8-2017

Scalable High-Speed Communications for Neuromorphic Systems

Aaron Reed Young
University of Tennessee, Knoxville, ayoung48@vols.utk.edu

To the Graduate Council:

I am submitting herewith a thesis written by Aaron Reed Young entitled "Scalable High-Speed Communications for Neuromorphic Systems." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Engineering.

Mark E. Dean, Major Professor

We have read this thesis and recommend its acceptance:

James S. Plank, Garrett S. Rose

Accepted for the Council:
Dixie L. Thompson

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

Scalable High-Speed Communications for Neuromorphic Systems

A Thesis Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Aaron Reed Young

August 2017

Abstract

Field-programmable gate arrays (FPGA), application-specific integrated circuits (ASIC), and other chip/multi-chip level implementations can be used to implement Dynamic Adaptive Neural Network Arrays (DANNA). In some applications, DANNA interfaces with a traditional computing system to provide neural network configuration information, provide network input, process network outputs, and monitor the state of the network. The present host-to-DANNA network communication setup uses a Cypress USB 3.0 peripheral controller (FX3) to enable host-to-array communication over USB 3.0. This communications setup has to run commands in batches and does not have enough bandwidth to meet the maximum throughput requirements of the DANNA device, resulting in output packet loss. Also, the FX3 is unable to scale to support larger single-chip or multi-chip configurations. To alleviate communication limitations and to expand scalability, a new communications solution is presented which takes advantage of the GTX/GTH high-speed serial transceivers found on Xilinx FPGAs. A Xilinx VC707 evaluation kit is used to prototype the new communications board. The high-speed transceivers are used to communicate to the host computer via PCIe and to communicate to the DANNA arrays with the link layer protocol Aurora. The new communications board is able to outperform the FX3, reducing the latency in the communication and increasing the throughput of data. This new communications setup will be used to further DANNA research by allowing the DANNA arrays to scale to larger sizes and for multiple DANNA arrays to be connected to a single communication board.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Neuromorphic computing is becoming increasingly common. As the limits of conventional computation are reached, new architectures that break away from the traditional von Neumann architecture will need to be researched, developed, and deployed. One promising class of post von Neumann architectures are the brain-inspired, neuromorphic architectures. Spiking neural networks, one type of neuromorphic architecture, are event-based networks with an inherent notion of time [26]. The neurons in this network create a fire event when their charge exceeds a threshold. This fire event results in a spike with a weight value. The connected neurons receive the spike and increase their charge by the weight. The input, output, and internal communication is all done via spikes. In order to develop these neuromorphic devices, traditional von Neumann based computers must be able to communicate with the new neuromorphic devices over a fast and flexible communication channel. This communication setup should be fast enough to allow the von Neumann hardware to run alongside a neuromorphic processor in real-time, thereby allowing real-time applications to be run on the system. The communication setup also needs to allow for larger array sizes and multiple chips, enabling the setup of larger neuromorphic arrays. Creation of a communication setup that meets the growing needs of larger neuromorphic hardware is a challenge. After a comparison of various setups, the implementations and design choices of one possible communication setup are discussed in detail and evaluated for maximum scaling potential.

Chapter 2

Related Work

Many research groups are working on neuromorphic hardware. Some are modeling the neuromorphic components in processors; others are creating custom circuits to perform the computations. No matter how the computational elements are designed, developing ways to connect neuromorphic elements to each other and to off-chip devices is essential. The human brain has billions of neurons, each with thousands of connections, so as researchers work to scale up their designs, they will be faced with additional communication challenges. Although the challenges and goals are similar, the solutions that the various research groups have chosen are diverse.

One challenge developers face is turning fire events into network packets. A solution to this is through Address-Event Representation (AER), whereby unique addresses are sent over a bus to represent the source or destination of a fire pulse [6]. AER is used by many neuromorphic projects to communicate spikes, and because of this various research groups have implemented high-speed serial AER using very-large-scale integration (VLSI) and FPGA boards [14, 5]. The AER boards are typically connected to a traditional computer using PCIe or USB.

Researchers at Stanford University have created a mixed-analog-digital system called Neurogrid [28]. The fundamental neuromorphic component of the Neurogrid is the silicon neuron. These neurons are placed in a 256×256 silicon-neuron array inside a Neurocore. Neurogrid totals a million neurons by using 16 Neurocores [7, 16]. A deadlock-free multicast tree packet router is used to transmit data between the 16 Neurocores. Each Neurocore

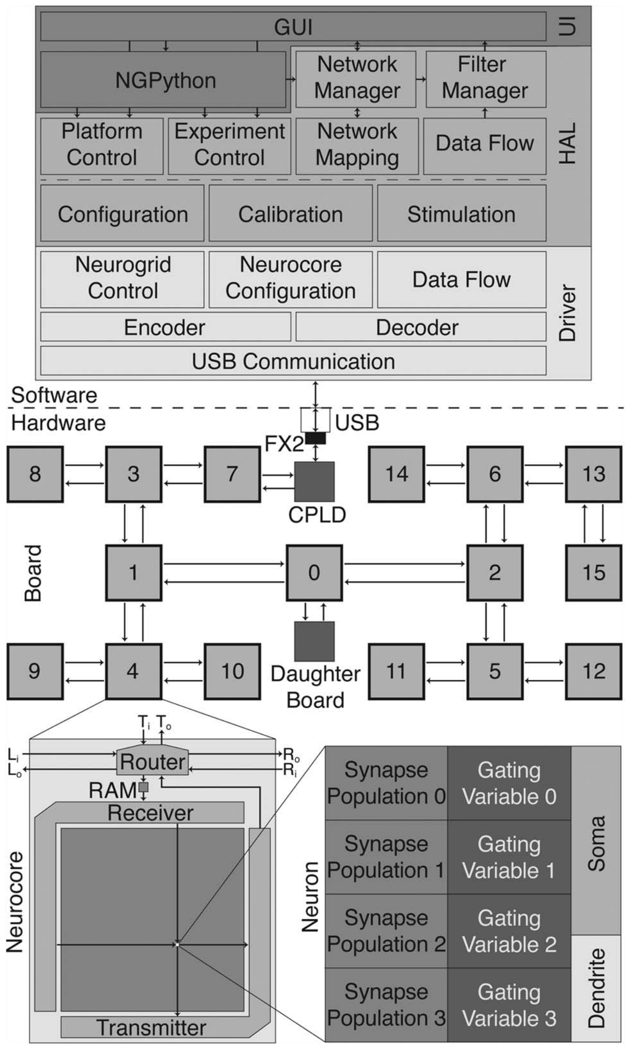


Figure 2.1: Neurogrid communication setup [4].

has its own full-custom asynchronous VLSI implementation of this router. This network router is well-suited for neuromorphic applications and is able to deliver 1.17 Gwords/s across the sixteen-chip network with only 1 µs of jitter. Neurogrid communicates off-board to a computer running the software stack via USB 2.0 with the use of a Cypress EZ-USB FX2LP (FX2) [4]. Figure 2.1 shows the complete communication setup of the Neurogrid. Neurogrid packets are sent from the software stack to a Lattice ispMACH CPLD (CPLD) via an FX2. The CPLD enables the FX2 to communicate with the Neurocores.

Manchester University, in the United Kingdom, is working on a large neuromorphic system called SpiNNaker [27]. SpiNNaker is a digital computer architecture designed to simulate many neurons and synapses in real-time. It accomplishes this by using 2^{16} chip multiprocessors

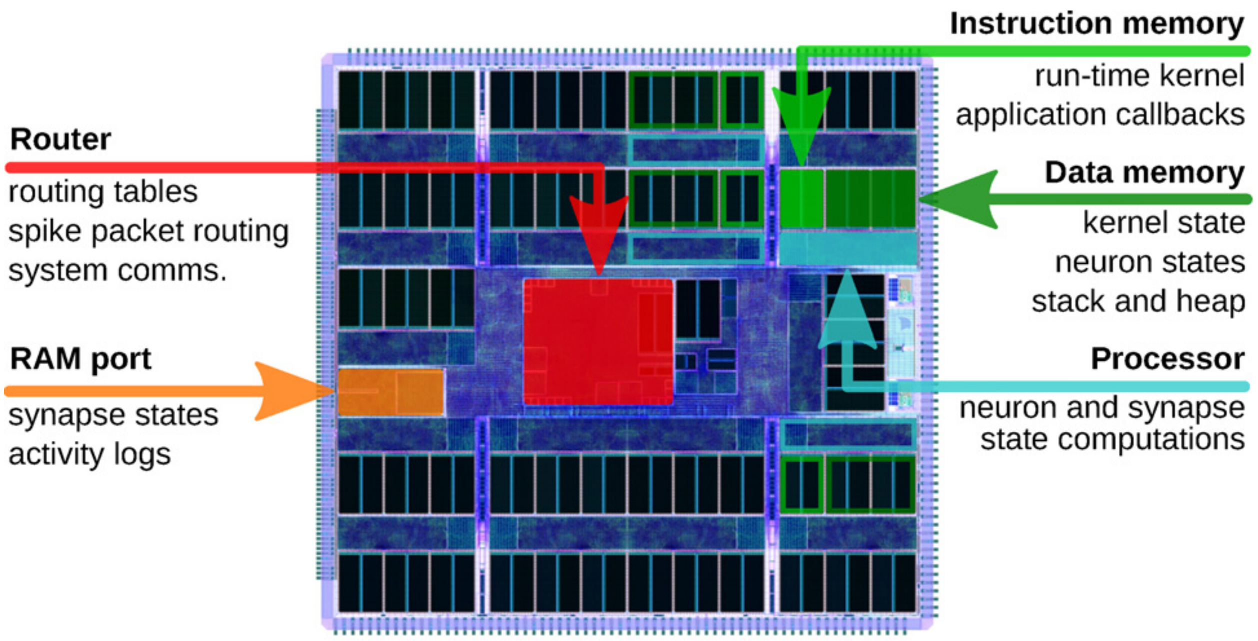
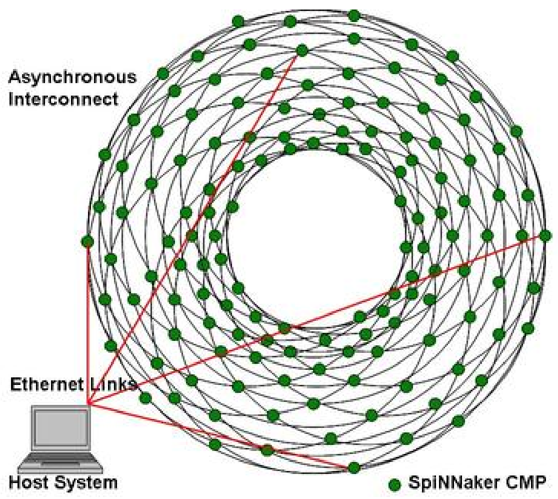
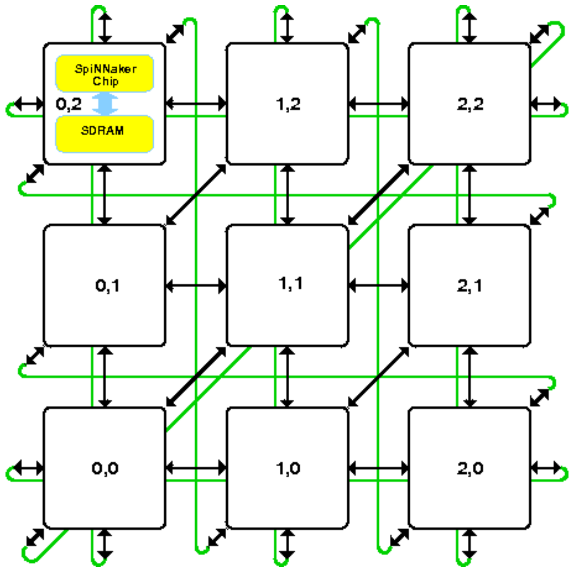


Figure 2.2: A SpiNNaker chip containing eighteen processors and specialised communication hardware [27].

(CMPs), each with eighteen homogeneous ARM968 processors. One processor is used for administration, sixteen are used for simulation. The last processor is a backup in case there is a fault with the other processors. Each processor has its own communications controller which communicates with an on-chip router to send neural spike signals. Figure 2.2 shows a diagram of one of the SpiNNaker chips. Each SpiNNaker chip contains a packet-switched router to form links between all of the eighteen on-chip processor cores and to communicate with the routers of the six neighboring chips. A 32 bit key is used to uniquely identify each neuron. Routing tables are used by the on-chip routers to forward messages to the correct neighbor so that the packets can reach the correct neuron. Just like other neuromorphic machines, SpiNNaker is controlled by a conventional von Neumann computer. This computer is used to specify the neuromorphic model, trigger the simulations, and retrieve the results. Ethernet is used to connect the host machine to one or more SpiNNaker CMPs [15]. Each of the CMPs is connected in a two-dimensional toroidal mesh. Figure 2.3a shows a high-level view of the system links and connections to the host. Figure 2.3b shows a more detailed look at the two-dimensional toroidal mesh connections the CMPs make with their nearest 6 neighbors.



(a) System connections with host.



(b) CMP connection details.

Figure 2.3: SpiNNaker connectivity [15].

The Human Brain Project has developed waferscale neuromorphic hardware through the FACETS and BrainScaleS projects [20]. Waferscale integration technology made it possible to create a large-scale, high-density neuromorphic system with 40 million synapses and up to 180 thousand neurons. Figure 2.4 shows a diagram of this system. A special communication infrastructure had to be developed to support communication needs of the waferscale neuromorphic system [22, 21]. The communication infrastructure can be divided into two parts, on-wafer and intra-wafer. The on-wafer communication is transferred by a high-density routing grid made up of post-processed metal interlinks added to the top of the wafer. Intra-wafer communication is handled by a packet-based network which connects the wafer to surrounding wafers and host PCs. The main component of the packet-based network is an application-specific integrated circuit called a digital network chip (DNC). The DNC employs synchronous high-speed serial packet communication to transmit time-stamped spike events. The packet network is setup hierarchically with eight high input count analog neural networks (HICANNs) connected to one DNC. Four DNCs are connected to a custom FPGA-AER board. This board is then connected to other FPGA-AER boards and also to the host PC. A picture of this communication tree is shown in Figure 2.5.

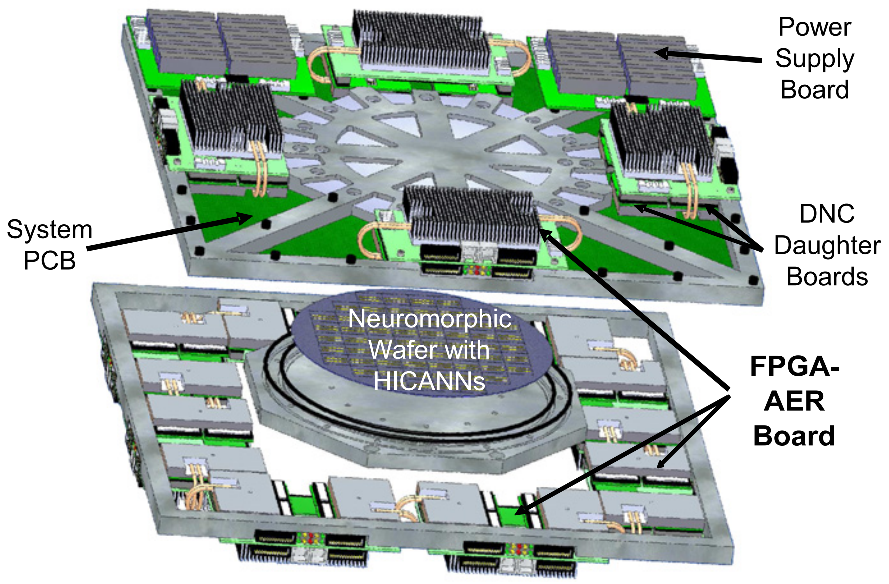


Figure 2.4: Overview of one wafer module of the FACETS/BrainScaleS waferscale system [21].

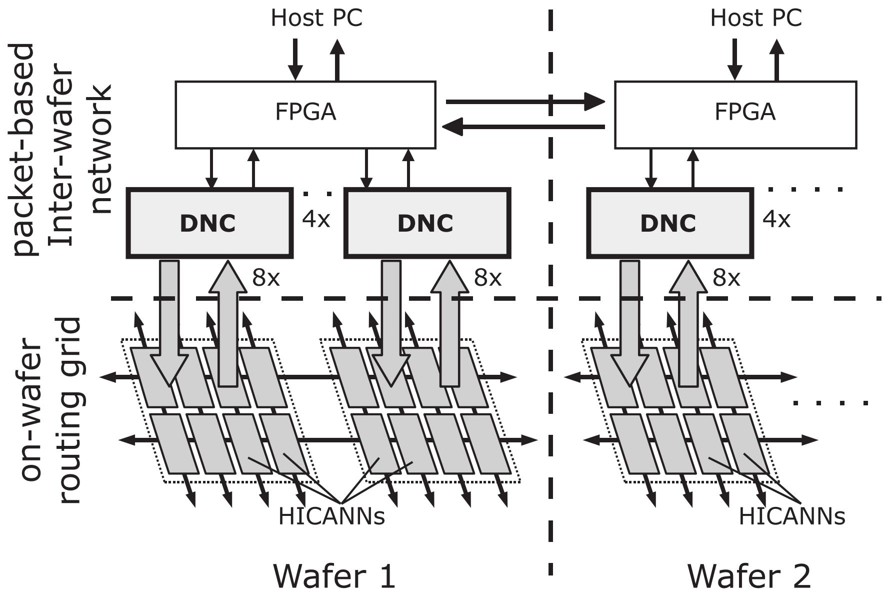


Figure 2.5: Logical structure of the off-wafer packet-based network and the on-wafer routing grid [22].

IBM has also developed a neuromorphic platform called TrueNorth as part of the DARPA SyNAPSE project [8, 19, 1]. TrueNorth is composed of a scalable network of neurosynaptic cores. Each core contains components that function as neurons, dendrites, synapses, and axons. The cores operate in parallel and are event driven, conserving power when there is no activity. Communication is handled via unidirectional messages containing spike information that are sent from a source neuron to a receiving axon. In 45 nm silicon, IBM's neurosynaptic cores contain 256 neurons and 64k synapses. These neurosynaptic cores can then be connected together through a communications network to form large neuromorphic arrays. In 2011, IBM used Samsung's 28 nm process to fabricate a chip with 4,096 neurosynaptic cores connected via an on-chip network [17]. The chip contained 5.4 billion transistors and had one million neurons and 256 million synapses. Figure 2.6 shows a diagram of the TrueNorth chip. The chip is built by interconnecting a network of lightweight neurosynaptic cores. Short range connections are implemented with a intra-core crossbar memory and long range connections are implemented through an inter-core spike-based message-passing network.

Each research group is using the human brain as inspiration to create new computer architectures that have the same desirable characteristics as the brain. The brain is a marvelous machine capable of highly parallel processing and high efficiency with low power usage. All of the research groups are also looking to use their neuromorphic elements to build up large scale neuromorphic arrays. A common theme is that the elements should be generic and flexible supporting many various configurations. Handling communications between all the elements when scaling up the networks presents a common challenge. There are similarities in this effort; the communication infrastructure has to support configuring, running, and monitoring the network as it operates. Sometimes this communication structure is also used to facilitate scaling to multiple chips. The details on how this gets accomplished vary. Neurogrid uses a multicast tree, TrueNorth a crossbar, SpiNNaker a toroidal network, and BrainScaleS a waferscale routing grid.

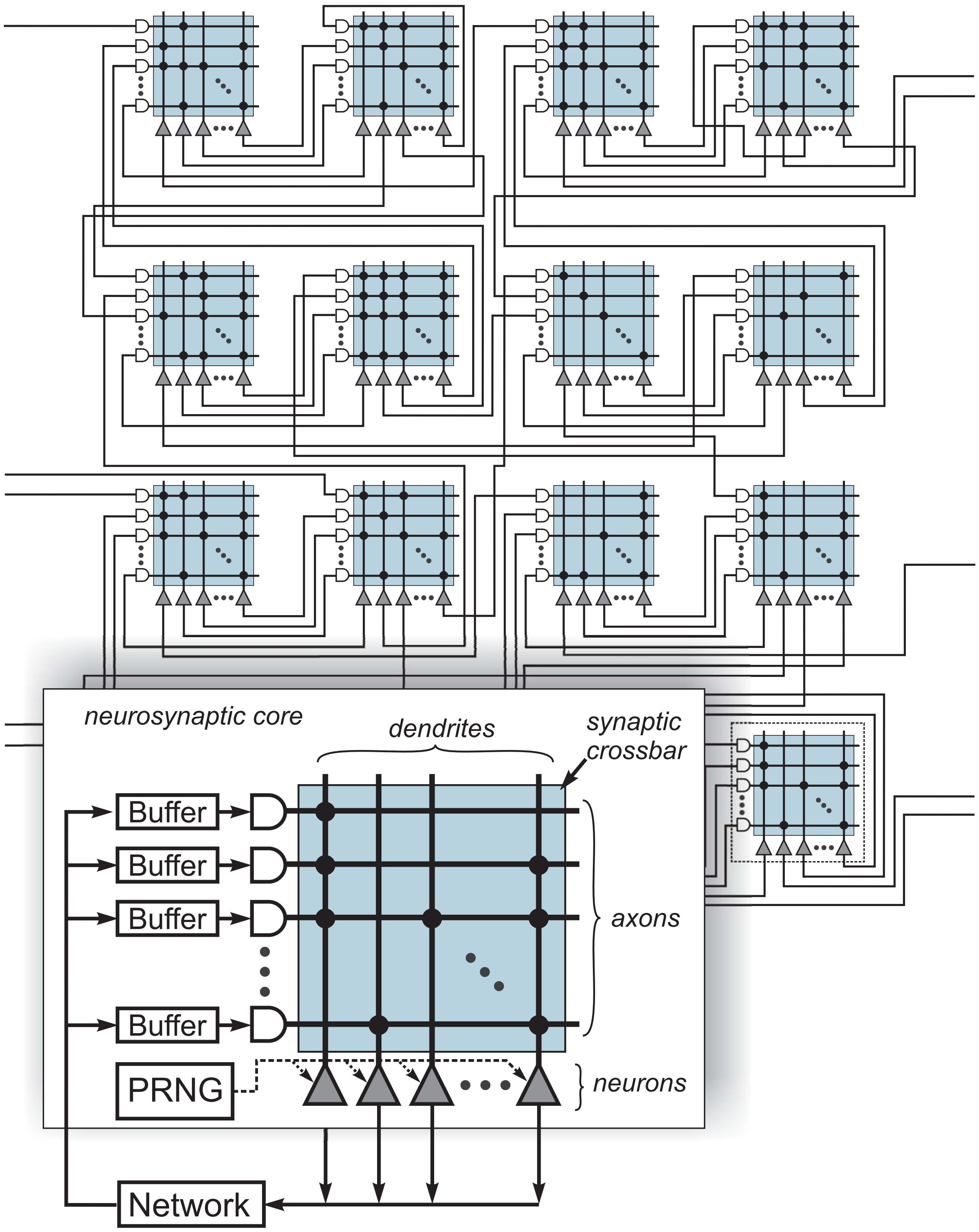


Figure 2.6: TrueNorth chip architecture diagram [8].

Chapter 3

Previous Work

The new scalable high-speed communication system, discussed in this thesis, builds upon work done previously in the neuromorphic computing research group at the University of Tennessee. The primary goal of the group to design computer systems using principles discovered from studying the brain, and to address the constraints found in traditional computing systems, including distance between processing and data components, communications bottlenecks, scalability, power efficiency, and other constraints.

Neuroscience-Inspired Dynamic Architecture (NIDA) was the first neural network computer architecture developed by the group [25, 24, 26]. NIDA's main features are its dynamic behavior and three-dimensional structure. The elements that make up NIDA are modeled after neurons in the brain. These elements are event-driven and use firing events, or spikes, to signal neighboring elements. The elements are scattered in three-dimensional space and connected together to form a large network. The communication between elements has a varying delay value which is inherited from the geometric distance between the elements. Therefore, spacial relationships and connectivity between the elements play a large role in the functionality of the network. NIDA is also used to explore the effects of particular biologically-inspired mechanisms to determine the role they have in training and generalization performance. Two such mechanisms are simple potentiation/depression weight-change mechanisms on the synapses and leak on the neurons which were explored by Schuman [23].

The other hardware modules from the group include the Dynamic Adaptive Neural Network Array (DANNA), which is two-dimensional representation of NIDA suited for implementation with digital circuits, and mrDANNA, which is a version of DANNA designed to use memristive devices [26, 9, 18].

All of these models are trained using evolutionary optimization (EO). Further information about EO can be found in "Neuroscience-Inspired Dynamic Architectures" [26]. EO is inspired by neuroevolution. It generates a population of random networks and then uses a fitness function to calculate a fitness value from each network. This fitness value represents the performance of the network and is calculated based on how close the network behaves to the desired behavior. The most fit networks, the networks with the highest fitness values, are selected to seed the next generation of networks. The offspring networks are generated from performing mutation and crossover from the previous generation. The mutation operation will randomly change some of the properties of the network and will generally be a small change. Crossover will take parts of two parents and combine both parts into a single child network. New generations of the population will continue to be generated as the algorithm runs and the fitness of the best network will continue to increase. The algorithm stops once a target fitness value has been reach, or a specified number of cycles has occurred.

3.1 DANNA Overview

Dynamic Adaptive Neural Network Array (DANNA) is a two-dimensional representation of NIDA well-suited for implementation with digital circuits [11]. The FPGA implementation of DANNA is written in VHDL and is designed for Xilinx FPGAs. DANNA uses a two-dimensional grid of elements that can be either configured as a neuron or a synapse. The array size is limited by the capacity of the chip or chips used for implementation. NIDA, on the other hand, can place an arbitrary number of elements anywhere in space. DANNA uses a configurable synaptic delay instead of using the geometric distance to calculate the delay. NIDA also uses floating point values whereas DANNA uses fixed point integers. DANNA includes a potentiation/depression mechanism to change the weight of synapses depending on whether the synapse causes the neuron to fire. In order to implement DANNA in hardware, the possible connections between the elements also has to be limited. Each element can

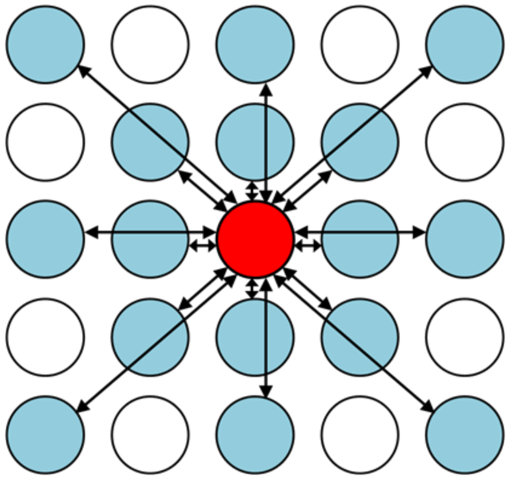


Figure 3.1: DANNA element connectivity [10].

connect to each element directly adjacent to it and to the next nearest element to that for a total of 16 possible connections. Figure 3.1 shows this connectivity.

DANNA elements are configured to form neuromorphic networks. The input to the network is applied to the first column, or left side, of the network. The output from the network comes from the last column, or right side.

3.1.1 Elements

Neurons can send and receive fire events to and from multiple synapses. Synapses receive fire events from one neuron and transmit fire events to one neuron. Multiple synapses can be chained together to connect distant neurons. Each neuron has a configurable threshold and a mechanism to choose which inputs, out of 16 possible inputs, to enable. The threshold specifies how much charge is needed to cause the neuron to fire. Each neuron can receive input from any number of its neighbors. Elements configured as synapses have a weight, refractory period, delay, and input enable. The weight specifies the value of the charge the synapse will send to the connected neuron. The refractory period specifies how many cycles have to take place before potentiation or depression can occur again. The delay specifies how long to hold the charge before it is applied to the output. This variable delay allows the fixed grid of elements to have a greater variety in the "distance" between the elements. Since the geometric distance between the elements is fixed, the delay field allows for a wide range of

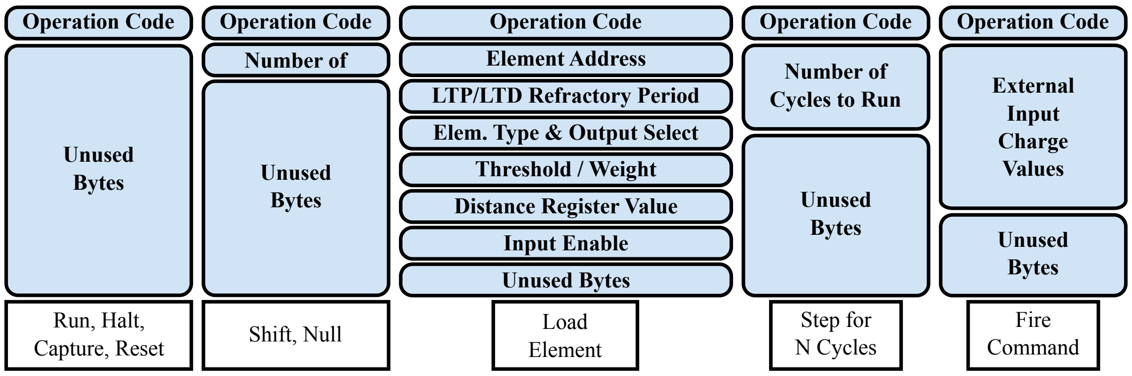


Figure 3.2: Input packet overview.

apparent distances to be configured. The input enable specifies which neighboring element it uses to know when to fire. Ideally, the network would consist of neurons which are connected to synapses which are connected to neurons and so forth. This setup makes sense with the biological model, since the neuron represents the dendrite inputs and cell body, which build up a charge until the cell creates a spiking fire event, represented in a digital circuit as a logic pulse. The synapse represents the delay in the propagation of the charge along the axon and the biological chemical synapse between cells. Synapse-to-synapse connections are used to create long "distances" between neurons or to support routing of connections in an array.

3.1.2 Communication Interface

The DANNA networks are configured, controlled, and monitored over a custom programming interface. The term communication interface is used to refer both to the structure of the packet sent between the host and the DANNA network and the hardware module responsible for parsing and generating the packets. Input packets are the packets that originate from the host computer and are used as input to the DANNA communication interface. Output packets are the packets generated from the communication interface and are sent back to the host computer. The commands provided by the communication interface can be broken down into three main functionalities of configuring, controlling, and monitoring the network. Figure 3.2 shows the different input commands and their structure. The input commands are further discussed in Subsection 3.1.3.

Configuring

Configuring the network is accomplished using the load command. It specifies an element at a particular row and column to be configured as a neuron or as a synapse. If the element is a neuron, the command also specifies the threshold and which of the 16 inputs are enabled. If the element is a synapse, the fire weight, delay, refractory, enabled input, and enabled output are all specified. In order to configure the entire array, each active element needs to be loaded. The command reset makes it easy to clear a loaded network by resetting the network back to the initial un-configured condition.

Control

Control commands provide instructions directed at running the neuromorphic network. They allow input fires to be sent to the network and provide control over the clocks that drive the execution of the network. The control commands are run, step, halt, fire, and null.

Run, step, and halt control the clock. Run starts the network and allows the network to run until the halt command is sent. The step command provides more control over how many cycles are executed by telling the network to run for a given number of cycles and then stop.

The fire and null commands are used to provide input to the array. The fire command specifies which input neurons to fire on and the weight for each fire. Since fire commands are applied during the same cycle the packet is received, the null commands are used to align fires to the desired cycle by representing cycles with no fire event. The null command takes an argument which specifies how many cycles occur without a fire event. Therefore, if you wanted to fire on cycle 10, a null of 9 would be used so that the fire command would be processed on the tenth cycle.

Monitoring

Two commands, capture and shift, work together to provide the ability to monitor the state of the neural network. The capture command will tell each element to record statistics about itself into its “shift register”. The shift command will in turn pull one bit from each shift register from elements at the top of the network. Internally, the data from all the shift

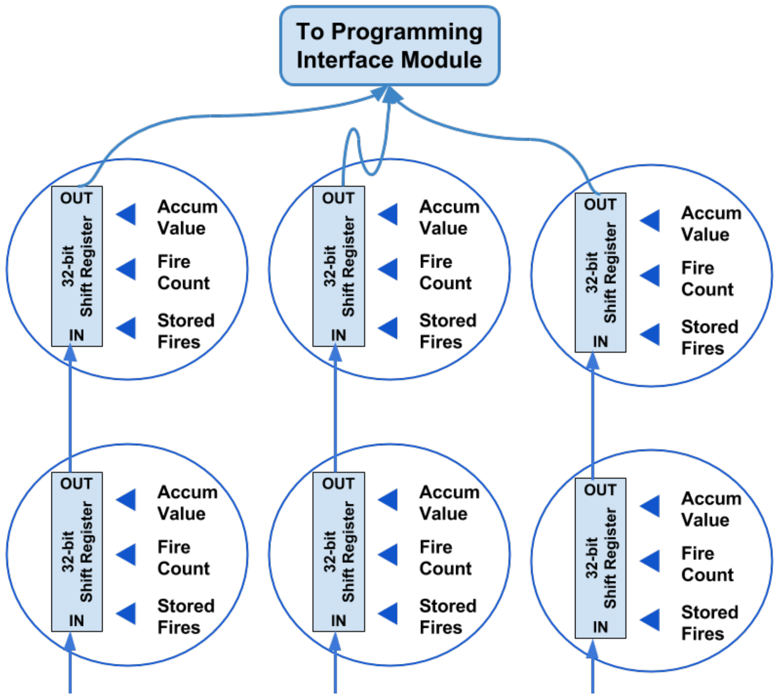


Figure 3.3: Monitoring capture/shift diagram [10].

registers is shifted up. In order to shift out all the data from each element, the number of shifts issued needs to be the size of the element's shift register times the number of rows in the network. Figure 3.3 illustrates the capture shift process. Each DANNA element stores the contents of the accumulator, the fire count since the last capture command, and the number of stored fires it holds (if a synapse), when it receives the capture command. This information provides insights into the state of the internal neural network and is used for EO, debugging, and visualization programs. The visualization programs generate graphics that show the activity in the network. The output of the visualization program shown in Figure 3.4 depicts a DANNA network as it is being run. Each frame of the visualization represents one run of the network. The color of the element represents the charge for neurons and weight for synapses. The border around the elements shows the fire count of the element for the run. A darker border represents a higher count.

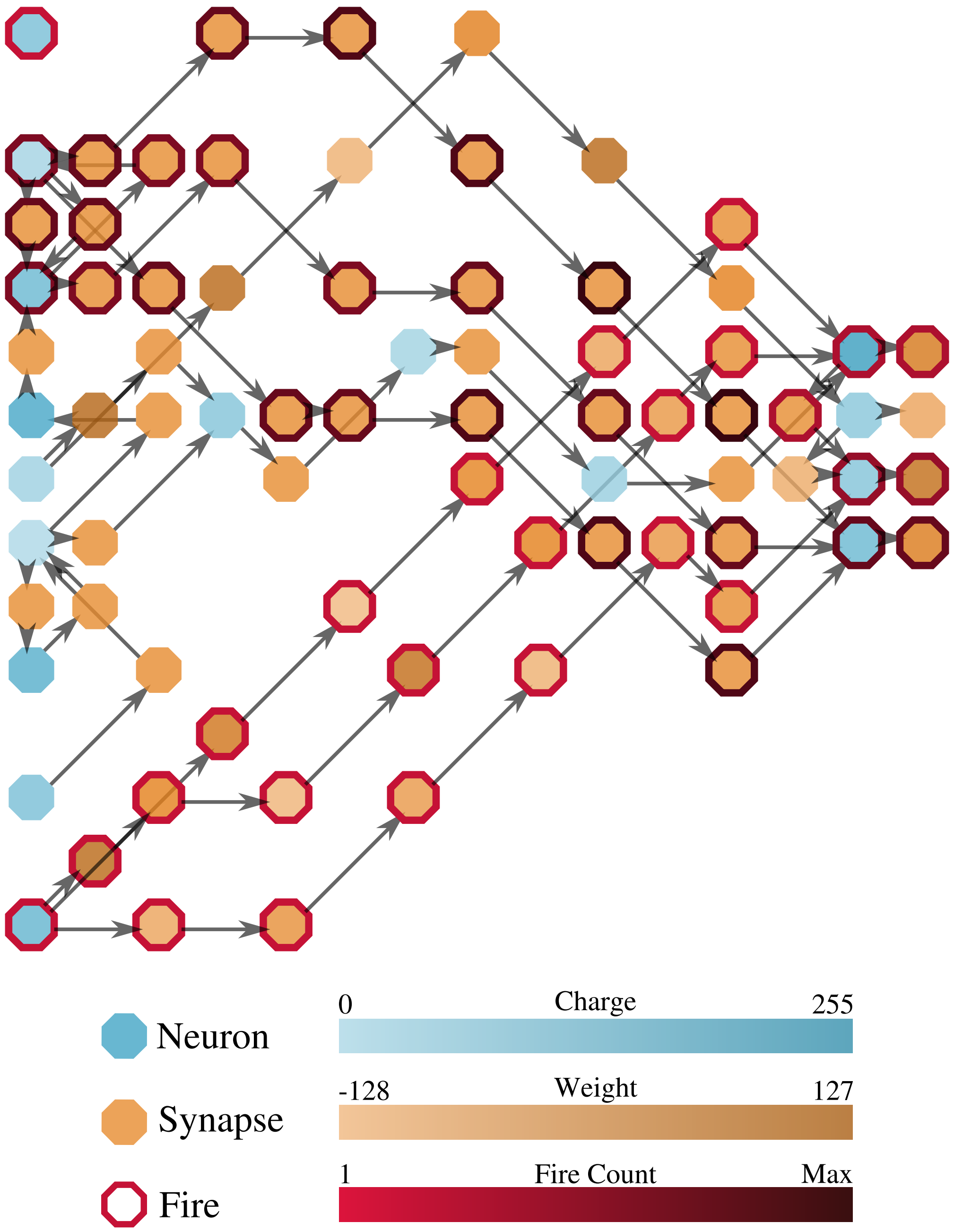


Figure 3.4: Image of the DANNA visualization tool.

3.1.3 DANNA Input Packet Structure

The communication interface defines the expected format of the input packets. Input packets have a fixed size of 36 bytes. The command type is given by an 8-bit opcode at the start of the input packet. Each command type has a different structure, designed to accommodate the parameters for the command. The different command types are null, load, halt, run, step, fire, reset, capture, and shift. The commands run, halt, capture, and reset do not have any parameters and therefore consist only of the operation code. Shift, null, and step all include an 8-byte field, which specifies how many times to apply the command. The fire command contains the input fire weight for each of the input neurons and is presently limited to 32 8-bit weights. The fire command has the most data and is the command that drives the packet size. The load command has parameters that specify the properties of the neuron or synapse. The same packet structure is used for both a neuron or a synapse. The fields take on a different meaning or are unused depending on the element type bit. Figure 3.2 summarizes the input packet structure.

3.1.4 DANNA Output Packet Structure

The communication interface creates the output packets. An output packet is only generated on a cycle containing a fire response, halt response, or shift operation. The output packet has a fixed size of 64 bytes and all of the values are in a fixed location in the packet. The output packet contains the current time stamp of the DANNA network. This time stamp gets set to 0 on a reset and counts the number of cycles that have passed. Next in the packet are the output weights for all of the output elements. Then there are 16 bytes of shift data, with one bit from each column of elements. Since this field is 16 bytes wide, the maximum number of columns that can be supported is 128. Next are the status flags. These flags specify if the packet is a halt packet or if the packet contains shift data. Finally, there is a configuration ID which is an ID unique to the network that can be verified. Figure 3.5 summarizes the output packet structure in a figure.

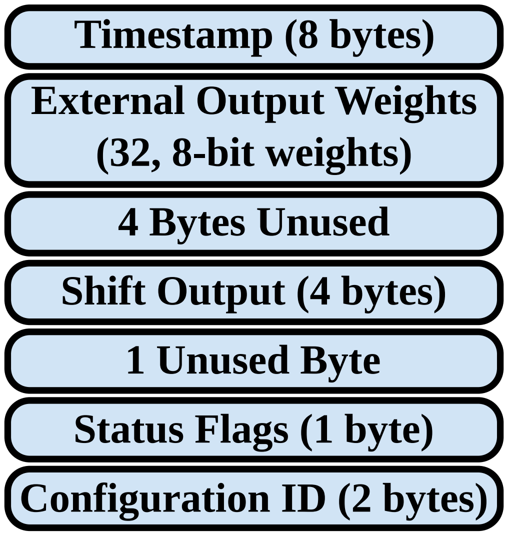


Figure 3.5: Output packet overview.

3.2 Single DANNA FPGA with PCIe

The first iteration of the DANNA FPGA implementation was on a single FPGA and communicated with the host machine via PCIe [11, 31]. This interface used an older, but similar, packet structure to the one covered in subsections 3.1.3 and 3.1.4 and was used for external configuration, to control the network, and to monitor the characteristics of the network. One major difference with this older implementation was that each element could only be connected to its nearest 8 neighbors.

This first iteration revealed some of the problems with using PCIe on a single FPGA. For example, the PCIe implementation used a significant part of the FPGA's resources. This resource usage can be reduced by only using one lane of the PCIe bus (PCIe 1x).

3.3 DANNA Kit with FX3

Using a PCIe connection to connect DANNA to a host machine has some downsides. PCIe is a complex protocol that includes many more features than are necessary to communicate with the communication interface. All of the complexity of the PCIe interface takes up valuable space on the FPGA that could be used for more array elements. Additionally, using PCIe limits the FPGA boards to those designed to be mounted inside a computer.

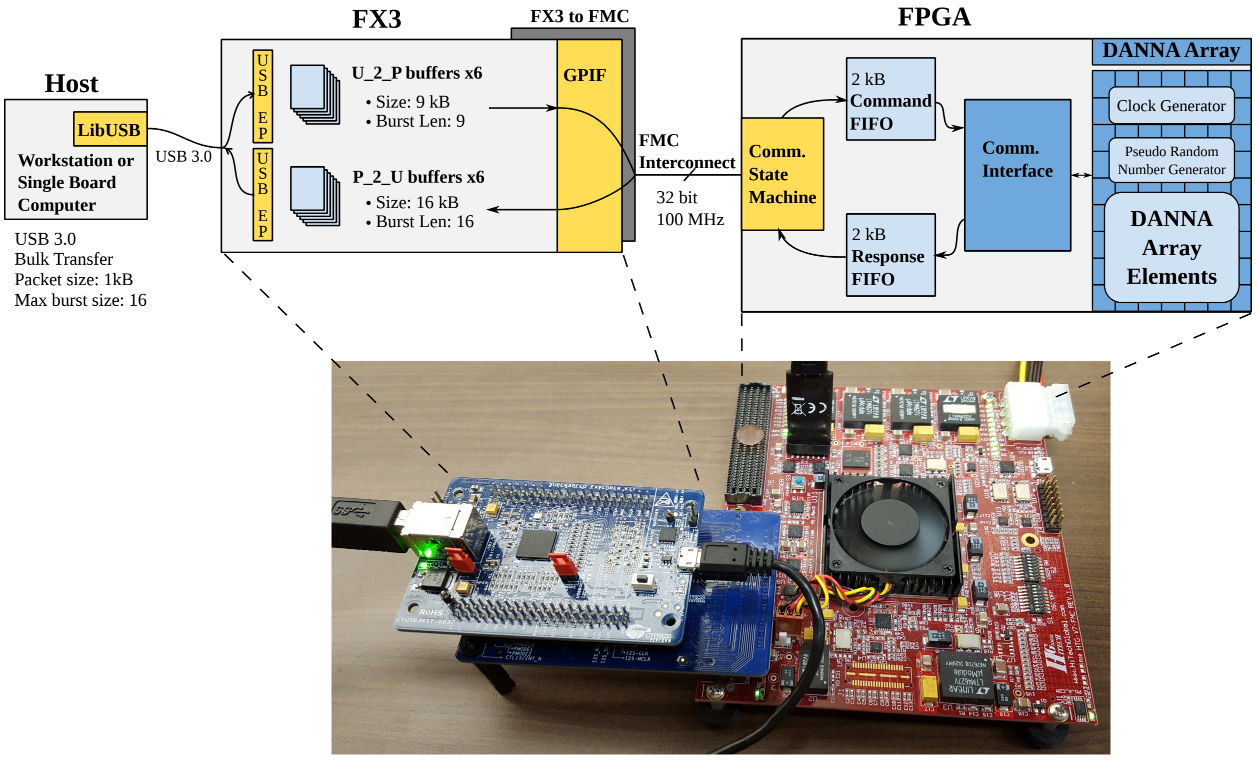


Figure 3.6: Diagram of the FX3 communication setup.

Because of all these limitations, DANNA was redesigned to communicate over USB to the host computer with the help of a Cypress USB 3.0 peripheral controller (hereafter referred to as an FX3) [31, 10]. Initially, the setup only worked with USB 2.0, but it was later expanded to work with both USB 2.0 or USB 3.0. Figure 3.6 shows a diagram of the communication setup using an FX3. The host machine is any Linux machine. The current tested operating systems are Redhat and Ubuntu. In theory any Linux OS will work as long as it has a C++ compiler and the LibUSB driver. The host machine communicates with the FX3 using a LibUSB driver and USB 2.0 or 3.0. The FX3 has a direct memory access (DMA) controller which moves the data from the USB endpoints to on-board buffers. These buffers are then accessed by the General Programming Interface (GPIF) state machine to send data to the FPGA. The GPIF pins are mapped to pins on an FPGA Mezzanine Card (FMC) connector with an FX3 to FMC interconnect board. In total 47 signals are used for the communication [29]. The data bus accounts for 32 of the signals. There is also a clock signal, reset signal, 5 control signals, 2 address signals, and 6 buffer status flags. Figure 3.7 shows a diagram of this connection interface. A state machine on the FPGA is the interface master and controls

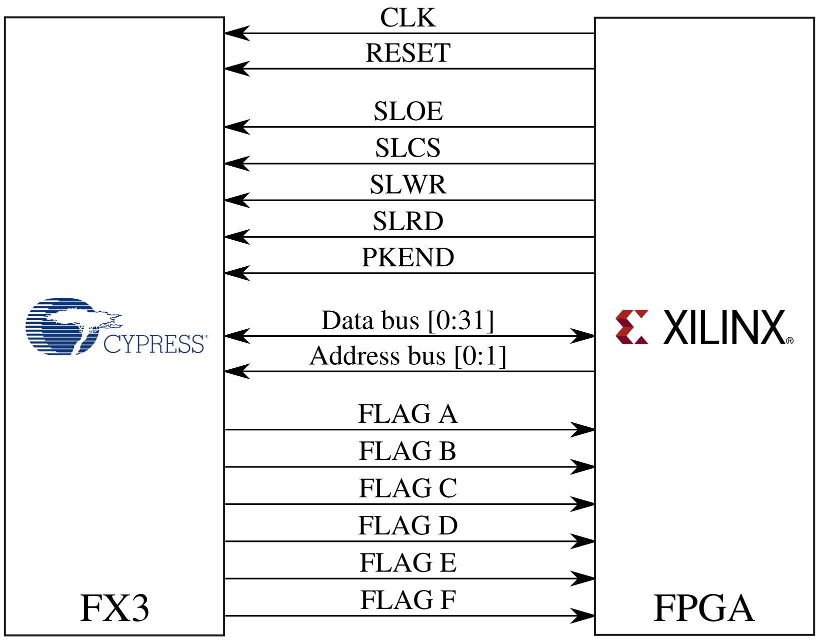


Figure 3.7: Diagram of the FX3 communication interface.

the transfer of data. The data connection is 32 bits wide and operates at 100 MHz. The FPGA state machine then sends the packets to on-chip buffers which are used to feed the communication interface. The communication interface then runs the DANNA Network.

Around this same time an application development platform (ADP) or kit was developed to make running neuromorphic networks on hardware much easier [12]. The hardware kit along with the corresponding Software Development Kit (SDK) allows other researchers to use the DANNA infrastructure and hardware to build DANNA-based neuromorphic computing systems. The kit is a self-contained system, in a 10in x 10in x 4in 3D printed enclosure. A picture of this kit can be found in Figure 3.8. The kit includes an ARM Wandboard which is the host, an FX3 which provides a connection from the host to the DANNA array, and a FPGA on which the DANNA array runs. With this kit the DANNA hardware can be accessed by using the Wandboard directly, or remotely by connecting to the Wandboard through another machine. The Wandboard only has a USB 2.0 controller and is thus limited in the speed in which it can interface with the DANNA Array.

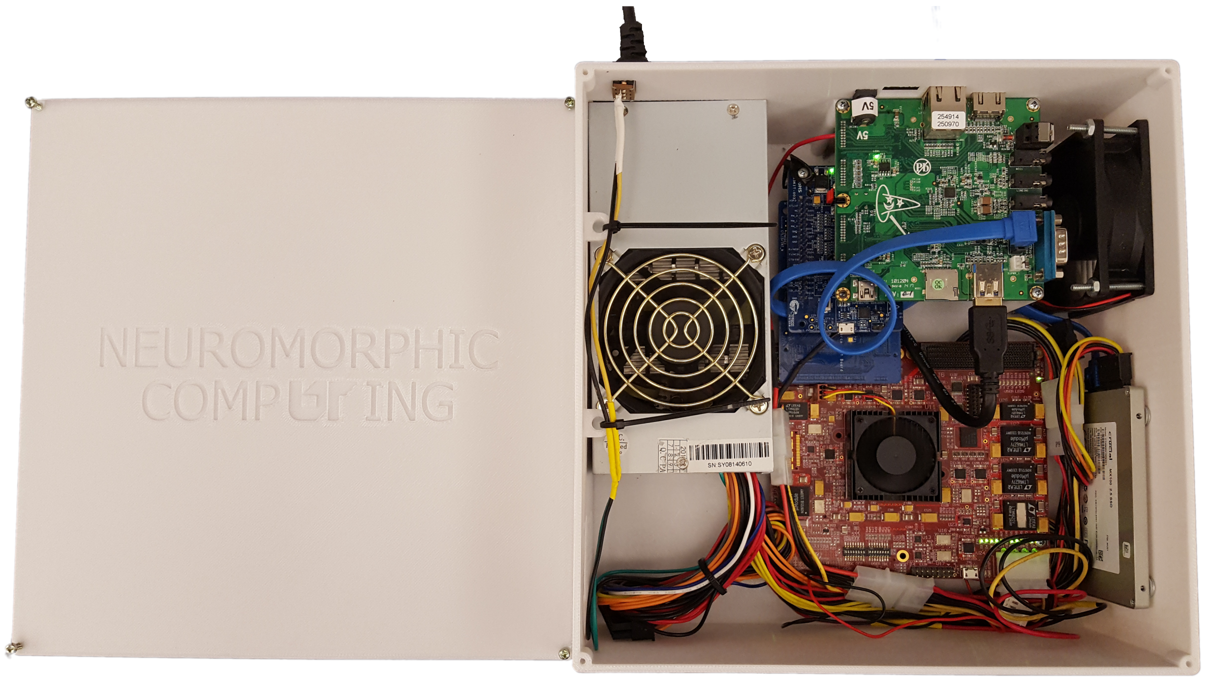


Figure 3.8: Picture of the DANNA Kit.

Chapter 4

New Improvements

The new communications setup is designed to provide many new improvements over the previous communication setups.

4.1 Problems with the Previous Communication Implementations

The prior DANNA communication setups utilized PCIe with a single board and USB via the FX3 board. They both have been well tested and are valuable methods for connecting with the custom hardware. Just as the USB communication setup replaced single board PCIe to provide an easy lightweight kit and reduce the FPGA logic dedicated to the PCIe interface, the USB communication setup will be replaced to allow for greater scaling and flexibility with high-performance, larger and/or multi-chip DANNA neural network implementations.

4.1.1 PCIe Single Board Limitation

The initial PCIe connection with a single FPGA was implemented using a Xillybus module on the FPGA to handle the complexities of the PCIe interface [11]. This initial design sent one packet at a time and waited for the response before sending the next packet. Even with this lockstep transmission, the PCIe connection had sufficient performance to support the communication needs. The limitations of this design were inflexibility as to which FPGAs could be used and the large resource utilization of the Xillybus core. With this setup only FPGAs that had a PCIe edge connector and were designed to fit in a computer chassis could be used. This greatly limited the selection of potential FPGA boards available for use. The other main drawback was the large portion of the FPGA resources that were taken up the by Xillybus core. This prevented the resources from being used for DANNA elements, reducing the maximum array sizes that could be built.

4.1.2 USB via FX3 Limitations

Switching to using USB to communicate with the FX3 board as an interconnect greatly opened up the connection flexibility. This approach allows any DANNA hardware to connect to any host computer with a USB connection, greatly expanding host machine options. Now mobile computers like laptops and lightweight ARM boards can be used, like the Wandboard [30]. The FPGA board selection also increased; now any board with an FMC connection can be easily used.

Although the FX3 works for simple tests, it is not without issues. The way the DANNA array and communication are designed for USB, packets are dropped if the communication framework falls behind and the communication buffers fill up. The communication can fall behind on any size of network, since the packets are a fixed size and all the events for a cycle are included in the packet. The faster the network runs, the fewer network cycles are completed before the communication buffers are exhausted. The packet drops can be observed on any size neural network, running at a global clock of 1 MHz for $\sim 30,000$ or more cycles, with input and output fire events occurring every cycle. LibUSB proves to be very resource-intensive, which causes the bandwidth to vary depending on the power of the host. This means that lightweight hosts can not be used to run communication intensive networks. To try and fix this bandwidth issue, the design was changed from using USB 2.0 to using USB 3.0. With any USB connection there is a direct trade-off between latency and bandwidth. The USB transfer type of bulk transfers, with deep buffers and maximum packet bursting, was selected to maximize communication bandwidth of DANNA. This switch to USB 3.0 helps reduce packet drops, but also forced running commands a batch at a time. Even with this switch, the performance of the communication is greatly effected by the power of and the current load on the machine. Additionally, the communication network is unable to keep up when there is a fire command and an output command every network cycle for a long running network.

Although the GPIF interface uses far fewer resources, the state machines on both the FX3 and the FPGA are complex and are inflexible to change. This prevents making changes to the packet length without major changes to the FX3 and FPGA implementation, since the state machines and buffers are designed to transfer and hold complete packets. The maximum size of the DMA buffers also is limited on the FX3. The current FX3 design maximizes buffer utilization on the FX3 with six 9 KB command buffers and six 16 KB response buffers. Another limitation is the 40 µs processing time the FX3 firmware takes to swap DMA buffers [13].

4.2 Goals for a New Communication Implementation

The new communications setup should both remove the limitations of the previous implementations and build upon their strengths. The communications setup should have a small footprint on the DANNA FPGA to allow the majority of the resources to be spent on DANNA elements.

It should also have sufficient resources to support real-time and cycle-accurate operation. This means that there should be low enough latency and high enough bandwidth to allow a response fire from the network to drive inputs to the network within a few cycles of the network. Low latency and high bandwidth additionally allows real-time processing to occur, enabling real world inputs and outputs that can drive actuators in the real world.

The communications setup should also enable effective monitoring of the neuromorphic array. High bandwidth is needed to allow the monitoring information to be sent back to the host along with the fire information. With higher bandwidth, more detailed monitoring data can be collected.

Flexible, high-capacity packet management should also be supported. Such support would allow for easy changing of packet structure to support new features. Previously, the packet size was fixed and making a change to it would require reworking the GPIF interface.

Furthermore, the FX3 forces constraints in the buffers and sizes used. A more flexible, high-capacity packet management scheme would allow for such features as easy packet size changes and for variable length packets. A high-capacity transfer size needs to be maintained so that the high bandwidth can still be achieved.

The communications board should also support flexible physical connectivity so that multiple DANNA arrays can be connected at once and different types of FPGA boards can be connected. It should also be fairly flexible with the host connection so that there are not too many constraints on the PCs that can be used.

Finally, the communications board should scale in both performance and bandwidth, enabling it to support DANNA networks as they scale to higher clock speeds and larger arrays.

The main objective of the new communications board is to support future research in neuromorphic computing. With the current FPGA chips, the DANNA implementation is limited to an approximately five thousand element array operating at 1 MHz. New ASIC and VLSI implementations of DANNA will have 100 million element arrays operating at 10 MHz. Additionally, multiple chips will be connected together to create multi-chip implementations of DANNA arrays. The communication board will have to be able to facilitate host-to-chip communications for all of the devices.

Chapter 5

Comparison of Solutions

There are many different ways to set up a communication system, each with its own strengths and weaknesses. Many different configurations were considered when deciding how to implement the new communication system.

Previously, the FX3 used a parallel bus that was 32 bits wide and ran at a frequency of 100 MHz to communicate to the FPGA. This communication channel was shared in both directions and could in theory have a maximum bandwidth of 400 MB/s in one direction. Thus, the maximum expected transmission rate is 200 MB/s in each direction. In practice, however, cycles are spent when the state machines change directions of data flow, which results in a smaller actual bandwidth of 100 MB/s. The USB 3.0 side could in theory transfer data faster than the GPIF interface, and in practice the GPIF interface seems to be the bottleneck. USB 3.0 has a maximum theoretical bandwidth of 5 gbps which translates to 640 MB/s. However, this connection is much slower with small transfer sizes because of the overhead in the USB protocol, the limited power of the FX3 board, and the overhead in the LibUSB library. Further details on the measured FX3 performance can be found in Chapter 11.

5.1 Synchronous or Asynchronous Bus

The first main decision to make is whether to use a synchronous or asynchronous bus. Synchronous buses are simpler to implement, but could require a much larger number of pins to obtain high throughput. Throughput and latency calculations are easy with a synchronous bus. Theoretical maximum throughput can be calculated as shown in (5.1). The equation for the minimal latency is shown in (5.2). Note that the equations do not include software, protocol, or packet processing overheads.

$$\text{Max Throughput} = \text{Bus Width} \times \text{Bus Frequency} \tag{5.1}$$

$$\text{Min latency} = \left\lceil \frac{\text{Length of data}}{\text{Bus Width}} \right\rceil \times \text{Bus Frequency} \tag{5.2}$$

The downside of synchronous buses is that they require a large number of signal lines to both increase throughput and to reduce the minimum latency. The bus frequency also has an upper bound; if the frequency is increased too much the data between the signals will start to become misaligned with each other, resulting in a higher rate of transmission errors. A synchronous bus usually transmits data in parallel.

The second option is to use an asynchronous bus. This allows high-speed serial transceivers to be used since the clocks no longer have to be in sync with each other. High-speed transceivers use a Serializer and Deserializer (SERDES) to convert the parallel data into a serial stream, transmit it across the physical cable, then receive the data and convert it back into parallel data [3]. The main advantage to using high-speed serial is speed. Since high-speed serial uses special transceivers to transmit the data over differential pair wires, high transition frequencies are possible. The Series 7 FPGAs from Xilinx have built-in, power-efficient transceivers called GTX or GTH that support line rates between 500 Mb/s to 12.5 Gb/s for GTX and up to 13.1 Gb/s for GTH transceivers [32]. These transceivers are highly configurable and tightly integrated with the programmable logic resources in the FPGA. The different FPGA parts have different numbers of the GTX and GTH transceivers. It is important to note that the transceivers can be used together as long as the chosen data rate is supported by both. At 12.5 Gb/s or 1600 MB/s per lane, the serial high-speed transceivers are much faster than using a synchronous parallel bus. The transceivers can also be used together to set up a connection with multiple lanes. Each transceiver is full-duplex and has a dedicated lane for each direction. This means that the bandwidth is not shared between transmitting and receiving. The FMC Vita57.1 protocol that is used by many FPGA boards provides access to 8 transceivers, so the total maximum bandwidth is 100 Gb/s or 12.5 GB/s each direction. Counting both directions, the total potential throughput is 200 Gb/s or 25.0 GB/s. This theoretical maximum of 12.5 GB/s is far greater than the maximum from the parallel GPIF interface, which has a maximum of 400 MB/s. This maximum speed calculation is the maximum speed of the physical layer and does not account for the overhead needed to support reliable, in-order, communication channels. Typically, a line encoding scheme is used when transmitting over a serial link to ensure that words are aligned and there are enough transmissions to keep the link active. One widely adopted scheme is the 8B/10B encoding scheme developed by IBM [3]. The encoding scheme will be an additional source of overhead that is not typically found in a parallel bus.

Given the goal of scalability to large-scale system structures, GTX/GTH transceivers were chosen to support the physical layer of the communication system.

5.2 Communication Board or Direct Connection

The next important decision is whether to have the host machine directly connected to the DANNA FPGA board, as in the initial implementation of DANNA, or if there should still be an intermediary communication board, like the FX3 communication setup. The advantage to keeping the setup on a single board is that only one board is needed instead of two, and the latency in the communication would be reduced. The advantage to using a separate board is the communication method used to connect to the communication board could be different than the communication board's interface with the DANNA array, allowing greater flexibility. This means that the method used to connect to the DANNA array could use far fewer resources on the board than the communication method selected to connect to the host. Using two boards also lets multiple DANNA arrays be connected to the same communication board, allowing scaling in the number of DANNA networks used. Furthermore, the communication setup between the host and communication board could be interchanged without affecting the communication between the communication board and the DANNA arrays. With a communication board, multiple methods could be used to connect the FPGAs together, allowing a wide range of FPGA boards to be used. ASIC and

VLSI implementations will not have Xilinx GTH/GTX transceivers, but could have off-chip, custom, or no transceivers. A separate communication board will make it possible to interface with the ASIC/VLSI implementations regardless of the interface chosen, and the board will still be able to communicate with the connected FPGAs implementations. Because of all the additional flexibility and scaling it provides, the communications board was selected to be an intermediary between the host machine and the DANNA arrays.

5.3 Host to Communication Board

The connection between the host and the communication board is limited to protocols that are available both to an FPGA and to a PC. The main options are USB, PCIe, Ethernet, fiber optic, Serial ATA (SATA), and UART. UART is quickly eliminated by being the slowest by far with a maximum speed of 115200 b/s. USB was previously used and its shortcomings have been discussed. Gigabit Ethernet (GbE) over twisted pair cables has a maximum speed of 1000 Mb/s or 125 MB/s and is widely available on most PCs. However, at this speed it is slower than USB 3.0.

Fiber optics could be used to run 40Gb Ethernet (40GbE) with a maximum speed of 5 GB/s. The disadvantage is special computer hardware would be needed since commercial PCs do not come with fiber optic ports.

SATA has a maximum speed of 16 Gb/s or 2 GB/s, which makes it a compelling option; however, SATA connectors are not commonly found on FPGA boards without an adapter and interface protocol intellectual property (IP) would have to be licensed or custom-designed in order to use it. Furthermore, SATA on an FPGA is commonly used to connect to a hard drive and not to interface with another machine.

PCIe is the fastest option other than 40GbE. If Xillybus is used to aid in PCIe communication between the host and the FPGA, the maximum bandwidth is 800 MB/s, 1700 MB/s, and 3500 MB/s for revisions A, B, and XL respectively. The downside to Xillybus is the required license fee for commercial implementations. The alternative to using Xillybus is joining the PCIe group and building a custom design. Since PCIe is a common PC interface available on all desktop computers, fast, and easy to implement with the help of Xillybus, it was chosen as the interface between the host PC and the communication board.

5.4 Communication Board to DANNA

Even more options are available when choosing an interface between the communication board and the DANNA boards. In order to have a complete communication setup, decisions have to be made about the physical connection, the encoding of the data, the link level protocol, and the transport level protocol. Prepackage solutions exist, but most of them are not free. Rapid IO is one such solution, with a license to use their IP costing $25,000. The other options are writing original code to interface with the transceivers or using the IP provided by Xilinx. Luckily, Xilinx has a LogiCORE IP called Aurora, which is an open link-layer protocol that uses the high-speed serial transceivers on the FPGA. The Aurora core is lightweight, scalable, and provides many configuration options to the user. The Aurora core can take full advantage of the GTX/GTH transceivers and can use up to 16 transceivers for a channel, which results in a throughput that ranges from 480 Mb/s to over 84.48 Gb/s. Aurora was selected as the link-layer protocol because of its availability, flexibility, cost, and speed. Aurora uses either 8B/10B or 64B/66B line encoding. The 8B/10B encoding is widely used with many serial technologies, such as Ethernet and PCIe. The 64B/66B encoding is used for 10 Gigabit Ethernet and has less encoding overhead than 8B/10B [3]. A disadvantage to 64B/66B is a lower ratio of sync bits to payload bits, which could result in the possibility of a slight DC bias, longer alignment times, and more complex encoders and decoders. Because of the downsides of 64B/66B encoding, 8B/10B encoding was chosen for the first implementation. If the overhead of 8B/10B proves to be too great, the encoding can be changed to 64B/66B at a later time.

Deciding to use Aurora as the link-layer protocol is only part of a board to board communication solution; a physical connector still needs to be chosen. Most Xilinx FPGA boards route the high-speed transceivers to either a special purpose connector, like PCIe or SFP, or to a general purpose connector, like an FMC connector. Since the FMC connector is commonly found on most FPGAs, and since it has the highest number of high-speed

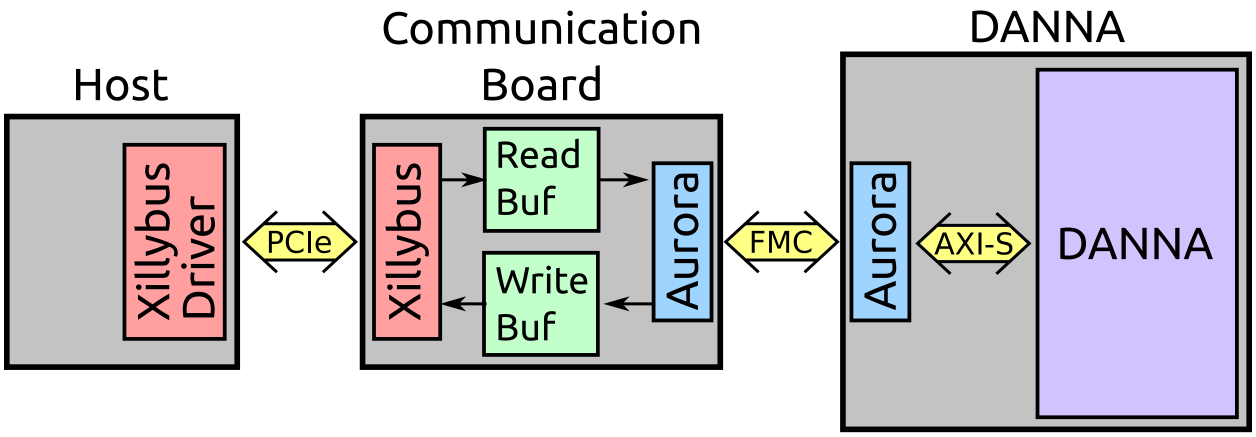


Figure 5.1: Diagram of the communications board.

signals, it was the logical choice. Some FPGAs are designed to stack and can be directly connected together. Other FPGAs need an intermediary. FMC is designed primarily to connect FPGAs to a daughter card and is not designed to be able to connect two FPGAs together unless one of the FPGAs is designed to stack. Because of this, the 8-port FMC to SMA daughter card and SMA cables were chosen to connect FPGA boards that are not designed to stack. Using SMA cables to connect the FPGAs together is logical since they can handle the high-speed differential signals and provide maximum flexibility with each transceiver being wired up independently. The main downside is the large number of cables that will have to be connected—2 cables per lane per direction, resulting in 4 cables needed to connect one duplex Aurora lane. However, this solution is still the best available, resulting in the fastest speeds and the most flexibility. A direct FMC connection was used in the prototype design and a connection using SMA cables has not yet been tested.

5.5 New Communication Board Design

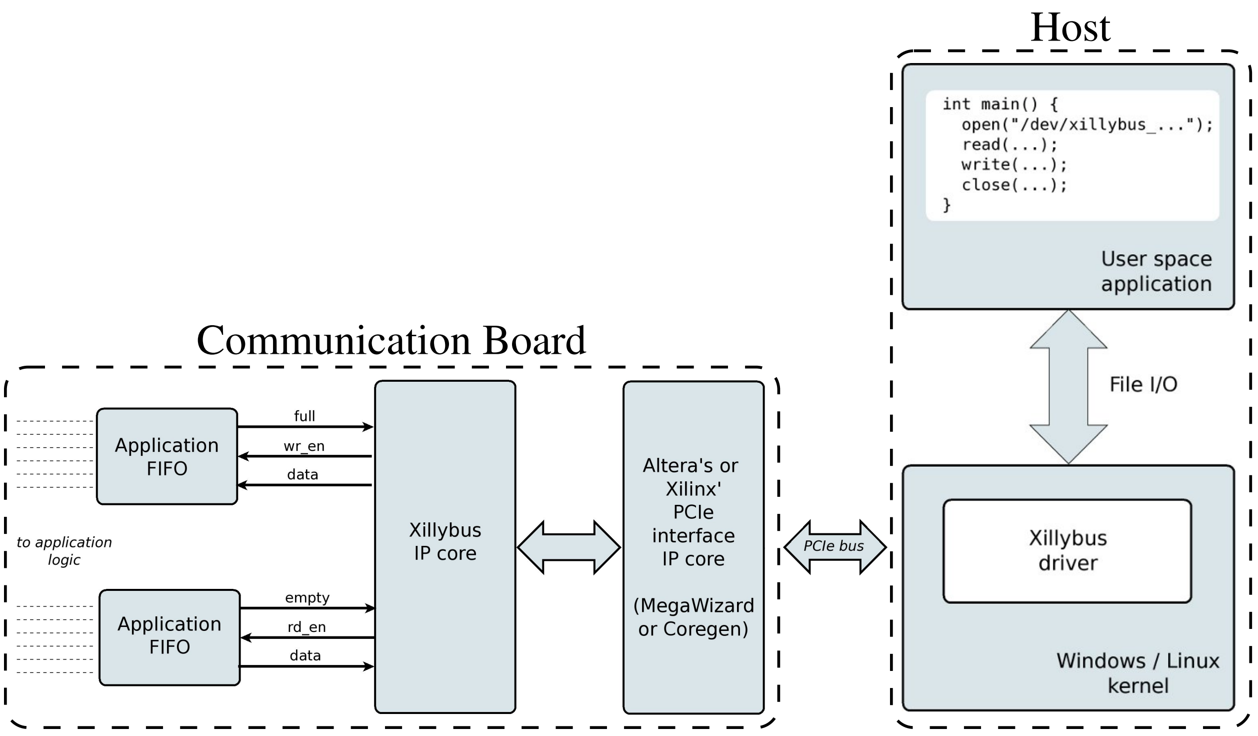
After evaluating various possible communication solutions, a new communications board was designed using the best options. Figure 5.1 shows a high-level block diagram of the new communication setup. The communication board sits in the middle and facilitates communication between the host PC and the DANNA array. The communication board connects to the host over PCIe using Xillybus. The Xillybus driver has to be installed on the host to connect to the communications board. The Aurora protocol is used to transfer data from the communication board and the DANNA array. The transceivers used by Aurora are connected via an FMC connector. Data sent to the communication board is stored in buffers until the destination is ready to receive it. The buffers are asynchronous FIFOs and provide synchronization between the clock regions used by Xillybus and Aurora. On the DANNA FPGA, an AXI4-stream bus is used to connect Aurora to the DANNA array.

The next few chapters will look at evaluating the different components of this design starting with PCIe in Chapter 6. Chapter 7 will evaluate the Aurora link layer protocol and Chapter 8 will discuss the AXI4-stream bus.

Chapter 6

PCIe

PCIe is a complex protocol with many low-level details that must be implemented correctly in order to create a successful design. Because of this, Xillybus is used to handle the data flow between the host machine and the FPGA. From the Xillybus product brief, "Xillybus is a straightforward, intuitive, efficient DMA-based end-to-end turnkey solution for data transport between an FPGA and a host running Linux or Microsoft Windows [36]." It "offers several end-to-end stream pipes for application data transport [37]." Xillybus is designed to be a complete solution for different types of workloads, and has been checked for robustness and dependability on many different FPGAs. Adding Xillybus into a design is easy since the Xillybus IP core can interface directly with standard application first in, first out memory (hereafter referred to as FIFOs) in the FPGA design. On the Linux host side the data pipes look like standard streaming I/O devices and use the same standard operating system calls. Figure 6.1 shows a block diagram of Xillybus. The communication board box highlights the component that is on the FPGA. The Xillybus IP core is connected to the application FIFOs, which in turn is connected to the application logic. This core is then connected to the Xilinx PCIe interface, which connects to the PCIe bus. Xillybus uses the PCIe interface core to utilize the transceivers to communicate over the bus. Xillybus provides a wrapper around the PCIe interface to handle the details of the interface and communicate to the host. The host box highlights the component that runs on the host machine. The Xillybus driver is included in the Linux kernel starting with version 2.6.36. The user application uses standard IO calls as shown in the user space application block. Xillybus was previously used in the



```
int main() {
  open("/dev/xillybus_...");
  read(...);
  write(...);
  close(...);
}
```

Figure 6.1: Simplified FPGA block diagram of Xillybus using PCIe transport with host interface block [36].

initial single board implementation of DANNA. This use of Xillybus worked well, with the main disadvantage being the heavy resource usage of the Xillybus and PCIe IP cores in the design. The new communication solution will no longer have this downside since a dedicated FPGA will be used to handle the communication.

6.1 Setting Up Xillybus

Xillybus has made the process of getting started easy by providing many working demo bundles that are compatible with various FPGAs. To get started, the demo bundle for the VC707 was downloaded and its basic behavior was tested using Unix utilities to send and receive data from the Xillybus device files. Once this simple design worked, the design was customized for DANNA.

6.1.1 Core Configuration

The main way to customize the Xillybus core is through the use of their custom IP core factory [39]. It allows for defining, generating, and downloading custom configurations of the Xillybus IP core. These configurations define a number of streams with customizable attributes to best meet the needs of the application. The custom IP core files are designed to partially replace the files included in the demo design for the board. Because of this, the user's guide highly recommends getting the demo example to work before customizing the core.

The configuration of the Xillybus core went through many iterations while trying to find the best configuration. In order to test the effects of the different configurations, a loopback project was set up, along with code that could test the latency and bandwidth of the stream. Initially, the core factory was allowed to auto set the internals of the stream, but better performance and control over the design was obtained by manually specifying the internal buffer sizes. The final configuration used the type "data exchange with coprocessor" and used 1024 buffers, each with a size of 128kB (total size of 128MB). Buffers of this size are the largest recommended for a single stream. If larger buffers were used then there would have to be changes to the configuration of the kernel driver. The buffer size used is larger than needed for the design to work, but will allow for future growth. Latency is not a problem with the larger buffers, since the driver can be forced to flush the buffers by using the write system call with a zero length buffer [40].

Another main configuration decision is whether to use synchronous or asynchronous streams. Synchronous streams cause the host program calls to run synchronously with the FPGA, blocking as the data transfer takes place. Asynchronous streams allow for background flow of data by transferring data independently of the host application. Asynchronous streams have better performance since they can communicate data between the host's kernel level software and the FPGA without the involvement of the user level application software. In order to maximize bandwidth and because the DANNA SDK already uses an asynchronous data model, an asynchronous data stream configuration was chosen.

Table 6.1: Xillybus revision summary [38].

Revision	Related demo bundle	Bandwidth multiplier	Maximal bandwidth	Internal data width	Width of PCIe block bus	Allowed user interface data widths
Revision A	Baseline	×1 (baseline)	800 MB/s	32	64[1]	8, 16, 32
Revision B	Baseline	×2	1700 MB/s	64	64	8, 16, 32, 64, 128, 256
Revision XL	XL bundle	×4	3500 MB/s	128	128	8, 16, 32, 64, 128, 256

6.1.2 Xillybus Revision B

Xillybus has been making improvements since the last time it was used in DANNA development. The main improvements are the addition of Revision B and XL Xillybus cores that were introduced in 2015 [38]. These cores remove the bandwidth limits of the Revision A cores by changing the internals of the design. The main change made in these revisions to increase bandwidth is substituting the 32-bit internal data bus found in Revision A with a 64-bit or 128-bit data bus in Revision B and XL, respectively. Revision B is a drop-in replacement for Revision A, and the two configurations can easily be swapped in and out using the IP core factory. Revision XL is a larger change than Revision B and is based on a different demo bundle, which prevents Revision XL from being a drop-in replacement for Revision A. In order to transfer data at the maximum throughput, the data width of the stream has to be the same size as the internal data width or larger. This means that the benefit of switching to Revision B or XL will only be seen when data is 64 or 128 bits wide, respectively. Table 6.1 summarizes the differences between the Xillybus revisions. Revision B was chosen for the final configuration since it has higher bandwidth than Revision A, but it is still compatible with the Revision A demo bundle. Revision XL could be used in the future if it is determined that more bandwidth is needed. In order to maintain packet alignment, the width of the input stream is 32 bit and the width of the output stream is 64 bit.

6.2 Testing Xillybus

In order to test Xillybus, a loop back project was set up and a host application was written to measure the throughput and latency with adjustable parameters. From initial latency tests,

[1]On Xilinx series-7 FPGAs. For other FPGAs, the connection with the PCIe block is 32 or 64 bits wide.

it became apparent that the manual flushing of the buffers was not working as expected. The observed latency was 11.9 ms, which closely matches the automatic buffer flush that occurs when nothing is written for a specific amount of time (typically 10 ms). The issue is with the Xillybus driver that is included with Ubuntu 16.04. Installing the Linux driver fresh from `http://xillybus.com/pcie-download` fixed the flushing problem.

Figure 6.2 shows the difference changing the driver had on round trip time. The graph shows that the original driver with flushing had the same latency as the new driver without flushing. The new driver with flushing has a much lower round trip time. The plateau without flushing is caused by the 10 ms timeout that forces the flushing of the buffers. This graph shows that manual buffer flushing was now working in the original driver.

Figure 6.3 shows that calling the flush command does cause the throughput to be slightly reduced in small transfers, and that moving to the new driver had no effect on latency.

Figure 6.4 shows the effect that using a synchronous stream has on the throughput. From the graph it is clear that using an asynchronous stream leads to a more predictable throughput since the transfer does not have to stay in sync with user space code.

Figure 6.5 shows the effect that switching to Revision B has on throughput. In order to get the max throughput from Revision B, the bus width needs to be 64 bits. Using Revision B with 32 bit bus only gives a slight increase over Revision A since it cannot take full advantage of the 64 bit internal data width.

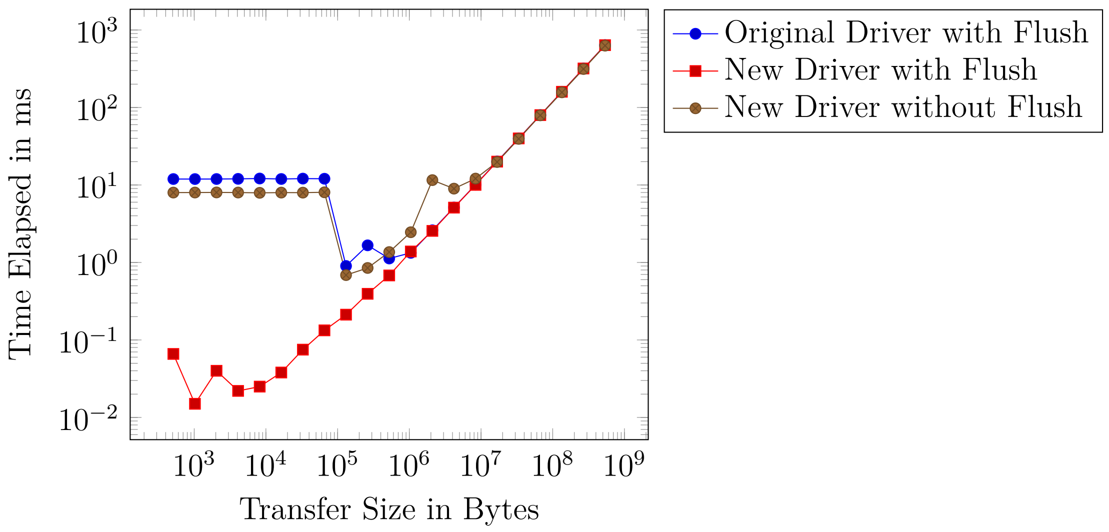


Figure 6.2: Driver change effect on latency (block size kept at 512 Bytes).

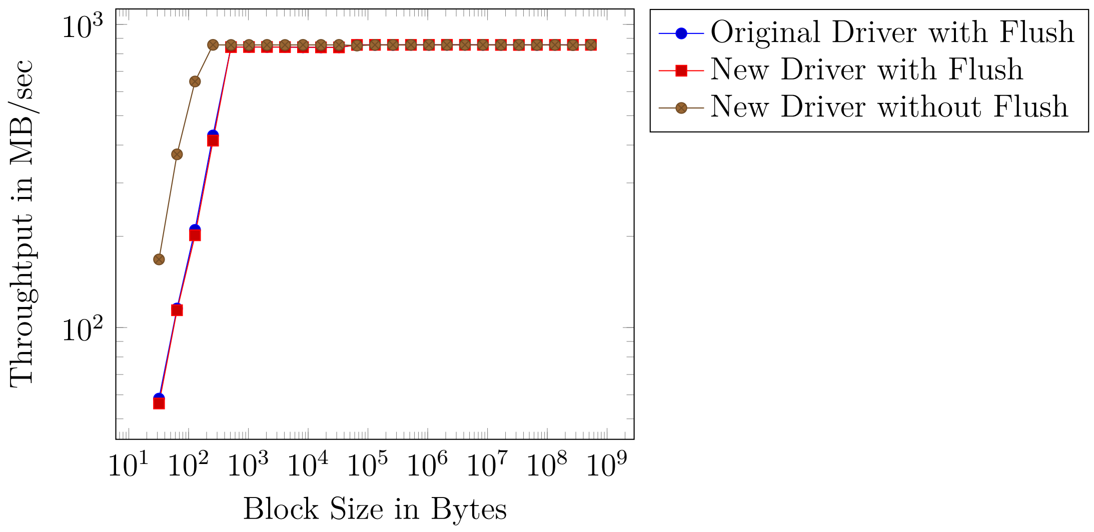


Figure 6.3: Driver change effect on throughput (data size kept at 512 MB).

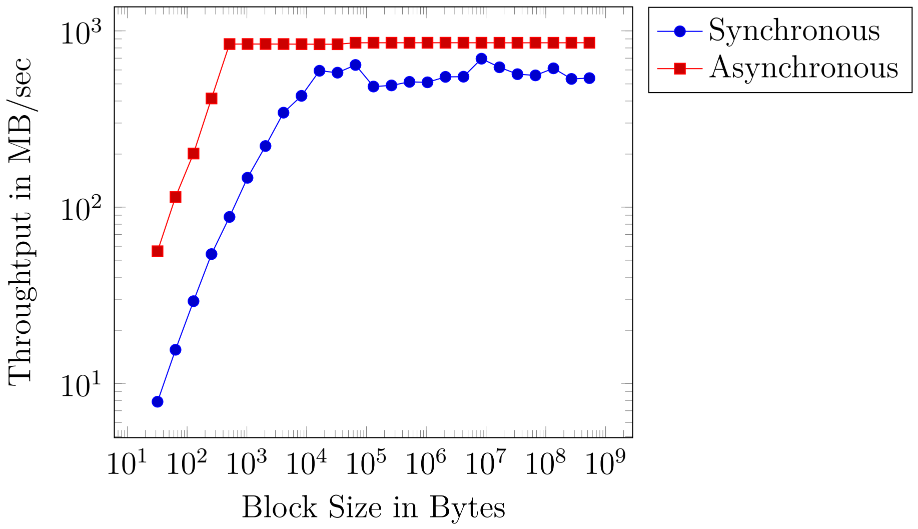


Figure 6.4: Throughput of synchronous vs. asynchronous (data size kept at 512 MB).

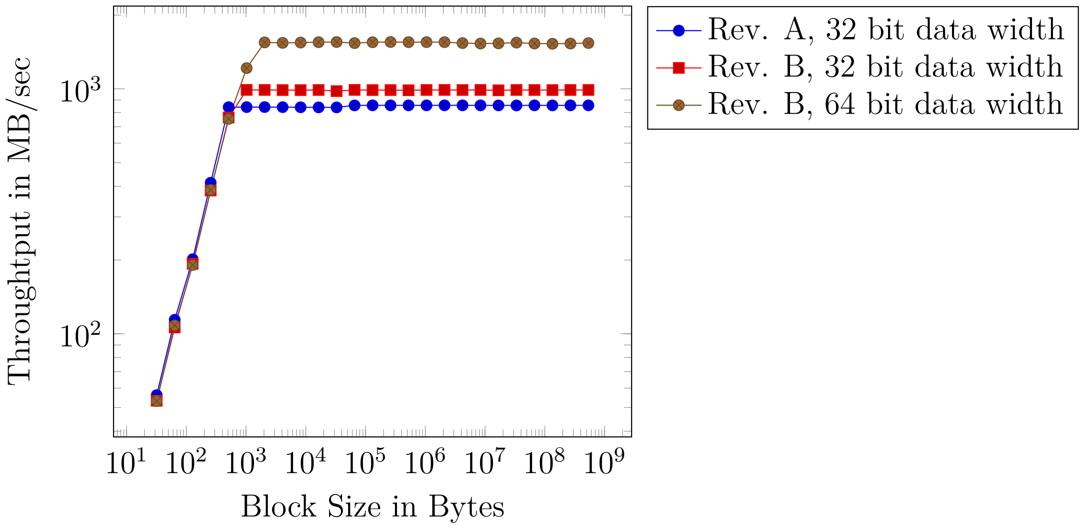


Figure 6.5: Throughput Revision A vs. Revision B (data size kept at 512 MB).

Chapter 7

Aurora

"The Aurora 8B/10B core is a scalable, lightweight, link-layer protocol for high-speed serial communication [33]." The Aurora LogiCORE IP is provided to developers by Xilinx and is used to establish serial links. The core uses the on-chip Xilinx GTX, GTP, and GTH transceivers to transfer the data. Up to 16 transceivers each running at a link speed of up to 6.6 Gb/s can be used at a time to establish a link. Each transceiver has a separate transmit and receive signal allowing the data channel to operate in full-duplex or simplex mode. The Aurora core adheres to the Aurora 8B/10B Specification v2.2 (SP002) [34]. The Aurora IP core also has built-in framing, flow control, and cyclic redundancy check (CRC). It will also setup and maintain the communication channel. Figure 7.1 shows an overview of an Aurora channel and includes the different terminology of the connection. The user application connects to the Aurora core via a user interface and the data over this interface is the user data. See Chapter 8 for more information on this user interface. Two Aurora cores connect to become Aurora channel partners, establishing a communication channel. This channel is made up of multiple lanes, each lane corresponds to one transceiver. Each lane can be setup as either full-duplex or simplex. Full-duplex lanes can transmit in both directions whereas simplex lanes can only send data in one direction. The data is sent across the lanes as 8B/10B encoded data.

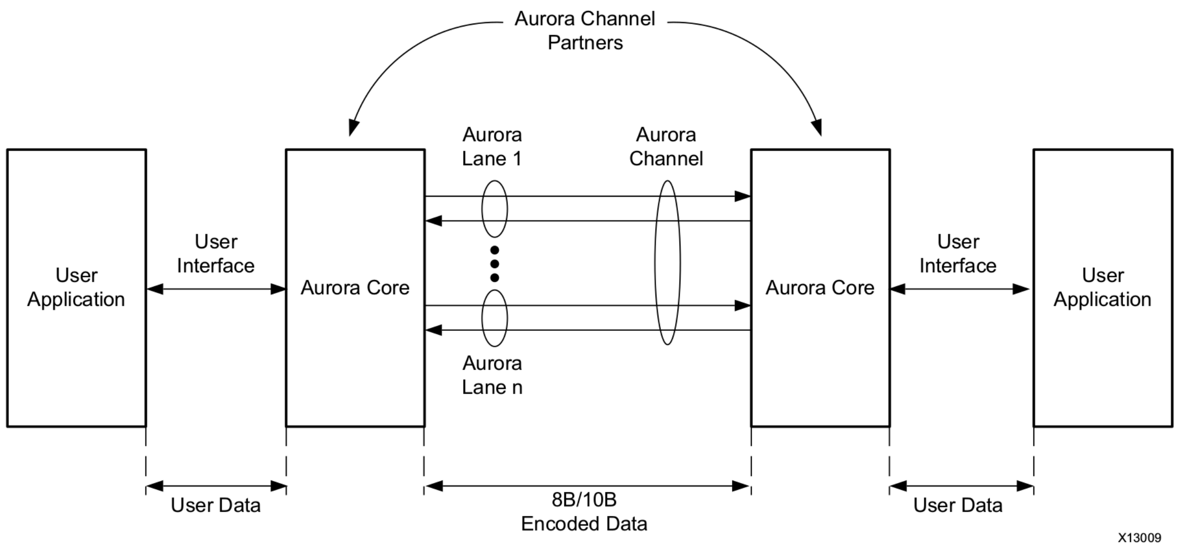


Figure 7.1: Aurora 8B/10B channel overview [33].

7.1 Aurora Example Design Setup

The example design was used to verify the correct setup of an Aurora communications channel. The design was first simulated using the included test bench. Figure 7.2 shows a diagram of the included example design. The example design included the Aurora component with a frame generator and a frame checker. The frame generator is used to feed input into the design and is setup in such a way that the frame check component can determine if the generated sequence is correct. The demonstration test bench then verifies if the frame checker determines that the output packets are correct. The test bench ran successfully and helped show how the ports of the Aurora core should be connected.

Running the example design on hardware proved more difficult. The ports of the Aurora IP are constrained so that they had to be connected to certain external ports. Because of this, an internal loop-back design could not be constructed since the transmit (TX), receive (RX), and clock (CLK) signals all have to be connected to an external port of the correct type and cannot be connect to internal logic. Two boards were used to perform the hardware test. A HiTech Global 690T was attached as a FMC daughter card to a Xilinx VC707 Evaluation Kit board. The example design was then split to run on two FPGAs and tested using debug

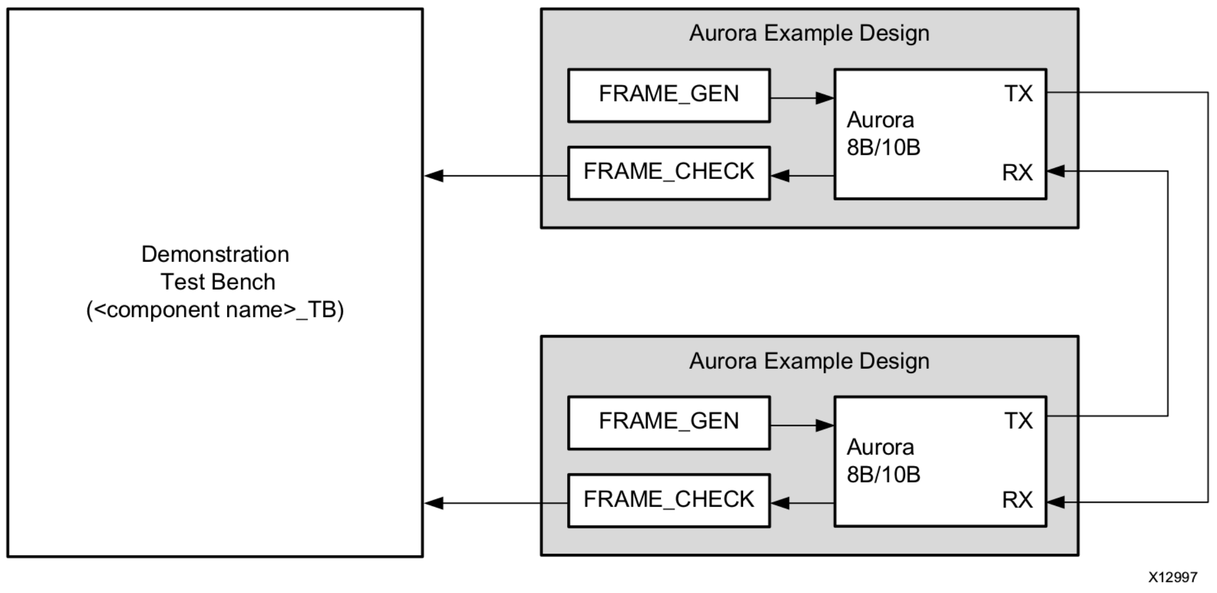


Figure 7.2: Aurora example design [33].

probes and LEDs. Both the probes and the LEDs indicated that the communication channel was up and that no errors were detected in the communication.

7.2 Verification

Since the example design functioned correctly in hardware, a loop back was setup. This loop back used the same two FPGAs and Xillybus PCIe design from Chapter 6. Data was sent to the VC707 from the computer, which was then sent to the 690T via Aurora. From the 690T the data would be sent back to the VC707 by Aurora and then back to the host over PCIe. The loop back was successful and verified that Aurora was working as expected. Results from benchmarking Aurora can be found in Chapter 11.

Chapter 8

AXI4-Stream

Previously, the native FIFO interface has been used to interface with the DANNA array. With the new communication setup, AXI4-Stream was a more logical interface choice. Both interfaces are similar and VHDL code can be written to convert from one interface to the other. AXI4-Stream has the advantage of being the native interface of the Aurora module. In addition, AXI4-Stream is part of the ARM standard and is also supported by multiple Xilinx IP blocks. In particular, Xilinx provides AXI4-Stream width converters and FIFOs. The main reason to switch to AXI4-Stream is that it supports framing. Framing is already used in Aurora so that the CRC check can be performed. Framing in DANNA would allow for easier implementation of variable length packets and for easy changing of the length of the fixed packets. The remainder of this chapter provides a brief introduction to AXI4.

Advanced eXtensible Interface 4 (AXI4) is a protocol defined as part of the Advanced RISC Machine (ARM) Advanced Microcontroller Bus Architecture 4 (AMBA4) released in 2010 [35]. There are three types of AXI4 interfaces. AXI4 is for high-performance memory-mapped operations. AXI4-Lite is for simpler, low-throughput memory-mapped communication. AXI4-Stream is for high-speed streaming data. The user interface for Aurora uses the AXI4-Stream interface to transfer user data to and from the IP core [33]. In order to use CRC error checking, the user interface must also use framing. Xilinx provides many different IP blocks to handle AXI4-Stream buses. The main ones used in the communication design are the AXI4-Stream Data FIFO and the AXI4-Stream Data Width Converter.

Table 8.1: AXI4-Stream interface signals [2].

Signal	**Source**	**Description**
ACLK	Clock source	The global clock signal. All signals are sampled on the rising edge of **ACLK**.
ARESETn	Reset source	The global reset signal. **ARESETn** is active-LOW.
TVALID	Master	**TVALID** indicates that the master is driving a valid transfer. A transfer takes place when both **TVALID** and **TREADY** are asserted.
TREADY	Slave	**TREADY** indicates that the slave can accept a transfer in the current cycle.
TDATA[**(8n-1):0**]	Master	**TDATA** is the primary payload that is used to provide the data that is passing across the interface. The width of the data payload is an integer number of bytes.
TKEEP[**(n-1):0**]	Master	**TKEEP** is the byte qualifier that indicates whether the content of the associated byte of **TDATA** is processed as part of the data stream. Associated bytes that have the **TKEEP** byte qualifier deasserted are null bytes and can be removed from the data stream.
TLAST	Master	**TLAST** indicates the boundary of a packet.

AXI4-Stream is a master/slave single direction interface with data being transferred from the master to the slave. For bidirectional data transfer, both endpoints need to have both a master and slave interface. The signals used by the AXI4-Stream interface are shown in Table 8.1, which is reproduced from the specification document [2]. TVALID is used to specify when data on the bus is valid. TREADY tells the master that the slave is ready for data. A stream transfer is defined by a single TVALID, TREADY handshake. TLAST is used to specify the end of a frame. TKEEP specifies how many bytes are part of the frame in the last data payload. This chapter provides only a brief introduction to AXI4-Stream. The full AXI4-Stream interface specification can be found in [2].

Chapter 9

New Communication Board Implementation

Now that all of the components of the communication board have been verified, the complete system can be assembled. The complete system was setup using the general design found in Section 5.5.

9.1 Hardware Components

The new communication board was prototyped using a Xilinx VC707 evaluation board. The evaluation board uses a Virtex7 XV7VX485T-2FFG1761C FPGA. The VC707 was selected for the communication prototype since it has a FPGA that can be used to configure the communication logic. Furthermore it has both a PCIe connector and two FMC high pin count (HPC) connectors. Selecting a prototype board which has two FMC connectors is advantageous since it allows for multiple DANNA FPGAs to be connected. In order to test the communication board, the test setup also includes a host PC and a HiTech Global HTG-777 with a Xilinx Virtex7 X690T. Figure 9.1 shows a picture of the hardware setup used to test the communication board. The communication board is connected to the host PC via PCI express x8 gen2 edge connector. The HiTech Global board will have the DANNA array loaded and is connected to the communication board with an FMC Vita 57.1 connector.

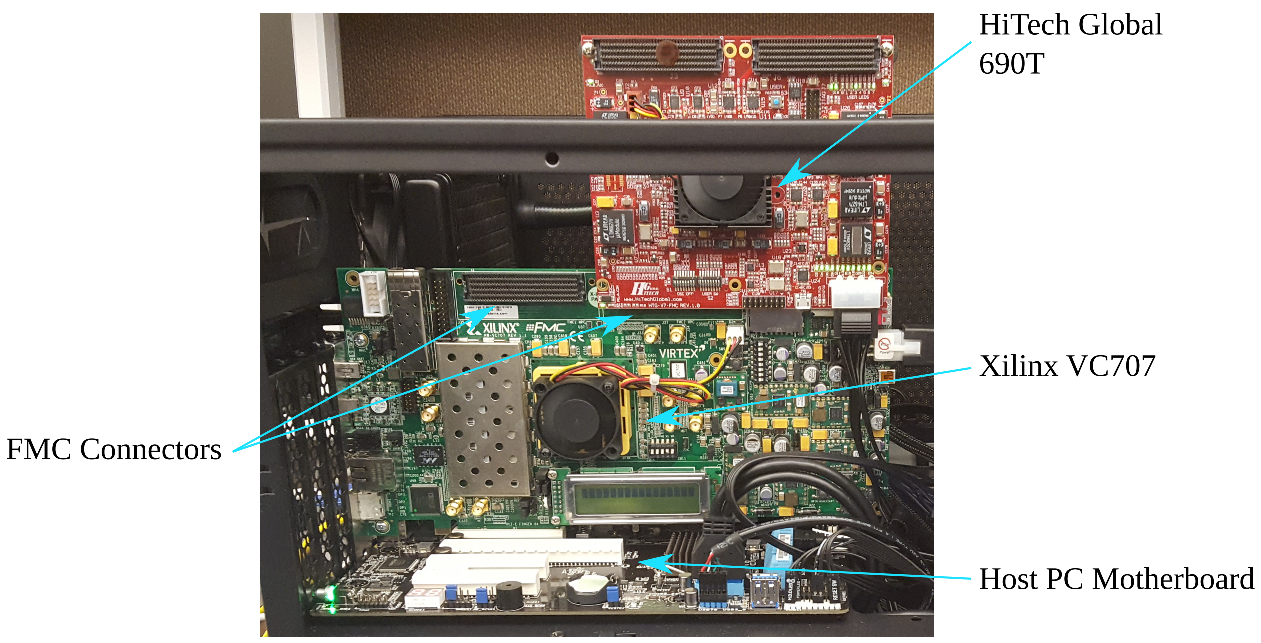


Figure 9.1: Hardware setup.

The host system can be any x86-64 computer that also has a PCI express x8 gen2 slot. This host system consists of an Asus P10S-M micro ATX motherboard, an Intel Xeon e3-1275 processor, 32 GB of DDR4-3333 memory, and a Thermaltake case. The computer is running Ubuntu 16.04.2 LTS. The only software needed to connect and run the DANNA network is the DANNA SDK, discussed in Section 3.3. Because of the buffer flushing problem with the old Xillybus driver (detailed in Section 6.2), the newest Xillybus driver needs to be downloaded and installed. Another important detail is the machine has to be rebooted once the bitfile is loaded onto the communication FPGA. This reboot is necessary so that the operating system can discover the new PCIe device.

The DANNA FPGA chosen is the same FPGA that is used in the DANNA Kit. This FPGA is large enough to fit a 45 by 45 DANNA array and also has a FMC connector designed to be stacked onto another FPGA.

9.2 Implementation Details

This section takes a more detailed look at the implementation details of the communication board's design.

The packets from the host to the DANNA array are called input packets. These packets are 36 bytes long. Packets from the DANNA array to the host are called output packets and are 64 bytes long. To make data alignment easier and to more closely match the previous communication design, the bit width of the buses was chosen so that it can evenly divide the total packet size. The previous FX3 design used 32 bit buses to transfer the packets. For convenience and since 32 bits is a common word size, a 32 bit bus was used for the majority of the design. The one exception is the width of the Xillybus read stream which uses a 64 bit bus and is discussed in Subsection 9.2.1.

The main components of the design are the Xillybus IP core, read buffer, write buffer, reset logic, Aurora IP core, packet framer, and native flow control. Each of these components will be looked at in more detail in the following sections.

9.2.1 Xillybus configuration

Xillybus provides a Custom IP Core factory to customize the configuration of the Xillybus Core. Based on the Xillybus testing, Revision B customized with a 64 bit bus provides roughly twice the performance of Revision A and 32 bit Revision B. The problem is a 36 byte packet size does not divide evenly into 64 bits ($36\,\text{bytes}/64\,\text{bits} = 4.5$). In order to get better performance out of Xillybus and to maintain packet alignment in buffers, a bus width of 32 bits was used for input packet and a bus width of 64 was used for output packets. This allows for the larger output packets to be transferred faster than the smaller input packets can be sent. The output buffer on the communication board is used to convert the 32 bit bus used by Aurora to the 64 bit bus used by the Xillybus read stream. Xillybus has an end-of-file flag which is currently set to '0', but it can be used in the future to indicate the end of the output from the DANNA array.

9.2.2 Communication Board Buffers

The communication board buffers were implemented using FIFO IP blocks. There are two FIFOs on the communication board, which buffer the packets while they are waiting to be sent to the next board. Both FIFOs are created using the Xilinx FIFO Generator (12.0).

Table 9.1: FIFO settings for communication board buffers.

Property	Input Packet FIFO	Output Packet FIFO
Interface Type	Native	Native
FIFO Implementation	Independent Clocks Block RAM	Independent Clocks Block RAM
Synchronization Stages	2	2
Read Mode	First Word Fall Through	Standard FIFO
Write Width	32	32
Write Depth	512	1024
Read Width	32	64
Reset Pin	✓	✓
Enable Reset Synchronization	✓	✓
Full Flags Reset Value	1	1
Dout Reset Value	0	0
Valid Flag	Active High	None
Programmable Full Threshold	511	1008
Programmable Empty Threshold	4	15

One FIFO is for input packets and the other is for the output packets. Table 9.1 shows the settings used to generate each of the FIFOs. The FIFOs have to change clock domains between the PCIe bus clock from the Xillybus IP core and the user clock from the Aurora IP core. The PCIe bus clock operates at 100 MHz while the Aurora user clock operates at 156.25 MHz. In order to allow independent clocks and different bus widths the independent clocks block RAM implementation was chosen. Both FIFOs are set up to use the native FIFO interface so that they can be directly connected to the Xillybus IP Core. The signals for both FIFOs are set up to connect to an AXI4-Stream bus on the Aurora side. Aurora uses an AXI4-Stream interface for the user data. Therefore the Input Packet FIFO is set up to be first word fall through and has an active high valid flag. Both of the settings allow a native interface FIFO to be connected to a AXI4-Stream interface. Additional logic is also needed to drive the TLAST signal used to signify the end of a frame. This logic was set up to raise TLAST on the 9th count of every work being read from the FIFO. This set each frame boundary on the input packet boundary. No special settings are needed for connecting from AXI4-Stream to Native FIFO. The framing information is lost, but it has no meaning for the Xillybus IP Core.

The data widths are 32 bit, with the exception of the output for the output packet FIFO which is 64 bits. Both FIFOs are configured to hold 512 words of the largest bus size. This means there are 512 32-bit words for the input packet FIFO and 1024 32-bit words (512 64-bit words) for the output packet FIFO. This size proves to be plenty large to hide the interface crossing given the current DANNA design. If needed, the buffer size can be increase to further utilize burst transfers. Or, since the buffers are not fully used, they can be reduced in size to save resources on the communication board.

The threshold values are chosen so less than a full packet is empty and if there is not enough room for a full packet it will be full. This is to allow full packets to be added and removed from the FIFO, further helping to keep the data aligned.

9.2.3 Aurora Configuration

Aurora has multiple configuration options that allow the communications channel to be set up to meet application needs. Since Aurora can operate full duplex, only one IP Core is needed to handle both sending and receiving packets. Table 9.2 shows the settings that are selected for the Aurora core. The lane assignments are not shown in the table, but the correct transceivers are selected to connect the FPGAs together over the FMC connector. The Aurora IP Core is configured the same on the communication board FPGA and the DANNA FPGA apart from the Gigabit Transceivers (GTs) used in the lane assignment. The bit width of the Aurora user data interface can be calculated as shown in (9.1).

$$\text{data width} = \text{number of lanes} \times \text{lane width} \times 8 \tag{9.1}$$

In order to have a data width of 32 bits and one lane, a lane width of 4 bytes is chosen.

The default frequency of the reference clock on the 690T provided to the transfers is 156.25 MHz so a Line Rate of 6.25 Gbps is chosen. 6.25 Gbps is the closet value to the max of 6.6 Gbps that also use 156.25 MHz as the reference clock. Full duplex is chosen to allow

Table 9.2: Aurora 8B10B (10.2) configuration settings.

Property	Value
Lane Width (Bytes)	4
Line Rate (Gbps)	6.25
GT Refclk (MHz)	156.250
Dataflow Mode	Duplex
Interface	Framing
Flow Control	Completion NFC
Little Endian Support	✓
CRC	✓
Lanes	1
Shared Logic	Include Shared Logic in Core

one channel to handle both directions of communication. CRC error detection is enabled to allow errors in packet transfer to be detected. Framing has to be enabled to allow CRC to be enabled. Framing also allows for one packet to be added to one frame, therefore the packet boundary is also the frame boundary and errors are detected on the packet level. One lane is all that is needed to provide ample bandwidth to the current DANNA design. This can be expanded in the future as more bandwidth is needed. Since only one core is needed, the shared logic is included in the core. If multiple cores are used, then some of the cores would not need the shared logic included.

9.2.4 Reset Logic

Reset logic is used to reset the data in all of the FIFOs, both on the communication board and on the DANNA FGPA. This ensures that old data is not left in the buffers between runs. The device open signals are used to drive the reset. If neither the read nor write Xillybus device file descriptors are open on the host, then the reset signal is asserted. With this setup the host can clear out the communication buffers by closing and reopening both device file descriptors. This reset does not reset the DANNA array and a reset command will still have to be sent to reset the array. With the FX3 implementation the only way to reset the communication buffers was to press the physical reset button on the FX3. Now a buffer reset can be performed via software. The main reason a reset would be needed is if the software sends a short or long packet resulting is the packets becoming misaligned.

9.2.5 Aurora Flow Control

Aurora has two options for implementing flow control. The first option is native flow control, and it allows the receiver to regulate the rate at which the sender can send. The second option is user flow control, which provides the user with a high priority side-band in which they can implement flow control themselves. For our design, native flow control provides enough functionality. If a new design is needed it can be expanded later. Native flow control can be used to request idle cycles or to request that the user starts or stops sending data. Currently the implementation is set up to request that the user stops sending when the buffers fill up past a certain threshold and to request that transmission resumes once the buffers empty past the same threshold.

9.2.6 DANNA FPGA Details

On the DANNA FPGA, an AXI4-Stream data width converter is used to take the 32 bit data bus and convert it to a 288 bit bus (36 bytes) for input packets. Another data width converter is used to convert the 512 bit output packet (64 bytes) to a 32 bit data bus. An AXI4-Stream wrapper is used around the DANNA Array to buffer packets and provide the AXI4-Stream interface to input and output packets.

9.3 Verification

Since the components had been tested and verified independently, building and testing the complete system was easier. The main challenge with building a working system was getting the byte order of the data correct so both DANNA and the host interpreted the bytes in the correct order. Once the packets were sent in the right order, neural networks could be run on the DANNA array and the results could be interpreted. The main method used to verify correctness is running networks and comparing the output between the simulator and different hardware implementations. The DANNA FPGA implementation and the simulator have been shown to match output for multiple test networks. A selection of test networks have been chosen to build up a suite of test networks. If the simulator and the hardware match output on all of the test networks, then they are declared to both work as designed. The previous FX3 implementation would match the output of the simulator until the buffers in the communication path would fill up and packets would be dropped. Communication stress test networks were designed with input elements firing on every cycle, causing output response packets on every cycle. They generate the most communication traffic possible. The new communication network was subjected to the same test suite. It is able to match the simulator on all of the tests including the communication stress tests.

Another method of testing the correctness of the communication setup is to use debug probes to view the number of items stored in the buffers as the network runs. When the buffers on the DANNA FPGA were probed while the communication stress test was run, the probes showed that the input buffers remained at the full threshold and the output buffers only had at max one full packet. This setup shows experimentally that the communication setup is sending and receiving data faster than the DANNA array can process it.

Further testing was done to characterize the max performance of the communication boards. The results from this testing serve as further verification of the communication design and can be found in Chapter 11.

Chapter 10

Benchmarking Setup

In order to compare the performance of the previous FX3 implementation and the new communication board, multiple tests were set up. Each test setup was designed to measure either the complete communications path or an individual component of the communications path. Each test implements a communications loop back. The host could then send messages and measure how long it took the messages to be received. The packet size for the message was chosen to be 64 bytes long. This matches the output packet size, which is the larger of the packet sizes. The following sections will look at each test setup, discussing both the setup and what component the setup is trying to measure. The results from testing the setups can be found in Chapter 11. The section titles correspond to how the test setup will be labeled in the results sections.

10.1 FX3

The FX3 test setup was designed to measure the performance of the prior FX3 DANNA communication setup. The setup used USB 3.0 via the FX3 with the DANNA array removed and replaced with a loop back. Some additional changes were also required. In order to send and receive 64 byte packets, the DMA buffer flags on the FX3 input side had to be changed from using 36 byte packets to using 64 byte packets. The counters in the communication state machine also had to be modified to expect 64 byte input packets. This test setup used the FX3 and the 690T boards.

10.2 PCIe

The PCIe test was setup to test the performance of the PCIe component of the communication board design and therefore used the same Xillybus configuration. It used Xillybus Revision B with a 32 bit write data stream and a 64 bit read data stream. The host DMA buffers were set to use 1024 buffers, with each buffer being 128 kB. This means the total DMA buffer size for one stream is 128 MB. Both streams were set to be asynchronous. Only the VC707 board was used for this test and a loop back FIFO was on the FPGA to connect the input stream to the output stream.

10.3 PCIe with FX3 Emulator (PCIe GPIF)

The PCIe with FX3 emulator test was designed to measure the performance of the 100 MHz 32 bit interface that the FX3 uses to connect to the DANNA FPGA. The communication board was setup to emulate the logic of the FX3 GPIF state machine, with the flags matched to facilitate 64 byte packets. The same design on the 690T that was used for the FX3 test was also used in this test setup. This test setup uses both the VC707 and the 690T.

10.4 PCIe 64

The PCIe 64 test was designed to show the performance improvement that can be achieved by using 64 bit buses for both PCIe streams. The test setup was the same as the PCIe test, but had a 64 bit write stream. This test used only the VC707 board.

10.5 Aurora x1

Aurora x1 was designed to test the performance of Aurora using only one lane. Xillybus is configured the same as in the PCIe test. Aurora is configure with one lane using the same settings as the communication board with DANNA. The DANNA FPGA replaced DANNA for a loop back connection, but uses the same Aurora setup. This test uses both the VC707 and the 690T.

10.6 Aurora x2

Aurora x2 is the same test as Aurora x1, but uses two Aurora lanes instead of only one. It is designed to test the performance gain that can be achieved by adding more Aurora lanes.

Chapter 11

Results

This chapter covers the benchmarks that were performed and their results. The test setups used in the benchmarks are described in Chapter 10. Each test setup implements a loop back with different components of the communication design being used in the transfer.

The two main metrics measured are round trip latency and round trip throughput. A benchmarking program was written to make taking measurements easier. The benchmarking program can perform a variety of tests and the parameters can be specified on the command line. The program also performed multiple measurements and could average the results together with a mean value and a standard deviation. The benchmark program was written so it could be used to send packets over LibUSB to measure the FX3 design and device files to measure the PCIe-based designs.

There are two main variables that can be changed in the benchmark program. The first is the size of the buffer used when making a call to the transfer and receive functions. A larger buffer means that more data can be transfered before the user program has to be involved. This parameter is called the transfer size.

The other main parameter is the total amount of data that is transfered. This total amount is transfered one transfer size at a time until the total amount is reached. Figure 11.1 shows pseudo code to illustrate this distinction. The main difference is that the user program keeps needing to be entered to make the next transfer call when the total size is bigger than the transfer size. The user program will have to be entered $\frac{\text{total transfer size}}{\text{transfer size}}$ times. The benchmark makes the assumption that the total transfer size is a multiple of the transfer size.

```
// In the sent thread.
// Keep sending data while the amount sent is less than the total to
    send.
while (sent < total_transfer_size)
{
  // Send a buffer of length transfer size and add the amount to the
      amount sent.
  write(buffer, transfer_size);
  sent += transfer_size;
}

...

// In the read thread.
// Keep reading until the amount received is equal to the total
   amount.
while (received < total_transfer_size)
{
  // Read in transfer size at a time.
  read(buffer, transfer_size);
  received += transfer_size;
}
```

Figure 11.1: Pseudo code showing the difference between transfer size and total transfer size.

11.1 Latency Benchmarks

The first set of benchmarks are aimed at measuring the latency of one round trip transfer of a 64 bytes packet. This means that the total transfer size and the transfer size were both kept to 64 bytes. In order to obtain clean measurements, the computer was taken off of the network and run without a graphical user interface. In addition, the benchmark program would send 1000 round trip packets before sending a packet that is measured. The program would then average 1000 of the measured packets together to get the mean and standard deviation for the data point. The latency benchmark was run on all of the test setups presented in Chapter 10 and the results of the benchmark can be found in Figures 11.2 and 11.3. Figure 11.2 shows the results from all the benchmarks while Figure 11.3 only shows the results from the benchmarks that use PCIe. The FX3 test setup had by far the highest round trip latency, with a latency value of 80.38 µs. All of the other test setups have a round trip latency of around 6 µs. The measurement obtained had a low variance, with the standard deviation from the FX3 benchmark being 2 µs and the variation of the PCIe-based benchmarks being 0.2 µs. The low variance in part indicates that there were no measurement artifacts in the data collected. Additionally, the measured values are as expected. The documented FX3 firmware processing time for each DMA buffer is about 40 µs. Since a round trip transfer has to be processed twice by the DMA engine, a total round trip time of 80 µs seems very reasonable [13]. The much lower values of the PCIe benchmarks is also logical. Since the Xillybus allows for explicit flushing of the DMA buffers, the latency of the round trip packet is much lower.

Looking more closely at the various PCIe-based benchmarks, they all appear as expected. Taken into account the standard deviation values, the measurements for PCIe are all roughly the same. However, the mean is higher for PCIe with FX3 emulator and the two Aurora tests. One would expect the mean for those to be higher since they communicate to the 690T and back whereas the PCIe test does not. The theoretical latency for both the GPIF and Aurora can be calculated. The GPIF latency is calculated as shown in (11.1).

$$\text{GPIF latency} = \text{clock frequency} \times (\text{data cycles} + \text{overhead}) \tag{11.1}$$

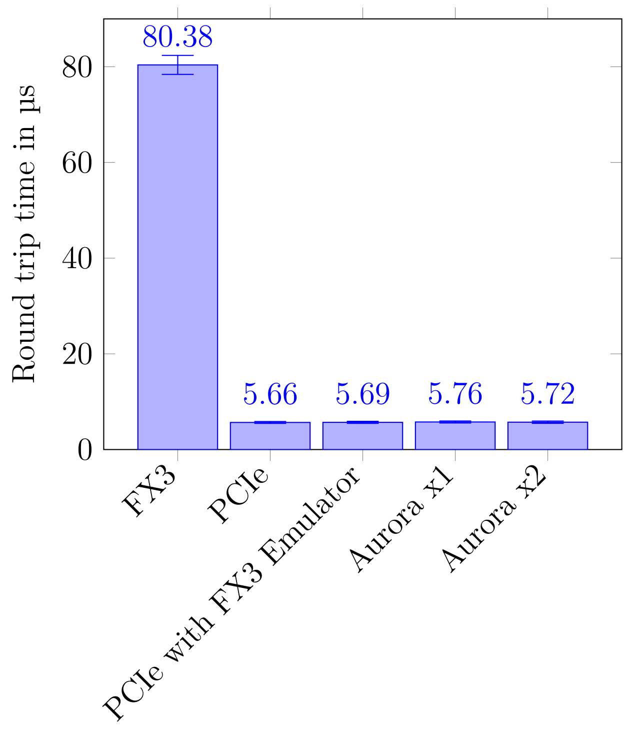


Figure 11.2: Round trip time comparison (all).

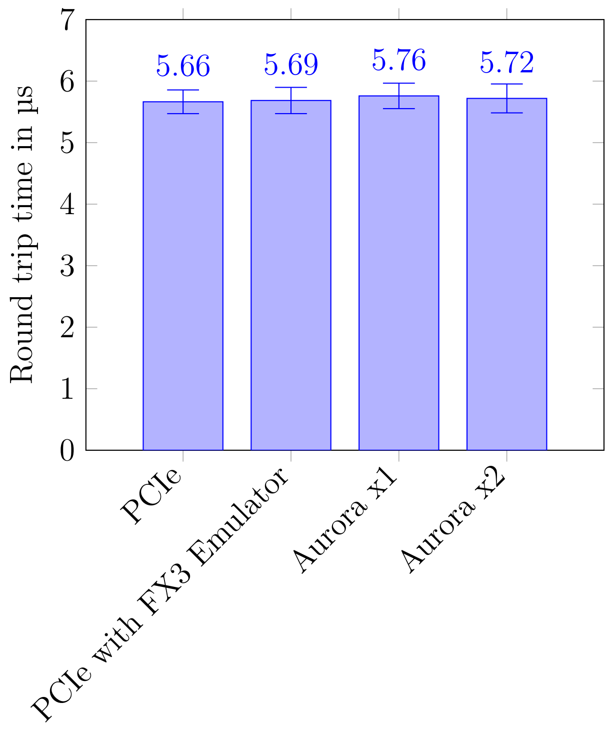


Figure 11.3: Round trip time comparison (only PCIe).

Assuming the overhead is around 10 cycles, then the round trip transfer time is 0.4 µs, as calculated in (11.2).

$$\text{round trip time} = 100\,\text{MHz} \times ((16 \times 2) + 10) = 0.4\,\mu\text{s} \tag{11.2}$$

The theoretical increase of 0.4 µs is larger than the observed increase of 0.03 µs, but taking into account that the transfer can start while the PCIe transfer is still in progress, the observe increase seems reasonable.

A theoretical calculation for Aurora can similarly be made. The Aurora latency can be calculated as shown in (11.3), which results in a theoretical latency of 0.1638 µs.

$$\frac{\text{Bits to transfer}}{\text{Transfer Rate}} = \frac{64 \times 16}{6.25\,\text{Gbps}} = 0.1638\,\mu\text{s} \tag{11.3}$$

The theoretical increase of 0.16 µs is about the same as the observed increase of $\approx$ 0.1 µs. Again, the Aurora transfer can start while the PCIe transfer is still taking place, which explains why the observed value is less than the theoretical value.

11.2 Throughput Benchmarks

The second set of benchmarks are aimed at measuring the max throughput of each design. Figures 11.4 and 11.5 both show the same data. Figure 11.5 has a log scale on both axes whereas Figure 11.4 only has a log scale on the x-axis. Both figures show throughput measured for each test design when the total transfer size is held constant and the transfer size is varied. The FX3 setup has the lowest throughput. It starts off at about 1 MB/s and increases linearly to 108 MB/s. This increase in throughput is a result of making better use of the USB 3.0's bursting capabilities. In order to maximize the FX3's performance, a large burst length and buffer size is required. The upper bound of the FX3's performance is caused by the implementation of the GPIF interface [13]. The GPIF's max throughput is shown by the PCIe with FX3 emulator line. According to "Optimizing USB 3.0 Throughput with EZ-USB" [13], the max throughput of the FX3 is 450 MB/s. This means the FX3's performance is limited by the implementation of the GPIF interface logic. The max theoretical throughput

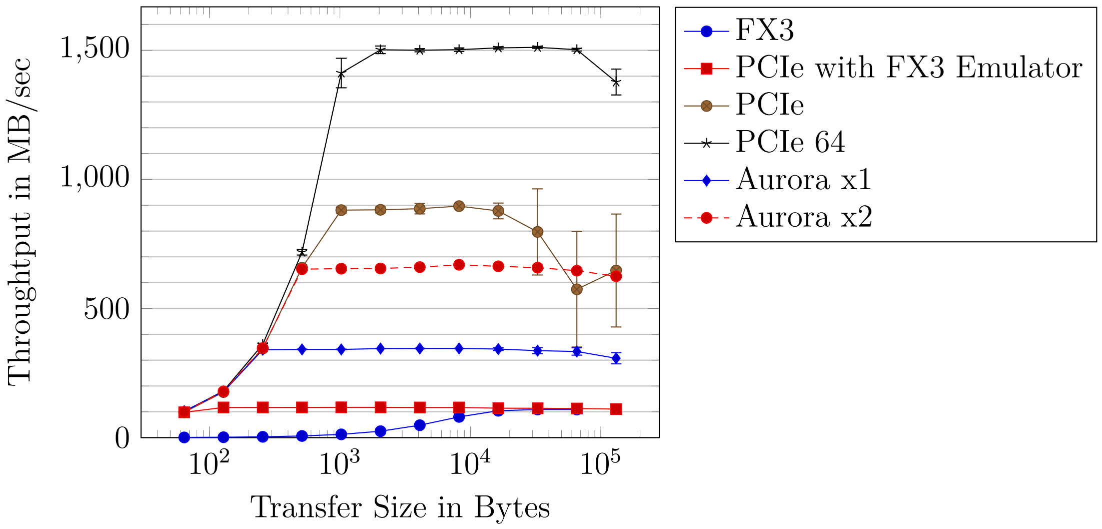


Figure 11.4: Vary size per function call.

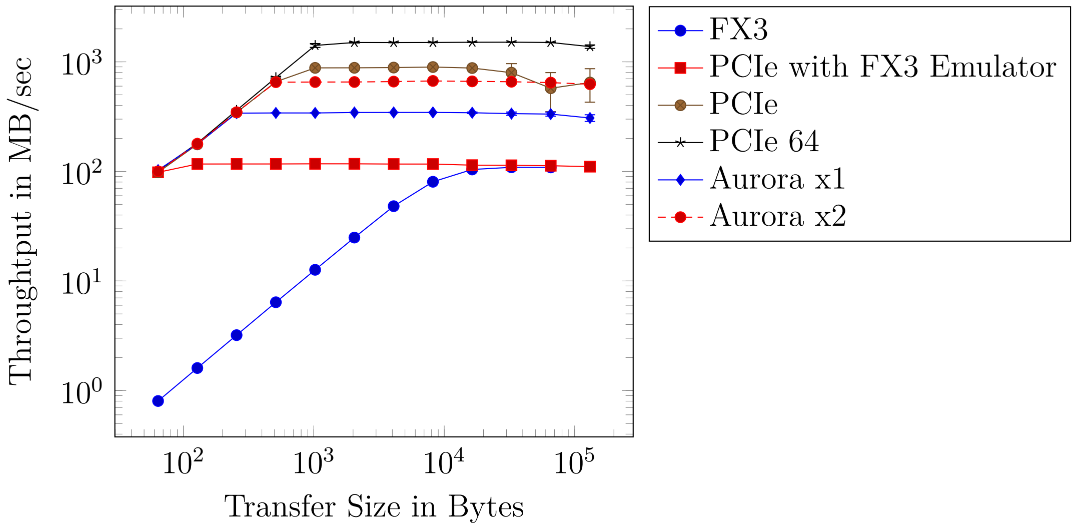


Figure 11.5: Vary size per function call (log Y axis).

of the GPIF interface is 32 bits $\times$ 100 MHz = 400 MB/s for both directions. Both directions share the 400 MB/s so each direction only gets 200 MB/s. By adding in communication overhead, the measured GPIF throughput of 117 MB/s in the PCIe FX3 emulator test seems reasonable.

The max throughput of the PCIe test setup is 896 MB/s. Since this max is much greater than the Aurora or PCIe with FX3 emulator tests, it can be inferred that PCIe was not the bottleneck in the other tests. The PCIe test shows the upper throughput limit with the PCIe implementation used in the communication board. If more bandwidth is needed, then 64 bit streams can be used. The max throughput of PCIe 64 is 1511 MB/s. In order to reach the max throughput, a transfer size of 1K and 2K is needed for PCIe and PCIe 64, respectively. All of the implementations have the same rate of change in the beginning region before the max is achieved. The PCIe implementations all have the same values. This means that the Xillybus PCIe bus transfer is the limiting factor and that the limit is the same for 64 bit as it is for 32 bit.

One lane of Aurora achieves a max throughput of 345 MB/s. Moving from one lane to two lanes roughly doubles the max throughput to 669 MB/s. Aurora should maintain this trend as more lanes are added up until its throughput starts to match the limits of the PCIe implementation. There is an interesting artifact in the data as the PCIe transfer size starts to exceed 32768 bytes. The throughput starts to drop and the variance in the data greatly increases. This could be caused by exceeding the size of the buffers on the communication board or from exceeding the size of the host DMA buffers. PCIe 64 shows a similar dip in performance but the change happens with a larger transfer size.

The DANNA array is currently operating at 1 MHz. Given that input packets are 36 bytes and output packets are 64 bytes, the max throughput requirement of DANNA is 100 MB/s. When running communication stress tests with the FX3 and the emulated FX3, both setups dropped output packets some of the time, but would also pass the tests some of the time. Since the max throughput of both of these setups is around 100 MB/s, the observed behavior makes sense. If there is any slowdown at all in the communication, the communication path would no longer be able to supply the bandwidth needed by DANNA, resulting in packet drops. The Aurora communication setup passed all of the communication tests. This also seems reasonable since it is able to supply a little over three times the max required bandwidth.

From this graph, many helpful conclusions can be made. Firstly, the new communication board has room to scale. If the single lane Aurora limit is reached, then two or more lanes can be used. If the PCIe limit is reached, 64 bit PCIe can be used. Once the 64 bit PCIe limit is reached, Revision XL can be used. The new communication setup can scale far beyond the limits of the FX3 and GPIF interface. Secondly, the max throughput is only reached when large blocks are transferred at a time, with the sweet spot seeming to be 1 KB of data.

Figures 11.6 and 11.5 both show the same benchmark data. This benchmark varied the total transfer size but kept the transfer size to one packet (64 bytes). Figure 11.5 has a log scale on both axes whereas Figure 11.6 only has a log scale on the x-axis. The purpose of this test is to show the maximum performance that can be expected from processing one packet at a time. All of the test setups that used PCIe have the exact same performance. The max throughput achieved is 100 MB/s. This means that the performance bottleneck is in the multiple transfer calls with the Xillybus driver and not in Aurora or the FX3 emulator. The performance of the FX3 is even worse, maxing out at 1 MB/s. These results show that transfer calls have to include multiple packets in order to achieve high throughput.

11.3 Aurora Footprint

The space used by Aurora is very comparable to the space used by the GPIF state machine. On a 10 by 10 array the utilization on the 690T with the GPIF interface is 6.20% of the slices. The same DANNA size with Aurora used 6.69%. On a 45 by 45 DANNA array the space used with the GPIF interface is 99.09%. The same DANNA size with Aurora used 99.29%. The same DANNA sizes were able to be generated as when using the GPIF interface. The communication board uses 7.71% of its slices to implement PCIe and Aurora. The communication board loop back design, which uses PCIe but not Aurora uses 7.01% of its slices. This means that in both cases Aurora used less than 1% of the FPGA's resources.

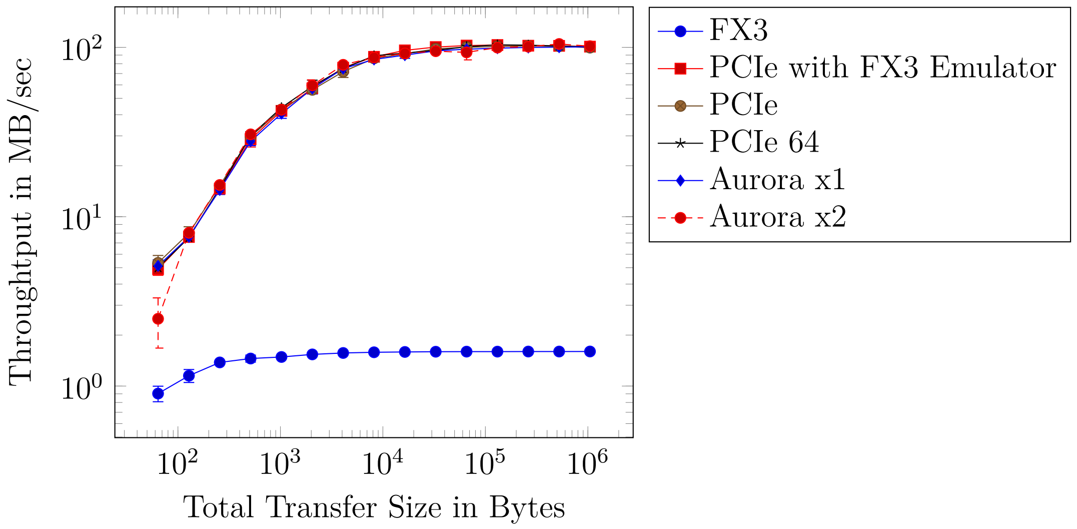


Figure 11.6: Vary total transfer size.

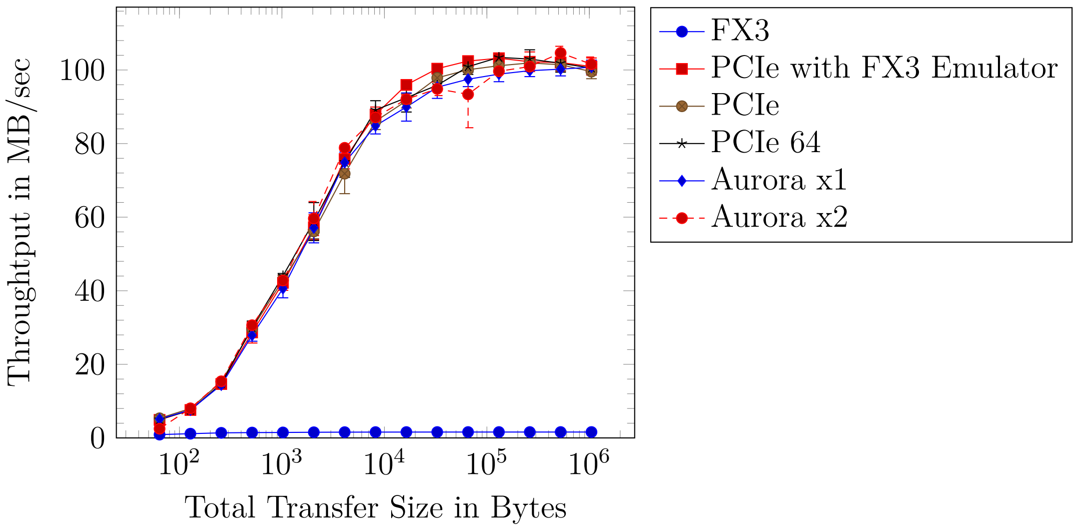


Figure 11.7: Vary total transfer size (log Y axis).

Chapter 12

Future Work

Now that the new communication system has been built and shown to scale successfully, work can be done to build larger systems. The first step is to implement retransmission on errors for the Aurora channel. Aurora already has a built in CRC checker to detect when errors occur, but it does not have a method to automatically retransmit the data when the CRC checker detects an error. Such a feature would be implemented in the transport layer. Aurora is a link layer protocol. Testing has shown that errors occur very infrequently, but creating a transport layer around Aurora, which implements retransmission of errors, would make the communication system more robust.

Currently the system has only been tested with the FMC connector directly. More future work would include using an FMC to SMA adaptor and transmitting the data using SMA cables. SMA cables will allow more flexibility in how the Aurora lanes are connected to different DANNA FPGA boards. It will also support easy connection of multiple DANNA FPGA boards.

The next step is to start scaling the communication design. The first level of scaling is to have multiple DANNA arrays working together on the same FPGA. Next, multiple DANNA FPGAs can be connected together, each potentially running multiple DANNA arrays. The communications board will have to be able to connect to each of the DANNA FPGAs and have the ability to run all of the DANNA arrays as one large array or as multiple parallel smaller arrays. The DANNA arrays can also be implemented in VLSI resulting in the ability to make arrays as large as 1000×1000 and capable of running at 100 MHz. The next point of

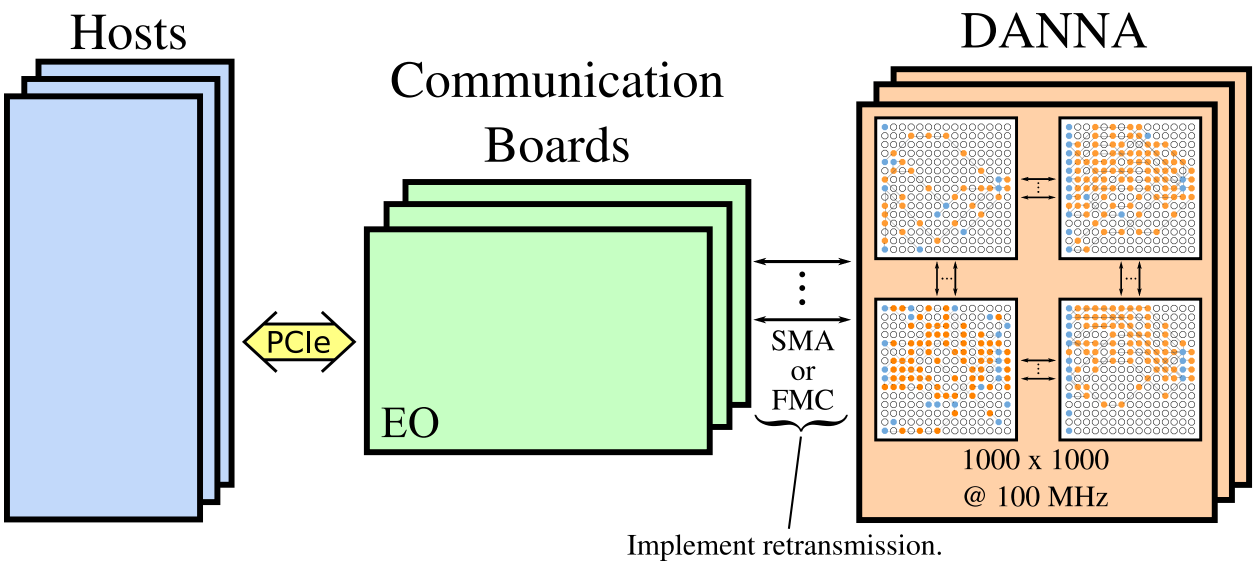


Figure 12.1: Future scaling.

scaling would be to start using multiple communication boards, each with multiple DANNA FPGAs attached to it, all connected to the same host. Finally, the system could scale into a neuromorphic supercomputer by adding multiple hosts, each with multiple communication boards, each with multiple DANNA FPGAs. Each host should be able to coordinate with all the other hosts via Ethernet or some other communication channel. Figure 12.1 shows a diagram representing the future work. Multiple boards can be stacked together to implement a large neuromorphic system.

Chapter 13

Conclusion

A new communications system for neuromorphic devices was designed and its performance measured. It was shown to out-perform the prior FX3-based communications setup and can be used to scale-up to communicate with multiple FPGAs simultaneously. PCIe is used to connect the host machine to the communication board and Aurora is used to connect the communication board to the DANNA FPGA boards. Both have been benchmarked and shown to have much higher throughput and lower round trip latency than the FX3 communication setup. The new communication board offers more flexibility, both in terms of the ease at which the communication packet structure can be modified and in terms of how FPGAs can be connected. The new communications setup should be able to scale to meet future needs of the neuromorphic research group at the University of Tennessee so that the communications setup is no longer the bottleneck preventing the scaling of neuromorphic arrays.

Bibliography

[1] Arnon Amir et al. “Cognitive computing programming paradigm: a corelet language for composing networks of neurosynaptic cores”. In: *Neural Networks (IJCNN), The 2013 International Joint Conference on*. IEEE. 2013, pp. 1–10 (cit. on p. 7).

[2] ARM. *AMBA 4 AXI4-Stream Protocol. Specification*. Version 1.0. ARM. 2010 (cit. on p. 43).

[3] Abhijit Athavale and Carl Christensen. *High-Speed Serial I/O Made Simple*. Apr. 2005. URL: https://www.xilinx.com/publications/archives/books/serialio.pdf (visited on 06/07/2017) (cit. on pp. 26, 27, 29).

[4] Ben Varkey Benjamin et al. “Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations”. In: *Proceedings of the IEEE* 102.5 (2014), pp. 699–716 (cit. on p. 3).

[5] H. K. O. Berge and P. Hafliger. “High-Speed Serial AER on FPGA”. In: *2007 IEEE International Symposium on Circuits and Systems*. May 2007, pp. 857–860. DOI: 10.1109/ISCAS.2007.378041 (cit. on p. 2).

[6] K. A. Boahen. “Point-to-point connectivity between neuromorphic chips using address events”. In: *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing* 47.5 (May 2000), pp. 416–434. ISSN: 1057-7130. DOI: 10.1109/82.842110 (cit. on p. 2).

[7] Kwabena Boahen. “Neurogrid: emulating a million neurons in the cortex”. In: *Conf. Proc. IEEE Eng. Med. Biol. Soc.* 2006, p. 6702 (cit. on p. 2).

[8] Andrew S Cassidy et al. “Cognitive computing building block: A versatile and efficient digital neuron model for neurosynaptic cores”. In: *Neural Networks (IJCNN), The 2013 International Joint Conference on*. IEEE. 2013, pp. 1–10 (cit. on pp. 7, 8).

[9] G. Chakma et al. “A Hafnium-Oxide Memristive Dynamic Adaptive Neural Network Array”. In: *International Workshop on Post-Moore’s Era Supercomputing (PMES)*. Salt Lake City, UT, Nov. 2016 (cit. on p. 10).

[10] Jason Chan. “Implementation of a Neuromorphic Development Platform with DANNA”. MA thesis. University of Tennessee, 2015 (cit. on pp. 11, 14, 18).

[11] M. E. Dean, C. D. Schuman, and J. D. Birdwell. “Dynamic Adaptive Neural Network Array”. In: *13th International Conference on Unconventional Computation and Natural Computation (UCNC)*. London, ON: Springer, July 2014, pp. 129–141 (cit. on pp. 10, 17, 21).

[12] M. E. Dean et al. “An Application Development Platform for Neuromorphic Computing”. In: *International Joint Conference on Neural Networks*. Vancouver, July 2016 (cit. on p. 19).

[13] Manaskant Desai and Karthik Sivaramakrishnan. *Optimizing Usb 3.0 Throughput With Ez-Usb Fx3*. AN86947. Version C. May 9, 2017 (cit. on pp. 23, 57, 59).

[14] D. B. Fasnacht, A. M. Whatley, and G. Indiveri. “A serial communication infrastructure for multi-chip address event systems”. In: *2008 IEEE International Symposium on Circuits and Systems*. May 2008, pp. 648–651. DOI: 10.1109/ISCAS.2008.4541501 (cit. on p. 2).

[15] Stephen Furber and Andrew Brown. “Biologically-inspired massively-parallel architectures-computing beyond a million processors”. In: *Application of Concurrency to System Design, 2009. ACSD’09. Ninth International Conference On*. IEEE. July 2009, pp. 3–12. DOI: 10.1109/ACSD.2009.17. URL: https://eprints.soton.ac.uk/270985/1/PID871138.pdf (cit. on pp. 4, 5).

[16] Paul Merolla et al. “A multicast tree router for multichip neuromorphic systems”. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 61.3 (2014), pp. 820–833 (cit. on p. 2).

[17] Dharmendra S. Modha. *Introducing a Brain-inspired Computer. TrueNorth’s neurons to revolutionize system architecture*. 2017. URL: http://www.research.ibm.com/articles/brain-chip.shtml (visited on 07/04/2017) (cit. on p. 7).

[18] W. Olin-Ammentorp et al. “Applying Memristors Towards Low-Power, Dynamic Learning for Neuromorphic Applications”. In: *42nd Annual GOMACTech Conference*. Reno, NV, Mar. 2017 (cit. on p. 10).

[19] Robert Preissl et al. “Compass: A scalable simulator for an architecture for cognitive computing”. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press. 2012, p. 54 (cit. on p. 7).

[20] J. Schemmel et al. “A wafer-scale neuromorphic hardware system for large-scale neural modeling”. In: *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*. May 2010, pp. 1947–1950. DOI: 10.1109/ISCAS.2010.5536970 (cit. on p. 5).

[21] Stefan Scholze et al. “A 32GBit/s communication SoC for a waferscale neuromorphic system”. In: *Integration, the VLSI Journal* 45.1 (2012), pp. 61–75 (cit. on pp. 5, 6).

[22] Stefan Scholze et al. “VLSI Implementation of a 2.8 Gevent/s Packet-Based AER Interface with Routing and Event Sorting Functionality”. In: *Frontiers in Neuroscience* 5 (2011), p. 117. ISSN: 1662-453X. DOI: 10.3389/fnins.2011.00117. URL: http://journal.frontiersin.org/article/10.3389/fnins.2011.00117 (cit. on pp. 5, 6).

[23] C. D. Schuman. “The Effect of Biologically-Inspired Mechanisms in Spiking Neural Networks for Neuromorphic Implementation”. In: *IJCNN: The International Joint Conference on Neural Networks*. Anchorage, May 2017 (cit. on p. 9).

[24] C. D. Schuman and J. D. Birdwell. “Dynamic Artificial Neural Networks with Affective Systems”. In: *PLoS ONE* 8.11 (Nov. 2013), e80455. DOI: 10.1371/journal.pone.0080455 (cit. on p. 9).

[25] C. D. Schuman and J. D. Birdwell. “Variable Structure Dynamic Artificial Neural Networks”. In: *Biologically Inspired Cognitive Architectures* 6 (Oct. 2013), pp. 126–130. DOI: 10.1016/j.bica.2013.05.001 (cit. on p. 9).

[26] Catherine D. Schuman. “Neuroscience-Inspired Dynamic Architectures”. PhD thesis. University of Tennessee, May 2015 (cit. on pp. 1, 9, 10).

[27] Thomas Sharp et al. “Power-efficient simulation of detailed cortical microcircuits on SpiNNaker”. In: *Journal of neuroscience methods* 210.1 (2012), pp. 110–118 (cit. on pp. 3, 4).

[28] Stanford University. *NeuroGrid*. 2006. URL: https://web.stanford.edu/group/brainsinsilicon/challenge.html (cit. on p. 2).

[29] Rama Sai Krishna V. *Designing with the EZ-USB FX3 Slave FIFO Interface*. AN65974. July 17, 2017 (cit. on p. 18).

[30] Wandboard. *Wandboard*. 2017. URL: https://www.wandboard.org/ (cit. on p. 22).

[31] J. C. Willis. "Middleware and Services for Dynamic Adaptive Neural Network Arrays". MA thesis. University of Tennessee, 2015 (cit. on pp. 17, 18).

[32] Xilinx. *7 Series FPGAs GTX/GTH Transceivers*. UG476. Version v1.12. Xilinx. Dec. 19, 2016. URL: https://www.xilinx.com/support/documentation/user_guides/ug476_7Series_Transceivers.pdf (cit. on p. 26).

[33] Xilinx. *Aurora 8B/10B. LogiCORE IP Product Guide*. PG046. Version 11.0. Xilinx. Oct. 5, 2016. URL: https://www.xilinx.com/support/documentation/ip_documentation/aurora_8b10b/v11_0/pg046-aurora-8b10b.pdf (visited on 06/14/2017) (cit. on pp. 39–42).

[34] Xilinx. *Aurora 8B/10B Protocol Specification*. SP002. Version 2.3. Xilinx. Oct. 1, 2014. URL: https://www.xilinx.com/support/documentation/ip_documentation/aurora_8b10b_protocol_spec_sp002.pdf (visited on 06/14/2017) (cit. on p. 39).

[35] Xilinx. *AXI Reference Guide*. UG761. Version 13.1. Xilinx. Mar. 7, 2011. URL: https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf (visited on 06/14/2017) (cit. on p. 42).

[36] Xillybus Ltd. *IP core product brief*. Feb. 16, 2017. URL: http://xillybus.com/downloads/xillybus_product_brief.pdf (visited on 06/12/2017) (cit. on pp. 32, 33).

[37] Xillybus Ltd. *Principle of Operation*. Xillybus Ltd. 2017. URL: http://xillybus.com/doc/xilinx-pcie-principle-of-operation (visited on 06/12/2017) (cit. on p. 32).

[38] Xillybus Ltd. *Revision B/XL user notes*. 2017. URL: http://xillybus.com/doc/revision-b-xl (visited on 06/12/2017) (cit. on p. 35).

[39] Xillybus Ltd. *The Custom IP Core Factory*. 2017. URL: http://xillybus.com/custom-ip-factory (visited on 06/12/2017) (cit. on p. 34).

[40] Xillybus Ltd. *Xillybus host application programming guide for Linux*. Version 2.2. 2017. URL: http://xillybus.com/downloads/doc/xillybus_host_programming_guide_linux.pdf (visited on 06/12/2017) (cit. on p. 34).

Appendix

A Abbreviations and Symbols

AER	Address-Event Representation
AMBA	Advanced Microcontroller Bus Architecture
ARM	Advanced RISC Machine
ASIC	Application-Specific Integrated Circuit
AXI	Advanced eXtensible Interface
CRC	Cyclic Redundancy Check
DANNA	Dynamic Adaptive Neural Network Arrays
DMA	Direct Memory Access
EO	Evolutionary Optimization
EZ-USB FX3	Cypress USB 3.0 Peripheral Controller
FIFO	First In, First Out
FMC	FPGA Mezzanine Card
FPGA	Field-Programmable Gate Array
GPIF	General Programmable Interface
HPC	High Pin Count
IP	Intellectual Property
NIDA	Neuroscience-Inspired Dynamic Architecture
OS	Operating System
PCIe	Peripheral Component Interconnect Express
PC	Personal Computer
SATA	Serial ATA
SDK	Software Development Kit
SERDES	Serializer/Deserializer
SMA Connector	SubMiniature Version A Connector
USB	Universal Serial Bus
VLSI	Very-Large-Scale Integration

Vita

Aaron Reed Young is from Knoxville, Tennessee. He graduated from Hardin Valley Academy in 2012 and then began his studies at the University of Tennessee, Knoxville, in the pursuit of a Bachelor of Science degree in Computer Engineering from the Department of Electrical Engineering and Computer Science in the College of Engineering. During the summers he completed internships at Siemens Medical Systems, Oak Ridge National Laboratory's Manufacturing Demonstration Facility, and Garmin International. He graduated summa cum laude, with his Bachelor of Science degree in Computer Engineering, Chancellor's Honors and Electrical Engineering and Computer Science Honors Programs in May of 2016. He earned his Master of Science degree, also in Computer Engineering, in August 2017. Aaron is continuing his graduate education at the University of Tennessee in pursuit of a Doctor of Philosophy.