) and Nest (Gewaltig and Diesmann, 2007) all use commodity PC hardware. While there is support for using multiple processors in these PC-based systems, they are ultimately limited to running on a single PC as the hardware is not designed to support shared memory between separate systems. For example, Steven Marsh found that a single-threaded neural network simulator written in C required 48.8 s to perform a neural computation of 300 ms of activity for 256k Izhikevich neurons, using the benchmark neural network presented in Section 8.2, which is 163× slower than real-time (Moore *et al.*, 2012). This is broadly comparable to the timescale of supercomputer-based systems, but for a massively smaller neural network scale.

It would be relatively trivial to perform neural computations of many unconnected neural networks using many PCs to create computations with the same total number of neurons as the neural networks used with supercomputer-based neural computation systems, but this is not comparable as the separate, small computations would not model the full set of synaptic connections in the larger neural network. This provides support for the Communication-Centric Hypothesis.

2.4.3 GPU-based systems

Between supercomputer-based and PC-based systems sit a more recent class of neural computation systems based on Graphics Processing Units (GPUs). These can provide real-time performance for reasonably large neural networks (for example Fidjeland and Shanahan (2010) report computations of up to 3×10^4 Izhikevich spiking neurons in real-time), but they struggle to scale to larger neural networks using additional GPUs as communication between GPUs is only available via the CPU in the host PC, a limit which supports the Bandwidth Hypothesis.

2.5 Hardware-based neural computation

Hardware-based neural computation systems typically separate modelling neurons, modelling the topology of a neural network and modelling synaptic connections into separate, inter-connected tasks that operate in parallel, using separate circuits. We will survey methods for performing each of these tasks separately as different methods are often combined in systems by other researchers.

2.5.1 Modelling neurons

Hardware-based neural computation systems can model neurons using both analogue circuits and mathematical models. Between these two modelling methods are models that directly map each neuron into hardware using a dedicated digital circuit with similar behaviour to an analogue neuron model.

Analogue models

Analogue neuron models use an analogue circuit with components that mimic the behaviour of biological neurons. These range from neuron models that closely replicate the structure of biological neurons such as those implemented by Mahowald and Douglas (1991), Saighi *et al.* (2005) and Alvado *et al.* (2004), which all implement the Hodgkin-Huxley model (1952), to those that replicate only the spiking behaviour of biological neurons, such as the array of leaky integrate and fire neurons implemented by Indiveri *et al.* (2006). Basu *et al.* (2010) use a combination of hardware blocks that allows computation with both integrate and fire neurons and more complex multichannel neuron models such as Hodgkin-Huxley.

In some cases the parameters of the neuron model can be altered, using an analogue memory (Saighi *et al.*, 2005) or a more conventional digital memory (Alvado *et al.*, 2004), while in others the ability to alter parameters is more limited, for example only allowing spike threshold voltages to be altered (Mahowald and Douglas, 1991). Basu *et al.* allow the behaviour of their configurable blocks to be altered using programmable floating gate transistors.

The biggest weakness of analogue neurons models is their lack of scalability, for example Indiveri *et al.* (2006) implement 32 neurons on a chip, while Saighi *et al.* (2005) implement 5 and Basu *et al.* up to 84 integrate and fire neurons or a smaller number of Hodgkin-Huxley neurons.

Analogue components have large physical size compared to the transistors used in digital systems, and hence the number of neurons that can be modelled on a single device using analogue neuron models is limited.

Direct mapping into hardware

An evolution of analogue neuron models are digital models which replace each analogue neuron with a digital neuron implemented in a hardware description language (HDL) such as Verilog or VHDL (Bailey, 2010; Upegui *et al.*, 2005). The number of neurons that can be modelled using a single chip is broadly comparable with analogue models. Bailey (2010) implements 100 neurons and 200 synaptic connections on a Xilinx Virtex 5 100T FPGA, while Upegui *et al.* (2005) implements 30 neurons with up to 900 synaptic connections in a more constrained topology on a Xilinx Spartan II xc2s200 FPGA.

The scale of a neural computation system using direct mapping is limited by needing a separate HDL block to represent each neuron, although it is possible to time-multiplex HDL blocks in some cases, such as if the topology of a neural network forms a regular grid (Yang *et al.*, 2011). Maguire *et al.* (2007) take another approach, with neuron state being stored in memory and HDL blocks being allocated to neurons by a coordinating processor, allowing these blocks to be time-multiplexed. This allows significantly more neurons to be modelled per chip, around 2×10^3 on an unspecified Xilinx FPGA.

Mathematical models

Hardware-based neural computation systems can use the same mathematical neuron models as software-based neural computation systems, particularly simple spiking neuron models. Using discrete-time mathematical neuron models allows compute resources to be shared between neurons using time-multiplexing, allowing the number of neurons that can be modelled by each resource to be increased significantly. The proportion of total resources required by each neuron depends on the sampling interval of the neuron model.

One method of evaluating neuron modelling equations in a hardware-based system is using many small processor cores running simple embedded software. This approach is taken by Jin *et al.* (2008), who use a network of custom ASICs with up to 20 small ARM processor cores per chip, with around 10^3 neurons per core.

Another method of evaluating neuron modelling equations in a hardware-based system is converting them into a data-flow pipeline (Thomas and Luk, 2009; Cassidy *et al.*, 2011; Martinez-Alvarez *et al.*, 2007; Rice *et al.*, 2009). Mathematical neuron models are amenable to being converted to pipelines with many stages (Rice *et al.* use 23 stages for an Izhikevich neuron model), as the neuron modelling equations must be evaluated for every neuron in a neural network once per sampling interval, resulting in simple control flow with no branches that would require a pipeline to be flushed.

Since each neuron is represented by a flow of data rather than by physical resources, the throughput of both the embedded software and data flow approaches and hence the number of neurons in a real-time neural computation is limited by the ability of the neural computation system to fetch this data from memory. This agrees with the Bandwidth Hypothesis.

2.5.2 Modelling neural network topology

Modelling the topology of a neural network requires some form of interconnect in a neural computation system. Within an individual device (either custom ASIC, FPGA or CPU) this interconnect can be provided by internal wiring (either fixed, reconfigurable or an on-chip network), but if a neural computation system is to scale to multiple devices (as suggested by the Scalability Hypothesis), some kind of inter-device interconnect will be needed.

Analogue wiring

Pure analogue neural models use voltages on wires between neuron blocks to model the topology of a neural network. For example, Saighi *et al.* (2005) connect their analogue neuron modelling blocks together using reconfigurable wiring controlled by data in a SRAM, similar to the method used to control routing in FPGAs.

Digital wiring

Digital wiring can be used to model the topology of a neural network, particularly when neuron models are directly mapped into hardware using Field Programmable Gate Arrays (FPGAs), and also in some custom neural computation systems, such as those by Alvado *et al.* (2004) and Basu *et al.* (2010). While FPGA-based neural computation systems using this approach are much easier to implement than custom systems as FPGAs are readily available to researchers, the topo-

logy of a neural network can only be altered by resynthesising a FPGA bitstream, which is time-consuming. Altering a neural network topology at run-time is possible using dynamic partial reconfiguration, as done by Upegui *et al.* (2005), but this requires that a portion of the FPGA bitstream is resynthesised for every topology change, which rules out frequent changes.

Modelling the topology of a neural network using wiring that directly mirrors this topology is often limited to single-chip systems as extending this type of topology modelling to multiple chips requires more pins than can be implemented on a chip package, which suggests that this method of neural network topology modelling is not suited to creating neural computation systems for large neural networks.

The number of neurons and connections that can be modelled per chip is also limited by routing resources, as connections between neurons tend not to form regular grid pattens, unless the arrangement of these connections is artificially limited by the architecture (Upegui *et al.*, 2005). This provides support for the Communication-Centric Hypothesis.

Packet-based signalling

The topology of a neural network can be modelled more abstractly by modelling neuron spikes (or other types of communication) using packets that indicates the identity of the source neuron (and possibly other information), commonly called AER packets (Boahen, 2000). These packets must be transmitted to target neurons using some form of network.

The topology of a neural network is then modelled using a routing system that identifies which source neurons are connected to which target neurons. This requires a routing table, which is typically stored in off-chip memory in larger neural computation systems (Jin *et al.*, 2008; Cassidy *et al.*, 2011). The memory needed to store this routing table is typically orders of magnitude larger than that needed to store neuron modelling equation parameters when the number of target neurons per source neuron is high (high fan-out), and hence the scale of neural network that can be handled by these neural computation systems in real-time is bounded by memory bandwidth, supporting the Bandwidth Hypothesis.

Packet-based signalling is particularly suited to modelling the topology of a neural network in neural computation systems that span multiple chips, for example Indiveri *et al.* (2006) use analogue neuron and synapse models with packet-based signalling for inter-chip communication. It can also be used to model the topology of a neural network within a chip using an on-chip network (Dally and Towles, 2001). Packet-based signalling could be used just for long-distance connections,

such as in the system implemented by Emery *et al.* (2009), which arranges neurons into tiles and uses programmable logic for connections within a tile and packet-based signalling for long-distance and inter-chip connections. Other systems such those implemented by Jin *et al.* (2008), Thomas and Luk (2009) and Cassidy *et al.* (2011) use an on- and inter-chip network to model the entire neural network topology.

2.5.3 Modelling synaptic connections

Synaptic connections can be modelled using either analogue models or arithmetic models, both with varying levels of complexity.

Analogue models

The behaviour of synaptic connections can be modelled using analogue hardware, with a separate hardware block for each synaptic connection. This allows for a more complex range of synaptic behaviour (particular plasticity) than delay-weight models. For example Rachmuth *et al.* (2011) implement a detailed model of a single synaptic connection on a single chip using a complex model involving ion channels and supporting synaptic plasticity. Even with less complex analogue synaptic connection models, the number of neurons and synapses that can be implemented on a single chip is limited as analogue synaptic connection models take significant chip area, for example Indiveri *et al.* (2006) report 32 neurons and 256 synapses per chip. This is a mean fan-out of only 8, compared to 10^3 in the human brain. Given the resource requirements of each synapse, it would appear that implementing a neural computation with biologically-plausible fan-out using an analogue synapse model is not possible.

Arithmetic models

If a simple delay-weight synapse model is used then modelling synaptic connections requires determining what the delay and weight of each connection should be, applying the delay and then summing the weights of each synaptic update for each neuron. Jin *et al.* (2008), Maguire *et al.* (2007), Cassidy *et al.* (2011) and Rice *et al.* (2009) all store the delay and weight of each synaptic connection in memory and then fetch it when a spike is received from a source neuron. Basu *et al.* (2010) encode synaptic weights in the interconnect between neuron blocks using floating gate transistors.

Jin *et al.* apply the delay and sum the weights using embedded software and temporary memory while Maguire *et al.*, Cassidy *et al.* and Rice *et al.* use dedicated hardware. Maguire *et al.* provide a dedicated hardware block for each synapse while Cassidy *et al.* use a data-flow processing pipeline and store the resulting summed value in memory before it is input to the neuron model.

With the parameters of each synaptic connection stored in memory, memory bandwidth limits the number of synaptic connections that can be modelled by a neural computation system in a given time period, as suggested by the Bandwidth Hypothesis.

2.6 Design decisions

The evaluation of neural computation systems reveals several major decisions that need to be made when designing a massively parallel neural computation system:

Software or hardware implementation

Software-based neural computation systems allow a wide variety of neuron models to be used, but they are not capable of performing massively parallel neural computation in real-time on platforms available to the majority of researchers. Hardware-based systems add many constraints but show significantly more potential for massively parallel computation.

Neuron modelling

While neuron modelling using analogue hardware has the potential to be significantly more accurate than digital modelling (since analogue hardware operates in continuous-time and digital in discrete-time), analogue hardware implementations are restricted to custom ASICs and show limited potential to scale to massively parallel computations, particularly as analogue components use a large amount of chip area. Hence a discrete-time, digital neuron model is most appropriate for massively parallel neural computation. The Izhikevich model will be used as it has an attractive combination of biological plausibility and computational complexity compared to other models such as integrate and fire.

Synaptic connection modelling

Similarly synaptic connections can be modelled using either analogue or digital hardware. While analogue again gives potential for greater accuracy and complex modelling of synapse behaviour, each synaptic connection will need dedicated hardware, which is unfeasible in a system with millions of neurons and billions of synaptic connections. In the digital domain it is possible to model the behaviour of synaptic connections using a range of models, but a simple model of discrete delay and weight shows the most potential for scaling to a massively parallel neural computation.

Neural network topology modelling

A variety of methods have been used to model the topology of a neural network in neural computation systems. Many are based on providing dedicated wiring that mimics each connection between source and target neurons, often using the reconfigurable routing resources in a FPGA, but these methods exhibit poor scaling. Modelling neural network topology using AER packets shows significantly more scaling potential, particularly when combined with an on- and inter-chip network. This means that the neural network topology will need to be stored in a routing table. The routing table can held in on-chip memory, but this cannot scale to massively parallel neural computations, particularly computations of neural networks with high fan-out as this results in large routing tables that will not fit in on-chip memory. Hence at least some of the data describing the topology of a neural network will need to be held in off-chip memory.

Resource sharing

A system using a discrete-time neuron model allows for the hardware used to model neurons and their communication to be time-multiplexed. Given that the equations used in the neuron model will need to be evaluated at around 1 ms intervals (see Section 3.2.3) and that the pipelines used to evaluate these equations can operate at 100s of MHz, the potential scale of a neural computation is significantly increased by time-multiplexing resources.

Numerical precision

Researchers who aim to create large-scale neural computation systems often use fixed-point arithmetic for their neuron models, as it is less complex than floating-point and hence has greater scaling potential. There will be a loss of accuracy as a result, manifested largely in differing spike times, but this is not particularly concerning as the overall spike pattern resulting from a neural computation (and in particular the probability density of the distribution of these spikes (Berger and Levy, 2010)) is of more interest than the exact timing of individual spikes.

Time-scale

Neural computation systems created by other researchers range from being orders of magnitude slower than real-time to orders of magnitude faster than real-time. The time-scale used by each system depends on the goals of the researchers who created it, and on limitations imposed by the system itself, such as maximum clock frequency or memory and communication bandwidth.

Selection of a time-scale is hence somewhat arbitrary, particularly if a neural computation system is implemented using data-flow pipelines where the volume of data that can be processed in a given time period is effectively fixed by available memory bandwidth, and hence the size of neural network that can be handled by the neural computation system is proportional to the time-scale used and the number of memory channels in the system.

This work uses a time-scale of real-time (that is that the computation of 1 ms of neural network activity takes 1 ms) as this will ease comparison with other large neural computation systems such as that by Jin *et al.* (2008), and also allow for interfacing to real-time systems such as external sensors in the future.

2.7 Conclusion

When creating a massively parallel neural computation system that operates in real-time and that can scale to computations of large neural networks with high fan-out, we have much to learn from other neural computation systems. It is clear that a massively parallel neural computation system must model neurons using a digital, discrete-time neuron model (such as the Izhikevich spiking neuron model) as neural computation with analogue neuron models has only been demonstrated for very small scale neural networks.

It is also clear that neural network topology should be modelled using packet-based signalling over an on- and inter-chip network, as other neural network topology modelling methods do not allow a neural computation system to scale beyond a single chip, which is essential for large-scale neural computation systems. Packet-based signalling also allows time-multiplexing of resources, which allows a massive increase in the scale of a neural computation. Further resource saving and hence increased scale can be achieved using fixed-point arithmetic in place of floating-point in neuron models.

The biggest unsolved challenge is how to efficiently model neural network topology and synaptic connections, which can be referred to as communicating and applying synaptic updates. It is clear that an on- and inter-chip network should be used to do this. The amount of data and bandwidth that will be needed to model the topology of a large-scale neural network and to communicate and apply synaptic updates in real-time must be analysed, as the Bandwidth Hypothesis suggests that the scale of neural network that can be handled by a neural computation system in real-time is bounded by inter-device communication bandwidth and memory bandwidth. We begin by analysing the high-level requirements of massively parallel neural computation in the next chapter and then develop these requirements into system resource requirements in Chapter 4.

Chapter 3

High-level requirements

3.1 Introduction

We have chosen to use the Izhikevich spiking neuron model (2003) to perform massively parallel neural computation. While it is a "black-box" model and its equations bear little direct relation to the internal processes of biological neurons, the model is capable of reproducing the spiking patterns of a wide variety of types of biological neuron (Izhikevich, 2004). Also it has an attractive combination of biological plausibility and computational complexity compared to other models such as integrate and fire, as discussed in Section 2.3.2.

In its original form the model is unsuitable for implementation in a time-multiplexed, digital system as its equations are designed to operate in continuous-time, and so they must be converted into discrete-time alternatives. Further conversion can be applied to replace floating-point arithmetic with fixed-point, to pre-compute constants and to eliminate terms relating to the length of the sampling interval. This yields much simpler versions of the model's equations which are suited to massively parallel neural computation.

The communication properties of biological neural networks are examined to form some initial bounds on the character and number of synaptic updates that a massively parallel neural computation system will need to handle, which will allow the resource requirements of massively parallel neural computation to be analysed in Chapter 4 and help to evaluate the hypotheses proposed in Chapter 1.

3.2 The Izhikevich spiking neuron model

Izhikevich's spiking neuron model (2003) uses a pair of differential equations to model the behaviour of a single neuron. Equation 3.1a calculates the membrane potential of the neuron and Equation 3.1b the refractory voltage. These equations have two variables (v and u) and two constants (a and b). An additional variable (I) represents the sum of all the synaptic updates targeted at the neuron.

An action potential ('spike') is produced if $v \geq 30\,\text{mV}$, in which case v and u are reset by Equation 3.1c and Equation 3.1d respectively. This requires two more equations with two constants (c and d).

$$\frac{dv}{dt} = 0.04v^2 + 5v + 140 - u + I \tag{3.1a}$$
$$\frac{du}{dt} = a(bv - u) \tag{3.1b}$$

If $v \geq 30\text{mV}$ after $\frac{dv}{dt}$ and $\frac{du}{dt}$ are evaluated then

$$v = c \tag{3.1c}$$
$$u = u + d \tag{3.1d}$$

The model appears to be capable of reproducing the behaviour of a range of types of biological neuron by altering parameters a to d. Izhikevich provides several examples (2003).

3.2.1 Conversion to discrete-time

Equation 3.1 operates in continuous-time, which makes it unsuitable for evaluation using a digital system, particular if we wish to increase the scale of a neural computation by time-multiplexing resources.

Thankfully it is straightforward to convert Equation 3.1 to operate in discrete-time. The simplest method is to use Euler integration. Given equations of the form in Equation 3.2 they can be converted to discrete-time equations, sampled at intervals of length δt by using Equation 3.3.

$$\begin{aligned} \frac{dy}{dt} &= f(x, y) \\ y(x_0) &= y_0 \end{aligned} \tag{3.2}$$

$$\begin{aligned} x_{n+1} &= x_n + \delta t \\ y_{n+1} &= y_n + \delta t \cdot f(x_n, y_n) \end{aligned} \tag{3.3}$$

Applying this method to Equation 3.1 yields the discrete-time version of the Izhikevich neuron model in Equation 3.4.

$$v_{n+1} = v_n + \delta t(0.04(v_n)^2 + 5v_n + 140 - u_n + I)$$
$$u_{n+1} = u_n + \delta ta(bv_n - u_n)$$

If $v_{n+1} \geq 30\text{mV}$ after v_{n+1} and u_{n+1} are evaluated then

$$v_{n+1} = c$$
$$u_{n+1} = u_{n+1} + d \tag{3.4}$$

3.2.2 Conversion to fixed-point

Equation 3.4 operates in discrete-time, but still relies on floating-point arithmetic, which is computationally expensive, particularly when aiming for a massively parallel system. By adapting the work of Jin *et al.* (2008), we can convert Equation 3.4 to use fixed-point arithmetic, which can be evaluated using comparatively simple hardware, particularly if multiplication or division can be approximated to a power of 2, allowing simple bit shifts to be used in their place.

I will extend the work of Jin *et al.* by providing additional intermediate steps to preserve δt (they used the simplifying assumption $\delta t = 1$), and to assist in understanding the process involved in transforming the equations.

The first step is to transform Equation 3.4 into Equation 3.5 by factoring out v_n and multiplying by a. The components of the equation that are applied if $v \geq 30\text{mV}$ will be omitted for now as they do not require transformation.

$$v_{n+1} = v_n + \delta t[v_n(0.04v_n + 5) + 140 - u_n + I]$$
$$u_{n+1} = u_n + \delta t(abv_n - au_n) \tag{3.5}$$

The floating-point variables and parameters can now be converted to fixed point by applying scaling factors according to the mapping in Table 3.1 on the facing page. Additional multiplication / division by these scaling factors is used to keep the equations balanced. The result is Equation 3.6.

Variable / Parameter	Scaling Factor
v	p_1
u	p_1
a	p_2
b	p_2
c	p_1
d	p_1

Table 3.1: Mapping of variables / parameters to scaling factors (Jin *et al.*, 2008)

Original Value	Replacement
vp_1	V
up_1	U
abp_2	A
$-ap_2$	B
cp_1	C
dp_1	D

Table 3.2: Mapping used to simplify Equation 3.6 (Jin *et al.*, 2008)

$$v_{n+1}p_1 = v_n p_1 + \delta t[v_n p_1((0.04p_2 v_n p_1)/p_2 + 5p_1)/p_1 + 140p_1 - u_n p_1 + Ip_1]$$
$$u_{n+1}p_1 = u_n p_1 + \delta t([(abp_2)v_n p_1]/p_2 + [(-ap_2)u_n p_1]/p_2)$$

If $v_{n+1} \geq 30$mV after $v_{n+1}p_1$ and $u_{n+1}p_1$ are evaluated then

$$v_{n+1}p_1 = cp_1$$
$$u_{n+1}p_1 = u_{n+1}p_1 + dp_1 \tag{3.6}$$

Notice that the majority of the scaling factors are applied directly to variables or parameters, with just a few additional scaling factors needed to keep the equations balanced. Equation 3.6 can be simplified by storing parameters in memory with the relevant scaling factors already applied. Using the mapping in Table 3.2, this results in the simplified Equation 3.7.

$$V_{n+1} = V_n + \delta t[V_n((0.04p_2 V_n)/p_2 + 5p_1)/p_1 + 140p_1 - U_n + Ip_1]$$
$$U_{n+1} = U_n + \delta t(AV_n + BU_n)/p_2$$

If $V_{n+1} \geq 30$mV after V_{n+1} and U_{n+1} are evaluated then

$$V_{n+1} = C$$
$$U_{n+1} = U_{n+1} + D \tag{3.7}$$

Scaling Factor	Value
p_1	256
p_2	65536

Table 3.3: Scaling factors selected by (Jin *et al.*, 2008)

Operation	Equivalent Shift
Multiply p_1	$<< 8$
Multiply p_2	$<< 16$
Divide p_1	$>> 8$
Divide p_2	$>> 16$

Table 3.4: Mapping of scaling factors to bit shifts (Jin *et al.*, 2008)

The accuracy and ease of evaluation of Equation 3.7 depends on the values of the scaling factors p_1 and p_2. After experimentation, Jin *et al.* selected the values in Table 3.3. Since these are powers of 2, simple bit shifts can be used in place of comparatively expensive multiplication and division to apply these scaling factors, resulting in the equivalent bit shifts in Table 3.4. After applying these transformations (and with the values of p_1 and p_2 fixed as in Table 3.3), Equation 3.7 becomes Equation 3.8.

$$\begin{aligned} V_{n+1} = V_n + \delta t[& \\ & V_n(((0.04 << 16)V_n) >> 16 + 5p_1) >> 8 \\ & + (140 << 8) - U_n + (I << 8) \\ &] \\ U_{n+1} = U_n + & \delta t(AV_n + BU_n) >> 16 \end{aligned} \tag{3.8}$$

Finally the shifted constants in Equation 3.8 can be simplified, resulting in Equation 3.9.

$$\begin{aligned} V_{n+1} = V_n + \delta t[& \\ & (V_n((2621V_n) >> 16 + 1280)) >> 8 \\ & + 35840 - U_n + (I << 8) \\ &] \\ U_{n+1} = U_n + & \delta t(AV_n + BU_n) >> 16 \end{aligned}$$

If $V_{n+1} \geq 30\text{mV}$ after V_{n+1} and U_{n+1} are evaluated then

$$\begin{aligned} V_{n+1} &= C \\ U_{n+1} &= U_{n+1} + D \end{aligned} \tag{3.9}$$

Note that while it might appear that the calculation of U_{n+1} in Equation 3.9 can be simplified by removing the shift operation and reverting to using parameters ab and $-a$ rather than A and B, both of the original groups of parameters need to be shifted to allow them to be stored using a fixed-point representation.

3.2.3 Sampling interval size

In order to use Equation 3.9 to perform neural computation, a value must be chosen for the sampling interval size (δt). If a neural computation system uses time-multiplexed resources then the sampling interval size has a direct effect on the number of neurons in a real-time computation. Given the same resources, using a smaller sampling interval size will result in a more accurate computation of fewer neurons, while a larger sampling interval size will result in a potentially less accurate computation of more neurons.

In addition, there is potential for the Euler integration method to become unstable and produce totally inaccurate results if the sampling interval size is too large. Particularly when used with coupled differential equations such as those in the Izhikevich neuron model, it is possible for the results given by Equation 3.4 to be radically different from those that would be given if Equation 3.1 were evaluated in continuous-time. The fixed-point arithmetic used in Equation 3.6 introduces further inaccuracies.

Parameter	Value
v_0	-70
u_0	-14
a	0.02
b	0.2
c	-50
d	2
I_n	15 for $n >= 22$

Table 3.5: Parameters used to illustrate instability produced by an overly-large sampling interval size. Note that these parameters correspond to those in Equation 3.1, and so must be transformed as described in Table 3.2 before being used in Equation 3.4

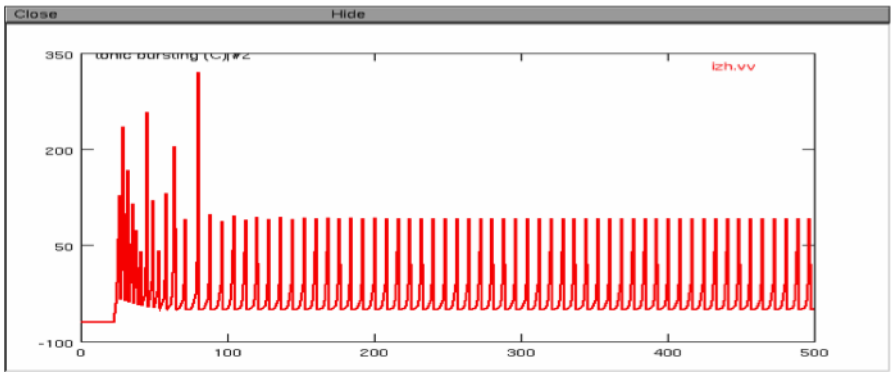


Figure 3.1: Result of evaluating Equation 3.4 with the parameters in Table 3.5 and $\delta t = 1.0$ms.

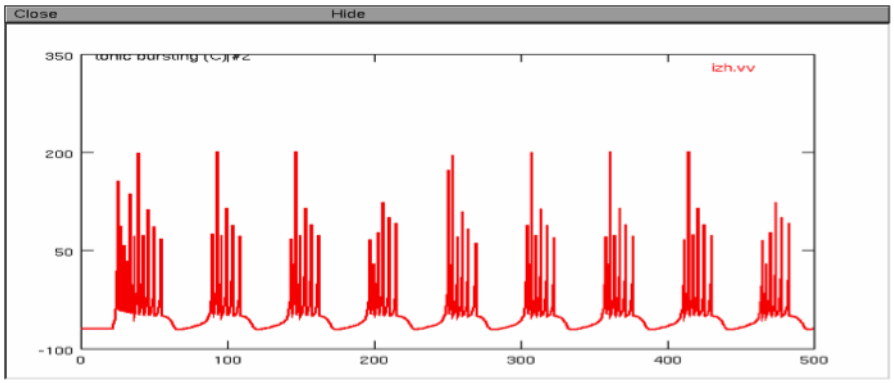


Figure 3.2: Result of evaluating Equation 3.4 with the parameters in Table 3.5 and $\delta t = 0.5$ms.

As an example, consider the effects of evaluating Equation 3.4 with the parameters in Table 3.5 on the preceding page. Figure 3.1 shows the results with $\delta t = 1.0$ms while Figure 3.2 shows the result of evaluating the same equation with $\delta t = 0.5$ms. These graphs were produced using NEURON (Hines and Carnevale, 1997). Figure 3.1 is clearly radically different from Figure 3.2, in particular there are several peaks in Figure 3.1 which correspond to extended flat areas in Figure 3.2.

This simple experiment illustrates that caution is needed when selecting a suitable value for δt. Further experiments with Equation 3.6 using both NEURON (using Equation 3.4) and a simple spreadsheet model in Microsoft Excel (using Equation 3.10) appear to indicate that increasing δt leads to a sudden, sharp transition from results similar (at least in the timing of peaks) to Figure 3.2 to results similar to Figure 3.1. It is likely that the exact value of δt when this transition occurs is dependent on the parameters of Equation 3.4.

Therefore there is a tradeoff between the number of neurons in a real-time computation and the accuracy of Equation 3.4 for a given value of δt. It is possible that there might be a value of δt that is safe regardless of the parameters used. This will not be investigated further – instead the value of δt will be fixed and the operation of the neural computation system will be validated using benchmark neural networks.

3.2.4 Final simplification

If, as suggested by Jin *et al.*, $\delta t = 1\,\text{ms}$, then a final simplification of Equation 3.9 can be made to produce Equation 3.10.

$$V_{n+1} = (V_n((2621 V_n) >> 16 + 1536)) >> 8 + 35840 - U_n + (I << 8)$$
$$U_{n+1} = U_n + (AV_n + BU_n) >> 16$$

If $V_{n+1} \geq 30\text{mV}$ after V_{n+1} and U_{n+1} are evaluated then

$$V_{n+1} = C$$
$$U_{n+1} = U_{n+1} + D \tag{3.10}$$

3.2.5 Numerical precision

The variables and parameters of Equation 3.10 could be stored using either floating-point or fixed-point precision. Using fixed-point precision reduces hardware complexity compared to floating-point and can also reduce the data storage (and hence memory bandwidth) requirements of processing the equation parameters. IEEE single precision floating-point requires 32 bits per value while IEEE double precision requires 64 bits. In comparison fixed-point precision can use any number of bits, though there will be a loss of accuracy compared to floating-point if fewer bits are used, manifesting itself as rounding errors.

Since reducing the memory size and bandwidth requirements of modelling each neuron increases the number of neurons that can be modelled per memory channel, 16 bit fixed-point precision will be used for the variables and parameters of Equation 3.10 (in common with Jin *et al.*), with 32 bit precision used during calculations to prevent significant loss of accuracy. With 2 variables and 4 parameters this means that $6 \times 16 = 96$ bits are needed for each neuron. The loss of accuracy relative to floating-point will manifest itself largely in slightly differing spike

times owing to accumulation of rounding errors. In practice it is likely that this can be mitigated by altering the parameters of Equation 3.10 to bring the behaviour of each neuron in line with what would be expected if floating-point precision were used.

3.3 Communication properties of neural networks

Along with analysing the neuron model, the communication properties of biological neural networks must be analysed. Analysing neural communication properties will facilitate design and evaluation of an algorithm to communicate and apply synaptic updates, as some algorithms have varying workload depending on the volume of synaptic updates. Other algorithms dedicate resources to all possible synaptic connections regardless of whether they are active or not, and hence do not experience additional workload in proportion to the volume of synaptic updates, but these algorithms do not scale well as network size and fan-out increase.

The volume of communication between neurons depends on their:

- Fan-out
- Locality
- Spike frequency
- Synaptic delays

Each of these properties of biological neural networks will be analysed to consider their effect on the volume of synaptic updates that must be communicated and applied and hence methods used to achieve this in a massively parallel neural computation system.

3.3.1 Fan-out

As a rough guide, we will assume that each neuron will send synaptic updates to (have a fan-out of) about 10^3 other neurons. This assumption is based on the human brain containing approximately 10^9 neurons, with 10^{12} synaptic connections. Likewise each neuron will be able to receive synaptic updates from about 10^3 other neurons. However, this is only an assumption regarding the average case, and in principle individual neurons can exhibit significantly greater fan-out.

3.3.2 Locality

Bassett *et al.* (2010) show 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. They find that communication in biological neural networks exhibits a significant degree of locality and modularity, with neurons which are close in physical space forming densely connected sets. Larger sets of neurons which perform more complex functions are then formed from smaller sets, with relatively few additional connections. Hence the majority of communication between neurons covers short physical distances, with there being some medium distance communication, and less still over longer distances. However, it is still possible that any neuron could form a synaptic connection with any other neuron, breaking these principles of modularity and locality.

This suggests that it would be efficient to place neurons which are close together in physical space close together in a massively parallel neural computation system too. We should plan for most communication to be relatively local, but must still allow for the possibility that some synaptic updates might need to be sent between neurons which are further apart in both physical space and in a computation system.

3.3.3 Spike frequency

Mead (1990) suggests that biological neurons can produce spikes at a rate of just a 'few hertz'. We will approximate this to 10Hz, although clearly there will be variation between individual biological neurons.

3.3.4 Synaptic delays

To model synaptic delays, we need to consider both their absolute magnitude (to obtain an upper bound) and the difference in their magnitude that appears to have an observable effect on the behaviour of target neurons (so that the number of distinct delay magnitudes can be determined). In common with Jin *et al.* (2008) we will assume that synaptic delays can range from 1 ms to 16 ms in 1 ms increments.

3.4 Conclusion

A variant of the Izhikevich spiking neuron model that is suitable for use in a massively parallel neural computation system has been developed and the communication properties of biological neural networks have been analysed.

We will assume that biological neurons have a mean spike frequency of 10 Hz and a mean fan-out of 10^3. While individual neurons will deviate from these means, they can be used to estimate the total number of spikes per 1 ms sampling interval and hence the number of neurons that will receive synaptic updates every 1 ms. The next chapter will explore this further, and consider the communication, memory and resource requirements of massively parallel neural computation system.

The benchmark neural network used to evaluate the massively parallel neural computation system developed in this work will build on these assumptions rather than using uniform, random spike activity, as these assumptions are biologically plausible. Many existing neural computation systems have been evaluated using benchmarks which use uniform, random spike activity, often with the mean fan-out of a neuron being proportional to network size (e.g. Thomas and Luk, 2009). Given the discussion of locality in biological neural networks in Section 3.3.2, it would appear that such benchmarks are inappropriate for evaluating a massively parallel neural computation system.

Chapter 4

System requirements

4.1 Introduction

The last chapter analysed the communication properties of biological neural networks and the Izhikevich spiking neuron model. This chapter builds on that analysis to derive some bounds on the communication, memory and resource requirements of real-time, massively parallel neural computation, and hence provide justification for the Bandwidth Hypothesis. The requirements of modelling 10^5 neurons per device with mean fan-out of 10^3 and mean spike frequency of 10 Hz are used as an example.

After analysing these requirements, an appropriate implementation technology can be selected. An implementation based on a network of Field Programmable Gate Arrays (FPGAs) is most appropriate for a number of reasons, particularly the availability of high-speed communication and memory interfaces in recent FPGAs. However, using FPGAs does bring some limitations compared to an Application Specific Integrated Circuit (ASIC) implementation, particularly limited on-chip memory, and so a neural computation system implemented on a multi-FPGA platform must take account of this.

4.2 Neural computation tasks

To model the behaviour of a complete neural network, a massively parallel neural computation system needs to:

1. Provide the variables and parameters of the neuron modelling equation (Equation 3.10) for every neuron every sampling interval
2. Communicate synaptic updates to target neurons when a neuron spikes
3. Delay synaptic updates
4. Sum synaptic updates to produce an I-value for every neuron every sampling interval, which is used in the neuron modelling equation

The last two tasks are perhaps the most subtle of the Izhikevich neural model, as they are not part its neuron modelling equation, or its derivatives. Any implementation of the model will have to delay and sum synaptic updates, as well as evaluating the equation for each neuron. Communicating synaptic updates between neurons also require thought, particularly if massively parallel neural computation is performed using a scaleable system consisting of multiple communicating devices as suggested by the Scalability Hypothesis.

We will now analyse the volume of activity for each of these tasks, which leads to analysis of their memory and communication requirements and hence to choice of an appropriate implementation technology for a massively parallel neural computation system.

4.3 Volume of activity

Neural computation can be broken down into four distinct but linked tasks:

1. Evaluating the neuron modelling equation for each neuron
2. Fetching and communicating synaptic updates
3. Delaying synaptic updates
4. Applying synaptic updates

Considering the volume of activity for each of these tasks provides a basis for analysis of the memory and communication requirements of massively parallel neural computation, which leads to justification of the Bandwidth Hypothesis.

4.3.1 Evaluating neuron modelling equations

The neuron modelling equations will need to be evaluated for every neuron at the desired sampling interval. Chapter 3 defined this as evaluating Equation 3.10 for every neuron every 1 ms. This activity is present throughout the life of a computation and its volume does not change in response to any factor. In particular the workload of evaluating the neuron modelling equations does not depend on the volume of spike activity.

This means that the workload of evaluating the neuron modelling equations provides an absolute lower bound on the resources required for a neural computation to operate in real-time. The only circumstances that might slow the evaluation of neuron modelling equations would be resource contention, for example if the volume of spikes were sufficiently high to cause contention for resources such as off-chip memory between processes evaluating neuron modelling equations and processes communicating and applying synaptic updates.

4.3.2 Synaptic updates

The workload involved in fetching and communicating synaptic updates depends on the volume of these updates that need to be processed. With a mean spike frequency of 10 Hz (Section 3.3.3) and 10^5 neurons per device, there will be $10 \times 10^5 = 10^6$ spike events per second per device. With a mean fan-out of 10^3 (Section 3.3.1) $10^6 \times 10^3 = 10^9$ synaptic updates will need to be fetched and communicated by every device every second.

4.3.3 Delaying synaptic updates

Based on the assumptions in Section 4.3.2, each device will receive a mean of $10^9/10^3 = 10^6$ synaptic updates every 1 ms. These updates will have varying delays, which could have a clear distribution or none at all. It is important to analyse the potential distribution of updates between delays as this has an effect on the total number of updates that are being delayed at any one time. If this total exceeds the capacity provided by the system then either it will deadlock or some updates will have to be discarded, both of which would invalidate a computation. This is in contrast to fetching the parameters of the neuron modelling equations or synaptic updates, where trying to fetch more parameters than there is available bandwidth for would cause a stall and mean that a computation was no longer in real-time, but would not necessarily invalidate its results.

Sample delay distributions

We will explore how the volume of synaptic updates that need to be delayed depends on the distribution of delays in incoming synaptic updates. This will allow us to determine the memory requirements of delaying synaptic updates in Section 4.4.3. Five sample delay distributions are used, with the first four being shown in Table 4.1 on the facing page. Each delay distribution divides the mean 10^6 synaptic updates that arrive every 1 ms between delay sizes from 1 ms to 16 ms.

We also consider a pathological case where the delay size for all 10^6 updates that arrive every 1 ms changes through time between the maximum and minimum so that 16×10^6 synaptic updates finish being delayed at the same time. This distribution is shown in Figure 4.1 on page 58.

Distribution →	Uniform	All min	All mid	All max
Delay / ms	Updates / ms $\times 10^6$			
1	1/16	1	0	0
2	1/16	0	0	0
3	1/16	0	0	0
4	1/16	0	0	0
5	1/16	0	0	0
6	1/16	0	0	0
7	1/16	0	0	0
8	1/16	0	1	0
9	1/16	0	0	0
10	1/16	0	0	0
11	1/16	0	0	0
12	1/16	0	0	0
13	1/16	0	0	0
14	1/16	0	0	0
15	1/16	0	0	0
16	1/16	0	0	1

Table 4.1: Distribution of synaptic updates between delay sizes for sample distributions

Distribution	Maximum updates with same delay completion time $\times 10^6$
Uniform	1
Minimum	1
Median	1
Maximum	1
Pathological	16

Table 4.2: Maximum updates with the same delay completion time for each delay distribution

We will use these sample delay distributions to determine the total number of synaptic updates in the process of being delayed at any one time and also the maximum number of updates that ever have their delays completing at the same time, which will be important when the memory requirements of delaying synaptic updates are examined in Section 4.4.3.

Figure 4.2 on page 59 shows how the number of synaptic updates in the process of being delayed varies over time for each of the delay distributions in Table 4.1, and for the pathological delay distribution in Figure 4.1 on the next page. These figures show that the maximum number of updates in the process of being delayed occurs when all updates need to be delayed for the maximum possible delay. With a maximum delay of 16 ms up to 17×10^6 updates can be in the process of being delayed at any time if 10^6 updates are received every 1 ms. This corresponds to 10^6

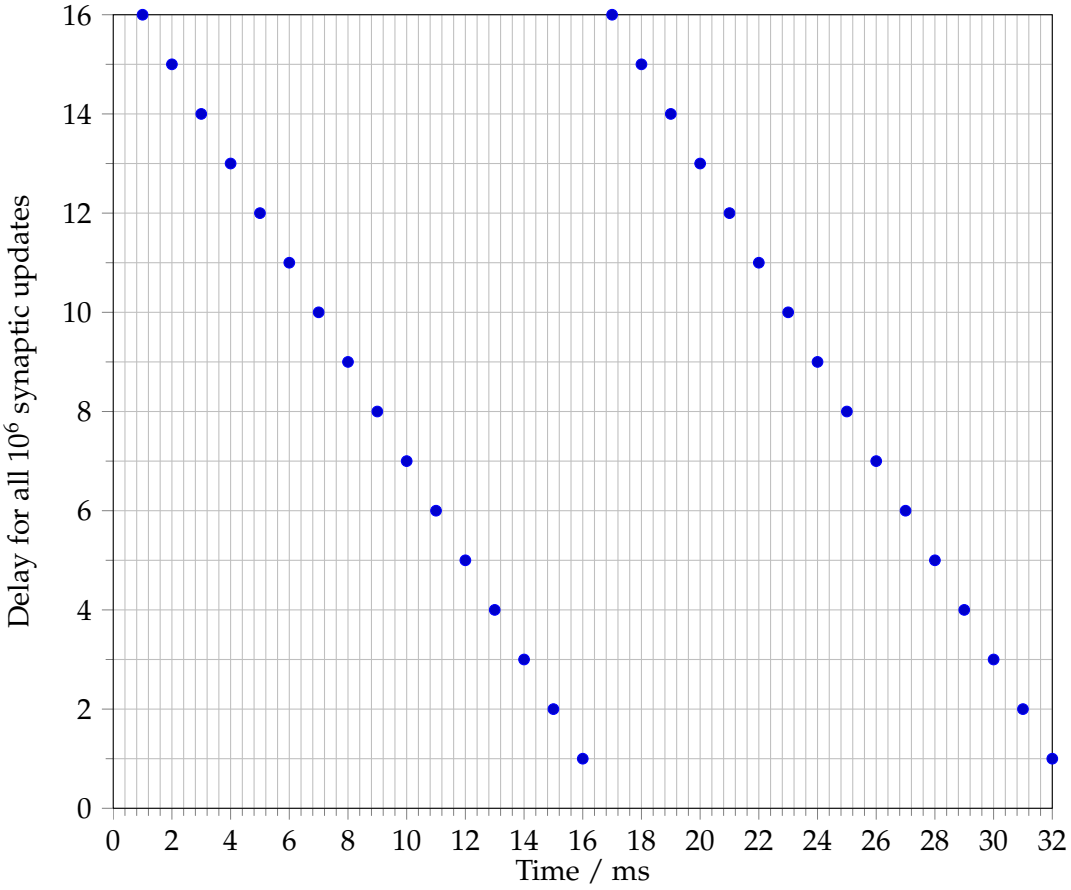


Figure 4.1: Distribution of synaptic update delay size over time that creates a pathological number of updates with the same delay completion time

updates for each possible delay size and a further 10^6 for a delay of 0, representing updates that are currently being applied to their target neurons. This maximum is also periodically exhibited by the pathological delay distribution.

Table 4.2 on the preceding page shows the maximum number of updates with the same delay completion time for each delay distribution. The effect of the pathological, time-dependent delay distribution is apparent here as it is possible for up to 16×10^6 updates to have their delays complete at the same time. Therefore the maximum number of updates that complete their delay at the same time must be considered alongside the total number of updates when designing a system to delay synaptic updates as this affects the bandwidth needed at the output of the delay buffers.

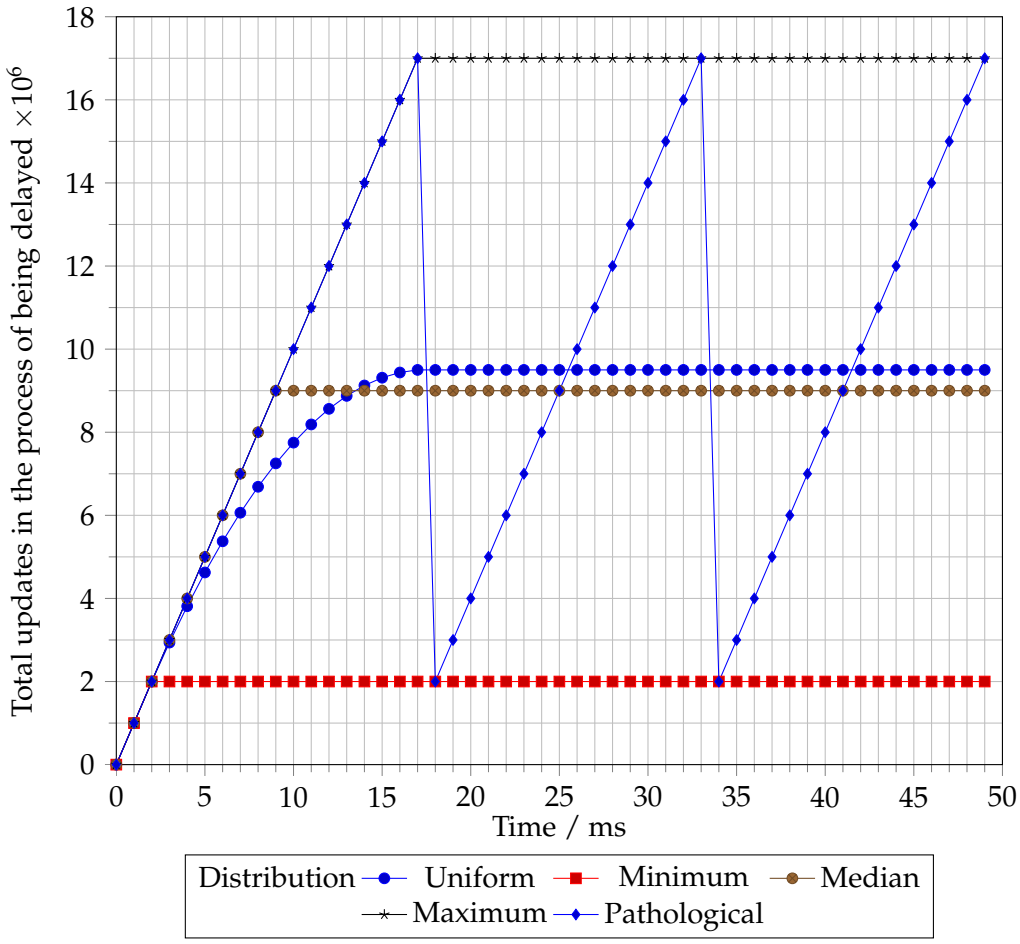


Figure 4.2: Total updates in the process of being delayed over time for each delay distribution. The delay distributions are described in Table 4.1 and Figure 4.1

This model predicts that there are a maximum of $u \times (1 + s)$ synaptic updates in the process of being delayed at any one time, where u is the number of synaptic updates received every 1 ms and s is the number of delay sizes. The total synaptic updates that are in the process of being delayed at any one time can be spread between delay completion times in various ways, with it being possible for up to $u \times s$ updates to have the same delay completion time.

4.3.4 Applying synaptic updates

The volume of synaptic updates that need to be applied by a device every 1 ms will depend on the volume that finishes being delayed. As shown in Section 4.3.3, this could be up to 16×10^6, depending on the distribution of synaptic updates between delay sizes. Any of these synaptic updates could target any neuron on the device.

Operation	Generic	10^5 neurons
Neuron equation evaluations	$E = N \times S$	10^8
Synaptic updates to fetch and communicate	$U = N \times H \times F$	10^9
Synaptic updates arriving to be delayed	$D = N \times H \times F$	10^9
Maximum synaptic updates:		
being delayed at one time	$M_D = (D/S) \times (1 + B)$	17×10^6
completing their delays at the same time	$M_C = (D/S) \times B$	16×10^6
to apply	$M_A = M_C \times S = D \times B$	16×10^9

Table 4.3: Volume of operations performed per second for each task of neural computation

Variable	Meaning	Example value
N	Number of neurons per device	10^5
S	Number of time steps per second	10^3
H	Mean spike frequency (Hz)	10
F	Mean fan-out	10^3
B	Maximum delay length (ms)	16

Table 4.4: Variables and example values used in Table 4.3

4.3.5 Summary

A summary of the volumes of each task of neural computation that need to be performed is shown in Table 4.3. This is shown in both a generic form and as an example for 10^5 neurons per device. The variables and their example values are defined in Table 4.4.

These calculations assume that there is no significant variation in mean spike frequency or mean fan-out. In reality there will be local variations in any biologically plausible neural network, which could cause some devices in a system performing a computation of a large network to be more heavily loaded than others. It is assumed that fan-out is balanced between devices, so for example if some fan-out is to other devices then an equal amount of fan-out will come from those devices to this device. If this were not the case then it would theoretically be possible for many devices to have fan-outs that all targeted a single device, with the number of synaptic updates being delayed multiplied by the number of source devices. This would massively increase the workload of communicating and applying synaptic updates on the target device, and is unlikely to be biologically plausible.

4.4 Memory requirements

We now consider the memory requirements of each task of neural computation, using the volumes of activity identified in Section 4.3. The variables in the equations that follow are identified in Table 4.4 on the facing page.

4.4.1 Evaluating equations

While the overall structure of the neuron modelling equation remains the same for each neuron being modelled, each neuron can have different behaviour defined by different equation parameters. The complete set of variables and parameters will need to be available once per sampling interval. The data size and bandwidth needed to achieve this depends on the data size of each set of neural modelling equation parameters.

Section 3.2.5 found that a set of neural modelling equation parameters can be stored in 96 bits. Rounding this up to 128 bits or 16 bytes allows for some extra data to be stored to simplify communication and application of synaptic updates (see Section 6.3.1), and allows a better fit with memory systems that deal with word widths that are a power of 2.

Therefore the total set of neural modelling equation parameters takes $128 \times N$ bits, which for the example of 10^5 neurons per device is 1.6 MB. The total bandwidth required to access these parameters is $128 \times N \times S$, which is 1.6 GB/s for the example. Since all of the parameters for all neurons will need to be accessed once per sampling interval, from start to end with no repeated accesses, there is no temporal locality but perfect spatial locality. This needs to considered when specifying an appropriate memory hierarchy to store and access these parameters.

4.4.2 Synaptic updates

To determine the data size and bandwidth needed to communicate and apply synaptic updates, we need to start by considering what items of data need to be stored for each synaptic update and how large each item needs to be. The items that are required, exactly where each item is stored and how much data is needed will vary depending on the implementation of the synaptic update communication and application algorithm.

However we will start by assuming a simple, worst-case algorithm that stores the target neuron, delay and weight for each synaptic update separately. The data size needed for each item can be determined by examining their expected maximum

Field	Minimum	Maximum	Bits
Target	0	4294967295	32
Delay	1	16	4
Weight	-2048	2047	12
		Total	48

Table 4.5: Ranges and bit sizes of each item in a synaptic update

and minimum magnitudes. These and the resulting number of bits required are shown in Table 4.5. Every potential target neuron is expected to have a unique identifier.

These data sizes allow for a computation of up to 4.2 billion neurons, with synaptic connections having positive and negative weights and delay sizes that match previous work (Jin *et al.*, 2008). In reality the synaptic update communication and application algorithm implemented in Chapter 6 finds each target neuron via two layers of indirection, allowing for computations of much larger numbers of neurons, but these figures are sufficient to illustrate the general data size and bandwidth requirements.

With 48 bits = 6 bytes of data per synaptic update, $6 \times N \times F$ bytes will be needed for the complete set of synaptic updates. For the example this is 6×10^8 bytes or 0.6 GB. This is almost three orders of magnitude more than is needed for the neural modelling equation parameters.

The number of synaptic updates that need to be fetched from memory per second is $U = N \times H \times F = 10^5 \times 10 \times 10^3 = 10^9$. At 6 bytes per synaptic update this will require $6 \times 10^9 = 6\,\text{GB/s}$ of bandwidth. This makes it clear that the scale of neural network that can be handled by a neural computation system in real-time is bounded by inter-device communication bandwidth and memory bandwidth, as suggested by the Bandwidth Hypothesis.

4.4.3 Delaying synaptic updates

Synaptic updates need to be held in some form of buffer to delay them. Since the delay size data for each incoming synaptic update can be discarded once the update has been buffered, and assuming that a buffered update must target a neuron on the local device (see Section 6.3.3), each buffered update will consist of a local neuron identifier and a weight. If local neuron identifiers are allowed to range up to 10^5 (matching the example number of neurons on a device) then $17 + 12 = 29$ bits will need to be buffered per update. We will round this up to 32 bits or 4 bytes.

If each of these updates were buffered individually than the maximum total size of the buffer (regardless of whether it is subdivided into separate buffers for each delay size or not) would be $4 \times 16 \times 10^6 = 64\,\text{MB}$, plus another 4 MB for the buffer currently being emptied. It is assumed that data will not be copied between buffers, rather they will be used to represent delays in a circular fashion (see Section 7.2.3).

The bandwidth required to this buffer depends on the volume of updates that need to be delayed and the distribution of their delay sizes, which was discussed in Section 4.3.3. With the pathological delay size distribution from Section 4.3.3 the maximum bandwidth required will be $4 \times 10^6 \times 10^3 = 4\,\text{GB/s}$ for input and $4 \times 16 \times 10^6 \times 10^3 = 64\,\text{GB/s}$ for output. In the more balanced cases exhibited by the other delay distributions, a maximum of 4 GB/s of bandwidth will be needed for both input and output, giving 8 GB/s total, another justification for the Bandwidth Hypothesis.

As mentioned previously, the delay buffer must not be allowed to fill up as this would either deadlock the computation or cause it to become invalid. We must find a way to ensure that this does not happen, with the worst case being that the computation slows so that it is no longer real-time, as would be the case if the volume of synaptic updates being processed at any given time were too high. It is also possible to significantly reduce the amount of data that needs to be buffered to achieve a delay to a set of synaptic updates, and hence the bandwidth needed, as will be shown in Section 6.6.2.

4.4.4 Applying synaptic updates

Each I-value is 16 bits, matching the precision of the other variables and parameters of the neuron modelling equation. Any of these I-values could be updated by any of the 16×10^9 synaptic updates that need to be applied every second. Since applying these updates is the critical inner loop of neural computation, this process will be particularly sensitive to any delay in fetching or updating the I-values. Two copies of the I-values need to be stored to ensure that the values being updated do not conflict with the values being read by the neural modelling equations.

With 10^5 neurons per device $2 \times 2 \times 10^5 = 400\,\text{kB}$ of storage will be needed for I-values. Up to 16×10^9 updates may need to be applied every second, requiring a bandwidth of $2 \times 16 \times 10^9 = 32\,\text{GB/s}$ for read and the same for write, a total of 64 GB/s. Since I-values need to be accessed relatively randomly to apply synaptic updates (assuming that incoming synaptic updates are applied in order of arrival), and any latency in these accesses would slow or deadlock processing, the I-values

will be best suited to being stored in any on-chip memory that is provided by an implementation technology. This is in contrast to memory accesses for other neural computation tasks, which deal with lists of instructions that are less sensitive to latency, provided that throughput is maintained.

4.4.5 Summary

This analysis shows that a massively parallel neural computation system will need the memory resources in Table 4.6 to perform a neural computation with 10^5 neurons per device, a mean spike frequency of 10 Hz and a mean fan-out of 10^3. These requirements will be essential when an implementation technology for a massively parallel neural computation system is selected in Section 4.6 and when it is decided how to partition data between on- and off-chip memory in Chapter 6.

Summing synaptic updates to produce I-values is the critical inner loop of the neural computation algorithm, and so the I-value data is the most sensitive to latency of all that used in neural computation, as well as having high locality of access. Therefore the I-value data should be given highest priority when deciding how to allocate whatever high-speed memories are provided by an implementation technology. The bandwidth required to delay synaptic updates is high relative to the total data size required, and therefore this data should be given the next highest priority for high-speed memory.

Operation	Size	Bandwidth
Neural modelling equations	1.6 MB	1.6 GB/s
Fetch synaptic updates	0.6 GB	6 GB/s
Delay synaptic updates	68 MB	8 GB/s
Apply synaptic updates	400 kB	64 GB/s

Table 4.6: Memory resources needed for a neural computation of 10^5 neurons

4.5 Communication requirements

When a neural computation scales from a single device to multiple devices, as the Scalability Hypothesis expects to be necessary for neural computations of large neural networks, there must be communication between these devices. This communication will be made up of control messages (particularly to start the computation and ensure that sampling intervals are synchronised) and messages resulting from the computation itself. We will concentrate on messages related to the computation rather than control messages as the latter will have very limited volume (of the order of a few tens of messages per sampling interval).

4.5.1 Type of communication infrastructure

It is assumed that an on- and inter-chip network will be used to communicate synaptic updates rather than dedicating resources to every possible synaptic connection, as the analysis of neural computation systems in Chapter 2 suggests that the latter will not scale to more than one device. This means that the volume of network traffic will be dependent on the frequency of neural spikes and their fan-out. The volume of network traffic together with routing efficiency will affect scalability, and hence bound the size of neural network that can be handled by a neural computation system in real-time.

4.5.2 Synaptic update communication

Messages between devices only need to be used when there are synaptic connections between neurons on one device and neurons on another device. We will initially assume that each synaptic update is transmitted from the source to the target device individually (unicast) as this will provide an upper bound on communication bandwidth requirements.

Section 4.4.2 found that each synaptic update required 6 bytes of data. If we conceptually consider that all synaptic updates will require some form of communication (even if it is just to the local device) then with 10^5 neurons per device, a fan-out of 10^3 and a mean spike frequency of 10 Hz, 6 GB/s of bandwidth will be needed to communicate synaptic updates.

The proportion of this bandwidth that needs to be provided between devices depends on the locality and overall size of a neural network. Since the discussion in Section 3.3.2 expects biological neural networks to exhibit significant locality, for very large neural computations, spanning many devices we expect to see a great deal of communication between neighbouring devices and very little communication over any distance.

An absolute upper bound on inter-device communication bandwidth can be found using the pathological case that all 10^5 neurons on a device have synaptic connections to neurons on other devices. In this case all 6 GB/s of bandwidth needs to be provided by inter-device links.

Communication latency is also an important consideration. For real-time neural computation we must ensure that all synaptic updates are communicated and applied to their target neurons in well under a 1 ms sampling interval.

4.5.3 Summary

An on- and inter-chip network will be used to communicate synaptic updates. The maximum bandwidth that may be needed between devices is 6 GB/s, which will decrease both with increased locality and if a more efficient method is used to communicate synaptic updates than unicast messaging. Since many other neural computation systems are not able to provide this amount of bandwidth between devices they are communication-bound as suggested by the Communication-Centric Hypothesis.

4.6 Implementation technology

The choice implementation technology for a massively parallel neural computation system is governed by many factors, particularly ability to provide the memory and communication resources that were identified in Section 4.4 and Section 4.5. We also need to consider other factors such as cost and usability. Implementation technology also affects many details of the design of a massively parallel neural computation system, particularly how synaptic updates are communicated and applied, and strategies that are used to make efficient use of memory resources. The principle choice of implementation technology is between Application Specific Integrated Circuits (ASICs) or Field Programmable Gate Arrays (FPGAs). We will explore how the requirements of a massively parallel neural computation system map to the characteristics of these technologies.

4.6.1 Memory resources

Compared to what can be implemented in an ASIC, FPGAs have limited on-chip memory, for example the Altera Stratix IV 230 FPGA has approximately 2 MB of Block RAM (BRAM). This constrains the implementation of a massively parallel neural computation system, for example it would not be possible to store all of the I-values for 10^5 neurons in BRAM on the example FPGA as well as the neuron modelling equation parameters. Certain types of structure are hard or impossible to implement in a FPGA, for example content addressable memories, which are often used as part of the implementation of operations such as multicast routing. Both ASICs and FPGAs are able to make use of large off-chip memory resources, for example DDR2 SDRAM.

4.6.2 Communication resources

The volume of communication that a massively parallel neural computation system needs between devices requires high-speed transceivers. While it is possible to implement these 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 compared to older processes. In contrast these transceivers are readily available in high-end commercial FPGAs and so they are available at significantly less cost in a FPGA implementation than in an ASIC implementation.

4.6.3 Reprogrammability

ASICs give no scope for reprogramming, and hence limited scope for fixing errors or altering many aspects of system behaviour. This means that massive effort is 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 as errors can be resolved by reprogramming, while much more effort is needed to avoid errors in ASICs as they could lead to an entire batch of chips and the photographic masks needed to fabricate them being written off.

Reprogrammability also allows significant scope architectural exploration. This could allow for a system to become more efficient over time, to have aspects of its behaviour altered (for example using a different neural modelling equation) or to be migrated to a different platform with comparatively little effort, for example moving to cheaper FPGAs that have sufficient resources for the design after an initial implementation with larger FPGAs.

4.6.4 Cost

ASICs have a very high initial cost, both as a result of the design effort noted above and for producing the photographic masks that are needed to produce chip wafers. The unit cost of each chip is then typically much lower than that of a FPGA, though it depends on yield, which is often a function of the complexity and size of a design. Overall this means that an ASIC implementation is not cost-effective if overall volumes are low, but it becomes competitive as volumes increase.

The unit cost of a FPGA is relatively independent of volume, though discounts for bulk purchases are often available. This makes FPGAs significantly more cost-effective than ASICs for low volume applications, becoming less cost-effective as

volumes increase, if there are no other factors affecting the choice of implementation technology. However FPGAs largely remove the risks involved in producing faulty batches of ASICs, and the costs of producing ASICs in a process that can implement high-speed transceivers need to be considered. This means that the volume at which an ASIC will be more cost-effective than a FPGA is much higher than would be the case for a simpler design, particularly one without high-speed transceivers.

Another factor to consider is the cost of producing a PCB to power each device and provide it with connectivity and peripherals. A custom ASIC implementation will need custom PCBs. These can be expensive to design and manufacture, particularly when high-speed signalling is involved. In contrast 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, allowing the evaluation board manufacturer to amortise PCB design and production costs.

4.6.5 Power consumption

An ASIC will typically have lower power consumption than a FPGA when performing an identical task. This is because the ASIC will provide precisely the resources that are required by that task, while the FPGA must implement these resources using its network of Lookup Tables (LUTs), which will typically require more transistor switching, and hence more power to achieve the same result. However FPGAs will often be implemented in a more modern, smaller-scale process, which will have the effect of reducing power consumption when compared to an ASIC using an older process.

4.6.6 Choice of implementation technology

Given these considerations I have chosen to implement a massively parallel neural computation system using a network of FPGAs. This choice is driven primarily by prior experience and the availability of high-speed transceivers. It will also have the useful side-effect of creating a platform that could be used for a wide range of applications beyond massively parallel neural computation. This will amortise hardware costs over many projects. It also gives an opportunity to experiment with a range of architectures for a massively parallel neural computation system and select whatever is most suitable, which would not be possible without reprogrammability.

4.6.7 Implications of this choice

When designing a massively parallel neural computation system using a FPGA implementation, we must consider the very limited amount of on-chip memory available (2 MB for an Altera Stratix IV 230). As discussed in Section 4.4.4, two copies of the I-values for the neurons being computed on the FPGA must be stored in on-chip memory (and hence FPGA BRAM) to avoid a performance bottleneck.

With two copies of the I-values at 16 bits per neuron the absolute maximum number of neurons that could be computed on a FPGA with 2 MB of BRAM is $\frac{2\,\text{MB}}{16\,\text{bit}\times 2} = 256\text{k}$. However in practice FPGA designs cannot make use of every available resource, and BRAM will also be needed for many other purposes, particularly delaying synaptic updates.

While there is clearly not enough BRAM available to delay the maximum possible number of synaptic updates using only the BRAM (e.g. 64 MB is needed in the pathological case from Section 4.3.3), there are many cases where the amount of BRAM available will be sufficient. Therefore a hybrid solution will be used to delay synaptic updates, with delays being implemented primarily using BRAM, with excess data being transferred to off-chip memory as required. This will be elaborated on in Section 7.2.3.

Beyond these uses of BRAM, all other data (other than structures such as temporary buffers for processing data) will be kept in off-chip memory. Given the calculations in Section 4.4.5, this would require a total off-chip memory data size of 0.6 GB and bandwidth of 7.6 GB/s for a neural computation of 10^5 neurons with a mean fan-out of 10^3 and mean spike frequency of 10 Hz.

Given the bandwidth available using off-chip memory (e.g. around 6 GB/s for DDR2 SDRAM), this gives some implementation challenges:

- Keeping utilisation of off-chip memory high so that bandwidth is not wasted
- Communicating synaptic updates between FPGAs without multicast routing that is reliant on large amounts of BRAM
- Optimising the organisation of data in off-chip memory to reduce the bandwidth required to access it to a level that is achievable using currently available technology

As Section 6.6 will show, using unicast routing to communicate synaptic updates has poor scaling as it uses communication resources in direct proportion to fan-out, so an algorithm to communicate synaptic updates must be designed that approximates multicast routing 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 communicated in real-time, and by extension the size and scalability of a neural computation system, as proposed by the Bandwidth Hypothesis. The routing algorithm must be designed to make maximum use of off-chip memory bandwidth, and this means making efficient use of burst read transactions.

4.7 Conclusion

I have chosen to implement a massively parallel neural computation system using a network of FPGAs. The multi-FPGA platform that will be used for this is introduced in the next chapter. FPGAs provide a good balance between performance and cost, and significantly reduce development costs and risks. However they pose implementation challenges, particularly related to the limited amount of on-chip memory that they provide.

The next chapter will introduce a multi-FPGA platform that is suitable for implementing a massively parallel neural computation system. Then Chapter 6 will develop an algorithm to communicate and apply synaptic updates in a massively parallel neural computation system that takes account of the limitations of a FPGA implementation.

Chapter 5

Multi-FPGA platform

5.1 Introduction

In the last chapter it was decided to implement a massively parallel neural computation system on a platform with a network of FPGAs on commercial off-the-shelf evaluation boards. The platform, christened "Bluehive" was constructed by Simon Moore and A. Theodore Markettos. This chapter discusses the design of this platform and the FPGA evaluation boards that it uses. Many of its features are pertinent to the design of the algorithm for communicating and applying synaptic updates in Chapter 6.

5.2 Choice of FPGA board

While it is clear that massively parallel neural computation needs FPGAs that provide high-speed transceivers and memory interfaces, these FPGAs must be placed on PCB boards to allow them to function, and to link them together via their transceivers to create a multi-FPGA platform. There are a number of design choices.

5.2.1 Number of FPGAs per PCB

In the past, the only way to create multi-FPGA platforms was placing many FPGAs on a single large, multi-layer PCB, as inter-FPGA links had to use parallel signalling, and so care was needed to ensure good signal integrity and low skew. This is still done (e.g various Dini Group products), but large PCBs for multi-FPGA systems are very expensive commercially, as a result of the raw cost of producing the large PCB, the cost of design effort (particularly routing the parallel links to avoid skew) and low volume of sales.

The introduction of high-speed serial transceivers has greatly simplified routing of links between FPGAs in some ways, as less wires are needed per link, however the higher clock rates involved mean that care is still needed to avoid signal integrity issues. However, differential signalling allows these links to operate at high speeds over relatively long cables. Factors such as the orientation of the cable and the proximity of other similar cables have relatively little bearing on the signal integrity of each link.

This means that it is easier, cheaper and more convenient to design a multi-FPGA platform which uses pluggable cabling for all links between FPGAs than a platform using multiple FPGAs on a single large PCB. With one FPGA per PCB the design effort of creating the PCB is significantly reduced, manufacturing costs are lower and volumes higher. All of these factors reduce the commercial unit cost of these single FPGA PCBs.

5.2.2 Memory interfaces

In Section 4.4 it became clear that each FPGA needs a high-bandwidth off-chip memory interface. This needs to support burst reads to allow the available bandwidth to be used as efficiently as possible, keeping delays for setting up reads and writes to a minimum.

The most suitable type of memory for this application is one of the variants of DDR SDRAM, as a well-implemented memory controller can fetch two words every memory clock cycle, and translate this to providing very long words to the main FPGA logic at a lower clock frequency.

5.2.3 Connectivity

A variety of physical interconnects have been used on FPGA boards to link them together. It is important that the physical interconnect can be used to construct topologies that are suitable for a massively parallel neural computation system. Some commercial FPGA evaluation boards have suffered from suboptimal interconnects in the past, for example the Terasic DE3 evaluation board provides interconnect using 4 High Speed Mezzanine Connectors (HSMC), using parallel signalling. These connectors are very short and not very flexible (to ensure signal integrity), and in addition the single DDR2 SDRAM connector shared pins with one of the 4 HSMC connectors. This constrained any multi-FPGA implementation using this board to a topology of either a stack, linear row or a ring, and in all cases the physical arrangement of boards had to match the topology, so a ring required that the boards be formed into a ring stood on end. This is not suited to creating a large multi-FPGA platform.

Therefore the connections between FPGA evaluation boards must use flexible cables, so a variety of communication topologies can be implemented without having to change the physical arrangement of boards in a multi-FPGA platform. Either the board must provide suitable connections or it must be possible to adapt what is provided.

5.2.4 Symmetry

Some multi-FPGA platforms (particularly those with multiple FPGAs on a PCB), have a different pin-out for each FPGA, for example some FPGAs may be connected to off-chip memory or other I/O devices while others are not. These could be called "non-symmetrical" platforms. Different FPGA bitstreams will be needed for some or all of the FPGAs when implementing a system on a non-symmetrical platform, which makes creating or modifying such a system complex and time-consuming.

An alternative approach is to use a "symmetrical" platform, where every FPGA has identical peripherals and connectivity. This allows the same bitstream to be used for all FPGAs if desired, which greatly simplifies creating and modifying a system implemented on the platform.

A symmetrical platform is most appropriate for implementing a massively parallel neural computation system as each FPGA performs a portion the computation and (assuming that the neural network description is held in off-chip memory) each FPGA can use an identical bitstream.

5.2.5 Availability of off-the-shelf solutions

If a commercial manufacturer can design a single product that can be used by many customers for a wide range of applications, it will make economic sense for them to do so, and unit costs are significantly reduced for all customers. This is significantly more likely to be the case for PCBs that have a single FPGA per PCB and that have a range of interconnects and peripherals than it would be for a PCB with multiple FPGAs or with only the features required by a given application.

Since such off-the-shelf solutions exist, it is not sensible to go to the effort and incur the cost of designing a custom PCB. However this could lead to having to make some design trade-offs if the mix of interconnect and peripherals provided by a off-the-shelf solution is not perfectly suited to the needs of a massively parallel neural computation system.

5.2.6 Summary

The most important considerations when creating a multi-FPGA platform for a massively parallel neural computation system are memory interfaces and interconnect. Given the benefits of using commercial off-the-shelf evaluation boards with

a single FPGA on each board, Simon Moore decided to identify and use a suitable board that meets the requirements of a massively parallel neural computation system and other similar projects. A combination of technical merit, suitable peripherals and prior experience of working with the manufacturer lead to the Terasic DE4 FPGA evaluation board being selected.

5.3 Terasic DE4 evaluation board

The Terasic DE4 evaluation board has a single Altera Stratix IV 230 FPGA, and the following features which are pertinent to the design of a massively parallel neural computation system:

- 228k logic elements
- 2 MB of Block RAM
- 2 DDR2 SO-DIMM sockets, which support up to 4GB of RAM each
- Bidirectional, high-speed serial transceivers:
  - 4 presented as SATA connectors
  - 8 presented as a PCI Express (PCIe) edge connector
  - 4 presented as 1 Gbit/s Ethernet
  - 8 presented in one HSMC connector
  - 4 presented in another HSMC connector

The layout of the board and location of each of these features is shown in Figure 5.1 on the next page.

As identified in Section 5.2.3 and Section 5.2.2, the high-speed serial transceivers and DDR2 memory are particularly pertinent to the design of a massively parallel neural computation system. At the time that construction of the multi-FPGA platform commenced, this evaluation board was one of only a few that provided high-speed transceivers, particularly in a physical format that facilitated constructing a scaleable multi-FPGA platform.

5.3.1 Memory hierarchy

There are three types of memory available on the DE4 board. Their capacity, latency and bandwidth are summarised in Table 5.1 on page 77. The bandwidth of BRAM depends on how it is accessed. There are multiple blocks, each with separate read

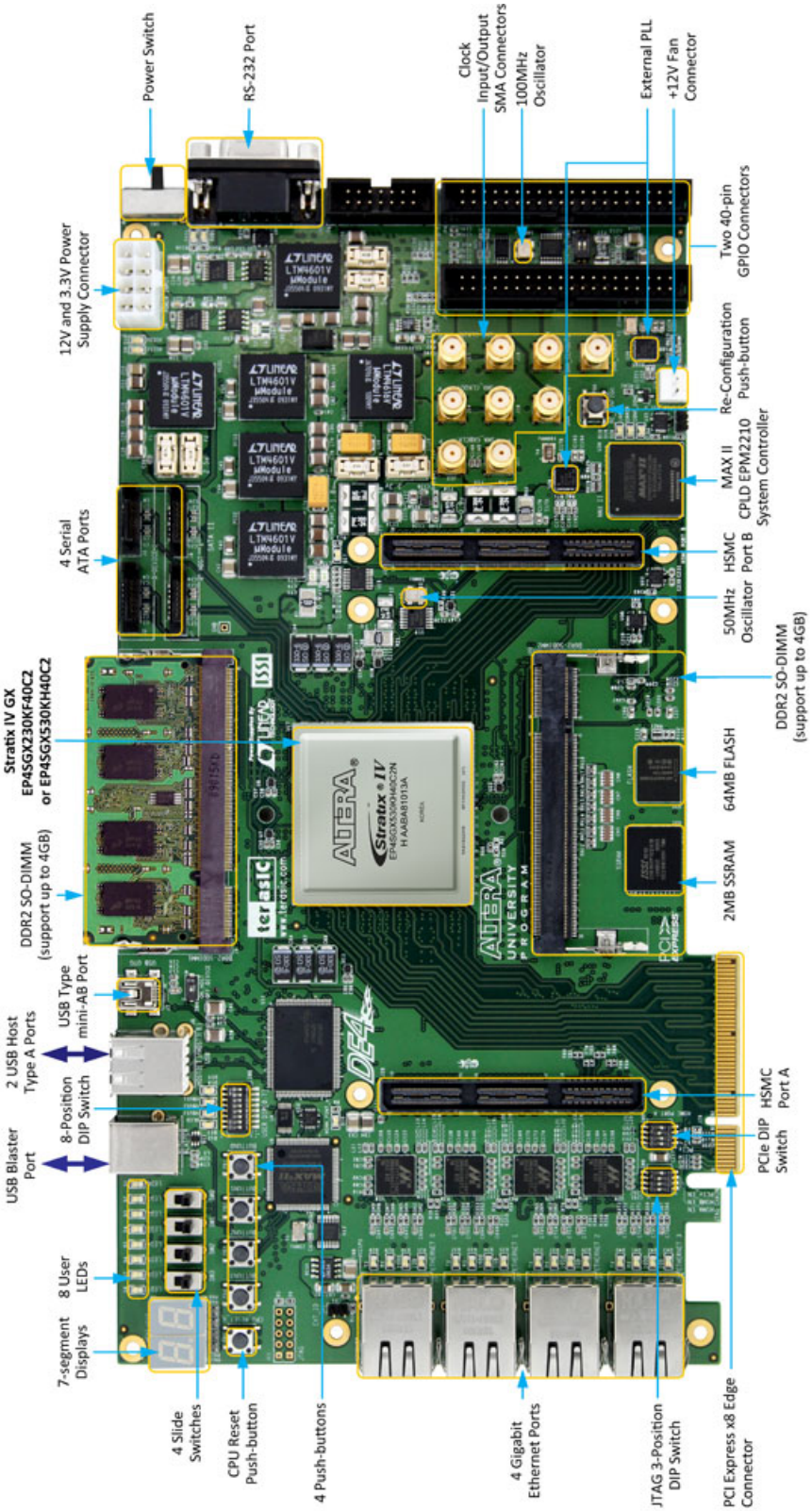


Figure 5.1: Terasic DE4 evaluation board and its major components (Image provided by Terasic)

Type	Size / MB	Latency / clock cycles	Bandwidth / GB/s
BRAM	2	1	Various
SRAM	2	3	0.2
DDR2	2000-4000	5	6

Table 5.1: Memories available on a DE4 board

and write ports. This allows for many parallel accesses and hence it is inefficient to construct large memories (in terms of the percentage of BRAM used) as this would effectively leave some read or write ports unused.

There are two DDR2 banks which are physically separate and use separate controller logic (although they may share some timing logic). Each supports up to 4 GB of RAM, although only a single bank of 1 GB is being used at present. Note that the latency of DDR2 is very dependent on access patterns. If burst reads and writes are used then the effective latency when accessing a contiguous set of words is reduced compared to accessing non-consecutive words

5.4 Creating a multi-FPGA platform

It was ascertained that 16 DE4 boards could be fitted in an 8U 19" rack box, 8 at the front and 8 at the back (Figure 5.2 on the next page). Suitable methods needed to be found to connect them and to provide power, programming support (JTAG) and management facilities.

5.4.1 Interconnect

Of the high-speed interconnects on the DE4 board:

- 4 presented as SATA connectors can be used directly
- 8 presented as a PCIe edge connector need adapting to be used outside of a host PC
- 4 presented as Ethernet are too slow for any use other than connecting to a network for management purposes. They are further limited by an Ethernet interface chip between the FPGA transceivers and the physical connectors
- 12 presented as HSMC connectors are difficult to use for interconnect directly, as they are designed for custom peripherals. Optical transceiver adapters are available, but they are expensive, and will be best suited to communication between racks in a larger multi-FPGA platform in future

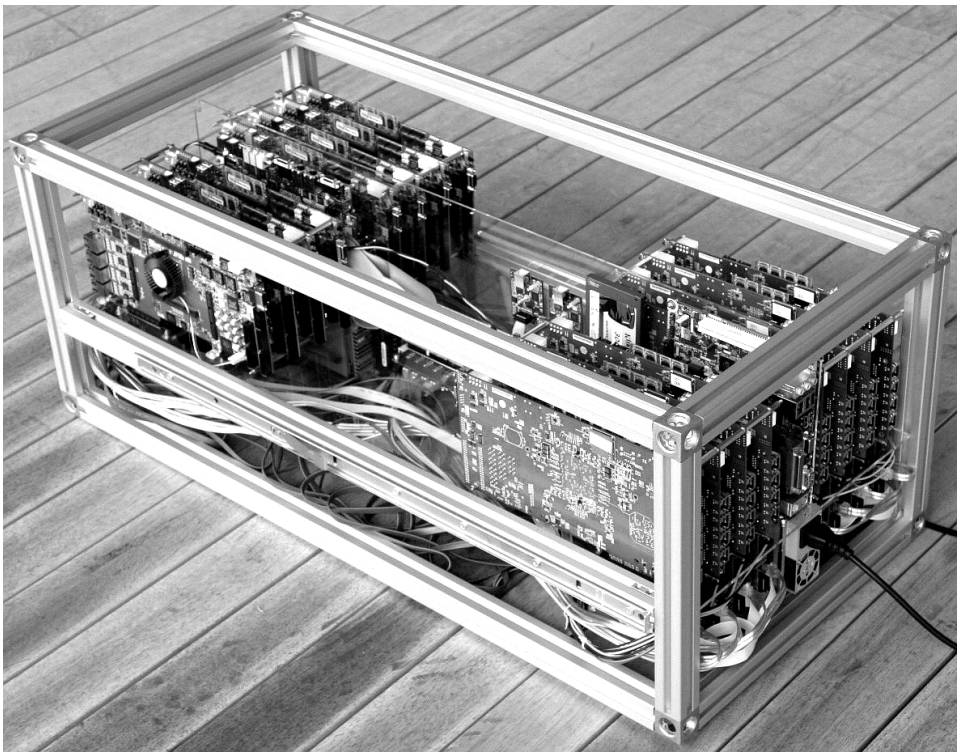


Figure 5.2: One of the multi-FPGA boxes with side panels removed showing 16 × DE4 boards at the top. There are 16 × PCIe-to-SATA adapters, SATA links and power supplies at the bottom.

The combination of SATA and PCIe gives 12 high-speed interconnects in total, thus $12 \times 6 = 72\,\text{Gbit/s} = 9\,\text{GB/s}$ of bidirectional communication bandwidth per FPGA board. This is higher than the pathological maximum bandwidth requirement of 6 Gbit/s for 10^5 neurons per FPGA with a fan-out of 10^3 and a spike frequency of 10 Hz identified in Section 4.5.2. Even allowing for overheads and before any optimisation this is more bandwidth than a massively parallel neural computation system is ever likely to need barring even more extreme pathological cases.

To allow the interconnects in the PCIe 8× edge connector to be used to link multiple FPGA boards, one large motherboard could have been designed, but as noted in Section 5.2.1 this would have been a complex and expensive PCB given the high-speed signalling.

Instead a small, four-layer PCIe-to-SATA adapter board was designed to break the DE4's PCIe channels out to SATA connectors (Figure 5.3 on the facing page). SATA links are normally used for hard disk communication since they have very low cost and yet work at multi-gigabit rates. Using SATA3 links Simon Moore and Theo Markettos found that it was easy to achieve 6 Gbit/s of bidirectional bandwidth per link (Moore *et al.*, 2012) with virtually no bit errors (less than 1 error in 10^{15} bits), a data rate that is much higher than that reported by Schmidt *et al.* (2008)

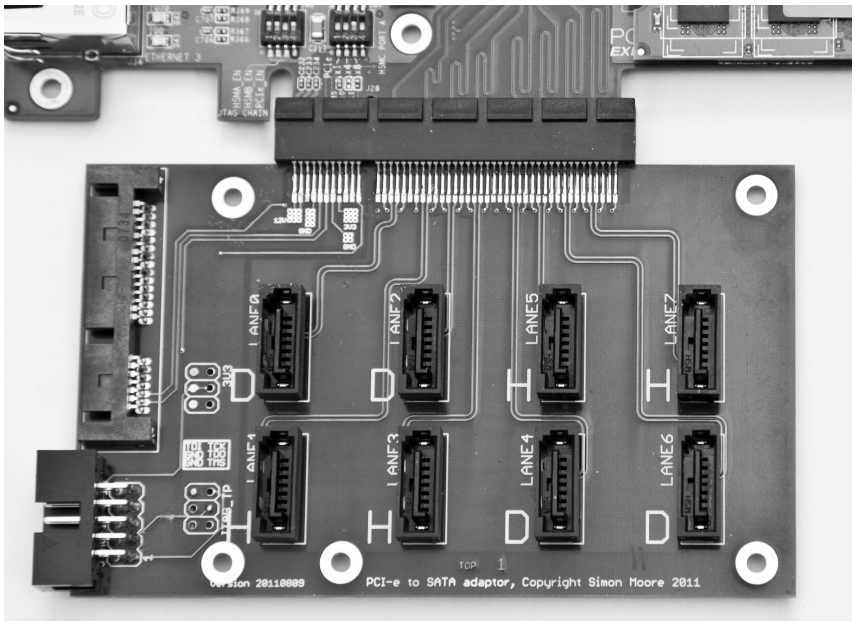


Figure 5.3: PCIe-to-SATA adapter board connected to a DE4 board

for FPGA-to-FPGA links. The SATA and PCIe connections are used at an electrical level only. A custom protocol is used rather than the SATA or PCIe protocols (see Section 7.4).

SATA is also available in an external variety, eSATA, offering better electrical characteristics and a more robust connector, so eSATA connectors are used for for box-to-box communication, connected to the 4 SATA ports on the board via an adapter. SATA and eSATA are capacitively decoupled which gives some board-to-board isolation. A common ground plane makes them suitable for rack scale platforms but probably no further due to cable length and three-phase power issues, necessitating the use of optical interconnects.

The PCIe-to-SATA adapter board permits intrabox communication to use a reconfigurable (repluggable) topology. A PCIe edge connector socket was chosen, 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 5.3), allowing a 3D "motherboard" of adapter boards to be constructed. The adapter board also provides power, JTAG programming and SPI status monitoring (e.g. of FPGA temperature).

5.4.2 Programming and diagnostics

The DE4 board has an on-board USB-to-JTAG adapter (Altera USB Blaster) which could have been used to connect multiple FPGA boards to a PC via USB hubs to program them. However Altera's `jtagd` is only capable of programming boards sequentially and, at 22 s per board, this becomes inconvenient for multi-FPGA platforms.

Given that it is often desired to program all FPGAs in a system with the same bitstream, a parallel broadcast programming mechanism was developed by Theo Markettos, using a DE2-115 (Cyclone IV) board to fan-out signals from a single USB-to-JTAG adapter. The DE2-115 board has selectable GPIO output voltage, allowing it to be easily matched to the DE4's JTAG chain requirements. A small NIOS processor system on the DE2-115 board facilitates communication with the programming PC, allowing the programming configuration to be selected. This allows everything from broadcast of a bitstream to all FPGAs in a system through to programming just one FPGA with an individual bitstream.

In broadcast mode, the PC only sees a JTAG chain with one FPGA, but the bitstream is sent to many FPGAs, depending on the programming configuration selected. It is also possible to make the FPGAs appear in one long JTAG chain, e.g. for communication with JTAG-UARTs on each FPGA board after configuration.

For diagnostic purposes the on-board temperature and power sensors on each DE4 can be monitored via an SPI interface. A small MAX II CPLD is used to multiplex the SPI signals between the DE4 boards and the DE2-115 board. This MAX II sits on a simple two-layer custom PCB (Figure 5.4). This PCB also fans-out the JTAG programming signals from the DE2-115 using the standard Altera JTAG header. The DE2-115 controls the configuration of the fan-out via the MAX II.

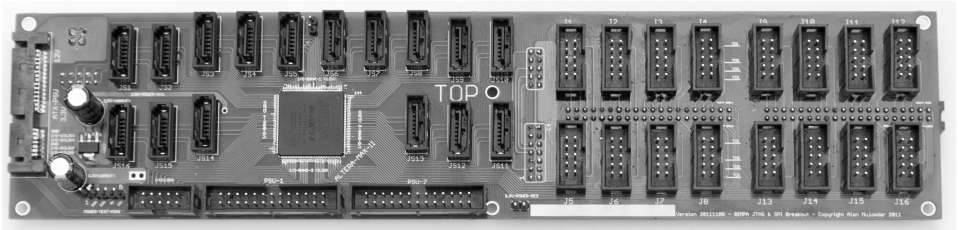


Figure 5.4: Parallel programming and status monitoring board

5.4.3 Management

There is a mini-ITX based Linux PC inside each box to act as a remote programming and management device. A small power supply powers this PC and the DE2-115 board. The PC can power up a server-class power supply, via the DE2-115, which powers the DE4 boards. This allows the multi-FPGA platform to be powered 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 and voltages and the server power supply status (via an I^2C link), and powers them down if they go outside of the desired operating range.

5.5 Conclusion

The Bluehive multi-FPGA platform will be used to implement a multi-FPGA, massively parallel neural computation system. It has many features that make it suited to this task, particularly interconnect and off-chip memory interfaces.

Section 4.4 found that BRAM needs to be conserved, with the majority of the data that describes a neural network kept in off-chip memory, and that access to this memory would become a bottleneck that limits the scale and speed of neural computation, as proposed by the Bandwidth Hypothesis.

On the DE4 board the majority of the off-chip memory is DDR2 SDRAM, which is most efficiently accessed using burst reads and writes. Therefore a massively parallel neural computation system implemented on the multi-FPGA platform introduced in this chapter must make efficient use of burst reads and writes, as this will make most efficient use of memory bandwidth and hence maximise the scale of a real-time neural computation.

Chapter 6

Communicating and applying synaptic updates

6.1 Introduction

Neural computation would be straightforward to parallelise if each neuron was modelled in isolation, without synaptic connections. However such a computation would not yield useful results, and so modelling synaptic connections must be included in the scope of a massively parallel neural computation system. This proves to be a significant challenge to parallelise, as predicted by the Communication-Centric hypothesis.

When a neuron spikes, we need to determine what synaptic connections exist, what effect each connection should have on other neurons and how long that effect should be delayed for. We then need to communicate these synaptic updates and apply their effects in the correct sampling interval so that target neurons are appropriately affected. This process will be referred to as communicating and applying synaptic updates.

Communicating and applying synaptic updates efficiently is critical if a massively parallel neural computation system is to operate in real-time, particularly with biologically plausible fan-out. The choice of a FPGA-based implementation platform gives many benefits (particularly high-speed transceivers), but it poses challenges to communicating and applying synaptic updates, particularly in relation to memory usage.

With high fan-out and significant locality, a multicast routing algorithm would appear to be appropriate to communicate synaptic updates. However, this requires a multi-stage memory lookup to determine which neurons are targeted by each synaptic update at target FPGAs. Since there is not room for these lookup tables in on-chip memory, they will need to be stored in off-chip memory. Scanning lookup tables in off-chip memory makes very inefficient use of bandwidth and limits the scale of a real-time neural computation. Therefore a multicast routing algorithm is unsuitable for a FPGA-based massively parallel neural computation system.

Unicast routing is also unsuitable as the number of messages it uses between FPGAs is proportional to the product of spike frequency and fan-out, which would use too much communication bandwidth. To counter both of these limitations, this chapter develops and evaluates an alternative algorithm to communicate and apply synaptic updates which aims to maximise memory efficiency, and finds that it provides a compromise between the communication bandwidth needed by unicast and the memory bandwidth needed by multicast.

6.2 Design goals

We aim to communicate and apply synaptic updates in massively parallel neural computations with high fan-out in real-time, within the constraints imposed by a FPGA implementation. Given the communication and resource constraints identified in Chapter 4, this leads to three main design goals:

1. Fetch synaptic updates from off-chip memory as efficiently as possible
2. Make good use of on-chip memory so that its limited quantity does not overly limit the scale of computations
3. Avoid excessive inter-FPGA communication, which would hamper scalability

Each of these goals will now be studied further.

6.2.1 Fetching synaptic updates efficiently

To make most efficient use of off-chip memory the characteristics of the memory system provided by the implementation platform must be considered. DDR2 memory supports memory accesses on both the rising and falling edges of a clock, and so using DDR2-800 memory at a memory clock frequency of 400 MHz, memory accesses can be made at an effective frequency of 800 MHz. Each memory access is 64 bits wide. Using buffering and clock-crossing logic, this is equivalent to making 256 bit memory accesses at 200 MHz. The Altera DDR2 memory controller supports pipelining of memory requests and burst reads and writes. It is currently set up to support burst reads of up to 8×256 bit words. The latency between a burst read request and the first word being returned is around 5 clock cycles, with the remaining words returned on each of the following clock cycles.

This means that the most efficient way to access off-chip memory on this platform is to to fetch data in blocks rather than performing multiple, separate single word reads, and to process it 256 bits at a time, making full use of all data returned. Memory requests should be queued in the memory controller so that it is never idle, either by making requests for multiple sets of sequential bursts from one part of a system, or independent requests from different parts of the system. If chains of memory requests are inter-dependent (for example the result of a single read contains the address for a further read) then the memory controller will be left idle when it could have been returning data, which reduces off-chip memory efficiency.

6.2.2 On-chip memory usage

As discussed in Section 4.6.7, a significant proportion of the on-chip memory provided by a FPGA is used to store two copies of the I-value for each neuron. Since the remainder of the data that describes a neural network is stored in off-chip memory, this leaves the remainder of the on-chip memory to delay synaptic updates and to implement neural computation hardware. Structures that use large amounts of on-chip memory, such as buffers with size proportional to the number of neurons in a computation or their fan-out must be avoided.

6.2.3 Inter-FPGA communication

Inter-FPGA communication must neither be excessive nor must it introduce excessive latency, as either could prevent a massively parallel neural computation system operating in real-time. Some form of multicast routing would appear to be appropriate for communicating synaptic updates, but this must not be at the expense of inefficient off-chip memory usage or excessive use of on-chip memory.

6.3 Design decisions

The task of a synaptic update communication and application algorithm at first appears straightforward – take each synaptic update that occurs when a neuron spikes and apply it to the target neuron. However, this becomes a rather complex process when the scale of the task is considered – around a thousand synaptic updates need to be communicated and applied in response to a single spike, with many thousands of neurons spiking every sampling interval resulting in millions of synaptic updates needing to be communicated and applied every millisecond. This means that this process dominates that of modelling the behaviour of neurons, as predicted by the Communication-Centric Hypothesis.

The task of communicating and applying synaptic updates can be broken down into several inter-dependant subtasks, each presenting a series of design decisions. Making appropriate design choices will produce an algorithm which makes efficient use of resources and is well suited to implementing a massively parallel neural computation system that operates in real-time.

6.3.1 Starting the update process

Each neuron has an associated set of synaptic updates that need to be communicated and applied to other neurons whenever it spikes. These will be stored in off-chip memory. When the neuron spikes the updates will need to be located. This could be done using:

Synaptic updates alongside neuron parameters
: The synaptic updates could be stored at subsequent memory addresses to the parameters of the neuron modelling equation for each neuron. This would avoid any indirection, but would also make memory accesses less efficient as fetching neuron modelling equation parameters would require either sparse storage or a table or chain of pointers to locate each set of neuron parameters.

A pointer found using a lookup table
: A lookup table could be used to find a pointer to the set of synaptic updates, keyed on the identity of the neuron that has spiked. This method is used by Jin *et al.* (2008) in the Spinnaker system, which uses a lookup table (implemented using a binary tree in on-chip memory) to find a pointer to a block of synaptic updates in off-chip memory.

A pointer alongside neuron parameters
: A pointer to the set of synaptic updates could be stored alongside the parameters of the neuron modelling equation. These parameters will have come from a different region of off-chip memory, and so the contents of one region of off-chip memory will be dependent on the contents of another.

Storing synaptic updates alongside neuron parameters is clearly unsuitable as it makes accessing these parameters complex and inefficient as burst reads cannot be used. Using a lookup table will be inefficient unless the table can be stored in on-chip memory, which is inconsistent with the aim of making good use of on-chip memory.

While storing a pointer to the set of synaptic updates alongside the neuron modelling equation parameters may make it slightly harder to update the structure of a neural network while computation is in progress (for example to implement learning), it provides a significant advantage that (unless there are no spare bits in the data used to represent the neuron modelling equation parameters) the pointer can be found without any extra memory accesses.

Therefore a pointer to the set of synaptic updates that need to be communicated and applied when a neuron spikes will be stored alongside the parameters of the neuron modelling equation.

6.3.2 Fetching synaptic updates

Synaptic updates will need to be fetched from off-chip memory so that they can be communicated and applied to their target neurons. These updates conceptually consist of tuples of (target FPGA, target neuron, delay, weight). Either all or part of these updates could be fetched at:

Source FPGA
Fetch the tuples at the source FPGA and send them to their target FPGAs using the inter-FPGA communication system.

Target FPGA
Fetch the tuples at target FPGAs in response to messages received over the inter-FPGA communication system.

Other FPGAs
Fetch the tuples at some point between the source and target FPGAs and send them to their target FPGAs.

Fetching all of the tuples at the source FPGA would mean sending them all between FPGAs, which would make excessive use of inter-FPGA communication and would not be consistent with the design goals. Therefore update tuples will be fetched in a combination of locations, some at the source FPGA, some at the target FPGA and some at intermediate FPGAs.

6.3.3 Delaying synaptic updates

Delaying synaptic updates requires some form of buffering at a point or several points in the neural computation system. This could be at either or several of:

Source FPGA
Delay synaptic updates at the source FPGA before communicating them to target FPGAs. This requires at least one message to be sent from the source FPGA to each target FPGA for each delay size.

Target FPGA
Delay synaptic updates at their target FPGAs. This has the advantage of hiding communication latency. Any synaptic update can be delayed in the inter-FPGA communication system for up to the smallest synaptic delay size without having any effect on the behaviour of the neural communication system, since the update will still arrive at the target FPGA in time to be processed without increasing the total delay from the point of view of the target neuron.

Other FPGAs

Synaptic updates could be delayed at some point between the source and target FPGAs. This would be difficult to implement, and since some of the synaptic updates might have the same source and target FPGA, it would reduce to being equivalent to one of the other two methods.

Delaying synaptic updates at the target FPGA is more suitable for several reasons. Firstly it helps to hide communication latency as synaptic updates are communicated to their target FPGA before they are delayed. As long as a synaptic update arrives at its target FPGA within the minimum delay size of 1 ms, it will appear to have arrived at the target FPGA with no extra latency. Conversely if synaptic updates are delayed at the source FPGA then any delay caused by communication latency in a real-time neural computation will be in addition to the delay specified by the synaptic update, potentially invalidating the neural computation.

Secondly the amount of inter-FPGA communication needed to communicate synaptic updates between FPGAs is reduced by delaying synaptic updates at their target FPGAs, as a single message can be sent from the source FPGA to target FPGAs in response to a spike, rather than having to send at least one message per delay size.

Given that it reduces complexity, helps to hide communication latency, and potentially reduces inter-FPGA communication, synaptic updates will be delayed at their target FPGAs.

6.3.4 Applying synaptic updates

Synaptic updates need to be applied so that they have an affect on their target neurons in the correct sampling interval. This could by done by:

Applying at the target FPGA

The target FPGA updates the I-value of the target neuron.

Applying from some other FPGA

The FPGA applying the update will need to write to the I-value memory on the target FPGA.

Applying synaptic updates at any point in the system other than the target FPGA requires shared memory accesses and infrastructure to allow these shared accesses. This adds significant complexity compared to applying synaptic updates at the target FPGA, which only requires FPGAs to be able to access their local memory and send messages to other FPGAs.

Therefore synaptic updates will be applied at their target FPGAs. This doesn't necessarily preclude the synaptic updates themselves being fetched at some other point in the system.

6.4 Passing pointers

Based on the preceding discussion, it is clear that:

- The update process should be started using a pointer stored alongside the parameters of the neuron modelling equation
- It is inefficient to fetch all of the synaptic updates at the source FPGA and communicate them to target FPGAs using the inter-FPGA communication system
- The synaptic updates must be stored in off-chip memory, and should be accessed using burst reads, making full use of all data returned
- Synaptic updates should be delayed at target FPGAs to hide communication latency
- Synaptic updates should be applied at target FPGAs so that shared memory is not required

The largest open question is how to arrange the synaptic updates in off-chip memory to make optimal use of bandwidth. One method to help achieve this is to make full use of pipelining in the memory controller by having multiple hardware blocks in a neural computation system, that can each make independent memory requests. This can be achieved by breaking synaptic update communication and application into separate stages, each of which accesses data in off-chip memory relevant to its function:

1. Determining the set of target FPGAs (fan-out)
2. Communicating with the target FPGAs (communication)
3. Delaying synaptic updates (delay)
4. Applying synaptic updates (accumulation)

Each of these stages will have an allocated region of off-chip memory which will contain a portion of each set of synaptic updates. This means that when each stage communicates with the next (either using the inter-FPGA communication system or internal wiring), the next stage will have to fetch data from its region of off-chip memory. Since lookup tables should be avoided (as they make inefficient use of

off-chip memory bandwidth), this means that each message between stages should contain a pointer to the data that will be used by the next stage. Therefore each stage's data (other than the last) will need to contain a pointer that will be used by the next stage alongside whatever other data that stage needs to perform its function.

6.4.1 Delaying pointers to synaptic updates

Implementing a delay requires buffering, with the total amount of buffering required being proportional to the combination of the size of each item of data being delayed, the number of items being delayed and the length of the delay. Synaptic updates need to be delayed from 1 to 16 ms, and so 16 parallel delay buffers will be needed, one for each size of delay. The size of these delay buffers should be minimised to minimise on- and off-chip memory usage, and therefore so should the size and volume of data that passes through them.

One way to reduce the size and volume of data being delayed is to delay a pointer to a block of (target neuron, weight) pairs (the remains of synaptic updates after delay information is removed) rather than the pairs themselves. This means that the delay stage will need to fetch tuples of (delay, pointer to updates) from off-chip memory. However, an extra off-chip memory access from the delay stage can be avoided by transmitting these tuples from the fan-out stage. The fan-out stage will hence fetch tuples of (target FPGA, delay, pointer to updates).

This uses extra inter-FPGA communication (up to 16$\times$ more if every possible delay size is used), but it will improve memory access efficiency by avoiding any off-chip memory accesses from the delay stage (other than if it needs to spill buffered data to off-chip memory) and increasing the size of off-chip memory accesses from the fan-out stage.

A pointer to the block of tuples that will be fetched by the fan-out stage in response to a neuron spiking will be kept alongside the parameters of the neuron modelling equation, and will be provided by the hardware that evaluates the equation whenever a neuron spikes. This completes the arrangement of synaptic updates in off-chip memory, which is shown in Figure 6.1 on the following page.

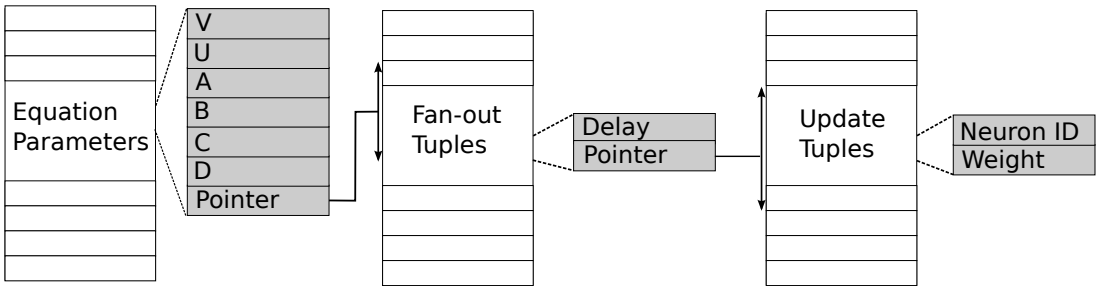


Figure 6.1: Arrangement of data used by synaptic update algorithm in off-chip memory

6.5 Synaptic update algorithm

The proposed synaptic update communication and application algorithm applies a complete set of synaptic updates by following a tree of pointers through regions of off-chip memory on the source FPGA, some intermediate FPGAs and target FPGAs. It proceeds as follows:

1. A pointer is found alongside the parameters of the neuron modelling equation for the neuron that has spiked
2. The pointer is used to burst read a block of tuples from the fan-out region of off-chip memory. These tuples consist of either:
   (a) (target FPGA, delay, pointer to update tuples) *or*
   (b) (target FPGA, pointer to fan out tuples)
3. All tuples are communicated to their target FPGA using the inter-FPGA communication system
4. Tuples of type 2a delayed at their target FPGAs
5. Tuples of type 2b repeat step 2
6. After being delayed each pointer is used to burst read a set of update tuples. These tuples consist of (target neuron, weight). Each weight is added to the target neuron's I-value to perform the synaptic update.

Repetition of step 2 is optional, and is suited to cases where either the number of delay sizes used by the fan-out to a given FPGA is high or 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 FPGAs in the group, with further fan-out then sending messages to the other FPGAs in the group.

6.5.1 Benefits of the algorithm

The proposed synaptic update communication and application algorithm minimises use of on-chip memory, with the neural network description being stored in off-chip memory. This facilitates implementation of the algorithm using FPGAs. Off-chip memory accesses are optimised (and hence the number of neurons that can be handled by each FPGA is maximised) in a number of ways:

- The pointer in step 1 is stored alongside the parameters of the neuron modelling equation, and so it does not need to be fetched from memory
- Separate tuples from the fan-out stage for each pair of target FPGA and delay avoid additional memory accesses at target FPGAs before applying delays
- The assumption of locality makes it likely that the majority of synaptic updates will target neurons which are either on the same FPGA as the source neuron or on nearby FPGAs. Combined with the assumption of high fan-out this means that a large number of fan-out tuples will be fetched by each pointer at the fan-out stage, which makes efficient use of burst memory accesses
- The pointer needed by the target FPGA in step 6 is supplied by the source FPGA, unlike multicast algorithms such as that proposed by Jin *et al.* (2008) that require multiple memory accesses to a lookup table to find the pointer based on the identity of the neuron that has spiked
- 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, which makes more efficient use of off-chip memory bandwidth than storing a single tuple 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 off-chip memory are populated before a neural computation can run, this is a similar situation to populating the routing tables used by multicast routing algorithms. Since all data required by a neural computation is stored in off-chip memory, it is simpler to load a neural network on to the neural computation platform since it is not necessary to transfer routing data into on-chip memories before the neural computation is run. Nor is it necessary to resynthesise the FPGA bitstream to change the neural network, as is the case with many previous FPGA implementations, such as that by Bailey (2010).

6.6 Evaluation of the algorithm

The performance of the proposed synaptic update communication and application algorithm can be evaluated by comparing it to algorithms that communicate synaptic updates using unicast and multicast routing. We will use mathematical models of the number of clock cycles needed to communicate and apply the synaptic updates produced by a single neuron spike with a fan-out of 10^3. There are two mathematical models. The first assumes that full sets of update tuples are delayed at their target FPGAs ("tuple delay") while the second delays pointers to subsets of these update tuples (grouped by delay) rather than the tuples themselves ("pointer delay"). This cannot be done with unicast routing, which has consistent behaviour using both models.

The models count the clock cycles needed for inter-FPGA communication, to perform off-chip memory accesses and to insert and remove update tuples or pointers from delay buffers. A number of distributions of target neurons relative to the source neuron are used to provide several points of comparison with varying degrees of locality.

6.6.1 Mathematical models

The mathematical models evaluate synaptic update communication and application algorithms based on the number of clock cycles used to communicate, delay and apply synaptic updates. Each clock cycle represents resource usage somewhere in a neural computation system, with lower resource usage translating to either ability to increase the scale of a computation, or a greater probability that a neural computation will operate in real-time.

We now discuss the distributions of target neurons relative to the source neuron that are used with the mathematical models and the method used to sample the clock cycles taken by off-chip memory accesses, inter-FPGA communication and delay buffer accesses.

Distribution number	0	1	2	3	4	5	⟵ Distance
	% of target neurons						Neuron-distance
1	100	0	0	0	0	0	0
2	80	20	0	0	0	0	200
3	70	16	8	6	0	0	500
4	50	20	15	10	5	0	1000
5	30	30	15	15	5	5	1500

Table 6.1: Target neuron distributions used to evaluate synaptic update communication and application algorithms

Distance	% of target neurons	FPGAs	Neurons / FPGA
0	70	1	700
1	16	4	40
2	8	8	10
3	6	12	5

Table 6.2: Number of target FPGAs and number of target neurons on each of these FPGAs for distribution 3 from Table 6.1

Target neuron distributions

The distributions of target neurons used with the mathematical models assume that a network of FPGAs is arranged in a 2-D mesh, with each FPGA having 4 neighbours. A varying percentage of target neurons are distributed between FPGAs a given distance from the FPGA holding the source neuron, using the distributions in Table 6.1 on the preceding page. Table 6.2 shows the number of target FPGAs and the number of target neurons on each of these FPGAs for distribution 3 from Table 6.1 as an example.

To quantify the degree of locality exhibited by each of these distributions, a measure that will be called neuron-distance is used. This is the sum of distance from the source neuron for all target neurons in the distribution. For example the neuron-distance for distribution 3 in Table 6.1 is $0 \times 700 + 1 \times 160 + 2 \times 80 + 3 \times 60 = 500$.

Off-chip memory accesses

The off-chip memory is modelled as having a word size of 256 bits, a burst size of 8 and a latency of 5 clock cycles between a burst (or single access) being requested and the first word being returned, with the remaining words returned on each of the following clock cycles, matching the assumed characteristics of the DDR2 SDRAM on the DE4 board in Section 5.3.1. A new read request can be made after the last word in a burst has been returned.

For the proposed algorithm it is assumed that 4 fan-out tuples or 8 update tuples are stored in a word (matching what is achieved in Chapter 7), and that these tuples are accessed using burst reads. A unicast algorithm is assumed to perform burst reads at the source FPGA to retrieve tuples of (target FPGA, target neuron, delay, weight), again with 4 tuples in a word and with no further memory accesses required at each target FPGA.

A multicast algorithm is assumed to perform memory accesses in the FPGAs at the "leaves" of the fan-out tree. In each of these FPGAs, a lookup table needs to be

scanned to find a pointer to either a set of tuples of (target neuron, delay, weight) in the tuple delay model, or a list of pointers to sets of (delay, weight) tuples, grouped by delay in the pointer delay model. The lookup table is assumed to contain 10^3 entries (equal to the fan-in), and so scanning it takes $\log_2 10^3 \approx 10$ clock cycles. Fetching tuples from off-chip memory takes the same number of clock cycles as the proposed algorithm.

Inter-FPGA communication

The number of times that an inter-FPGA message traverses a link between two FPGAs is counted. It is assumed that each message takes 5 clock cycles to traverse a link. A unicast algorithm sends one message per target neuron, with many of these messages traversing more than one link. A multicast algorithm sends messages in a minimum spanning tree between the source and target FPGAs. The proposed algorithm sends one message per target FPGA, with some traversing more than one link.

Delay buffer accesses

With the tuple delay model and with a unicast algorithm (regardless of delay model), 10^3 tuples need to be inserted into delay buffers and 10^3 tuples need to be removed, taking 2×10^3 clock cycles in total. With the pointer delay model the number of pointers that need to be inserted and removed from delay buffers is counted, which is dependent on the distribution of neurons between FPGAs, the number of delay sizes used and the threshold used to determine when a second fan-out stage should be used at target FPGAs. Inserting or removing a pointer from a delay buffer is assumed to take one clock cycle.

Delay sizes used

The performance of the pointer delay model is dependent on the number of delay sizes (from 1 to 16 ms) that are used by each set of synaptic updates. Using more delay sizes partitions the synaptic updates into more sets of tuples at target FPGAs, and results in more memory accesses for smaller sets of tuples, which results in decreased performance as the number of tuples fetched by each memory request drops below the burst size of off-chip memory. It is assumed that 8 delay sizes are used by each set of synaptic updates to begin with. The effect that the number of delay sizes used has on performance will be examined later.

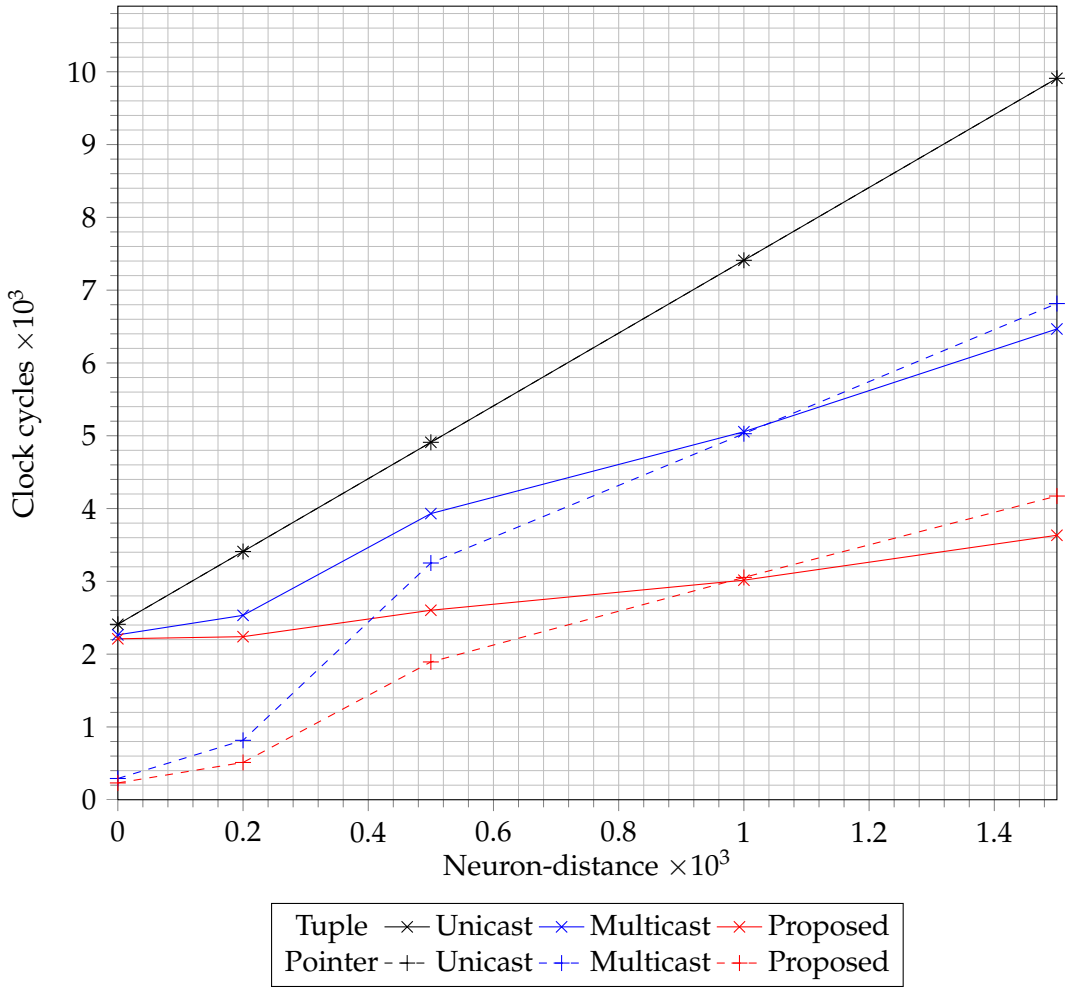


Figure 6.2: Clock cycles needed to apply synaptic updates for each algorithm using both tuple and pointer delay models. Note that the two plots for unicast overlap

6.6.2 Results

Figure 6.2 shows the total number of clock cycles needed to communicate and apply a set of 10^3 synaptic updates using each algorithm and both the tuple and pointer delay models. Figure 6.3 on the following page and Figure 6.4 on page 99 show how the clock cycles are divided between memory accesses, messaging and delay buffer accesses for each of the delay models. At first glance it is clear that the proposed algorithm consistently uses fewer clock cycles than either unicast or multicast using the same delay model, though which of the two delay models is better depends on the total neuron-distance.

The number of clock cycles needed by the pointer delay model is significantly less than for the tuple delay model when locality is high (lower neuron-distance), but this advantage is gradually eroded as locality decreases (higher neuron-distance), until the tuple delay models takes less clock cycles than the pointer delay model.

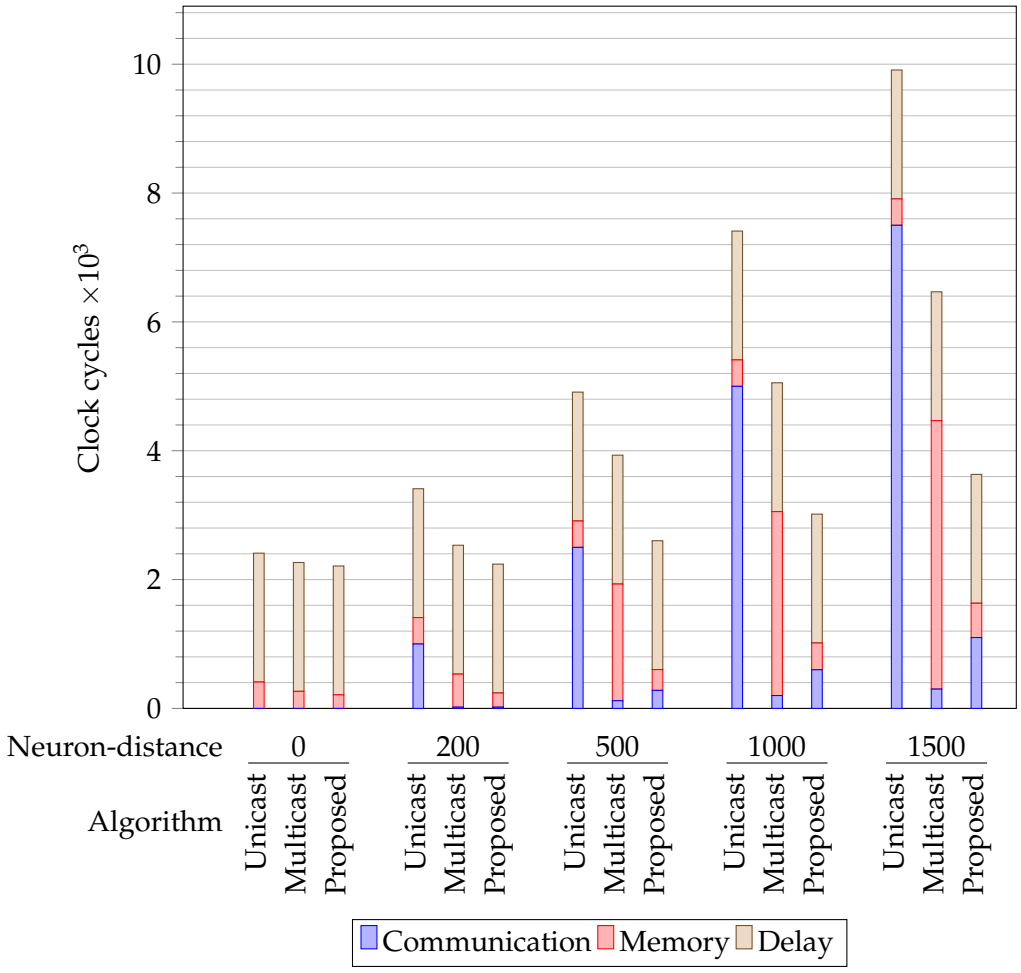


Figure 6.3: Breakdown of clock cycles needed to apply synaptic updates using the tuple delay model

This can be seen more clearly in Figure 6.5 on page 100, which compares the proposed algorithm and a multicast algorithm using both the tuple and pointer delay models, based on the clock cycles that they require as a percentage of those required by a unicast algorithm. A unicast algorithm takes the same number of clock cycles regardless of which delay model is used. Figure 6.5 clearly shows that the two delay models are almost equal at a neuron-distance of 1.0×10^3 and that the tuple delay model is more efficient when locality decreases further at a neuron-distance of 1.5×10^3.

This can be explained by considering how the two delay models partition the synaptic updates for each spike. While the tuple delay model divides the synaptic updates into sets of tuples for each target FPGA, the pointer delay model further subdivides these sets by delay size. If the number of target neurons on a given FPGA is already small (e.g one of the peripheral FPGAs in a case with low locality), then this results in there being pointers to very small sets of update tuples

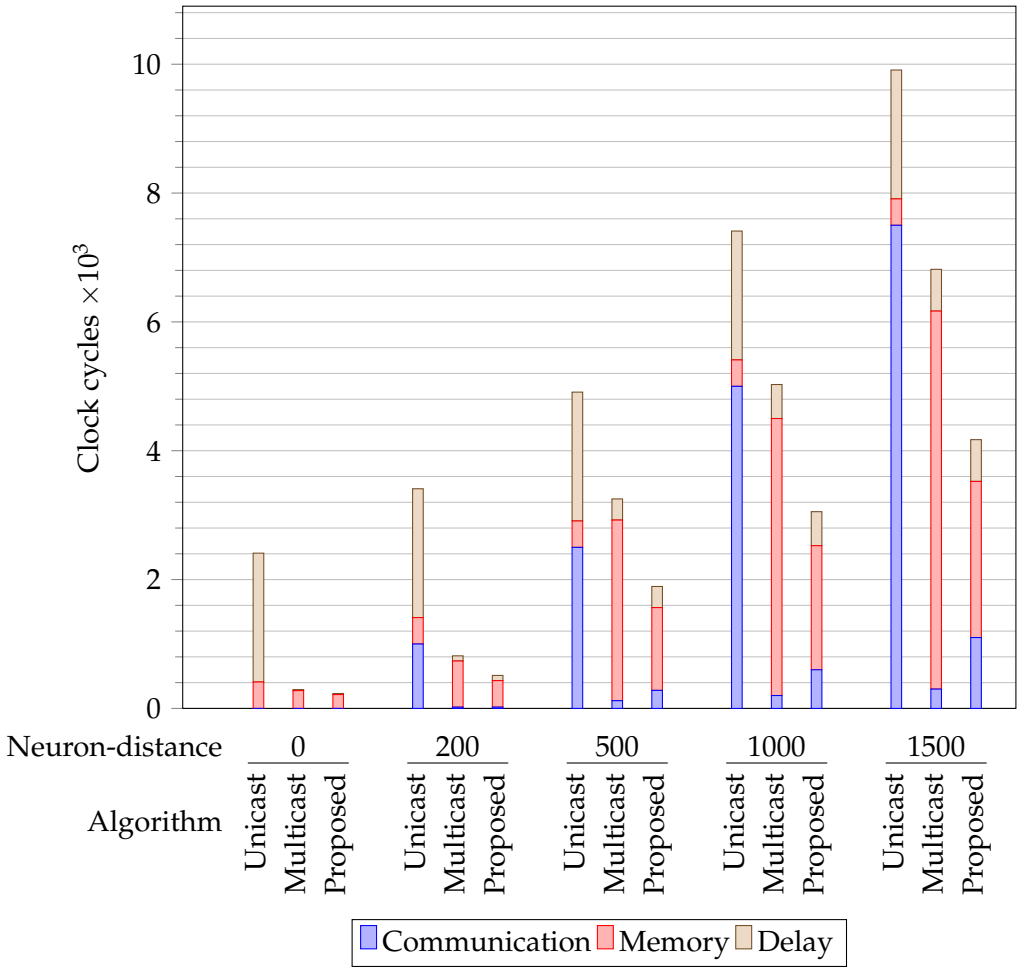


Figure 6.4: Breakdown of clock cycles needed to apply synaptic updates using the pointer delay model

e.g only one or two, which makes inefficient use of off-chip memory bandwidth and can result in the pointer delay model being less efficient than the tuple delay model.

This can be seen clearly in Figure 6.6 on page 101, which plots the number of clock cycles taken by the proposed algorithm as a percentage of those taken by a unicast algorithm against the number of delay sizes used in a set of synaptic updates for each of the five neuron distributions in Table 6.1. It can be seen that the proposed algorithm becomes significantly less efficient for the neuron distributions with less locality (higher neuron-distance) as more delay sizes are used, to the point that the proposed algorithm becomes less efficient than unicast for distributions with less locality when synaptic updates are distributed over more than 8 delay sizes.

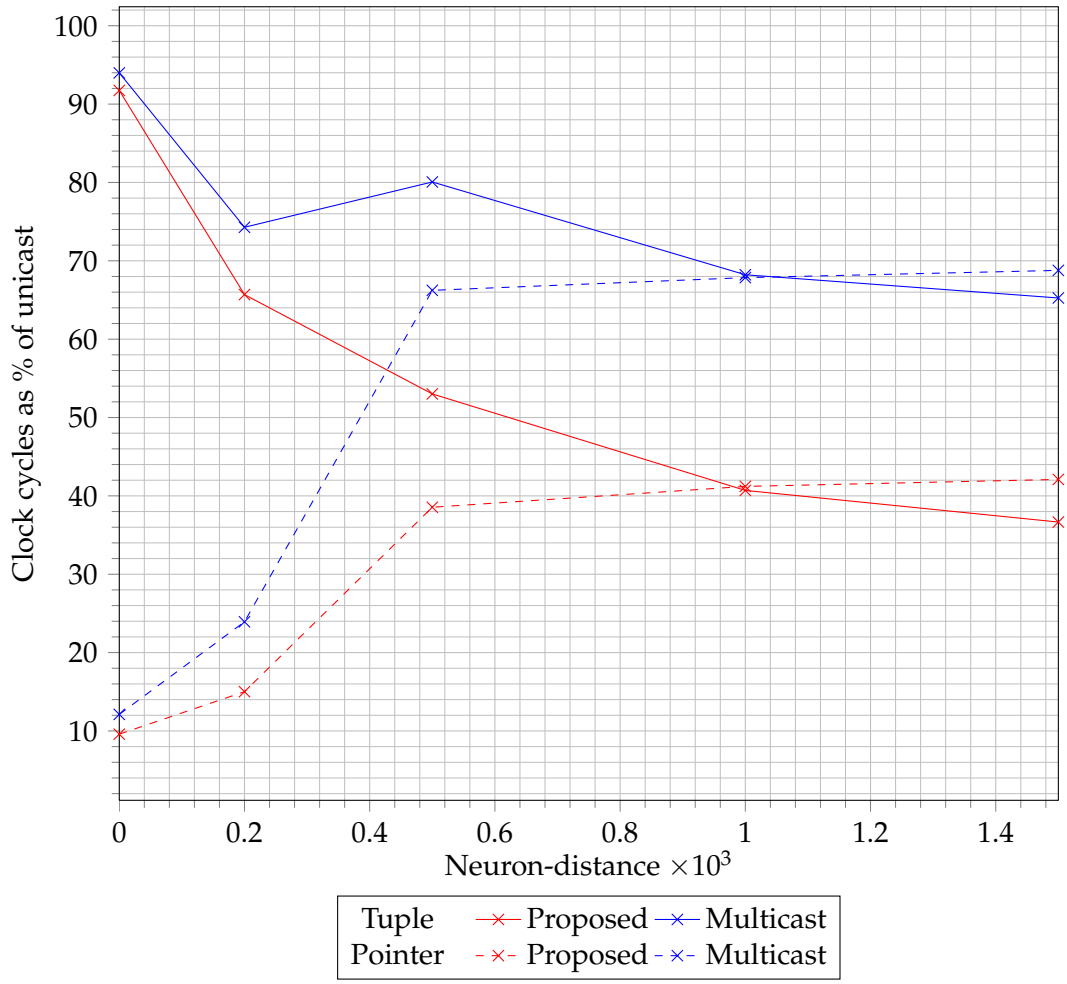


Figure 6.5: % of clock cycles compared to unicast used by the proposed algorithm and multicast for varying neuron-distance using both tuple and pointer delay models

6.6.3 Memory size

The memory requirements of communicating and applying synaptic updates were discussed in Section 4.4. The analysis assumed that the tuple delay model was used, and that synaptic updates were stored in memory in full, without any compression or indirection, resulting in a synaptic update data size of 0.6 GB and bandwidth requirement of 6 GB/s for a neural computation with 10^5 neurons per device, mean fan-out of 10^3 and mean spike frequency of 10 Hz.

Using the pointer delay model the size of the synaptic update data and hence the bandwidth needed to read it will depend on the locality of a neural network and the number of delay sizes used by synaptic updates.

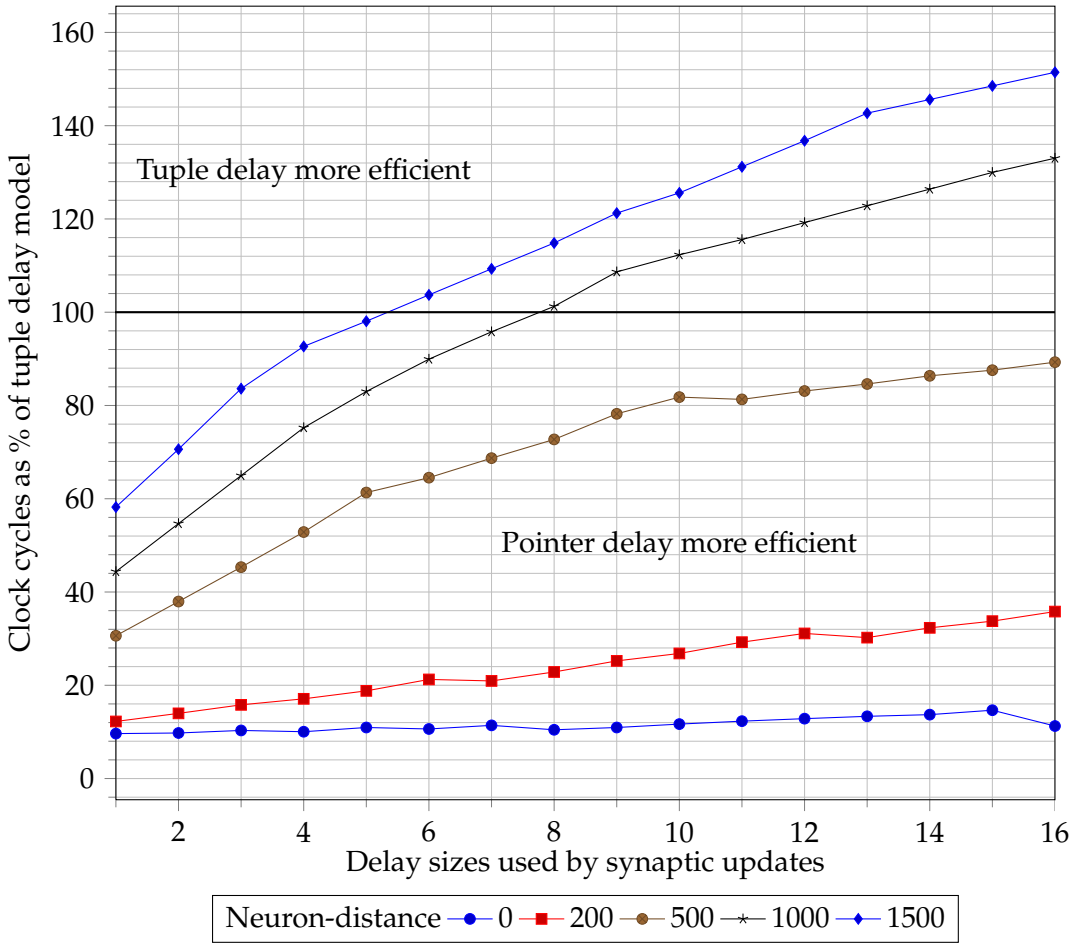


Figure 6.6: Clock cycles used by pointer delay model as % of those used by tuple delay model for different numbers of delay sizes used by synaptic updates

The memory size needed by the proposed algorithm using the pointer delay model is shown as a percentage of the size needed for a unicast algorithm in Figure 6.7 on the following page. This shows that the memory size needed is lower with the pointer delay model when locality is high, but it increases sharply as locality decreases, and further as the number of delay sizes increases. This means that a massively parallel neural computation system using the proposed synaptic update communication and application algorithm will use more memory and more memory bandwidth if a neural network exhibits poor locality. This is because the amount of memory used by the fan-out stage of the algorithm approaches the amount of memory used by the update stage when the number of target neurons on each target FPGA becomes small.

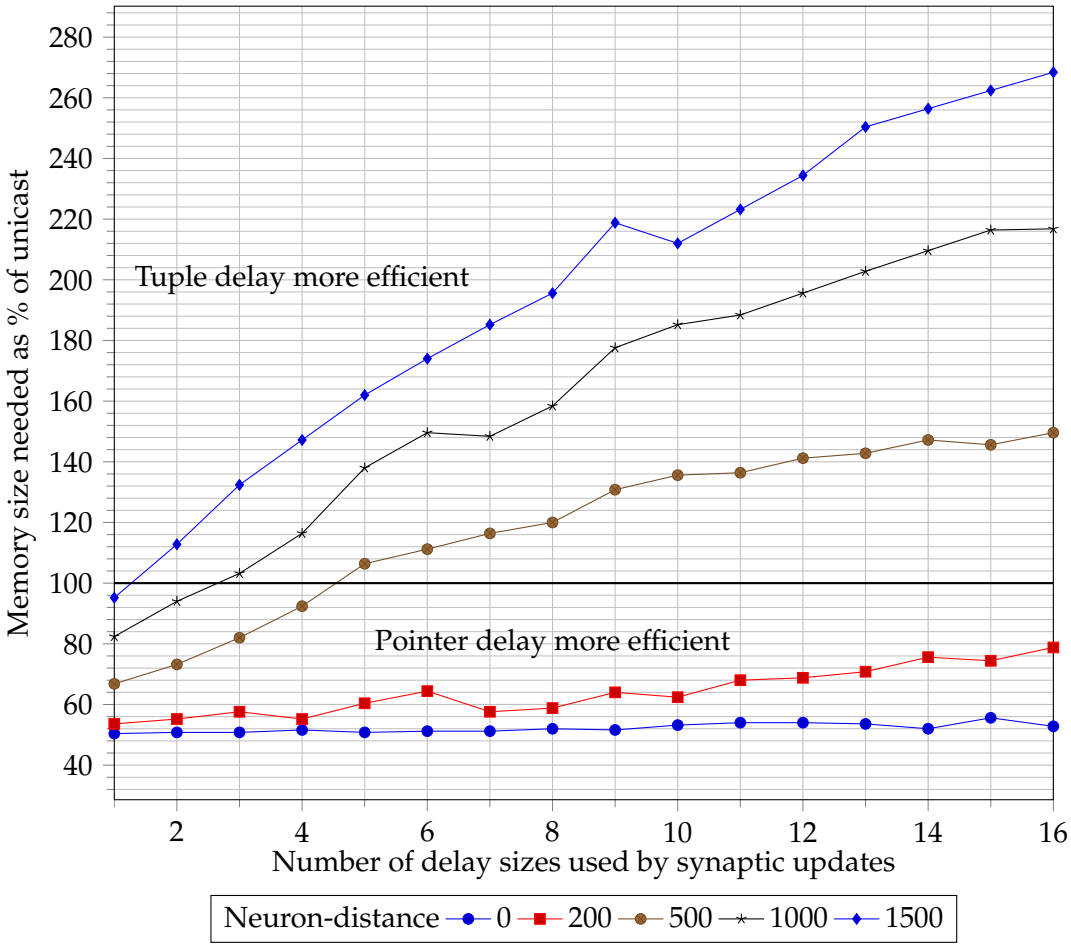


Figure 6.7: Change in memory size needed to store synaptic updates for differing numbers of delay sizes used by these updates when using the pointer delay model rather than the tuple delay model

Figure 6.8 on the next page shows the size of memory needed to delay synaptic updates using the pointer delay model as a percentage of the size needed by the tuple delay model for varying numbers of delay sizes used by these synaptic updates. This shows a similar trend to Figure 6.7, but in this case the memory size needed by the pointer delay model decreases by at least 50% compared to the tuple delay model, even with all delay sizes in use and little locality. If locality is high then the memory size needed by the pointer delay model decreases dramatically to less than 10% of that needed with the tuple delay model.

The memory needed to delay synaptic updates was originally calculated to be 68 MB in Section 4.4.3, needing a bandwidth of between 8 GB/s and 68 GB/s. If these requirements are reduced to 10% of the original then 6.8 MB of memory would be needed to delay synaptic updates, with a bandwidth of 0.8 GB/s to 6.8 GB/s.

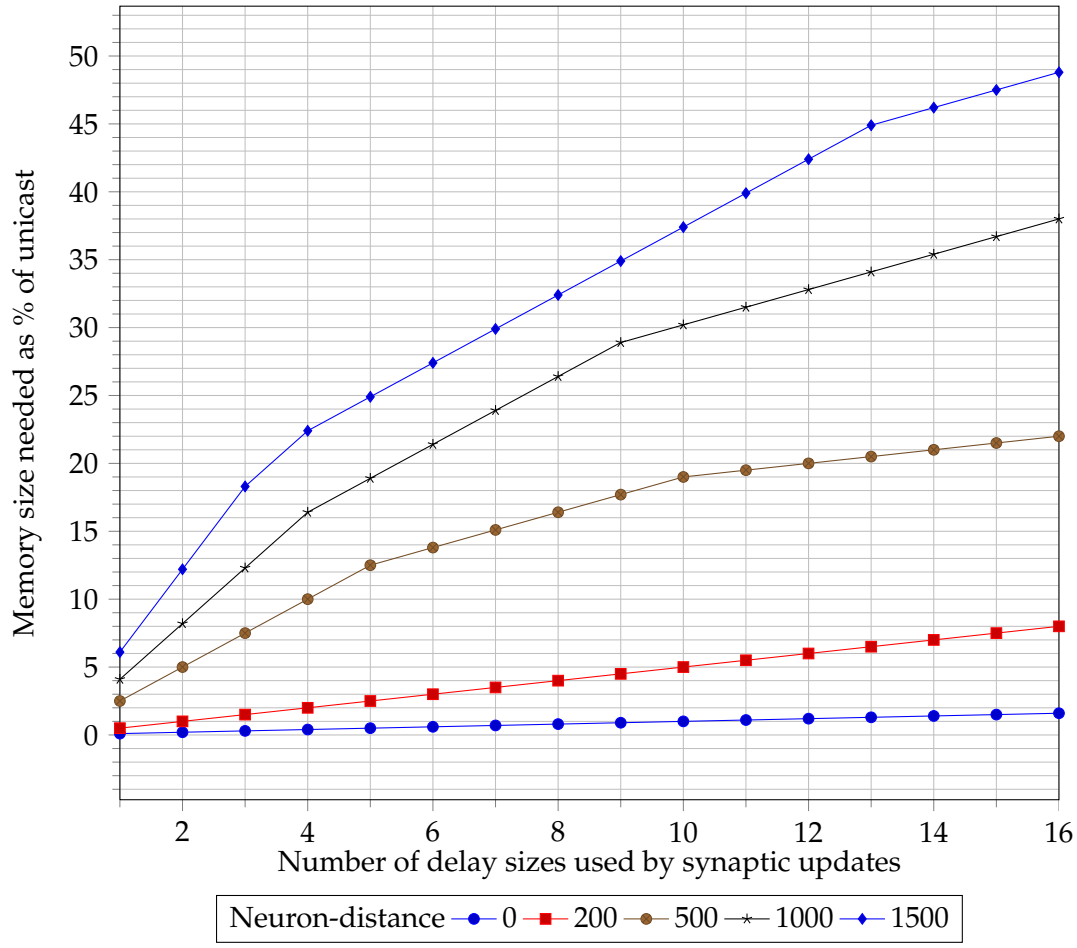


Figure 6.8: Change in memory size needed to delay synaptic updates for differing numbers of delay sizes used by these synaptic updates

While the increase in memory needed to store synaptic updates as locality decreases is less than ideal, the analysis of the locality of biological neural networks in Section 3.3.2 indicates that locality is expected to be high, and so there will be a large number of target neurons on each target FPGA. When combined with the dramatic reduction in memory needed to delay synaptic updates in all cases, this means that the proposed synaptic update communication and application algorithm appears to meet the goals of efficient off-chip memory usage, minimal on-chip memory usage and avoiding excessive use of inter-FPGA communication set in Section 6.2.

6.7 Conclusion

The proposed synaptic update communication and application algorithm reduces the memory size, memory bandwidth and inter-FPGA communication needed compared to unicast and multicast algorithms when locality in a neural network is high.

There is a more mixed outlook as locality decreases, particularly an increase in the amount of memory needed to store synaptic updates, but given the analysis in Section 3.3.2 that shows that biological neural networks exhibit high locality, any cases of low locality can be considered to be "out of spec." At worst a massively parallel neural computation system based on this algorithm would exhibit reduced performance and take longer than real-time if it were used to perform a neural computation using a neural network with low locality.

It was previously determined that all I-values will have to be stored in on-chip memory. It is clear that there is insufficient on-chip memory to store the remainder of the neural network description, and so it will need to be stored in off-chip memory, and hence off-chip memory bandwidth must be used as efficiently as possible to maximise the size of neural network that can be handled in real-time, as predicted by the Bandwidth Hypothesis.

Since the synaptic update communication and application algorithm is designed to group data into as large blocks as is possible, this means that high bandwidth usage efficiency can be achieved by using burst reads, unlike a multicast algorithm which has to use a large number of single-word reads, which makes much less efficient use of bandwidth.

The reduction in the amount of memory needed to delay synaptic updates compared to other algorithms is such that in some cases there will be sufficient on-chip memory (beyond that needed for the I-values) for the entire delay buffer to fit in on-chip memory, providing random access (needed to deal with incoming pointers as soon as they arrive and avoid deadlocking the inter-FPGA communication system or the fan-out stage of the algorithm) without inefficient use of memory bandwidth.

However this will not always be the case, and so a hybrid solution to delaying synaptic updates will be appropriate, involving a combination of on-chip memory and off-chip memory, with the "middle" values in each delay buffer (which are neither recently arrived nor about to be drained and processed) being "spilled" to off-chip memory. This would allow random access to a delay buffer for incoming and outgoing pointers as well as sufficient size to delay larger numbers of synaptic updates if needed.

Chapter 7

Implementation

7.1 Introduction

Given the analysis so far, it is clear that a multi-FPGA massively parallel neural computation system must be implemented with careful use of on-chip memory and making as efficient use as possible of off-chip memory bandwidth, as suggested by the Bandwidth Hypothesis.

This will be achieved by splitting the neural computation algorithm down into several stages, each implemented as a separate hardware block. Each of these blocks will make independent sets of off-chip memory accesses using burst reads. The aim is for there to always be a hardware block available to request a new off-chip memory access while other blocks are processing the data returned to them.

For each hardware block we will analyse how its design makes efficient use of off-chip memory bandwidth, as the Bandwidth Hypothesis suggests that this is critical when designing a massively parallel neural computation system that operates in real-time.

7.2 Splitting the algorithm

The implementation of a massively parallel neural computation system on the Bluehive platform has the following hardware blocks in each FPGA:

Equation processor
: Evaluates the neuron modelling equation for every neuron every 1 ms sampling interval. Passes a pointer to blocks of fan-out tuples to the fan-out engine when a neuron spikes.

Fan-out engine
: Performs the fan-out stage of the algorithm.

Router
: Provides inter-FPGA communication for both communicating synaptic updates and command and control of the neural computation system.

Delay unit
: Delays synaptic updates.

Accumulator
: Applies synaptic updates to target neurons.

Spike auditor
: Records spike events to output as the computation results.

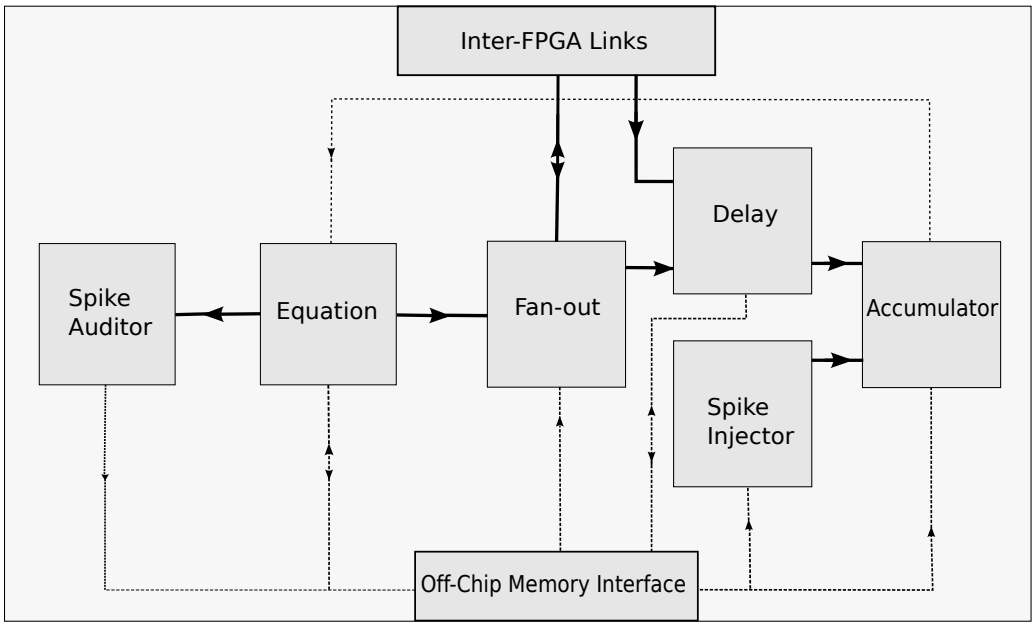


Figure 7.1: Architecture of an FPGA in the massively parallel neural computation system. Solid lines indicate a flow of messages while dashed lines indicate memory accesses

Spike injector
: Allows external spike events to be injected into a neural network. This is used to provide an initial stimulus. It could also be used to interface to external systems (e.g. sensors).

Control state machine
: Coordinates the setup and progress of a neural computation.

Control logic
: Sets up the neural computation system and advances the sampling interval in response to commands from the **control state machine**.

The architecture of each FPGA in the massively parallel neural computation system is shown in Figure 7.1. Note that the **control state machine** and **control logic** are not shown.

7.2.1 Equation processor

The **equation processor** needs to evaluate the neuron modelling equation (Equation 3.10) for every neuron every 1 ms sampling interval. It reads the parameters for each neuron from off-chip memory, evaluates the equation, writes back the results and sends a pointer to the **fan-out engine** if the neuron spiked.

Input

The main input is a stream of neuron modelling equation parameters. Each neuron has 128 bits of parameters, which are stored in off-chip memory. These are laid out as shown in Figure 7.2. There is also an I-value, which is fetched from the **accumulator** (described in Section 7.2.4), and a signal from the **control logic** that indicates that a new 1 ms sampling interval has started.

When a new 1 ms sampling interval starts this triggers a series of burst reads to read in the neuron modelling equation parameters for all of the neurons on the FPGA from off-chip memory. The data that is returned is a stream of 256 bit words, with each word containing the parameters for two neurons. There is enough buffer space to store the stream returned by two full burst reads of 8 words each. A new burst read is requested whenever there are at least 8 free spaces in the buffer, until all of the neuron modelling equation parameters have been fetched.

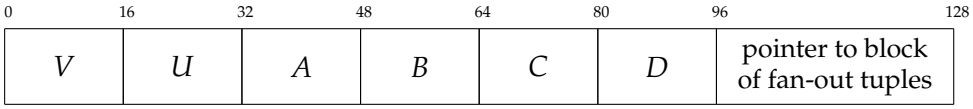


Figure 7.2: Layout of neuron modelling equation parameters in off-chip memory. The parameters are those of Equation 3.10

Function

The **equation processor** takes the stream of 256 bit words from off-chip memory and evaluates the next state for the two neurons that are represented by the two sets of equation parameters in each word. This is done using two parallel pipelines which have 6 stages each:

1. Request I-value from **accumulator**
2. $V_2 = ((2621 \times V) >> 16) + 1536$
   $U_2 = A \times V$
3. $V_3 = ((V \times V_2) >> 8) + 35840$
   $U_3 = (U_2 + B \times U) >> 16$
4. $V_4 = V_3 + (I << 8) - U$
   $U' = U_3 + U$
5. **if** $V_4 \geq (30 << 8)$ **then**
   $V' = C$
   $U' = U + D$
   spiked = true

else

$V' = V_4$

spiked = false

Send updated parameters to output stage

6. **if** spiked **then**

   Send pointer to **fan-out engine**

   Send neuron identifier to **spike auditor**

The original equation parameters are passed between stages alongside the additional intermediate values (V_2 to V_4 and U_2 and U_3). All intermediate values use 32 bit fixed-point precision to avoid loss of accuracy.

Output

The output of the **equation processor** is the next state for two neurons, represented by an updated 256 bit word (laid out as in Figure 7.2). The updated words are buffered until there are 8 of them and then they are written back to off-chip memory using a burst write. Because the off-chip memory controller has separate wiring for data being read and data being written it is possible to interleave this burst write with burst reads of parameters for other neurons, which significantly increases throughput compared to serialising burst reads and burst writes.

Note that the parameters A to D and the pointer do not change, and are only written back to off-chip memory along with V and U as the memory controller expects to receive write commands for whole 256 bit words.

If either or both the neurons being evaluated spikes, then one or two pointers to blocks of fan-out tuples will be passed to the **fan-out engine**. These pointers are laid out as in Figure 7.3. If both neurons spike then their pointers are serialised before being passed to the **fan-out engine**. This causes a stall if the **fan-out engine**'s input buffer is full, but in the vast majority of cases the gain in efficiency of evaluating the state of two neurons in parallel outweighs the penalty of these stalls. Identifiers for neurons that spike are also sent to the **spike auditor**.

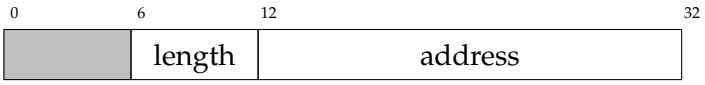


Figure 7.3: Layout of pointer to block of fan-out tuples

Summary

Input	off-chip memory (2×128 bit chunks every clock cycle)	(V, U, A, B, C, D) and pointer to block of fan-out tuples
	accumulator	I
	control logic	new sampling interval signal
Function	neuron modelling equation (Equation 3.10)	
Output	off-chip memory (2×128 bit chunks every clock cycle)	(V, U, A, B, C, D) and pointer to block fan-out tuples
	if $V \geq 30\,\text{mV}$ **then**	
	fan-out engine	pointer to block of fan-out tuples
	spike auditor	neuron identifier

Efficiency

The **equation processor** makes efficient use of off-chip memory bandwidth by:

- Using burst reads to fetch large blocks of input data
- Having requests for further blocks of input data ready so that they can be actioned whenever the memory controller is free
- Making full use of the 256 bit words returned by off-chip memory
- Evaluating the neuron modelling equation in a single clock cycle on average, which consumes input words as quickly as possible and avoids stalls
- Writing back updated values of V and U efficiently, interleaving writes with reads to avoid using extra bandwidth

7.2.2 Fan-out engine

Input

The **fan-out engine** receives pointers to blocks of fan-out tuples (Figure 7.3) from the **equation processor**.

Function

Each pointer is used to burst read a block of fan-out tuples from off-chip memory. The layout of these tuples is shown in Figure 7.4. These tuples are either targeted at the **delay unit** on this or another FPGA (delay $\neq 0$) or at the **fan-out engine** on another FPGA (delay $= 0$).

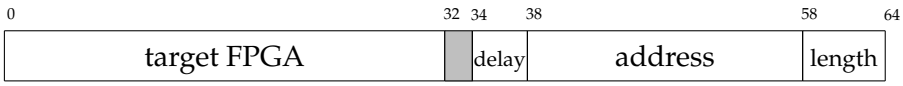


Figure 7.4: Layout of a fan-out tuple

Output

The fan-out tuples are sent to the **delay unit** if their target is the current FPGA, and are otherwise sent to the **router** to be communicated to their target FPGAs.

Summary

Input	**equation processor**	pointer to block of fan-out tuples
	off-chip memory	fan-out tuples
Function	fetch fan-out tuples	
Output	**delay unit**	fan-out tuples for current FPGA
	router	fan-out tuples for other FPGAs

Efficiency

The **fan-out engine** makes efficient use of off-chip memory bandwidth by:

- Using burst reads to fetch blocks of fan-out tuples
- Fitting multiple fan-out tuples into a 256 bit word
- Reducing the off-chip memory usage of the **delay unit**

7.2.3 Delay unit

Input

The **delay unit** receives fan-out tuples from the **fan-out engine**s on both the current FPGA and other FPGAs in the neural computation system (via the **router** and inter-FPGA communication system).

Function

The fan-out tuples are stripped of their target FPGA and delay fields to become pointers to blocks of update tuples. These have the same format as the pointers to blocks of fan-out tuples in Figure 7.3. These pointers are then inserted into a delay buffer.

The delay buffers are implemented by 16 FIFOs, one for each delay size (in 1 ms increments). FIFOs are assigned to delay sizes in a cyclical fashion and every 1 ms, the "2 ms" FIFO logically becomes the "1 ms" FIFO and the "1 ms" FIFO becomes the "0 ms" FIFO, and so on. The contents of the current "0 ms" FIFO are drained and become the output. This matches work by Jin *et al.* (2008).

As shown in Section 6.6.3, the maximum total size of the delay buffers depends on the locality of a neural neural network and the number of delay sizes in use, with an estimate of 6.8 MB, requiring a bandwidth of 0.8 GB/s to 6.8 GB/s. This is clearly too much data to fit into on-chip memory, so off-chip memory will have to be used. However, FIFO semantics should be retained, particularly as the **delay unit** must consume input immediately to avoid deadlocks elsewhere in the system.

Therefore the **delay unit** is implemented using 16 "spillable FIFOs." These use a small FIFO at the input and another at the output. When the input FIFO becomes full data is transferred to off-chip memory using a burst write. When the output FIFO becomes empty this data is fetched back using a burst read.

Output

The main output from the **delay unit** is a stream of pointers to blocks of update tuples. These are sent to the **accumulator**. These pointers share their format with that in Figure 7.3. There are also off-chip memory accesses from the spillable FIFOs.

Summary

Input	**fan-out engine** or **router**	fan-out tuples
	control logic	new sampling interval signal
	off-chip memory	burst read FIFO data previously spilled to external memory
Function	sort pointers into FIFO delay buffers	
Output	**accumulator**	pointer to block of update tuples
	off-chip memory	burst write FIFO data spilled to external memory

Efficiency

The **delay unit** makes efficient use of off-chip memory bandwidth by:

- Delaying pointers to blocks of synaptic updates rather than the updates themselves
- Allowing excess data to be spilled from premium on-chip memory to more abundant off-chip memory
- Using circular allocation for delay buffers to avoid needing to copy data between buffers every 1 ms

7.2.4 Accumulator

The **accumulator** sums synaptic updates for each neuron to produce I-values. Since this operation occurs at the leaves of a tree with high fan-out it is the inner loop of the neural computation algorithm, which makes it the most performance critical task in massively parallel neural computation.

Input

The **accumulator** receives a stream of pointers to blocks of update tuples from the **delay unit** and separately requests for I-values from the **equation processor**. It also receives a signal marking the end of a 1 ms sampling interval from the **control logic**.

Function

Burst reads are used to fetch streams of update tuples from off-chip memory. The layout of these tuples is shown in Figure 7.5. The tuples are packed in to 256 bit words with up to 8 tuples per word. To make good use of off-chip memory bandwidth all 8 tuples should be processed in parallel, in a single clock cycle on average. This is done by separating the core of the **accumulator** into 8 banks, each holding the I values for 1/8 of the neurons on the FPGA in on-chip memory.

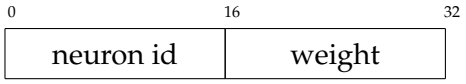


Figure 7.5: Layout of an update tuple

Neurons that frequently receive synaptic updates together (as a result of locality) are allocated to different banks so that they can be updated in parallel, which results in up to 8 updates being applied per clock cycle in the best case. Buffering is provided to prevent deadlock if multiple update tuples in an input word target neurons in the same bank. This statistical multiplexing means that there will only be stalls when there are an unusually high number of updates to neurons in the same bank.

The update tuples are fed into the core of the accumulator block as shown in Figure 7.6 on the next page. This figure shows 4 tuples per input word and 4 banks for clarity. This processes the update tuples as follows:

1. Each tuple is fed to a **bank selector**, which identifies which of the 8 banks holds the I-value for the neuron being updated by the tuple
2. The tuple is routed to a FIFO queue in its target bank. There is 1 FIFO per bank per position in the input word to allow update tuples to appear in any position in the input word. Without these FIFOs, 2 update tuples targeting the same bank would cause a stall
3. Update tuples are dequeued from the set of FIFOs in each bank in a round-robin fashion
4. The current I-value for the target neuron is fetched from on-chip memory
5. The weight in the update tuple is added to the current I-value
6. The new I-value is stored to on-chip memory

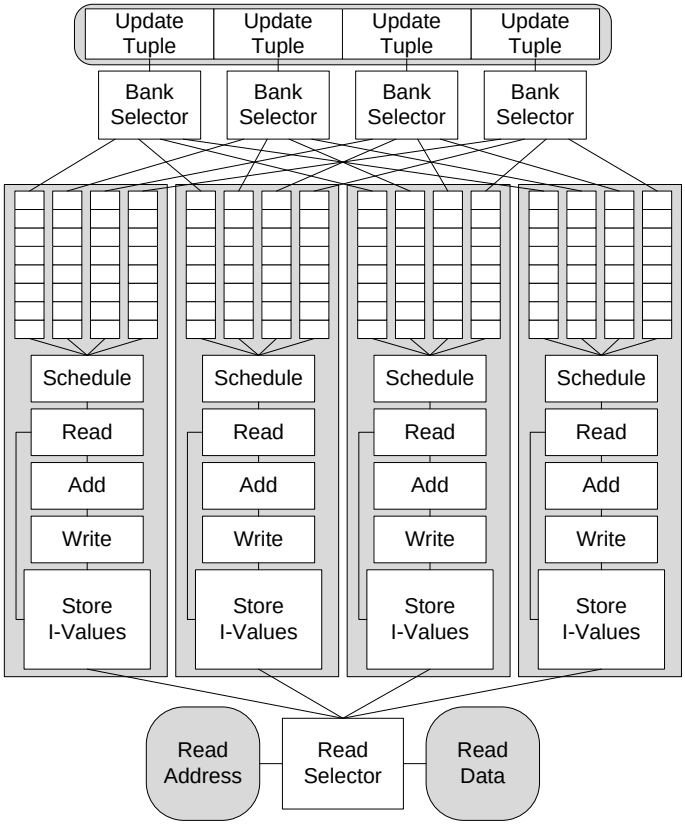


Figure 7.6: Layout of the core of the **accumulator**. 4 banks are shown for clarity, there are actually 8 banks

There are two complete copies of the I-values in the **accumulator**, one used to apply updates for the next sampling interval and the other used to supply the current I value for each neuron to the **equation processor**. These copies are swapped around every sampling interval.

Output

The output of the **accumulator** is a stream of I-values that are requested by the **equation processor**. Each of the I-values will be read once per 1 ms sampling interval.

Summary

Input	**delay unit**	pointer to block of update tuples
	off-chip memory	blocks of update tuples
	equation processor	requests for I-values
	control logic	new sampling interval signal
Function	apply the updates to the appropriate I-values in parallel	
Output	**equation processor**	I-values

Efficiency

The **accumulator** makes efficient use of off-chip memory bandwidth by:

- Consuming a whole 256 bit word from off-chip memory every clock cycle
- Providing buffering to prevent stalls in all but the most pathological distributions of update tuples between banks
- Keeping I-values in on-chip memory to allow for frequent updates with low latency

7.2.5 Memory spike source

The **memory spike source** allows a "script" of spikes that need to be injected into a neural network to be fetched from off-chip memory. This is particularly useful when providing some initial stimulus to a neural network at the start of a neural computation.

Input

A stream of tuples is fetched from off-chip memory. Each tuple contains the number of the sampling interval that the spike should be injected in, a neuron identifier and a weight, laid out as shown in Figure 7.7.

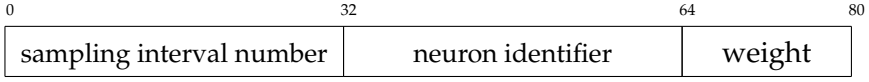


Figure 7.7: Layout of an injected spike tuple

Function

Injected spike tuples need to be fetched from off-chip memory in advance of the sampling interval when they need to be applied. This is achieved by buffering a whole burst of 24 tuples and then fetching a new burst when the buffer is almost empty.

Output

A stream of tuples is sent to the **spike injector** to be combined with injected spikes from other sources.

Summary

Input	**off-chip memory**	stream of injected spike tuples
Function	fetch tuples from memory in advance of their sampling interval	
Output	**spike injector**	stream of injected spike tuples

Efficiency

The **memory spike source** makes efficient use of off-chip memory bandwidth by fetching injected spike tuples using burst reads and buffering them until they are needed.

7.2.6 Spike injector

The **spike injector** allows injected spikes to be applied to neurons in a neural network from external sources It is used to provide some initial stimulus to start a computation (sourcing these injected spikes from the **memory spike source**), and could also be used to interface to external systems.

Input

Injected spikes are taken from multiple sources in the form of tuples of the same form as used by the **memory spike source** (Figure 7.7). Input can be either from the **memory spike source** or from other sources such as external sensors. It is assumed that the tuples presented from each source are already ordered by sampling interval number. The current sampling interval number and a next sampling interval signal are sourced from the **control logic**.

Function

If an input tuple is for the current sampling interval then it is sent to the output. Input tuples for sampling intervals that have already passed are discarded.

Output

Injected spike tuples are sent to the **accumulator** so that they can be applied to their target neurons.

Summary

Input	**memory spike source** and other sources **control logic**	streams of injected spike tuples current sampling interval number
Function	select tuples for the current sampling interval	
Output	**accumulator**	stream of injected spike tuples

7.2.7 Spike auditor

The **spike auditor** records spike events that are generated by a neural computation. Each FPGA maintains a record of spike events generated by its neurons in off-chip memory.

Input

A stream of neuron identifiers from the **equation processor** indicate which neurons have spiked. The current sampling interval number is provided by the **control logic**.

Function

Neuron identifiers are combined with the current sampling interval number to form tuples, laid out as in Figure 7.8 on the next page. These tuples are buffered until there are 64, which fill a whole burst write of 8×256 bit words.

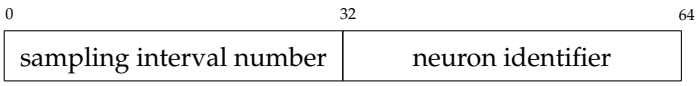


Figure 7.8: Layout of a spike auditor tuple

Output

A block of tuples is written to off-chip memory using a burst write. These tuples can be read by a host PC to produce an output from the neural computation.

Summary

Input	**equation processor control logic**	neuron identifiers current sampling interval number
Function	create spike auditor tuples	
Output	off-chip memory	blocks of spike auditor tuples

Efficiency

The **spike auditor** makes efficient use of off-chip memory bandwidth by writing out spike event records 256 bits at a time.

7.2.8 Control state machine

The **control state machine** runs on a single FPGA in the system. As well as controlling setup of the inter-FPGA communication system (see Section 7.4), it also sets up the neural computation system (including memory offsets for each hardware block) and controls the progress of the computation by determining when a sampling interval has ended and then requesting that all FPGAs advance to the next sampling interval.

Input

Messages are received from the **control logic** in all FPGAs in the system (including the FPGA that also contains the **control state machine**) to indicate when the sending FPGA has finished the setup process or finished processing all of the computation for the current sampling interval.

Function

In the setup phase each the messages that indicate that an FPGA has finished setup are counted until the total number of messages matches the total number of FPGAs in the system. At that point all FPGAs will have finished setup and the neural computation can be started.

Once the neural computation has been started a similar process is used to count messages from each FPGA that indicate that it has finished the computation for the current sampling interval. Once a message has been received from all FPGAs the next sampling interval is started.

Output

A message is sent to the **control logic** in all FPGAs in the system to start the setup process.

When a new sampling interval starts a message is sent to the **control logic** in all FPGAs in the system containing the new sampling interval number.

Summary

Input	**control logic** (via inter-FPGA communication)	finished setup and finished computation messages
Function	determine when setup has finished and when to start the next sampling interval	
Output	**control logic** (via inter-FPGA communication)	setup request and new sampling interval messages

7.2.9 Control logic

The **control logic** provides the current sampling interval number to other hardware blocks, determines when an FPGA has finished computation for a sampling interval and receives commands from the **control state machine**.

Input

Messages from the **control state machine** indicate when the setup process should be started. Other messages indicate when a new sampling interval has started and the number of this sampling interval. Signals from other hardware blocks indicate when they have finished computation for the current sampling interval.

Function

Status signals from other hardware blocks are monitored to determine when setup has finished and when computation for the current sampling interval has completed.

Output

A message is sent to the **control state machine** when setup has finished or computation for the current sampling interval has completed. A signal is sent to each hardware block when a new sampling interval begins as well as the number of this sampling interval.

Summary

Input	**control state machine** (via inter-FPGA communication) other hardware blocks	setup and next sampling interval messages finished computation signal
Function	determine when all other hardware blocks are ready for the next sampling interval	
Output	**control state machine** (via inter-FPGA communication) other hardware blocks	setup finished and ready for next sampling interval messages new sampling interval signal sampling interval number

7.3 Input data

A neural network is programmed into the massively parallel neural computation system using memory image files that are copied into the off-chip memory of each FPGA from a host PC. These image files are generated in advance on the host PC from a neural network description. Each of the hardware blocks in the system that accesses off-chip memory will accesses data from the memory image to perform its part of the neural computation.

7.3.1 Neural network description format

The input to the memory image file generator consists of two text files, one describing each of the neurons in a neural network and one describing the synaptic connections.

Neuron file

The neuron file contains the parameters of the neuron modelling equation for a single neuron on each line, separated by spaces. An example extract from this file is shown in Table 7.1. These parameters are for the original version of the Izhikevich neuron modelling equation (Equation 3.1). They are translated to the values required by Equation 3.10 as part of the memory image file generation process. There are also parameters for the magnitude of an initial injected spike (I_n) and the sampling interval when this initial spike should be injected (n).

Neuron Id	v_0	u_0	a	b	c	d	I_n	n
1000	-70.0	-14.0	0.02	0.20	-65.0	6.4	20	1
1001	-75.0	-15.0	0.03	0.25	-66.0	6.2	0	0
1002	-60.0	-13.0	0.02	0.20	-67.0	6.3	0	0
1003	-70.0	-12.0	0.03	0.25	-68.0	6.7	0	0

Table 7.1: Example extract from neuron file

Connection file

The connection file contains the parameters of a synaptic connection on each line, separated by spaces. An example extract from this file is shown in Table 7.2.

Source Neuron	Target Neuron	Weight	Delay
10	25	11	2
12	17	26	6
1012	557	-22	5
2018	2002	-9	15

Table 7.2: Example extract from connection file

7.3.2 Memory image file format

The format of a memory image file for each FPGA is shown in Figure 7.9 on the following page. After a header there are four major blocks of data: neural modelling equation parameters, fan-out tuples, update tuples and spike injector tuples. The format and meaning of each of these tuples was defined in Section 7.2. The layout of these tuples is reproduced for completeness in Figure 7.10 on the next page.

The header consists of a 256 bit word that contains the offsets of the neuron modelling equation parameter, fan-out tuple, update tuple and injected spike tuple regions. These offsets are read by the **control logic** as part of the system setup process and are then used by the **equation processor**, **fan-out engine**, **accumulator** and **memory spike source** when reading blocks of tuples from off-chip memory. The layout of the header is shown in Figure 7.11 on page 125.

7.3.3 Memory image file generation

As well as translating the input data files from Section 7.3.1 into a format that is usable by the system, the major task involved in generating memory image files for each FPGA is building blocks of tuples and populating other tuples with pointers to these blocks. This process is performed on the host PC in advance of a computation on the system. The memory image files are generated by 4 separate processes (one for each type of tuple), with some cross references between them as a result of the pointers to different types of tuple.

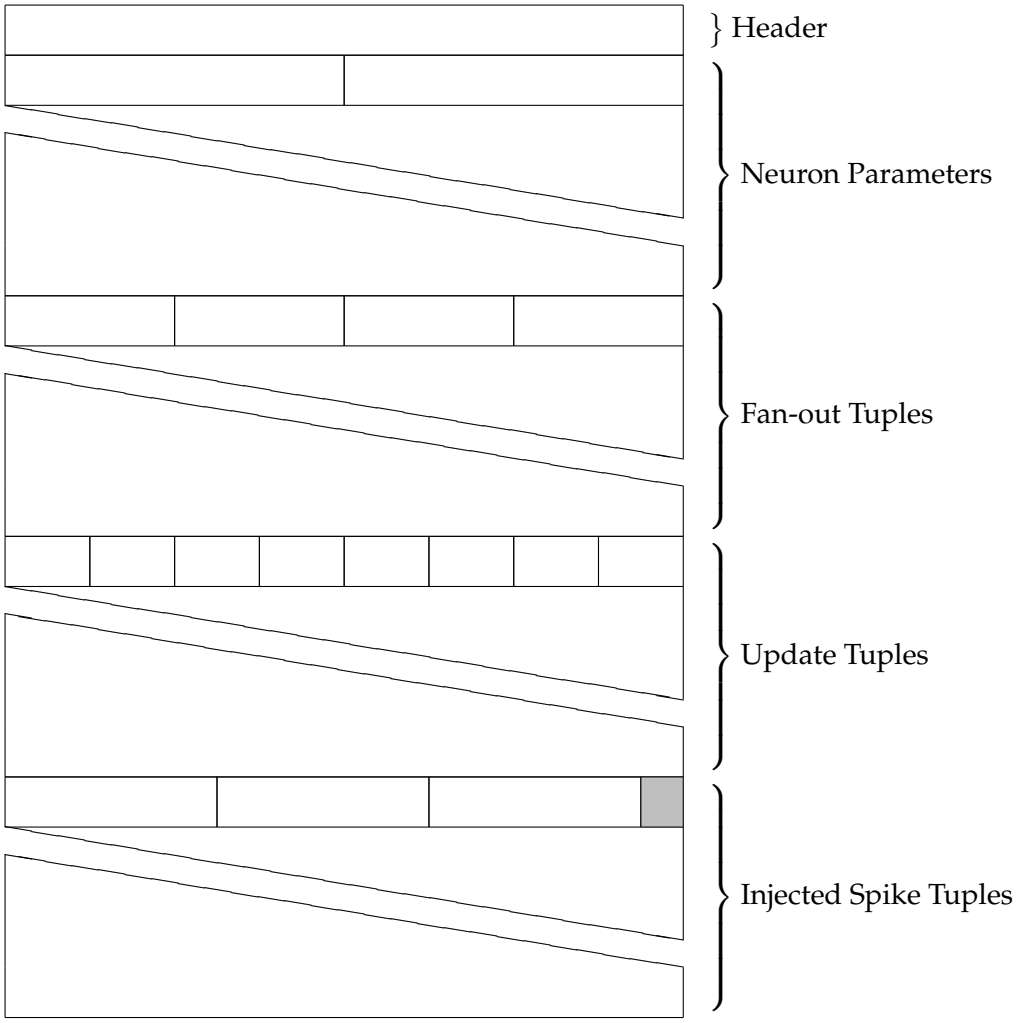
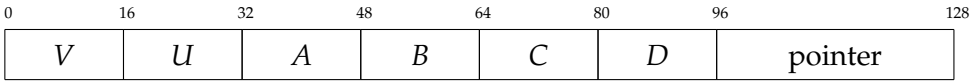
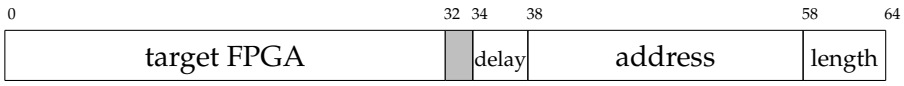


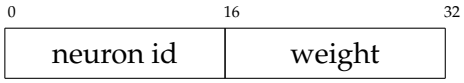
Figure 7.9: Format of a memory image file



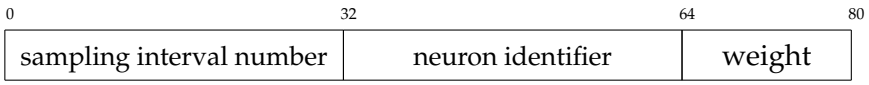
Neuron modelling equation parameters



Fan-out tuple



Update tuple



Injected spike tuple

Figure 7.10: Formats of tuples used by the neural computation system. Reproduced from Section 7.2

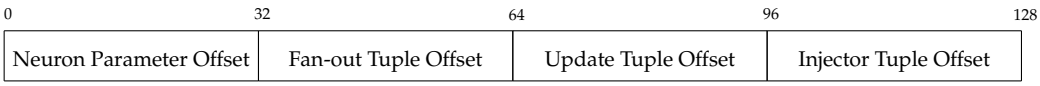


Figure 7.11: Layout of a memory image file header. The upper 128 bits are unused and are not shown

The memory image file generation algorithm maintains data structures that represent the off-chip memory of each FPGA, as well as data structures used to generate each type of tuple. These data structures are serialised, combined and a header added to produce the final memory image file for each FPGA.

Neuron modelling equation parameters

The neuron modelling equation parameters are generated from the neuron file and are converted to fixed-point parameters for Equation 3.10. The pointers to blocks of fan-out tuples are added once the update and fan-out tuples have been generated and allocated memory addresses.

Fan-out and update tuples

The algorithm used to generate blocks of fan-out and update tuples and pointers to these blocks works by replicating the path of every synaptic update through the multi-FPGA, massively parallel neural computation system. Every connection in the connection file is converted to an object that represents the progress of a synaptic update from its source neuron to its target neuron, with a linked list to record the path taken by the update. Another data structure (one for each FPGA) records the updates that have passed through or arrived at each FPGA, grouped by source neuron identifier and delay.

Each synaptic update is routed from its source neuron to its target neuron in the same way that it would be in the multi-FPGA, massively parallel neural computation system. When this process has finished, each FPGA will have lists of updates that are targeted at it, grouped by source neuron identifier and delay. These lists are converted to blocks of update tuples and are allocated to memory addresses, allowing pointers to these blocks to be produced.

The path taken by a single update in each block is then followed in reverse, allowing blocks of fan-out tuples to be produced. Again these tuples are allocated to memory addresses so that pointers to these blocks can be produced. Multiple blocks of fan-out tuples on different FPGAs can be generated for each source neuron.

Finally the pointer to the block of fan-out tuples for each source neuron that is on the same FPGA as that neuron is added to the neuron modelling equation parameters.

Injected spike tuples

The injected spike tuples are generated by reading the parameters I_n and n from the neuron file.

7.4 Inter-FPGA communication

The inter-FPGA communication system allows FPGAs in the massively parallel neural computation system to communicate, both to communicate synaptic updates and for coordination between the **control state machine** and **control logic** on each FPGA.

The inter-FPGA communication system makes use of the high-speed transceivers integrated into the Stratix IV FPGA on the DE4 board, as well as some custom physical connectivity, link layer protocols and a routing system. The end result is a communication system that allows any hardware block on any FPGA to communicate with any other block on any other FPGA, without being aware of the physical topology of the FPGAs, and with the assumption that communication is reliable.

7.4.1 Physical layer

Physical connectivity between FPGAs is provided using the high-speed transceivers provided by the Stratix IV FPGA. A combination of 4 SATA connectors on the DE4 board and a further 8 on a breakout board (see Section 5.4.1) gives 12 bidirectional, high-speed links per FPGA.

While SATA connectors and cabling are used because of their low cost, reliability and interoperability, this does not limit the links between FPGAs to the protocols and data rates defined by industry standards (nominally 3 Gbit/s). The Stratix IV transceivers support data rates of up to 10 Gbit/s in some cases. A data rate of 6 Gbit/s is used as this is the highest data rate that the Altera design tools will accept given the range of available input clock frequencies. Theo Markettos has confirmed that the physical links (using both the SATA connectors on the DE4 board as well as those on the breakout board) have a bit error rate of less than 1 error in 10^{15} bits at 6 Gbit/s. With 12 bidirectional links per board a variety of topologies could

be supported. However the current version of the massively parallel neural computation system only makes use of 4 links to produce a two-dimensional torodical mesh, as this is sufficient for the number of FPGAs in use.

7.4.2 Link layer

The link layer was developed by Simon Moore. It encapsulates each physical link to provide a communications channel which:

- Presents a FIFO interface to higher-level logic
- Marshals data from a flit-based format to individual 8B10B symbols
- Handles booting and resetting of the physical link and provides presence detection
- Provides reliability with retransmission of failed flits as needed

Each physical link on a FPGA is wrapped in link layer logic. The resulting reliable links are then connected to the routing infrastructure to produce an inter-FPGA communication system.

Data is sent from and received by higher-level logic using flits. The layout of a flit is shown in Figure 7.12. This has fields to indicate the start and end of a packet if flits are combined to produce larger packets and also a channel field that is used to indicate the message type and where it should be delivered and a data field.

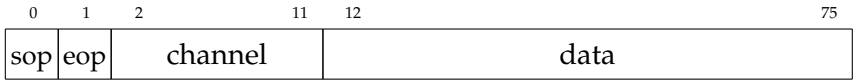


Figure 7.12: Layout of a link layer flit. sop and eop stand for "start of packet" and "end of packet" respectively

7.4.3 Routing

To allow communication between any pair of FPGAs in a multi-FPGA system, a routing system is needed. The routing system is designed to be used by any application on the multi-FPGA platform, regardless of the number of FPGAs and the number of dimensions in their topology. The only assumption is that the FPGAs are configured in an N-dimensional torus. It has the following features:

- Automatic end-to-end routing of messages from source to target FPGAs without any intervention from the application at intermediate FPGAs
- No requirement for the application to be aware of the physical topology of the FPGAs
- Automatic allocation of FPGA identifiers and discovery of the FPGA topology at boot time
- Support for multiple copies of an application on a FPGA via an additional topology dimension
- Support for some links between FPGAs to be omitted

Automatic identification system

The automatic identification system is used to allocate unique addresses to all FPGAs in the multi-FPGA platform, and to determine their topology. The following information needs to be provided for the system to perform this function:

1. The number of dimensions in the FPGA topology
2. The dimension allocated to each link on a FPGA, and whether messages following the link will reach a FPGA with a higher or lower address in that dimension
3. Which FPGA is the master

The system is then able to infer:

1. The address of each FPGA in the multi-FPGA platform
2. The total size of the multi-FPGA platform in each dimension
3. When all FPGAs have been allocated addresses and when the size of the multi-FPGA platform has been determined in all dimensions
4. Whether the FPGA topology is inconsistent in any way, for example a link has been connected between an incorrect pair of FPGAs

The identification process is started by the master FPGA. It is the first stage of booting any application on the multi-FPGA platform, as all other stages of the boot process rely on being able to send messages between all FPGAs in the platform. The identification algorithm is largely stateless, other than each FPGA gradually building up its address in each dimension and then the size of the multi-FPGA platform in each dimension.

Three types of messages used by the algorithm. All are sent point-to-point between adjacent FPGAs:

Address setup
: Contains the address of the sending FPGA. When another FPGA receives this message it knows that the FPGA whose address is given is at the other end of the link the message was received on. The recipient can then either infer its own address or (if it already has an address) or check that the local FPGA topology is consistent.

Dimension setup
: Contains the size of the multi-FPGA platform in the dimension in which the message was received. When a FPGA receives this message it updates its local view of the dimensions of the platform.

Identification finished
: Indicates that the identification system has finished identifying all FPGAs in the platform.

The identification algorithm starts by allocating address zero in all dimensions to the master FPGA. The master FPGA then sends an address setup message with its address on each of the links which connect to a FPGA which will have a higher address in each dimension (positive-going links). The remainder of the algorithm then proceeds based on the types of messages received by each FPGA in the platform, and whether that FPGA already has an address set, as shown in Figure 7.13 on the following page.

The master FPGA can determine the total number of FPGAs in the multi-FPGA platform as soon as it has generated dimension setup messages for every dimension. It then compares the number of FPGAs that it expects to exist with the number of identification finished messages that it receives. Once the correct number of messages is received the boot process can move to the next stage in the knowledge that the end-to-end dimension-ordered routing system is available. The address of each FPGA and the dimensions of the multi-FPGA platform are passed to the target to hops convertor.

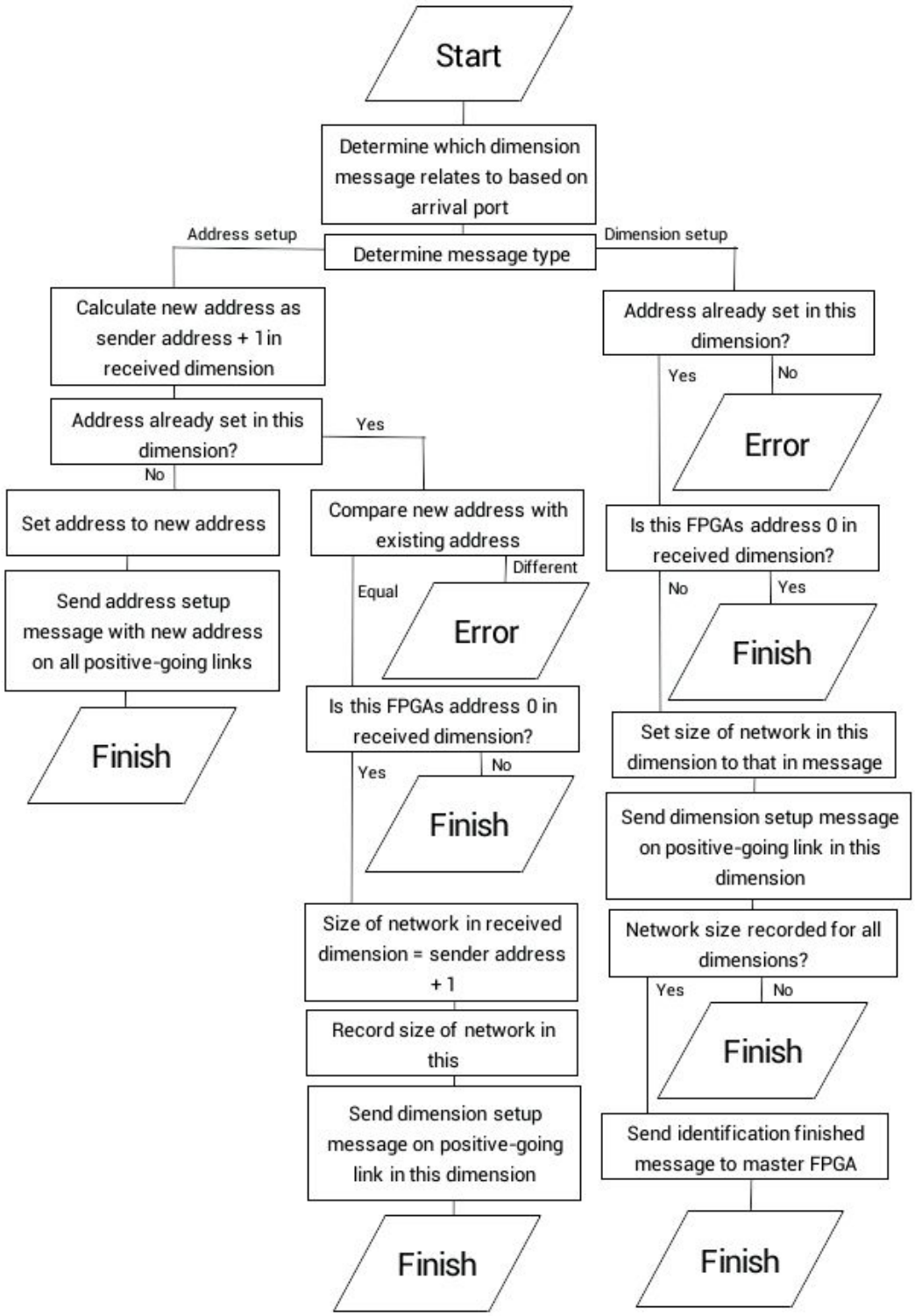


Figure 7.13: Processing of messages by automatic identification system

Target FPGA to hops conversion

The target to hops converter takes the address of a target FPGA and converts it to the number of hops needed in each dimension to reach the target FPGA. The hops will be either positive for all dimensions or negative for all dimensions (positive-going or negative-going), so that requests and replies are guaranteed to take disjoint paths through the inter-FPGA communication system, helping to prevent deadlock. Once the number of hops to reach the target FPGA has been determined, the message and the hops data are passed to the dimension-ordered router.

Dimension-ordered router

The dimension-ordered router sends and receives messages on each of the reliable links on a FPGA, as well as to the system implemented within the FPGA. The router is designed to introduce as little latency as possible, by avoiding complex calculations at intermediate routers between source and target FPGAs. The routing algorithm decides which link to send a message on based on the number of hops remaining, whether the hops are positive- or negative-going and the availability of links.

For each message:

1. If there are no hops remaining it is delivered to this FPGA
2. Otherwise send the message on the link in the highest dimension with remaining hops
3. If that link does not exist, send the message on a link in the next lowest dimension with an available link and remaining hops
4. If no exit route is available, an error has occurred

This routing algorithm allows for some links to be omitted in higher dimensions. For example if there were three dimensions, x, y and z, the x dimension could represent links between adjacent FPGAs, which will always exist, and the z dimension could represent links between boxes in a rack, where a link may not be provided between all pairs of FPGAs which logically straddle the divide between boxes.

7.4.4 Message filtering and delivery

When a message is delivered to a FPGA, it must be passed to one or more hardware blocks in the FPGA. This might be a higher-level block, for example the identification system, or a lower-level block such as part of the massively parallel neural computation system.

This is achieved using a hierarchy of simple switches which either deliver a message to logic at that level in the FPGA or pass it to a switch at the next level down. Switching decisions are made based on the channel field in a flit. Messages with a channel number lower than the switch's extraction threshold are extracted while other messages are passed to the next level. In addition to this usage of the channel number, the recipient can also use it for any application-specific purpose.

7.4.5 Application usage

In order to use the inter-FPGA communication system, an application (such as the massively parallel neural computation system) needs to:

1. Select the type of message using a channel number. As well as indicating to the eventual recipient what actions should be performed, the channel number is also used (as noted above) by the filtering and delivery system
2. Provide the address of the target FPGA and optionally application block within the FPGA. No information about the topology of the multi-FPGA system is required apart from knowledge of how many application blocks are in a FPGA
3. Provide a payload if appropriate

A message is formatted as in Figure 7.14 (which also shows how the message fits into a link layer flit). It is then converted into a flit and passed to a local switch, which will arrange delivery to the target.

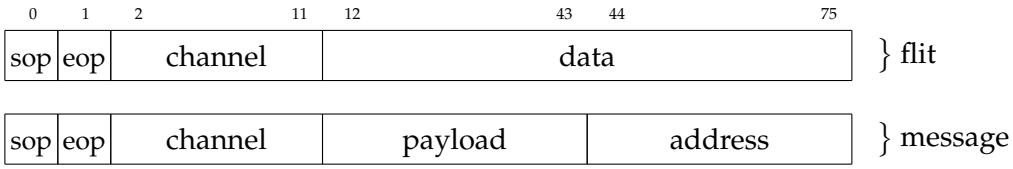


Figure 7.14: Layout of an inter-FPGA message showing how it fits into a link layer flit

7.5 Implementation languages

The majority of the multi-FPGA massively parallel neural computation system is implemented using Bluespec SystemVerilog (BSV) (Nikhil and Czeck, 2010), 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 allows the architecture to be easily expressed in a 'communicating sequential processes' style, which is very well suited to a system with multiple components.

BSV comes with a simulator (Bluesim) which allows for cycle-accurate simulation of the behaviour of a system which is significantly quicker than an RTL-level simulation such as ModelSim. Bluesim allows simulated memories to be populated using a data file, which allows the massively parallel neural computation system to be simulated using the same off-chip memory image files (with a few minor formatting changes) as are used to program the multi-FPGA system on the Bluehive platform.

Memory image file generation is performed using a program written in C++. Various Makefiles and Perl scripts are used to link system components together in simulation and to upload and download data from the multi-FPGA system.

7.6 Conclusion

The implementation of a massively parallel neural computation system on the Bluehive multi-FPGA platform consists of separate hardware blocks for neuron modelling equations and to perform the fan-out and accumulation stages of the synaptic update communication and application algorithm introduced in Chapter 6, together with supporting infrastructure.

Each hardware block is designed to make efficient use of off-chip memory bandwidth by using burst reads and processing whole words in a single clock cycle. This should maximise the size of neural network that can be handled in real-time, as suggested by the Bandwidth Hypothesis.

Chapter 8

Evaluation

8.1 Introduction

The multi-FPGA, massively parallel neural computation system will be evaluated using two metrics:

1. Scale of neural network that can be handled in real-time
2. Speed-up when adding compute resources while keeping the neural network constant

These methods are similar to those used by Ananthanarayanan *et al.* (2009) to evaluate their supercomputer-based neural computation system.

A benchmark neural network of sufficient scale and with mean fan-out and mean spike frequency that match the assumptions that were made in Section 4.3 is needed. Since there are no benchmark neural networks available that satisfy these criteria, a benchmark neural network is developed in Section 8.2. While the pattern of synaptic connections in this benchmark neural network cannot be claimed to be biologically plausible, in all other respects it is suitable for evaluating the massively parallel neural computation system.

The multi-FPGA, massively parallel neural computation system is able to handle up to 64k neurons per FPGA in real-time using the benchmark neural network. Results are provided for systems using 1, 2 and 4 FPGAs, which perform computations for networks of 64k, 128k and 256k neurons respectively. Analysis of the number of clock cycles of work for each sampling interval of the computation for each of these systems shows that there is a significant jump in the number of clock cycles of work per sampling interval between 1 and 2 FPGAs, but little extra between 2 and 4 FPGAs. This provides promise for scaling to larger neural networks with more FPGAs, and meets the requirements of the Scalability Hypothesis.

The speed-up when adding compute resources to a computation of a benchmark network of 64k neurons is almost linear with 1, 2 and 4 FPGA systems. This suggests that the massively parallel neural computation system is sufficiently successful at optimising communication and memory bandwidth usage that (at least for this particular benchmark neural network), neural computation has become compute-bound rather than communication bound, showing that scalability can be achieved by adding compute resources provided that there are also sufficient communication resources.

8.2 Benchmark neural network

The benchmark neural network is designed to provide a "load test" of the neural computation system, which scales to the number of neurons required and has a biologically-plausible fan-out size (but not fan-out pattern) and spike frequency. Taking these parameters from Section 3.3, the benchmark neural network will have a mean fan-out of 10^3 and a mean spike frequency of 10 Hz.

The benchmark neural network is made up of a number of synfire chains. A synfire chain (Abeles, 1982) is a neural network where groups of neurons have synaptic connections to other groups of neurons. Each group of neurons spikes synchronously after receiving spikes from another group of neurons.

In the synfire chain used in the benchmark neural network, each neuron has a combination of neuron modelling equation parameters and synaptic delays and weights (shown in Table 8.1) that result in it spiking 10 ms after it receives a synaptic update. They were selected after experiments using NEURON (Hines and Carnevale, 1997), which provides an Izhikevich neuron model, and allows the behaviour of a single neuron to be observed as its parameters are changed.

If an input voltage (I-value) of 16 mV is applied to a neuron with these parameters then it spikes immediately. If 10 of these neurons are connected in a chain with a synaptic delay of 10 ms, then each neuron will spike at 100 ms intervals, which is equivalent to a spike frequency of 10 Hz.

Biologically-plausible fan-out size is created by grouping neurons into blocks of 10^3. Each neuron in a block has 10^3 synaptic connections, one to each of the other neurons in the block. The weights of the synaptic connections from each set of 100 neurons to the next set of 100 sum to 16 mV, so that when the first 100 neurons in the block spike in unison this will result in the next 100 neurons in the block spiking in unison after 10 ms and so on, with synaptic connections from the last 100 neurons looping around back to the first 100 neurons. The result is a synfire chain where each neuron has a fan-out of 10^3 and a spike frequency of 10 Hz.

Parameter	Value
v_0	-70.0
u_0	-14.0
a	0.02
b	0.2
c	-65.0
d	6.0

Table 8.1: Parameters of neuron modelling equation (Equation 3.1) used to create benchmark neural network

The scale of a benchmark neural network is increased by observing that each block of 10^3 neurons has a set of 100 neurons spiking every 10 ms, with no spike activity in between. Therefore a network of 10^4 neurons that maintains a biologically-plausible fan-out size and mean spike frequency can be created by interleaving 10 of these 10^3 neuron blocks. The time of the initial injected spike that starts each block's spike chain is offset by increments from 0 to 9 ms for blocks 0 to 9 respectively.

The result is a benchmark neural network of 10^4 neurons, with biologically-plausible fan-out size and mean spike frequency. To create larger benchmark neural networks, multiple copies of this 10^4 neuron network are instantiated until the benchmark neural network reaches the required size. For example a benchmark neural network of 6.4×10^4 neurons will have 6.4 copies of the 10^4 neuron network. The decimal part is achieved using a network of 4×10^3 neurons with 4 offset blocks of 10^3 neurons. There will be slight variations in the workload produced by a benchmark neural network with a decimal part between sampling intervals, but fan-out and spike frequency remain unaffected.

8.3 Scale

The ability of the multi-FPGA massively parallel neural computation system to perform neural computations for large neural networks in real-time will be shown using a neural computation of a benchmark neural network of 256k neurons on an implementation of the system with 4 FPGAs. A plot of the spikes recorded from this neural computation is shown in Figure 8.1 on the next page. Figure 8.2 on page 140 shows the number of neurons spiking in each sampling interval and the number of clock cycles of work per sampling interval. As the number of clock cycles of work for each sampling interval is consistently less than 200k (dashed green line), the computation is running in real-time.

Neurons are allocated to FPGAs in groups of two e.g. neurons 0 and 1 are allocated to FPGA 0, neurons 2 and 3 to FPGA 1 etc. Since each neuron has a fan-out of 10^3 this means that synaptic updates from each neuron will target 250 neurons on each of the 4 FPGAs. Locality is deliberately reduced in this way to avoid an optimal allocation of neurons to FPGAs, which would place all of the neurons in each block of 10^3 on the same FPGA. This would result in no inter-FPGA communication and hence essentially 4 independent neural computations with 64k neurons each – not a fair test of a system designed for massively parallel neural computation.

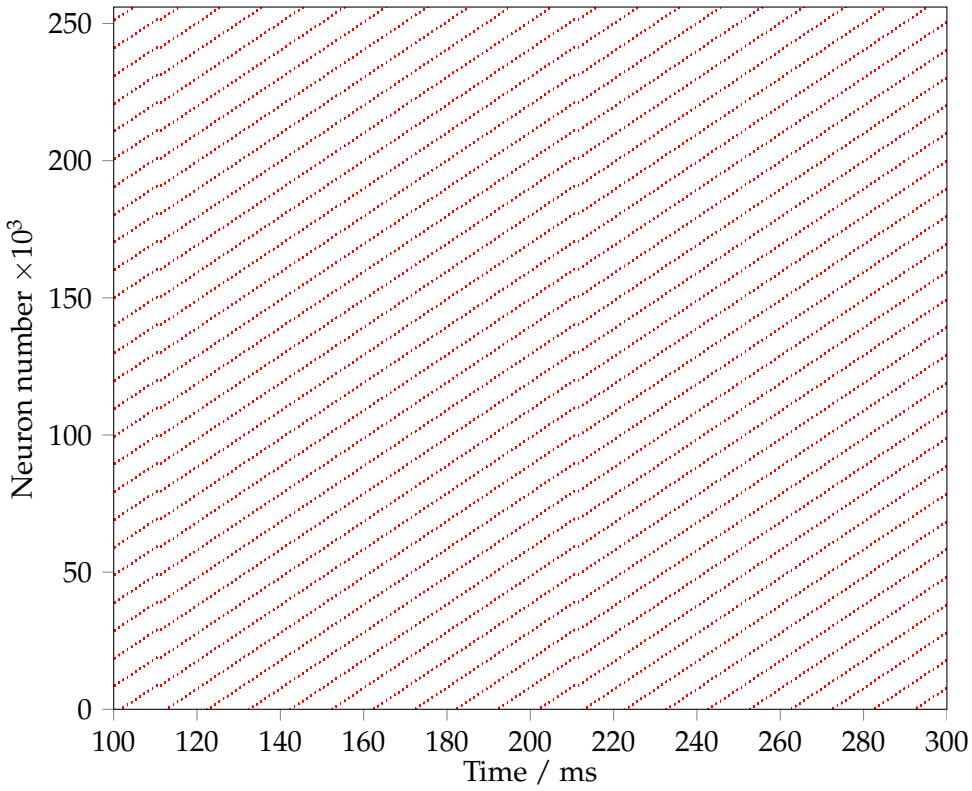


Figure 8.1: Spike pattern for a computation of 256k neurons over 4 FPGAs

Figure 8.3 on page 141 shows the number of clock cycles of work per sampling interval for benchmark neural networks of 64k, 128k and 256k neurons. These networks are distributed over 1, 2 and 4 FPGAs respectively so that there are 64k neurons on each FPGA. There are four causes of variation in the number of clock cycles of work per sampling interval between the 64k, 128k and 256k networks:

1. The jump in maximum clock cycles per sampling interval between 64k neurons and 128k and 256k neurons is most likely a result of the need to use the inter-FPGA communication system to communicate synaptic updates between FPGAs, which is not necessary with the 64k neuron network.

2. The major peaks and troughs are the result of "excess" neurons that are not part of a complete set of 10k neurons. For example the 128k neuron network has $12 \times 10\text{k} + 8\text{k}$ neurons, with the 8k neurons being constructed of $8 \times 1\text{k}$ neuron sets and 2 "gaps." This means that more neurons will spike and hence more synaptic updates will need to be communicated and applied in the first 8 1 ms sampling intervals in every 10 than in the last 2, and hence that the number of clock cycles of work per sampling interval varies in the same pattern.

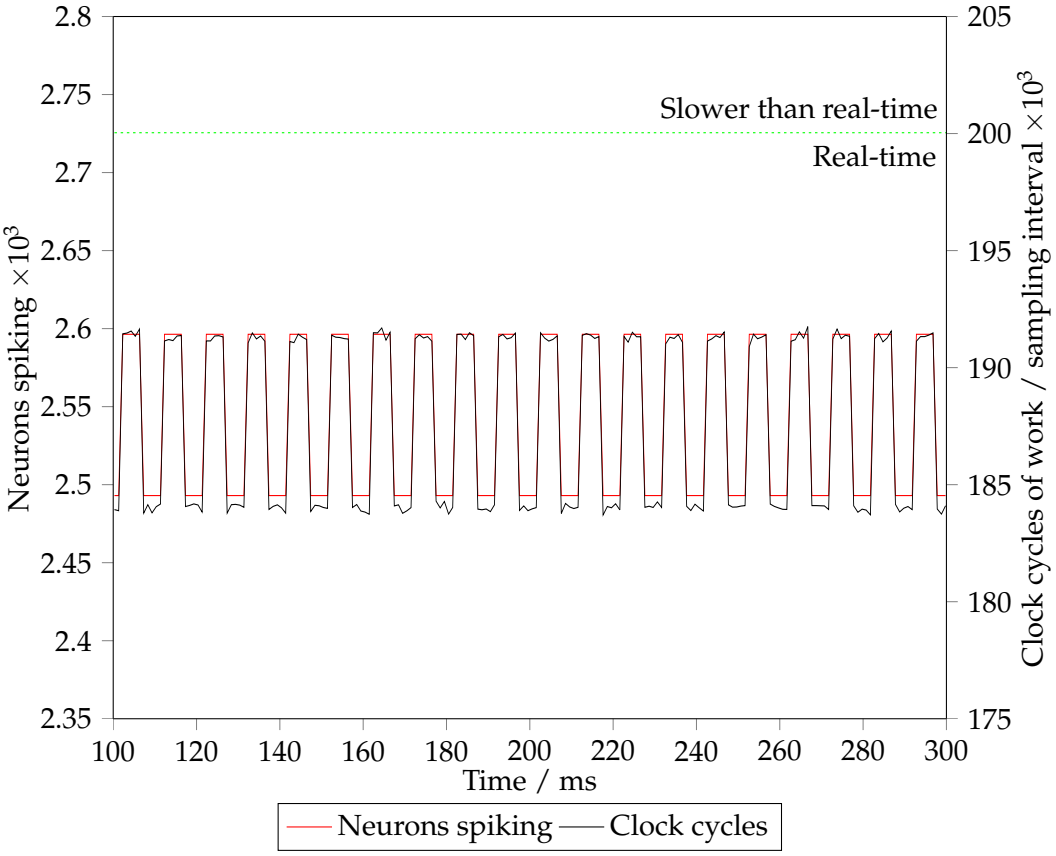


Figure 8.2: Neurons spiking per sampling interval and clock cycles of work per sampling interval for computation of 256k neurons using 4 FPGAs

For the 256k neuron network there are $25 \times 10\text{k} + 6\text{k}$ neurons, with the 6k neurons being constructed of $6 \times 1\text{k}$ neuron sets and 4 "gaps," and hence the peaks and troughs are more equal than with 128k neurons.

3. The amplitude of the peaks and troughs is dependent on the number of FPGAs that the "excess" neurons are distributed over. With 64k and 128k neurons each FPGA has 4k excess neurons while with 256k neurons each FPGA has 1.5k excess neurons, and hence the variation in clock cycles per sampling interval caused by communicating and applying synaptic updates from these excess neurons is lower.

4. Minor variations are an artefact of the mechanism used to determine when all computation for a sampling interval has been completed, which relies on each FPGA determining when no data remains in the input and output FIFOs of each component of the neural computation system and then signalling this to the master FPGA, which determines when all FPGAs have completed computation and signals that computation for the next sampling interval should be started.

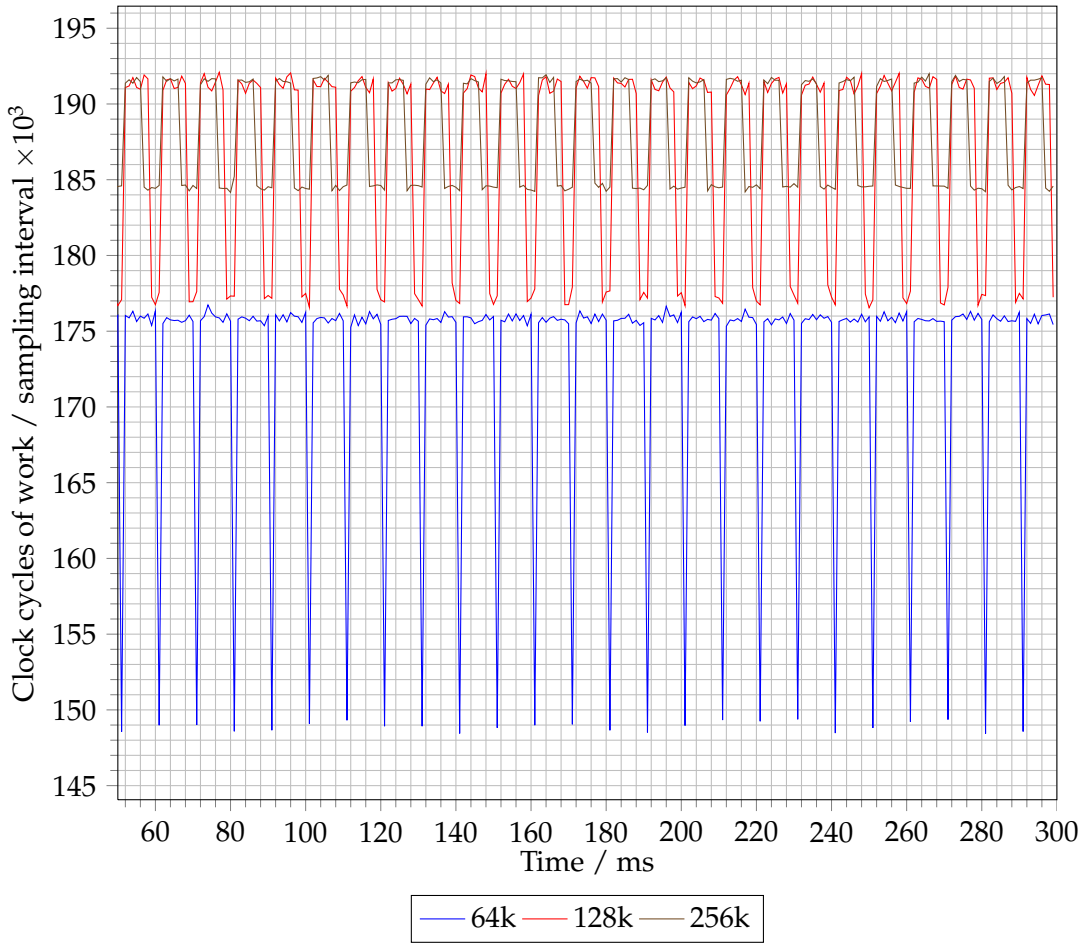


Figure 8.3: Clock cycles of work per sampling interval for benchmark networks of 64k, 128k and 256k neurons on 1, 2 and 4 FPGAs respectively

While this will need to be confirmed by further experiments, it is likely that a computation of a benchmark neural network of 512k neurons distributed over 8 FPGAs will have a maximum number of clock cycles of work per sampling interval which is close to that of that of the network with 256k neurons as the number of neurons per FPGA will remain constant. There will be 2k excess neurons in total, but only 256 excess neurons per FPGA, and so the number of clock cycles per sampling interval will show variation that has wider troughs than peaks but lower amplitude than that shown by the 256k neuron network. This suggests that the system will scale to computations of larger neural networks spread over more FPGAs while preserving real-time.

8.4 Communication overhead

The overhead of inter-FPGA communication and partitioning a neural network between multiple FPGAs can be examined by spreading a computation of a benchmark neural network of 64k neurons over 1, 2 and 4 FPGAs. If there were no overhead then the 2 and 4 FPGA computations would be expected to take 50% and 25% of clock cycles of work per sampling interval of the 1 FPGA computation respectively.

Figure 8.4 on the facing page shows the number of clock cycles of work per sampling interval for a benchmark neural network of 64k neurons when the computation is spread over 1, 2 and 4 FPGAs. Figure 8.5 on page 144 shows the same data for 2 and 4 FPGAs as a percentage of that for 1 FPGA. This shows that the actual percentages are close to the theoretical values of 50% and 25%. This shows an almost linear speed-up, and suggests that the massively parallel neural computation system has been sufficiently successful at optimising communication and memory bandwidth usage that (at least for this particular benchmark neural network), neural computation has become compute-bound rather than communication bound. This shows that scalability can be achieved by adding compute resources provided that there are also sufficient communication resources.

8.5 Validation

The behaviour of the neural computation system was cross-validated against a PC-based system written in Java by Steven Marsh. There was no deviation in the output of the two systems for a sample of 300 ms of activity of the benchmark neural network presented in Section 8.2. This activity included that shown in Figure 8.1.

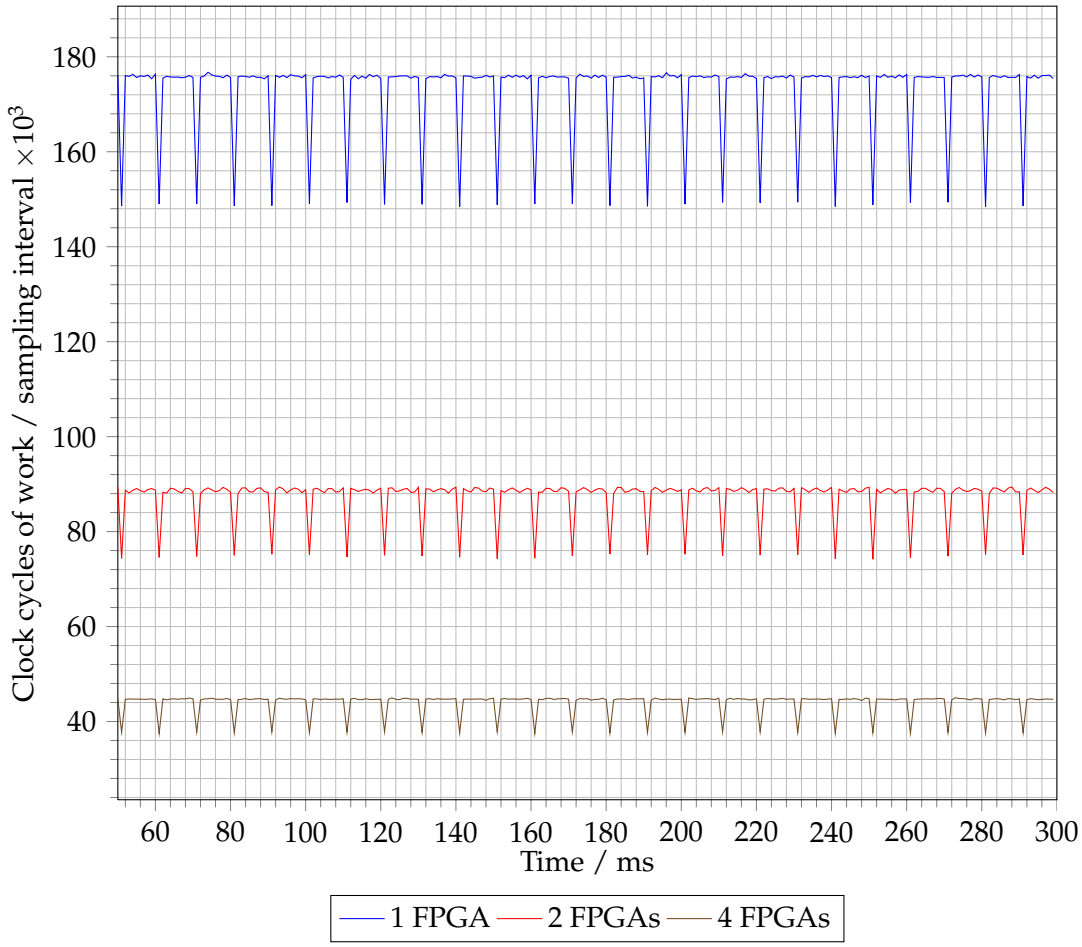


Figure 8.4: Clock cycles of work per sampling interval for a benchmark network of 64k neurons spread over 1, 2 and 4 FPGAs

8.6 Conclusion

The evaluation of the multi-FPGA massively parallel neural computation system implemented on the Bluehive platform shows that it is able to perform neural computation with up to 64k neurons per FPGA in real-time. Results are provided for benchmark neural networks of up to 256k neurons. The system shows promise of scaling to handle larger neural networks using more FPGAs.

The system shows almost linear speed-up when adding compute resources to a computation of a constant-sized neural network, which suggests that the massively parallel neural computation system is successful at optimising communication and memory bandwidth usage.

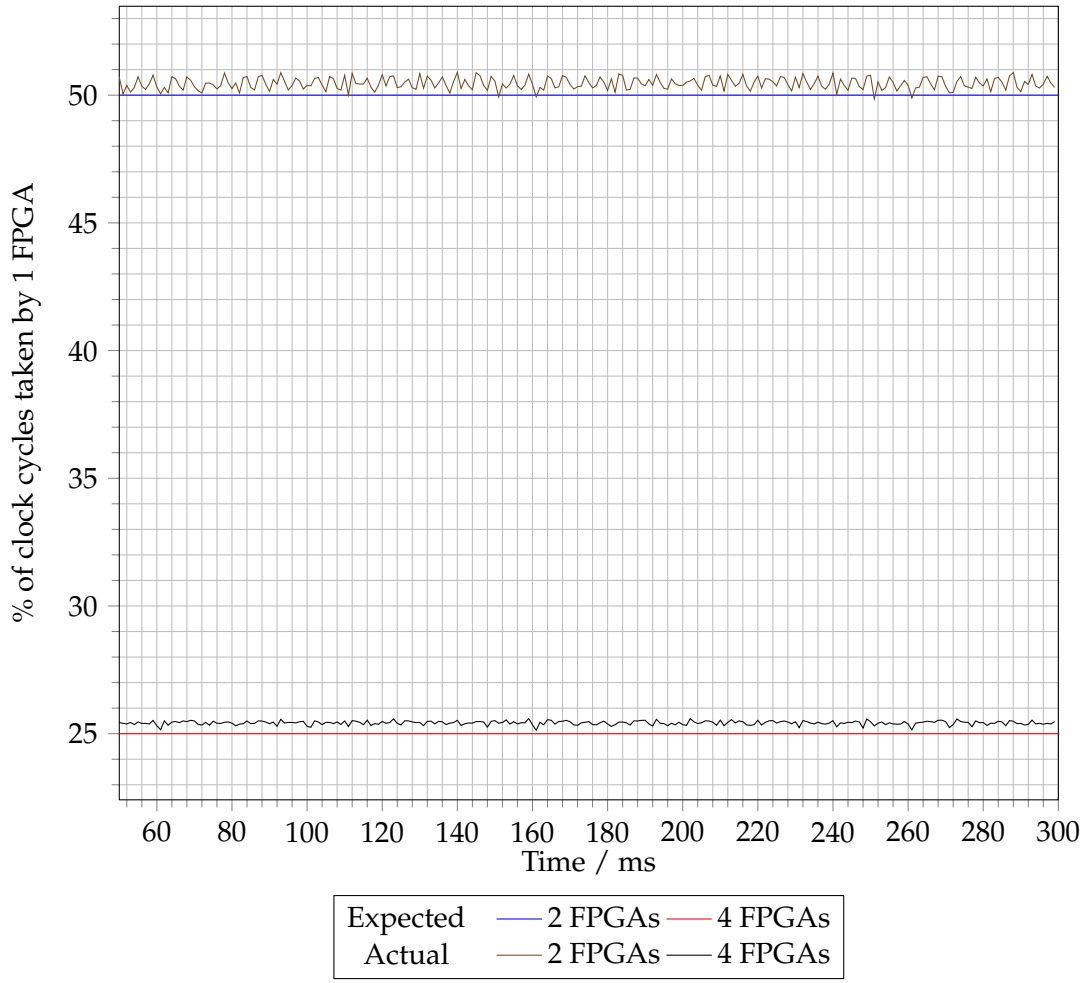


Figure 8.5: Clock cycles of work per sampling interval for a benchmark network of 64k neurons spread over 2 and 4 FPGAs as % of clock cycles taken by 1 FPGA

Chapter 9

Conclusions

The major conclusion of this thesis is that massively parallel neural computation systems must be designed with communication as a first-class design constraint. This manifests itself in several ways. Firstly it is clear that the implementation platform for a massively parallel neural computation system must allow the system to scale to multiple devices by providing high-bandwidth, low latency connections between the devices that make up the platform. This is made clear when the limitations of PC-based and GPU-based neural computation systems are considered.

Secondly, failure to consider the implications of communication in biological neural networks, and particular the operating rate of biological neurons, significantly hampers the scale of some types of neural computation systems. Those that dedicate resources to each neuron and synaptic connection cannot match the scale of those that time-multiplex resources, use an on- and inter-chip network for communication and store a neural network description in off-chip memory. Therefore all future massively parallel neural computation systems should use the latter approach.

Thirdly, the scale of neural computation that can be handled by a system implemented using today's electronics is limited by communication and memory bandwidth, and so neural computation systems must make maximum use of memory bandwidth by issuing memory requests whenever the memory controller is able to accept them, for example by using requests for contiguous blocks of data or having multiple parts of the system issue memory requests so that it is likely that a part of the system will be able to issue a request whenever the memory controller is able to accept even when other parts of the system are busy. Also, full use must be made of all data returned, for example by packing multiple items into long words and processing these items in parallel.

By taking a communication-centric design approach I was able to demonstrate a scalable that a FPGA-based massively parallel neural computation system can handle 64k neurons and 64M synaptic connections per FPGA. This system will be compared to other neural computation systems in Section 9.2.

9.1 Evaluation of hypotheses

Three hypotheses were introduced in Chapter 1. We will now examine how they are supported by the work in this thesis and the conclusion that communication must be considered as a first-class design constraint when implementing a massively parallel neural computation system.

9.1.1 Scalability Hypothesis

The scale of neural network that can be handled by a neural computation system in real-time must be able to be increased by scaling the system to multiple devices

Based on the review of related work in Chapter 2, it is clear that a neural computation system implemented using today's electronics needs to scale to multiple devices to support large-scale neural computation.

Recent technology scaling trends point towards future computation systems using multiple, small devices rather than fewer large devices. Given that many researchers ultimately aim to perform neural computations at a scale that matches the human brain, future neural computation systems will need to handle larger neural networks than current systems, and so it appears likely that future neural computation systems will need to be implemented using multiple, linked devices even if the size of neural network that can be handled by an individual device can be increased. This means that the implementation technology must provide suitable inter-device communication infrastructure.

9.1.2 Communication-Centric Hypothesis

The scalability of a neural computation system is communication-bound, not compute-bound

The analysis in Section 4.3 makes it clear that the volume of neural communication events that needs to be modelled per sampling interval is significantly greater than the volume of neuron modelling equation evaluations. With 10^5 neurons, mean fan-out of 10^3, mean spike frequency of 10 Hz and a sampling interval of 1 ms, 10^9 synaptic updates need to be communicated and applied and 10^8 neuron modelling equations need to be evaluated every second. Every synaptic update will require some kind of communication in a neural computation system, and so the volume of communication events that need to be handled is an order of magnitude greater than the volume of neuron modelling equation evaluations, which makes neural computation communication-bound rather than compute-bound.

9.1.3 Bandwidth Hypothesis

The scale of neural network that can be handled by a neural computation system in real-time is bounded by inter-device communication bandwidth and memory bandwidth

Section 4.3 and Section 4.5 found that the memory and communication bandwidth required by a neural computation of 10^5 neurons with mean fan-out of 10^3, mean

spike frequency of 10 Hz and sampling interval of 1 ms is at the limit of what is provided by today's electronics. Therefore it is impossible to increase the scale of neural network that can be handled by a neural computation system in real-time without either:

1. Making more efficient use of bandwidth by optimising data storage and access patterns and communication methods
2. Providing more memory and communication bandwidth to each FPGA, using either more memory and communication channels or new technologies
3. Using multiple FPGAs, which provides more memory and communication channels

Since it is possible that future technology might bring sufficient bandwidth that neural computation on that technology becomes compute-bound rather than communication bound, it is most accurate to say that:

The scale of neural network that can be handled by a neural computation system **that is implemented using today's electronics** *in real-time is bounded by inter-device communication bandwidth and memory bandwidth*

There is evidence for this in related work. For example the scale of neural network that can be handled in real-time by the GPU-based neural computation system implemented by Fidjeland and Shanahan (2010) is limited to whatever can be achieved using a single GPU as the implementation platform does not provide sufficient inter-device communication bandwidth to allow a computation to scale to multiple GPUs.

All three hypotheses point at communication needing to be considered as a first-class design constraint alongside computation when designing a massively parallel neural computation system.

9.2 Comparison to other systems

We will now compare the scale and performance of the FPGA-based massively parallel neural computation system to other neural computation systems, examining how their scalability is affected by the degree to which communication was considered in their design.

9.2.1 Supercomputer-based systems

Supercomputer-based neural computation systems must consider communication as a first-class design constraint as their programming model relies on distributing work over a very large number of CPUs and inter-process communication using custom interconnects and either shared memory or message passing communication models.

A direct comparison of the multi-FPGA neural computation system with supercomputer-based systems is difficult, as supercomputer-based systems have tended to use significantly more complex neuron models than other systems. The neural networks used in published work such as that by Ananthanarayanan *et al.* (2009) have higher fan-out (10^4 rather than 10^3) and a much larger number of neurons (10^9 compared to around 10^5) than the neural networks that have been used to evaluate other systems, including the system in this work. Supercomputer-based systems typically operate much slower than real-time e.g. around $800\times$ slower than real-time for a network with a mean spike frequency of 10 Hz.

A real-time neural computation of 10^9 Izhikevich neurons with a fan-out of 10^3 and mean spike frequency of 10 Hz using the architecture from this work would require 16×10^3 FPGAs, assuming that the architecture could scale sufficiently. Alternatively the architecture could handle a neural computation of around 2×10^8 Izhikevich neurons $800\times$ slower than real-time using 4 FPGAs, given the currently unrealistic assumption of each FPGA having 100 MB of on-chip memory or low-latency off-chip memory. If this could be scaled to more FPGAs, then 20 FPGAs should be able to handle a computation of 10^9 neurons.

This would still have a far less complex neuron model and lower fan-out than the supercomputer-based system, and so no direct conclusions can be drawn. However this does serve to illustrate the potential of FPGA-based neural computation systems. The cost of building a multi-FPGA system with 20 FPGAs is many orders of magnitude less than that of a supercomputer such as an IBM BlueGene/L.

In conclusion supercomputers are a suitable platform for neural computation as they provide sufficient communication and compute resources for a wide range of neuron and communication models, but their cost means that they are not readily available to researchers.

9.2.2 PC-based systems

Comparing the multi-FPGA massively parallel neural computation system to a commodity PC-based system gives an indication of the performance increase that can be expected by the majority of neuroscientists, who are currently using various PC-based systems. Since commodity PCs only support scaling using networking technologies such as Ethernet, which has a significant protocol overhead, this effectively limits PC-based systems to using a single PC.

Steven Marsh has written a neural computation system in C that uses the synaptic update communication and application algorithm from Chapter 6 and the fixed-point neural modelling equation from Chapter 3 (Moore *et al.*, 2012).

Using the same benchmark neural network that is used to evaluate the FPGA-based system, a single-threaded version of this PC-based system required 48.8 s to perform a computation of 300 ms of neural activity using a single thread of a 16-thread, 4-core Xeon X5560 2.80 GHz server with 48 GB RAM.

Therefore the 4-FPGA neural computation system is 162 times faster than this particular PC-based system. The PC-based system can clearly be improved, and in particular an adaptation of NEST (which is commonly used by neuroscientists for large neural computations) would provide a fairer "real-world" comparison, however this is sufficient to show that a neural computation system that has been designed without considering communication as a first-class design constraint has significantly inferior performance to the multi-FPGA system which was designed with communication and scalability in mind.

9.2.3 GPU-based systems

The GPU-based system implemented by Fidjeland and Shanahan (2010) provides an interesting point of comparison as it uses a discrete-time Izhikevich neuron model with a 1 ms sampling interval and benchmark neural networks with mean fan-out of 10^3, mean spike frequency of 10 Hz. This is the same neuron model that is used in the massively parallel neural computation system and both benchmark neural networks have the same statistical properties, so the two systems are readily comparable.

While the GPU-based system is able to handle neural networks with up to 55k neurons in real-time using a single GPU, it does not scale to larger networks using multiple GPUs. In comparison, the system in this work is able to handle up to 64k neurons per FPGA in real-time, with support for scaling to larger networks using multiple FPGAs.

GPUs have high communication bandwidth between each GPU and the CPU in the host PC, but if there are multiple GPUs then communication between them must all go via the CPU, which has high latency and does not allow the topology of a multi-GPU system to be adapted to suit an application. This shows that their scalability is hampered by not considering communication as a first-class design constraint.

If GPUs were provided with a more flexible communication topology with multiple links per GPU, then it would seem likely that they would be able to exhibit similar performance to FPGA-based systems, which suggests that future work could focus on neural computation systems that combine the more general-purpose compute architecture and high memory bandwidth of a GPU-based system with the flexible communication topology of a FPGA-based system.

9.2.4 Bespoke hardware-based systems

Many hardware-based neural computation systems have clearly not been designed with communication as a first-class design constraint as they do not support scaling to multiple devices or make inefficient use of communication resources. In particular FPGA-based systems that directly map neuron models into hardware and use FPGA routing resources to connect these "hardware neurons" support orders of magnitude less neurons and synaptic connections than the multi-FPGA system introduced in this work.

There is a similar situation with analogue hardware-based neural computations. These typically model small numbers of neurons and synaptic connections in greater detail than simple spiking neuron models, many times faster than real-time. But many of these systems are unable to support scaling to larger neural networks. Those systems that do support scaling to larger neural networks typically use some kind of packet-based signalling and an inter-chip network (often combined with an on-chip network), which shows that when communication is considered as a design constraint alongside computation the scale of neural computation can be increased.

Both of these types of neural computation systems are limited in scale if they dedicate resources to each individual neuron. If neural computation systems are designed to take account of the difference in the operating rate of biological neurons and today's electronics (a difference that is expected to increase in future) by time-multiplexing compute and communication resources then the scale of neural computation that can be supported per device is increase by orders of magnitude. Combined with the support for scalability that this approach brings, it would ap-

pear that all large-scale neural computation systems should use time multiplexed compute and communication resources rather than direct mapping of biological features to hardware.

This can be justified by comparing the scale and scalability of direct-mapped systems compared to systems with multiplexed resources. Bailey (2010) implements 100 neurons and 200 synaptic connections per FPGA using direct mapping into hardware and communication via FPGA routing resources, compared to 64k neurons and 64M synaptic connections per FPGA in the system implemented in this work. Another point of comparison is the system implemented by Cassidy *et al.* (2011), who also use time-multiplexed resources. They claim to perform computation of a network of up to 10^6 neurons on a single FPGA, far more than is likely to be possible with the system implemented in this work using currently available technology. However the fan-out of each neuron is very limited in this case (perhaps even to the point of each neuron having a fan-out of 1), and so the utility of this system is ultimately limited.

Given the scaling results in Chapter 8, the system in this work should be able to perform a computation of 10^6 neurons in real-time using 16 FPGAs. This would result in a network with the same number of neurons as used by Cassidy *et al.*, but at least $100\times$ or even $1000\times$ as much fan-out. Therefore it will ultimately outperform any system that has been designed to maximise the number of neurons per FPGA at the expense of fan-out in any application where biologically-plausible fan-out is required. Since increased fan-out leads to increased communication, could be considered another example of the limitations that occur when communication is not considered as a first-class design constraint.

9.3 Future work

There are two main targets for future work – increasing the scale of neural computation that can be performed by a massively parallel neural computation system and increasing their flexibility.

9.3.1 Increase in scale

Increasing a system's scale could be achieved by both using more FPGAs in a system and by changing the implementation platform to take advantage of new technology.

Using more FPGAs

The first target to increase the scale of a massively parallel neural computation system is to use more FPGAs. Our Bluehive system has 16 DE4 boards per rack box, so each rack box should allow neural computation of up to 10^6 neurons with a fan-out of 10^3 in real-time.

Change of implementation platform

Altera and Terasic have recently announced their DE5-NET FPGA evaluation board, which uses a Stratix V FPGA. This board has many features that are similar to the NetFPGA-10G board produced by the NetFPGA project, including 36 MB of QDRII SRAM and up to 8 GB of DDR3 SDRAM. While it lacks some of the peripherals of the DE4 board (such as an SD card slot), the SRAM would be sufficient to hold the parameters of up to 2×10^6 neurons. This would provide significantly more bandwidth for accessing neuron parameters.

This would leave the SDRAM dedicated to read-only accesses to fan-out, update and spike injector tuples, along with write accesses from the spike auditor. Combined with the significantly larger FPGA (950k logic elements compared to 228k on the DE4-230), this should allow for neural computations of at least 10^6 neurons with a fan-out of 10^3 in real-time on a single FPGA. This is equivalent to scale of neural computation that should be possible in real-time using 16 DE4 boards – the difference is a result of the higher off-chip memory bandwidth provided by the QDRII SRAM, once again illustrating that neural computation using today's electronics is primarily communication-bound.

Given the lack of peripherals on the DE5-NET board compared to the DE4, it may be appropriate to construct a multi-FPGA system using DE5-NET boards as the "core" and DE4 boards at the periphery to handle I/O and control functions. The high-speed transceivers on the DE4 and DE5-NET are expected to be compatible.

9.3.2 Increase in flexibility

The current system is a hand-optimised implementation of a massively parallel neural computation system for a specific neuron modelling equation. Allowing the neuron equation to be specified by a user would increase the number of potential users of the system. The NineML project (Raikov *et al.*, 2011) provides a framework to specify neuron modelling equations programatically (using a C-like syntax). By

analysing an equation using a simple parser it should be possible to derive an evaluation pipeline in Bluespec, and hence allow the system to use arbitrary neuron modelling equations.

A more flexible system could use a stream processor with a simple software program to evaluate neuron modelling equations while retaining the synaptic updates communication and application system. This would allow for computation of any combination of neuron modelling equation and neural network. Given that the performance exhibited by a GPU-based system using a single GPU is comparable to the FPGA-based system using a single FPGA, a flexible system for high-performance scientific computation of many kinds could be created by combining the compute architecture and high memory bandwidth of GPUs with the communication bandwidth of FPGAs.

Bibliography

ABBOTT, L. (1999). Lapicque's introduction of the integrate-and-fire model neuron (1907). *Brain Research Bulletin*, **50** (5-6), 303 – 304.

ABELES, M. (1982). *Local cortical circuits: an electrophysiological study*. Springer-Verlag.

ALVADO, L., TOMAS, J., SAÏGHI, S., RENAUD, S., BAL, T., DESTEXHE, A. and MASSON, G. L. (2004). Hardware computation of conductance-based neuron models. *Neurocomputing*, **58-60**, 109 – 115.

ANANTHANARAYANAN, R., ESSER, S. K., SIMON, H. D. and MODHA, D. S. (2009). The cat is out of the bag: cortical simulations with 10^9 neurons, 10^{13} synapses. In *High Performance Computing Networking, Storage and Analysis, Proceedings of 2009 Conference for*, pp. 63:1 – 63:12.

BAILEY, J. A. (2010). *Towards the neurocomputer: an investigation of VHDL neuron models*. Doctoral Thesis. University of Southampton, School of Electronics and Computer Science.

BASSETT, D. S., GREENFIELD, D. L., MEYER-LINDENBERG, A., WEINBERGER, D. R., MOORE, S. W. and BULLMORE, E. T. (2010). Efficient physical embedding of topologically complex information processing networks in brains and computer circuits. *PLoS Computational Biology*, **6** (4), Article Id: e1000748.

BASU, A., RAMAKRISHNAN, S. and HASLER, P. (2010). Neural dynamics in reconfigurable silicon. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pp. 1943 –1946.

BERGER, T. and LEVY, W. (2010). A mathematical theory of energy efficient neural computation and communication. *Information Theory, IEEE Transactions on*, **56** (2), 852 –874.

BOAHEN, K. (2000). Point-to-point connectivity between neuromorphic chips using

address events. *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, **47** (5), 416 – 434.

CASSIDY, A., ANDREOU, A. and GEORGIOU, J. (2011). Design of a one million neuron single FPGA neuromorphic system for real-time multimodal scene analysis. In *Information Sciences and Systems (CISS), 45th Annual Conference on*, pp. 1 – 6.

DALLY, W. J. and TOWLES, B. (2001). Route packets, not wires: on-chip inteconnection networks. In *Design Automation, Proceedings of the 38th Conference on*, pp. 684 – 689.

EMERY, R., YAKOVLEV, A. and CHESTER, G. (2009). Connection-centric network for spiking neural networks. In *Networks-on-Chip, Proceedings of the 3rd ACM/IEEE International Symposium on*, pp. 144 – 152.

FIDJELAND, A. and SHANAHAN, M. (2010). Accelerated simulation of spiking neural networks using GPUs. In *Neural Networks (IJCNN), The 2010 International Joint Conference on*, pp. 1 – 8.

GEWALTIG, M.-O. and DIESMANN, M. (2007). NEST (NEural Simulation Tool). *Scholarpedia*, **2** (4), 1430.

GOODMAN, D. F. M. and BRETTE, R. (2008). The Brian simulator. *Frontiers in Neuroscience*, **3** (26), 192 – 197.

HEBB, D. O. (1949). *The Organization of Behavior: A Neuropsychological Theory*. Wiley.

HINES, M. L. and CARNEVALE, N. T. (1997). The NEURON simulation environment. *Neural Computation*, **9** (6), 1179 – 1209.

HODGKIN, A. L. and HUXLEY, A. F. (1952). A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of Physiology*, **117** (4), 500 – 544.

INDIVERI, G., CHICCA, E. and DOUGLAS, R. (2006). A VLSI array of low-power spiking neurons and bistable synapses with spike-timing dependent plasticity. *Neural Networks, IEEE Transactions on*, **17** (1), 211 – 221.

IZHIKEVICH, E. (2003). Simple model of spiking neurons. *Neural Networks, IEEE Transactions on*, **14** (6), 1569 – 1572.

— (2004). Which model to use for cortical spiking neurons? *Neural Networks, IEEE Transactions on*, **15** (5), 1063 – 1070.

JIN, X., FURBER, S. and WOODS, J. (2008). Efficient modelling of spiking neural

networks on a scalable chip multiprocessor. In *Neural Networks, 2008 IEEE International Joint Conference on*, pp. 2812 – 2819.

MAGUIRE, L. P., MCGINNITY, T. M., GLACKIN, B., GHANI, A., BELATRECHE, A. and HARKIN, J. (2007). Challenges for large-scale implementations of spiking neural networks on FPGAs. *Neurocomputing*, **71** (1-3), 13 – 29.

MAHOWALD, M. and DOUGLAS, R. (1991). A silicon neuron. *Nature*, **354** (6354), 515 – 518.

MARKRAM, H. (2006). The Blue Brain project. *Nature Reviews Neuroscience*, **7** (2), 153 – 160.

MARTINEZ-ALVAREZ, J., TOLEDO-MOREO, F. and FERRANDEZ-VICENTE, J. (2007). Discrete-time cellular neural networks in FPGA. In *Field-Programmable Custom Computing Machines, 15th Annual IEEE Symposium on*, pp. 293 – 294.

MEAD, C. (1990). Neuromorphic electronic systems. *Proceedings of the IEEE*, **78** (10), 1629 – 1636.

MIGLIORE, M., CANNIA, C., LYTTON, W., MARKRAM, H. and HINES, M. (2006). Parallel network simulations with NEURON. *Journal of Computational Neuroscience*, **21**, 119 – 129.

MOORE, S. W., FOX, P. J., MARSH, S. J. T., MARKETTOS, A. T. and MUJUMDAR, A. (2012). Bluehive - a field-programable custom computing machine for extreme-scale real-time neural network simulation. In *Field-Programmable Custom Computing Machines, 20th Annual IEEE Symposium on*, pp. 133 – 140.

NIKHIL, R. S. and CZECK, K. R. (2010). *BSV by Example*. CreateSpace.

RACHMUTH, G., SHOUVAL, H. Z., BEAR, M. F. and POON, C.-S. (2011). A biophysically-based neuromorphic model of spike rate- and timing-dependent plasticity. *Proceedings of the National Academy of Sciences*, **108** (49), E1266 – E1274.

RAIKOV, I., CANNON, R., CLEWLEY, R., CORNELIS, H., DAVISON, A., DE SCHUTTER, E., DJURFELDT, M., GLEESON, P., GORCHETCHNIKOV, A., PLESSER, H., HILL, S., HINES, M., KRIENER, B., LE FRANC, Y., LO, C.-C., MORRISON, A., MULLER, E., RAY, S., SCHWABE, L. and SZATMARY, B. (2011). NineML: the network interchange for neuroscience modeling language. *BMC Neuroscience*, **12** (Supplement 1), P330.

RALL, W. (1959). Branching dendritic trees and motoneuron membrane resistivity. *Experimental Neurology*, **1** (5), 491 – 527.

RICE, K., BHUIYAN, M., TAHA, T., VUTSINAS, C. and SMITH, M. (2009). FPGA implementation of Izhikevich spiking neural networks for character recognition.

In *Reconfigurable Computing and FPGAs, 2009 International Conference on*, pp. 451 – 456.

ROWLEY, H., BALUJA, S. and KANADE, T. (1998). Neural network-based face detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, **20** (1), 23 – 38.

SAIGHI, S., TOMAS, J., BORNAT, Y. and RENAUD, S. (2005). A conductance-based silicon neuron with dynamically tunable model parameters. In *Neural Engineering, 2nd International IEEE EMBS Conference on*, pp. 285 – 288.

SCHMIDT, A., KRITIKOS, W., DATTA, S. and SASS, R. (2008). Reconfigurable computing cluster project: Phase I Brief. In *Field-Programmable Custom Computing Machines, 16th Annual IEEE Symposium on*, pp. 300 – 301.

SONG, S., MILLER, K. D. and ABBOTT, L. F. (2000). Competitive Hebbian learning through spike-timing-dependent synaptic plasticity. *Nature Neuroscience*, **3** (9), 919 – 926.

STEIN, R. B. (1965). A theoretical analysis of neuronal variability. *Biophysical Journal*, **5** (2), 173 – 194.

THOMAS, D. and LUK, W. (2009). FPGA accelerated simulation of biologically plausible spiking neural networks. In *Field-Programmable Custom Computing Machines, 17th Annual IEEE Symposium on*, pp. 45 – 52.

TOUMAZOU, C., GEORGIOU, J. and DRAKAKIS, E. (1998). Current-mode analogue circuit representation of Hodgkin and Huxley neuron equations. *Electronics Letters*, **34** (14), 1376 – 1377.

TRAUB, R. D., CONTRERAS, D., CUNNINGHAM, M. O., MURRAY, H., LEBEAU, F. E. N., ROOPUN, A., BIBBIG, A., WILENT, W. B., HIGLEY, M. J. and WHITTINGTON, M. A. (2005). Single-column thalamocortical network model exhibiting gamma oscillations, sleep spindles, and epileptogenic bursts. *Journal of Neurophysiology*, **93** (4), 2194 – 2232.

UPEGUI, A., PEÒA-REYES, C. A. and SANCHEZ, E. (2005). An FPGA platform for on-line topology exploration of spiking neural networks. *Microprocessors and Microsystems*, **29** (5), 211 – 223.

YANG, S., WU, Q. and LI, R. (2011). A case for spiking neural network simulation based on configurable multiple-FPGA systems. *Cognitive Neurodynamics*, **5**, 301 – 309.