

SNACC: The Scaled-up Neuromorphic Array Communications Controller

A Dissertation Presented for the
Doctor of Philosophy
Degree
The University of Tennessee, Knoxville

Aaron Reed Young

May 2020

© by Aaron Reed Young, 2020
All Rights Reserved.

Acknowledgments

I would like to thank the Department of Electrical Engineering and Computer Science's Bodenheimer Fellowship (especially Dr. Robert Bodenheimer and Mr. Mike Crabtree) for providing me with funding to continue pursuing higher education. I would also like to thank the Department of Electrical Engineering and Computer Science and TENNLab for allowing me to serve as a teaching assistant and research assistant throughout my time as a graduate student.

Much appreciation to Dr. Mark Dean for his continual support, feedback, and guidance throughout my time in graduate school, and also many thanks to the other members of my committee, Dr. James Plank, Dr. Garrett Rose, and Dr. Andy Sarles for their insight and support. I would like to thank the entire TENNLab team for being amazing co-researchers, for helping me identify a dissertation research area, and for the continued support of my work. I would especially like to thank my wife Courtney for her continual support of my education ever since we met in high school.

This material is based in part on research sponsored by The University of Tennessee (UT) Science Alliance Joint Directed Research and Development Program; and by the U.S. Department of Energy, Office of Science, and Office of Advanced Scientific Computing Research, under contract number DE-AC05-00OR22725. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Research Laboratory or the U.S. Government.

Abstract

Neuromorphic computers take inspiration from the brain to perform computation in a way similar to how a brain works. They perform computation by evaluating Spiking Neural Networks (SNN) to model how the electronic pulses travel through a collection of neurons. The neurons in the network spike, or fire a pulse, once the accumulated inputs to the neuron exceed a specific threshold. Field-Programmable Gate Arrays (FPGAs), Application-Specific Integrated Circuits (ASICs), and other chip/multi-chip level implementations can be used to construct Dynamic Adaptive Neural Network Arrays (DANNA) and its successor DANNA2. In many use cases, the neuromorphic hardware is connected to a traditional computing system that is used to load neural network configurations, provide network input, process network output, and monitor the status of the network. To ensure robust communication, a custom, hardware-based, go-back- n , automatic repeat request protocol is presented, which allows for high-throughput, low-latency, error-free communication using the Aurora link-level protocol over the GTX/GTH high-speed serial transceivers found on Xilinx FPGAs. Multiple DANNA2 element cores are tiled into a grid array and placed within a KCU1500 Kintex Ultrascale FPGA to build a reconfigurable hardware neuromorphic processor. For resource-constrained environments, these element cores can also be densely packed onto the FPGA for a specific network, requiring fewer resources for a non-reconfigurable neuromorphic processor. For high-performance computation, multiple reconfigurable neuromorphic processors with grid arrays are tiled together with a Neuromorphic Array Communication Controller (NACC) to build a large-scale neuromorphic system called Scaled-up NACC (SNACC). SNACC uses scalable, high-performance, point-to-point connections to network the neuromorphic processors into a two-dimensional array. The neuromorphic processors are connected back to the host PC through a hierarchical, high-speed network made possible through the use of one

or more NACCs. These new hardware DANNA2 neuromorphic processors are used to further research with recurrent spiking neural networks (RSNNs). Specifically, this work uses the new hardware for evolutionary optimization of neural networks using genetic algorithms, for reservoir computing, and for solving graph algorithms. Additionally, the hardware can be used for real-time processing as demonstrated with target acquisition and obstacle avoidance on a ground-roaming autonomous robot.

Table of Contents

1	Introduction	1
1.1	Scaled-up Neuromorphic Array Communications Controller	4
1.2	Outline	6
2	Related Work	7
2.1	Neuromorphic Projects Overview	8
2.1.1	Stanford University—Neurogrid & Braindrop	8
2.1.2	Human Brain Project—BrainScaleS & SpiNNaker	9
2.1.3	TrueNorth	11
2.1.4	Loihi	14
2.1.5	Darwin	14
2.1.6	Dynap-SEL	15
2.2	SNN Packets	16
2.2.1	AER Representation	17
2.2.2	Routing Method	18
2.2.3	Asynchronous Circuits	22
2.2.4	Packet Structure	23
2.2.5	Event-Based Sensors and Actuators	24
2.3	Local Communication Challenges	25
2.4	Global Communication Challenges	26
2.4.1	Tree	27
2.4.2	Large Grid	27
2.4.3	Hybrid	30

2.4.4	Robustness in Larger Communication	30
2.4.5	Host Connection	34
2.4.6	Synchronization	34
2.5	Scalability	36
2.6	Summary	39
3	Previous Work	42
3.1	TENNNLab Overview	42
3.1.1	Applications	42
3.1.2	Learning	44
3.1.3	Neuromorphic Processors	46
3.1.4	Summary	52
3.2	Biologically-Realistic Versus Biologically-Inspired Computers	53
3.3	DANNA2	53
3.3.1	Model	53
3.3.2	Hardware Architecture	56
3.3.3	Element Control	59
3.3.4	Array Control & Communication Interface	60
3.3.5	Simulator	64
3.3.6	FPGA Implementation	66
3.4	Communication with DANNA	66
3.4.1	PCIe with Xillybus	70
3.4.2	Aurora	72
3.4.3	AXI4-Stream	73
3.5	Tiled DANNA	75
4	Goals	83
5	Tools	85
5.1	Vivado	85
5.2	Xilinx	86

5.3	Texas Instruments	86
5.4	Xillybus	86
5.5	GHDL	86
5.6	VUnit	89
5.7	DANNA2 Network Visualizer	89
6	DANNA2 on a Single FPGA	92
6.1	Existing Work	92
6.2	New Design	96
6.2.1	Hardware Design	96
6.2.2	Build System Design	98
6.2.3	Driver Design	101
6.3	Verification	103
7	SNACC	104
7.1	DANNA2	106
7.2	NACC	108
7.3	Communications	110
7.3.1	ACK	111
7.3.2	Local Communications Between Sub-arrays	130
7.3.3	NACC and Sub-array	153
7.3.4	Host and NACC	154
7.4	Host	155
7.4.1	Build System Design	156
7.4.2	Driver Design	158
7.5	Verification	160
7.6	Power	165
7.7	Scalability	167
7.7.1	Size per Sub-array	167
7.7.2	Size per NACC	169
7.7.3	Size per SNACC	170

8	Applications and Performance	174
8.1	Performance Measurements	174
8.2	EONS	197
8.2.1	GenGraph support for SNACC	197
8.2.2	Performance	199
8.2.3	Streaming Training	202
8.3	Reservoir	204
8.4	New DANNA2 Features to Support Graph Algorithms	208
8.4.1	Requirements to Find the Shortest Path	210
8.4.2	Adding Element Reading	212
8.4.3	Testing and Debugging Hardware STDP	219
8.4.4	Adding Infinite Refractory Period	222
8.5	Evaluating Graph Algorithms using DANNA2 Networks	224
8.5.1	Results	226
8.5.2	Limitations and Conclusion	230
8.6	Real-time Network Evaluation	237
8.6.1	GRANT	238
8.6.2	Design	240
8.6.3	Results	244
8.6.4	Conclusion	254
9	Accomplishments	255
9.1	Literature Review	255
9.2	Aurora ACK	255
9.3	DANNA2	256
9.4	DANNA2 Visualizer	256
9.5	Single-board FPGA Implementation	256
9.6	SNACC	257
9.7	Build System	257
9.8	Applications	257

10 Future Work	259
10.1 Additional Neuromorphic Applications	259
10.2 Move SNACC Routing from Host to NACC	260
10.3 DANNA2 Array Sub-partitioning	261
10.4 SNACC Sparse Array Support	261
10.5 Increase Density and Performance	262
10.6 Streaming EO	263
11 Conclusion	264
Bibliography	265
Appendix	280
A Abbreviations and Symbols	281
B Pseudocode	283
B.1 Go-Back-N	283
B.2 Window Buffer	284
B.3 Aurora ACK Send	284
B.4 Aurora ACK Receive	287
Vita	289

List of Tables

2.1	Summary of neuromorphic hardware communication systems	41
3.1	Table of Programmable Element Parameters	62
3.2	Xillybus revision summary	72
3.3	AXI4-Stream interface signals	74
3.4	Tiled DANNA Clocks	78
6.1	IP Core Factory Device Files for Xillybus core Revision XL for Xilinx Kintex Ultrascale	97
7.1	Parameters for the SNACC simulator	138
7.2	Scaling Summary	171
8.1	Maximum Events per Cycle Where the Simulator is Fastest	193
8.2	Input Packet Opcodes	215
8.3	Output Packet Opcodes	215
8.4	STDP Lookup Table Values	221
8.5	Maze 40x20 Shortest Path Finding Time	228
8.6	City Map 40x20 Shortest Path Finding Time	229
8.7	Average Decision Update Times for Different Test Runs	244
8.8	GRANT Application Results	251

List of Figures

1.1	High-level overview of SNACC.	5
2.1	Neurogrid circuit board with 16 packaged Neurocore chips.	10
2.2	Diagram of BrainScaleS system.	12
2.3	Photo of the SpiNNaker circuit board.	13
2.4	Darwin chip die and demonstration PCB board.	15
2.5	The two-stage routing scheme used by DYNAPs.	21
2.6	Flow of information from sensors through a neural network to actuators. . .	25
2.7	A BrainScaleS wafer with 56 complete reticles.	28
2.8	Neurogrid routing tree.	28
2.9	Diagram of BrainScaleS’s hierarchical L2 routing.	29
2.10	The SpiNNaker system network connectivity.	29
2.11	TrueNorth cross-chip connectivity.	31
2.12	DYNAPs hybrid hierarchical-mesh routing scheme example.	32
2.13	TrueNorth NS1e, NS1e-16, and NS16e systems.	33
2.14	Barrier sync operation of Loihi.	35
2.15	The NS16e-4 scaled-up TrueNorth evaluation system.	38
2.16	Graphical summary of neuromorphic hardware communication systems. . . .	40
3.1	TENNNLab software stack.	43
3.2	An overview of the EONS algorithm.	46
3.3	DANNA element connectivity.	48
3.4	Monitoring capture-shift diagram.	48
3.5	DANNA2 element connectivity.	50

3.6	DANNA2 example networks.	51
3.7	The main loop of an application running in the TENNLab framework on a neuromorphic processor.	52
3.8	Diagram of a DANNA2 element.	57
3.9	Depiction of DANNA2's element control.	59
3.10	Layout of the initial addressing packet sent over the programming bus. . . .	60
3.11	Layout of synapse programming information sent over programming bus. . .	60
3.12	Layout of neuron programming information sent over programming bus. . . .	61
3.13	Configuration Packet Description.	62
3.14	Step and Fire Packet Description.	63
3.15	Reset Packet Description.	63
3.16	Output Fire Packet Description.	64
3.17	DANNA2 simulation flow chart.	65
3.18	Top-level interface for the DANNA2 array.	67
3.19	Schematic of DANNA2 element generated by Xilinx Vivado.	68
3.20	Diagram of the FX3 communication setup.	69
3.21	Diagram of the communications board.	70
3.22	Simplified FPGA block diagram of Xillybus using PCIe transport with host interface block.	71
3.23	Aurora 64B/66B channel overview	73
3.24	AXI4-Stream interface wiring diagram.	74
3.25	Examples of valid AXI4-Stream handshakes.	76
3.26	Tiled DANNA Structure.	77
3.27	Picture of the Tiled DANNA prototype.	78
3.28	Host to Tiled DANNA Communication Interface.	79
3.29	Tiled DANNA tile-to-tile communication interface.	80
3.30	Tiled DANNA Boundary Edge Element Nearest-Neighbor Connections. . . .	82
3.31	Fire event multiplexing and demultiplexing across the tile boundary.	82
5.1	Images of the Xilinx FPGA Boards used.	87

5.2	LMK03328EVM Ultra-Low-Jitter Clock Generator EVM with 2 PLLs, 8 differential outputs, and 2 inputs	88
5.3	An example network with labels displayed by the DANNA2 Viz.	90
5.4	Legend of the DANNA2 Viz.	91
5.5	DANNA2 Viz with a selected element.	91
6.1	Top-level design for DANNA2 single board communication.	93
6.2	Picture of the single FPGA DANNA2 hardware.	94
6.3	DANNA2 grid array design for a 2×2 grid.	95
6.4	Simplified schematic of a DANNA2 Element.	97
6.5	Directory structure of the DANNA2 build system.	99
6.6	DANNA2 build script settings.	100
6.7	Diagram of the TENNLab Framework with the DANNA2 Processor.	100
6.8	DANNA2 Device Virtual Functions.	102
7.1	High-level overview of SNACC.	105
7.2	NACC Design.	109
7.3	SNACC Communication Types.	110
7.4	Diagram showing how Aurora ACK is used.	116
7.5	Diagram showing the testing architectures for Aurora ACK.	117
7.6	Schematic showing the components of Aurora ACK.	118
7.7	Simplified diagram showing the design of Aurora ACK.	118
7.8	Aurora ACK header structure.	119
7.9	Top-level interface for the window buffer.	121
7.10	Design of the window buffer.	122
7.11	Round trip time comparison.	126
7.12	Throughput experiment, varying the amount of data transmitted per function call.	128
7.13	Logical view of a simulated fake DANNA2 chip.	134

7.14	Delay FIFO is used to move the synaptic delay from inside the element to make it part of the communications channel, thereby hiding the communication channel's latency.	135
7.15	Logical view of a simulated SNACC.	137
7.16	SNACC simulator post-simulate statistics.	137
7.17	SNACC Simulator visualization.	140
7.18	Host bottlenecked array state.	141
7.19	Top-level interface for the DANNA2 multi-array.	143
7.20	Top-level design for multi-array DANNA2.	145
7.21	S Clock Domain State Machine used to implement the AXI4-Stream Clock Converter.	147
7.22	Top-level design for the AXI4-Stream Clock Converter.	148
7.23	Single-bit n -stage synchronizer used in the AXI4-Stream Clock Converter.	149
7.24	Set and Clear Toggle Flip-Flop (TFF) design used in the AXI4-Stream Clock Converter.	150
7.25	Diagram of the extended fire array.	151
7.26	Picture of the previous DANNA communication setup hardware.	154
7.27	Picture of the SNACC hardware setup.	155
7.28	Directory structure for the SNACC build system.	157
7.29	Diagram of the TENNLab Framework with the DANNA2 SNACC device.	159
7.30	Flowchart of SNACC Driver Threads.	160
7.31	Debug probes showing the correct behavior of the max flood control.	162
7.32	Added delay verses average element clock cycle frequency.	165
7.33	Scaled up SNACC System.	172
8.1	Example passthrough, snake, and loop networks with a height of ten.	175
8.2	Network split to load DANNA2 arrays onto the SNACC system.	177
8.3	Network evaluation time comparison between select size sweeps for hardware and software for 10k cycles.	180
8.4	Simulator performance for a varying number of events.	182

8.5	Per cycle simulator performance for a varying number of events per cycles. . .	183
8.6	Time per simulation event for a varying number of events.	183
8.7	Time for the hardware device evaluation for a varying number of events. . .	184
8.8	Single-board DANNA2 processor evaluation time for a varying number of input fires.	185
8.9	SNACC DANNA2 processor evaluation time for a varying number of input fires.	185
8.10	The average time per cycle for each device when run for a given number of cycles.	187
8.11	The average time per cycle for each hardware device when run for a given number of cycles greater than or equal to 100k.	187
8.12	The effect the number of inputs has on single-board FPGA evaluation time for passthrough networks with fires every cycle	188
8.13	The effect the number of inputs has on SNACC evaluation time for passthrough networks with fires every cycle	188
8.14	The winners for evaluating networks for select network size sweeps for various array sizes and input activities.	189
8.15	Graphs showing the winning device type based on key variables for a varying number of cycles between 1 k and 50 k.	192
8.16	Fastest evaluation method for all data.	193
8.17	Fastest evaluation method for all data (zoomed).	194
8.18	Fastest evaluation method for non-loop networks.	194
8.19	Fastest evaluation method for non-loop networks (zoomed).	195
8.20	Fastest evaluation method for loop networks.	195
8.21	Fastest evaluation method for loop networks (zoomed).	196
8.22	Evaluation time versus epoch number for the pole balancing application training.	200
8.23	Simulation call time for pole balancing (100 cycles)	200
8.24	Simulation call time for Robonav (100 cycles)	201
8.25	Simulation call time for Robonav (10 k cycles)	201
8.26	Diagram of reservoir computing.	205
8.27	Graph of reservoir training time for different DANNA2 devices.	207

8.28	Pseudocode for maze graph to DANNA2 network.	209
8.29	Pseudocode for city graph to DANNA2 network.	210
8.30	A maze and the resulting DANNA2 network used to find the length of the shortest path through the maze.	211
8.31	A city graph and the resulting DANNA2 network used to find the length of the shortest path through the city.	211
8.32	New DANNA2 element read input packet description.	213
8.33	New DANNA2 element state output packet description.	214
8.34	Diagram of the element communication chain.	217
8.35	Layout of the initial addressing packet over the element communication chain with added element read bit.	218
8.36	Plot of STDP lookup table values.	222
8.37	One-shot loop.	223
8.38	Layout of the neuron programming information sent over the element communication chain with additional neuron fields added.	223
8.39	Pseudocode to read out the shortest path from a DANNA2 network modified by STDP.	225
8.40	Maze shortest path solving time comparison for different methods with a 40×20 maze.	228
8.41	City map shortest path solving time comparison for different methods with a 40×20 city.	231
8.42	Maze shortest path solving time for graphs with different numbers of vertices (i.e. DANNA2 elements).	231
8.43	Maze shortest path solving time for graphs with different numbers of vertices (software implementations only).	232
8.44	Maze shortest path solving time for the ideal neuromorphic solution versus length of the shortest path.	232
8.45	Maze shortest path solving time for graphs of different shortest path lengths.	233
8.46	Maze shortest path solving time for graphs of different shortest path lengths (software implementations only).	233

8.47	City map shortest path solving time for graphs of different numbers of vertices.	234
8.48	City map shortest path solving time for graphs of different numbers of vertices (software implementations only).	234
8.49	The Ground-Roaming Autonomous Neuromorphic Targeter.	239
8.50	GRANT SoC design block diagram.	241
8.51	Network evaluation performance improvement as different approaches are tried.	245
8.52	Regular update time for local and external runs via Ethernet as measured by GRANT.	247
8.53	Network evaluation time for external runs via Ethernet as measured by the external host PC.	248
8.54	Regular update time for external runs via Wi-Fi as measured by GRANT.	249
8.55	Line plot of the regular update time samples via Wi-Fi as measured by GRANT.	250
8.56	Sensor update evaluation time for local and external runs via Ethernet as measured by GRANT.	251
8.57	Line plot of the sensor update evaluation time samples for local and external runs via Ethernet as measured by GRANT.	252
8.58	Neural network switch update evaluation time for local and external runs via Ethernet as measured by GRANT.	253

Chapter 1

Introduction

I praise you, for I am fearfully and wonderfully made. Wonderful are your works; my soul knows it very well. For you formed my inward parts; you knitted me together in my mother's womb.

– Psalms 139:13-14 (ESV)

The human brain is an amazing computational machine, with many properties that surpass the capabilities of modern supercomputers.¹ For comparison, contemporary processors operate in the multi-Gigahertz range with a power density of 100 W/cm² [1]. The human brain, on the other hand, which has unparalleled performance on cognitive and perceptual tasks, does so by running at an average firing rate of 10 Hz and with a very low power density of 10 mW/cm². The brain, which takes up a total size of 2 L, weighs 1.5 kg, and uses 20 W of energy, is difficult and inefficient to simulate using traditional computing architectures. Part of the difficulty in simulating the brain is that researchers disagree on the level of detail and which features must be modeled in order to successfully simulate the workings of the brain. Markram argues that detailed, biologically accurate models, based on first principles, are required to accurately model the cortex [55]. On the other hand, Izhikevich et al. argue that the cortex can be accurately modeled by a simple system of coupled differential equations [49]. Regardless, many groups are interested in scaling up their models of neural networks to model neurons and synapses at a biologically realistic scale. In order to scale

¹The content of this paragraph is published in an IEEE Access journal article titled “A Review of Spiking Neuromorphic Hardware Communication Systems” © 2020 IEEE [120], reprinted with permission.

Ananthanarayanan et al.’s simulation of Izhikevich neurons to a similar scale as a human brain using a supercomputer, Sharp et al. estimate that 2^{16} BlueGene processors would be needed, requiring 8.4 GW of power and running at a rate of “642 seconds for one second of simulation per Hz of spiking activity” [4, 93]. Clearly the brain is functioning dramatically different from traditional von Neumann architectures, requiring far less space and power to operate. The brain is made up of approximately 86 billion computational elements, called neurons, which all operate independently, forming a massively parallel computer [7, 1]. The brain’s unmatched performance comes from this immense parallelism, but each individual neuron is relatively simple on its own. The behaviour of neurons can be studied and understood as computational units independent from the other neurons. This allows each individual neuron to be viewed as a low-power biological core, analogous to a small Reduced Instruction Set Computer (RISC) core. These neurons can be evaluated with multiple models, ranging in biological realism from the simplified leaky integrate-and-fire model to more biologically realistic models like the Hodgkin-Huxley model [1, 53]. Neurons have a short-term memory in the form of the charge stored as the neuron’s membrane potential. Leak causes neurons to forget and their membrane potential to return back to the resting value. The brain is clever because of its connectivity [107]. Each neuron has 1,000–10,000 synaptic connections to other neurons, forming a local memory for the neuron [1]. Just as there are a wide range of neuron models, there are also many synaptic models, with disagreement on the level of detail needed in the model. Simplified models treat synapses as point-to-point connections with a weight value associated with the connection’s strength. This weight value can change and the magnitude of the weight can be viewed as information stored locally to the connected neurons. More complex synapse models include support for short-term plasticity as well as modeling the complex, dynamic, nonlinear, stochastic properties of biological synapses. At a high level, the brain-inspired, neuromorphic architectures have parallel processors (neurons), which only perform relatively simple operations, operate at low-frequency, and have binary output, and they have local, distributed memory, which is stored at the connection points to other elements in the form of the synaptic connections to other neurons. This architecture avoids the von Neumann bottleneck by colocating computation with memory, using simple

components with simple communication, and by exploiting inherent and scalable parallelism in operation [90].

The end of Moore’s law and the limits of traditional computing paradigms are starting to be reached, as the heat generated from silicon circuitry increases, and the feature size is reduced to just a few atoms across [103, 65]. As the limitations of our current technology are reached, researchers are expanding their research into alternative computer architectures like quantum [52], molecular [101], and chemical computing [32]. The field of neuromorphic computing has arisen to use insights from the operation of biological brains to further advance our human-made computers. Neuromorphic computing looks towards the brain for inspiration in how to continue improving computers so that they can share the low-power, real-time, parallel nature of the biological brain. Over the past few years, The Laboratory of Tennesseans Exploring Neural Networks (TENNNLab) group at the University of Tennessee and Oak Ridge National Laboratory have been exploring spike-based neuromorphic computing models. We have several different models, including a software-based Neuroscience-Inspired Dynamic Architecture (NIDA) model [87], a memristor-based Dynamic Adaptive Neural Network Arrays (mrDANNA) model [17], and two digital models, Dynamic Adaptive Neural Network Arrays (DANNA) [26] and its successor, DANNA2 [63]. DANNA and mrDANNA have been fabricated using a Very-Large-Scale Integration (VLSI) design [16, 15] and the digital DANNA and DANNA2 designs can be implemented using a Field-Programmable Gate Array (FPGA). In addition to designing neuromorphic hardware, our group also focuses on how to make use of these neuromorphic systems to perform different tasks. As our ability to make use of the software increased, so too has our desire to use networks of larger size. One of the main challenges with designing neuromorphic hardware is how to interact with and scale these systems. This dissertation focuses both on how to connect neuromorphic hardware to a host computer and also how to scale neuromorphic systems to increase their capacity. This work builds upon the previous work from the TENNNLab group and looks at how to scale the neuromorphic FPGA designs to larger sizes using intermediate communication boards to facilitate the communication. First, the robustness of the communication protocol was improved by adding error detection and correction to ensure error-free, robust communication between the host machine and the neuromorphic array. Additionally, this work covers the

tiling of DANNA2 cores on FGPAs to allow neuromorphic networks to be mapped across the cores. The main focus of this dissertation is evaluating and demonstrating the scaling potential of DANNA2 across multiple FPGA devices using a Neuromorphic Array Communications Controller (NACC). The larger scaled systems are called the Scaled-up Neuromorphic Array Communications Controller or SNACC. This work covers the design, implementation, and testing of a communications controller to scale DANNA2 neuromorphic networks to larger sizes.

1.1 Scaled-up Neuromorphic Array Communications Controller

The design and testing of a way to enable scaling of a single-chip DANNA2 design to a multi-chip, scaled-up DANNA2 system is the main focus of this work. Many of the components of this dissertation center around designing and testing the components of this system, and once the system was built, testing the system using various workloads and comparing the system with other solutions. The SNACC system is comprised of the neuromorphic sub-arrays, the NACC, communications systems, and the host. The key component which enables the scaling of the neuromorphic system is the NACC. This communication controller organizes communications between the host and the neuromorphic arrays, allowing the size of the neuromorphic arrays to grow to larger sizes as multiple processing chips are combined into a large, uniform array. A high-level overview of this design is shown in [Figure 1.1](#). This figure shows an overview of how the NACCs are used to connect the host machine to multiple neuromorphic network arrays. The host is a traditional von Neumann computer and can be used to configure the arrays, send input fires, and process output fires. NACC is a communications controller board in between the host and the neuromorphic arrays, which enables the connection between these two and enables scaling the system up to larger sizes. The use of a intermediate communications board allows for a simple yet effective approach to scaling, which has been shown to be effective in scaling up a neuromorphic system with local connectivity patterns.

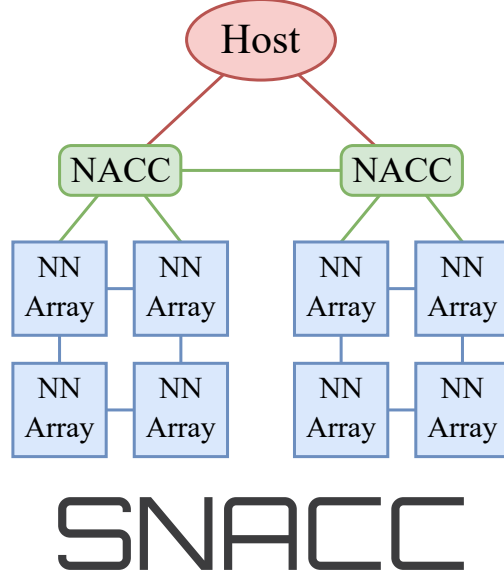


Figure 1.1: High-level overview of SNACC.

Two levels of scaling are shown in [Figure 1.1](#). The first level is the local tiling of the neuromorphic network arrays connected to a single NACC. The second level of scaling is obtained by connecting multiple NACCs together to scale the neuromorphic array further than what can be achieved with a single NACC. A third level of scaling is also possible where multiple host systems are connected together to form a neuromorphic computing cluster. The third level of scaling would be a looser scaling than the other two-types and would likely not be able to support running a large unified array due to a communication bottleneck between the host machines. The lines in this figure represent the communication channels between the components.

The use of NACC to scale individual neuromorphic processors into a large, uniform, multiprocessor array was tested by building the Scaled-up Neuromorphic Array Communications Controller (SNACC) system. This system was built using Field-Programmable Gate Arrays (FPGAs) to test and verify the design. Although the communication system setup is designed and tested on FPGAs, further improvements can be obtained by designing either the communications controller or the DANNA2 arrays in VLSI.

1.2 Outline

This work is structured into multiple sections organized as follows. [Chapter 2](#) provides an overview of relevant related work. In particular, this chapter covers the communication design of multiple well-known neuromorphic systems. The chapter also looks at challenges and considerations common to all spiking communication systems. [Chapter 3](#) summarizes relevant prior work from the TENNLab research group at the University of Tennessee. An overview of the TENNLab group is presented, along with an introduction to the DANNA2 neuromorphic processor, prior communication methods to the DANNA neuromorphic processor, and work on Tiled DANNA. [Chapter 4](#) discusses the goals this work seeks to accomplish. [Chapter 5](#) covers all of the programs, utilities, and hardware used in this work to build SNACC. [Chapter 6](#) discusses the work done to run DANNA2 on hardware using a single FPGA. [Chapter 7](#) discusses the design of SNACC. SNACC's design is subdivided into its three major components: neuromorphic arrays, neuromorphic array communications controller, and the host system. This chapter also covers the design of the communication system between these three main components, and details the high-performance, go-back- n , custom, automatic repeat request protocol. [Chapter 8](#) uses the neuromorphic hardware for multiple different applications and compares the use of the hardware with the use of the software simulators. [Chapter 9](#) summarizes the work completed as part of this dissertation. [Chapter 10](#) discusses future work. Finally, [Chapter 11](#) provides a brief summary and concludes the work.

Chapter 2

Related Work

Surely there must be a less primitive way of making big changes in the store than by pushing vast numbers of words back and forth through the von Neumann bottleneck. Not only is this tube a literal bottleneck for the data traffic of a problem, but, more importantly, it is an intellectual bottleneck that has kept us tied to word-at-a-time thinking instead of encouraging us to think in terms of the larger conceptual units of the task at hand. Thus programming is basically planning and detailing the enormous traffic of words through the von Neumann bottleneck, and much of that traffic concerns not significant data itself but where to find it.

– John Backus, 1977 ACM Turing Award Lecture [8]

The content of this chapter is published in an IEEE Access journal article titled “A Review of Spiking Neuromorphic Hardware Communication Systems” © 2020 IEEE [120], reprinted with permission. Different groups have tackled the challenge of designing a new type of computer, taking inspiration from our understanding of how the brain is able to achieve its great computational feats. Multiple issues must be faced to build a brain-inspired device, but perhaps the most challenging is how to tackle the dauntingly hard task of creating a system able to support the trillions of connections found in the brain. Building a neuromorphic computer is an interesting endeavor. Although the base components and their operations are easy to understand on their own, the difficulty comes from the vast number of these components found

in the brain. When designing a large neuromorphic system, the challenges stem from trying to create a system with a similarly large number of neurons and synapses as the brain using semiconductor technology. This review paper discusses how different neuromorphic hardware designs handle spiking communication between neurons, and shows recent design trends found in neuromorphic communication. [Section 2.1](#) introduces the neuromorphic projects reviewed in this paper. [Section 2.2](#) discusses various considerations when dealing with packets in a spiking neural network (SNN). [Section 2.3](#) looks at different solutions to local high fan-out and fan-in connections. [Section 2.4](#) considers solutions to supporting global synaptic connections. Prior papers have also compared various state-of-the-art neuromorphic systems. Recent notable papers include: “Large-Scale Neuromorphic Spiking Array Processors: A Quest to Mimic the Brain” which reviews details on the various design strategies of neural processors [98]; “Low-Power Neuromorphic Hardware for Signal Processing Applications” summarizes the operation of the brain and recent neuromorphic systems with an emphasis on signal processing [76]; *A Survey of Neuromorphic Computing and Neural Networks in Hardware* is an exhaustive review of research conducted in neuromorphic computing since the inception of the term [88]; and “Spiking Neural Networks Hardware Implementations and Challenges: A Survey” which surveys state-of-the-art spiking neuromorphic hardware and current trends in algorithm elaboration [13].

2.1 Neuromorphic Projects Overview

Initially, neuromorphic hardware design was only done by research institutions, but as academics have shown the potential of these brain-inspired models, corporations have started developing research chips, and partnering with research institutions to best design and utilize these new chips. Neuromorphic computing, and spiking neural networks in particular, is a growing field with many avenues of future research available.

2.1.1 Stanford University—Neurogrid & Braindrop

Researchers at Stanford University have created two separate neuromorphic hardware designs. The first design is a mixed-analog-digital system called Neurogrid, which is able to provide

computational neuroscientists with the capability to perform biological real-time simulations of the brain with millions of neurons and billions of synaptic connections [10]. Neurogrid is made up of analog neurons placed inside a 256×256 array fabricated in a 180-nm CMOS to make a Neurocore. To build the full Neurogrid system, 16 of the Neurocore chips are placed on a board and arranged into a tree structure. Figure 2.1 shows the complete Neurogrid circuit board.

Their second design is called Braindrop [67]. Braindrop, like Neurogrid, is a mixed-analog-digital design; however, unlike Neurogrid, Braindrop is designed to be programmed at a high level of abstraction. Braindrop uses the Neural Engineering Framework (NEF) as the theoretical underpinning for the abstractions used to hide the heterogeneity found when designing with analog neurons. Braindrop’s programming is unique because computations are specified as coupled, nonlinear, dynamical systems and an automated procedure is used to synthesize the systems to the hardware. Braindrop is fabricated in 28-nm FDSOI process and integrates 4,096 neurons onto a single Braindrop chip. In the future, multiple Braindrop cores will be integrated to build a larger Brainstorm chip.

2.1.2 Human Brain Project—BrainScaleS & SpiNNaker

The next group of neuromorphic systems all come from the Human Brain Project (HBP) [56]. HBP is funded by the European Union with the goal of “building a research infrastructure to help advance neuroscience, medicine and computing” [47].

BrainScaleS is a mixed-analog-digital waferscale neuromorphic hardware system developed by a collaboration of research groups including the University of Heidelberg and the Technische Universität Dresden [80, 83]. BrainScaleS builds on the work completed in the FACETS (Fast Analog Computing with Emergent Transient States) project [79]. The waferscale integration technology developed for BrainScaleS makes it possible to utilize an entire 20 cm wafer for a very-large-scale neuromorphic system with 40 million synapses and up to 180 thousand neurons. The BrainScaleS system is built up by implementing many analog neuron circuits and their synapses in a structure called the Analog Network Core (ANC). The ANC was fabricated using 180-nm CMOS to create a High Input Count Analog Neural Network

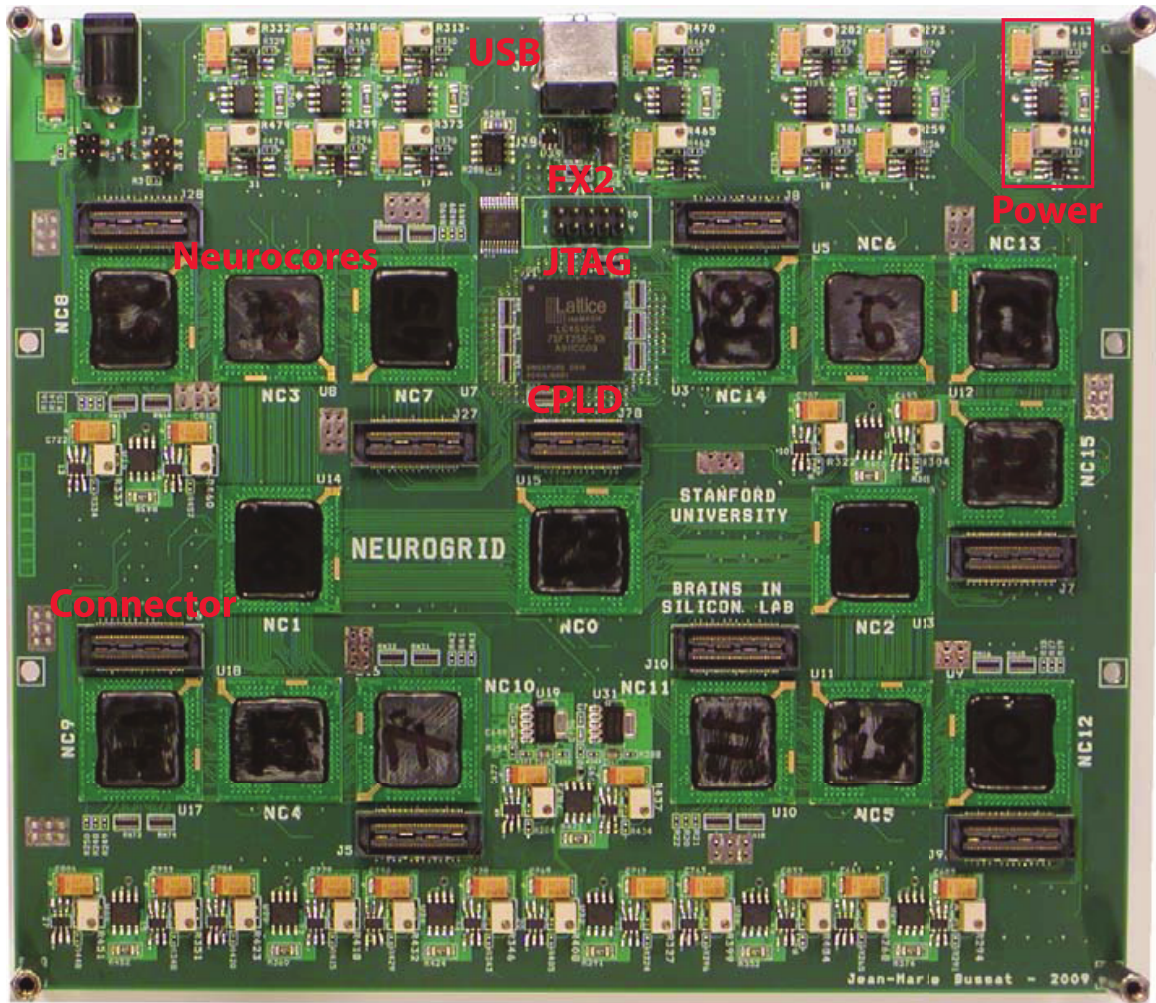


Figure 2.1: Neurogrid circuit board with 16 packaged Neurocore chips. The CPLD and FX2 enable a USB connection between a traditional computer and Neurogrid [61].

(HICANN). 352 HICANN chips are able to fit on a single wafer. A diagram of the complete wafer-scale BrainScaleS system is shown in [Figure 2.2](#).

A second generation of the BrainScaleS system is being designed and was revealed at the NICE Workshop in 2018 [81]. BrainScaleS-2 uses a more complex neuron model which supports nonlinear dendrites and structured neurons, along with other features. They also added the ability to use the neurons in a perceptron mode, where the neurons behave like traditional perceptrons and can be used to build non-spiking convolutional networks. This feature will also allow BrainScaleS-2 to combine both spiking neurons and perceptrons in the same experiment.

Also part of the Human Brain Project, researchers at the University of Manchester are working on a large digital neuromorphic system called SpiNNaker [38]. Instead of building custom neuron circuitry, SpiNNaker simulates the brain in real-time by connecting together over one million ARM processors. The ARM processors allow SpiNNaker to model a billion spiking neurons with biologically realistic connectivity (1,000–10,000 synapses per neuron) with 1 ms per step of simulation. Eighteen homogeneous ARM968 processors are integrated into one Chip MultiProcessor (CMP) fabricated in 130-nm CMOS. Sixteen processors are used for simulation, one processor for administration, and one processor is a backup in case a processor is faulty. The full system is constructed of 2^{16} CMPs connected in a two-dimensional toroidal mesh. A single SpiNNaker board contains 48 CMPs and is shown in [Figure 2.3](#).

The first version of SpiNNaker is only able to simulate 1% of the human brain. The Second generation, named SpiNNaker2, aims to be able to simulate the entire brain [45]. They plan to achieve this feat by scaling-up the design of the previous generation. SpiNNaker2 will have 144 ARM MF4 cores per CMP fabricated in the modern 22FDX process. Additionally, SpiNNaker2 will include new features such as dynamic power management, floating-point support, synchronous memory sharing to neighboring cores, multiple-accumulate accelerators, and other numeric accelerators.

2.1.3 TrueNorth

The TrueNorth neuromorphic platform has recently been developed by IBM as part of the Defense Advanced Research Projects Agency (DARPA) Systems of Neuromorphic Adaptive

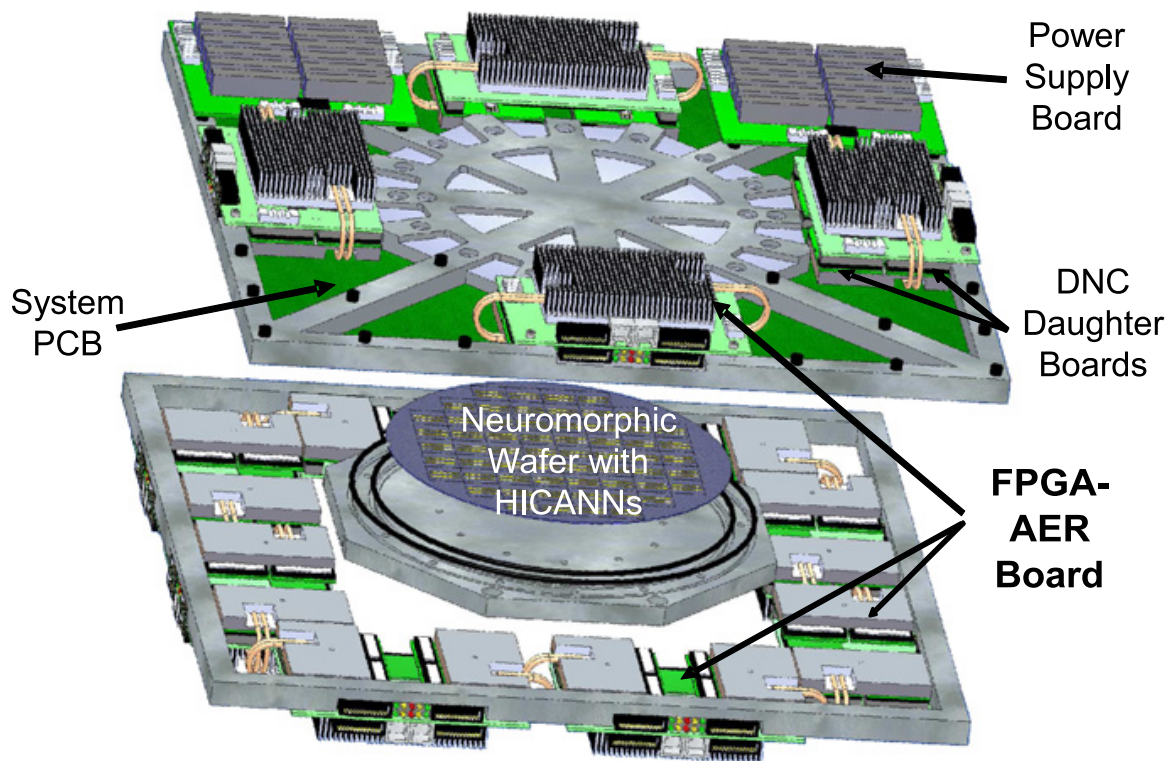


Figure 2.2: Diagram showing the complete BrainScaleS system [82].

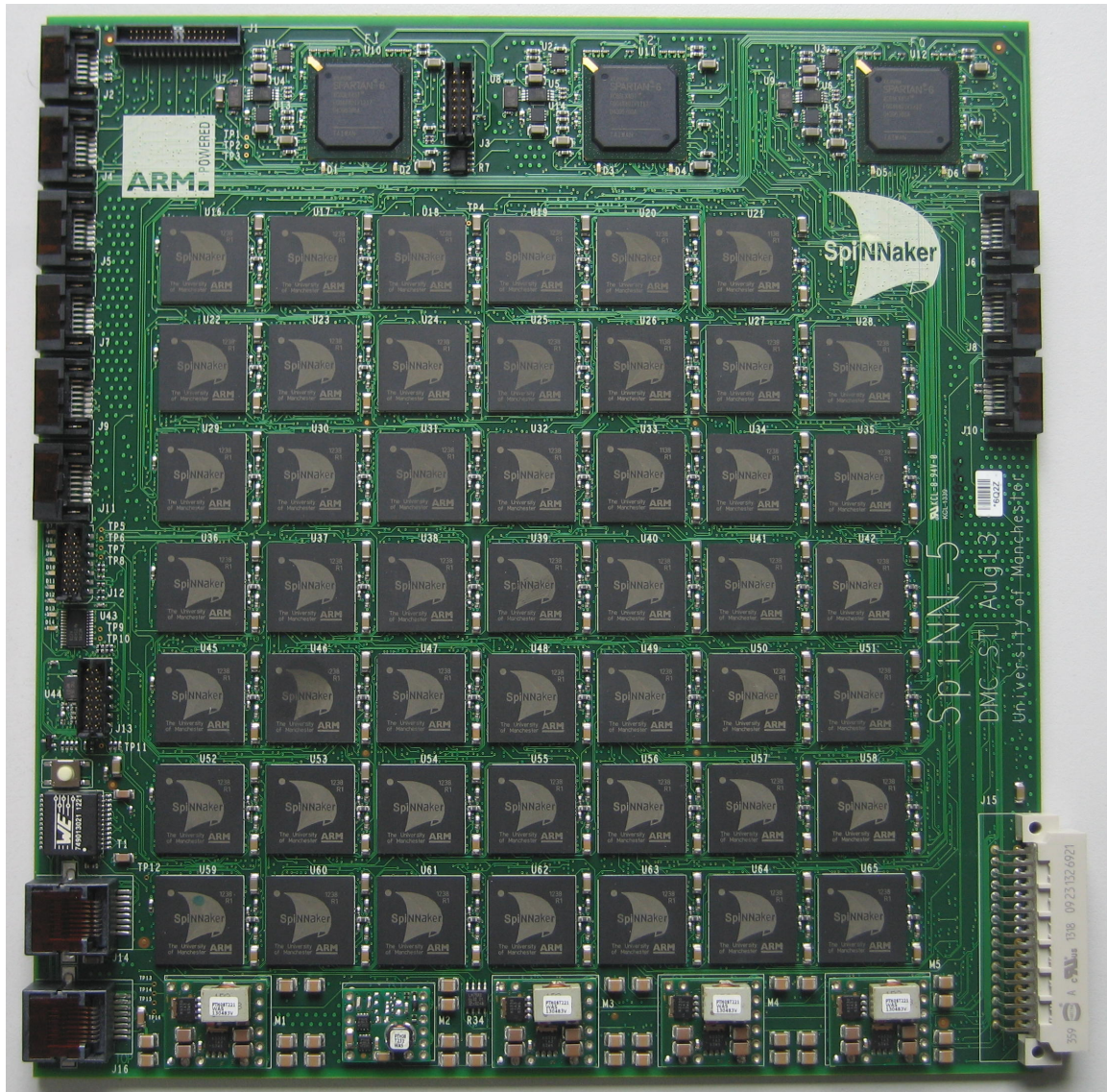


Figure 2.3: SpiNNaker circuit board, which is a building block for the SpiNNaker machine, contains 48 chips with a total of 864 ARM processors [9].

Plastic Scalable Electronics (SyNAPSE) program [1]. A single TrueNorth chip is composed of 4,096 neurosynaptic cores, with each core bringing together memory (“Synapses”), processors (“Neurons”), and communication (“Axons”), fabricated in IBM’s 45-nm SOI process [3]. Each core implements 256 digital integrate-and-fire neurons with 1024 axonal circuits for input connectivity organized as an Static Random Access Memory (SRAM) crossbar [60]. A single TrueNorth chip contains 5.4 billion transistors, and implements 1 million digital neurons and 256 million synapses, tightly integrated in an event-driven design [1]. Multiple chips have also been combined into a larger 16-chip NS16e platform to allow for the simulation of 16 million neurons and 4 billion synapses.

2.1.4 Loihi

Intel has also recently developed a digital neuromorphic research chip known as Loihi [23, 105]. Loihi has a unique programmable microcode learning engine for on-chip SNN training. Along with the 128-neuromorphic cores found on the chip, there are 3 x86 Lakemont cores which help with advanced learning rules and with managing the neuromorphic cores. Loihi is fabricated in Intel’s 14-nm process. The 128-neuromorphic cores implement 130,000 artificial CUBA leaky-integrate-and-fire neurons and 130 million synapses. The Loihi design supports scaling up to 4,096 on-chip cores and 16,384 chips.

2.1.5 Darwin

Research groups at Zhejiang University and Hangzhou Dianzi University in China have created the Darwin Neural Processing Unit (NPU), which is targeted for resource-constrained, embedded applications [94, 53]. The NPU constrains 8 physical neurons on the chip; but each neuron can be used to simulate 256 logical neurons with time multiplexing, resulting in the 2048 logical neurons for the entire chip. Each neuron is implemented with digital logic and can be connected arbitrarily to any other neuron resulting in a theoretical max of 4,194,304 synapses. In practice, however, the max number of synapses is limited by the size of the external memory used to store synapse information. This design was originally prototyped on a Field Programmable Gate Array (FPGA) and then fabricated in SMIC’s 180 nm process.

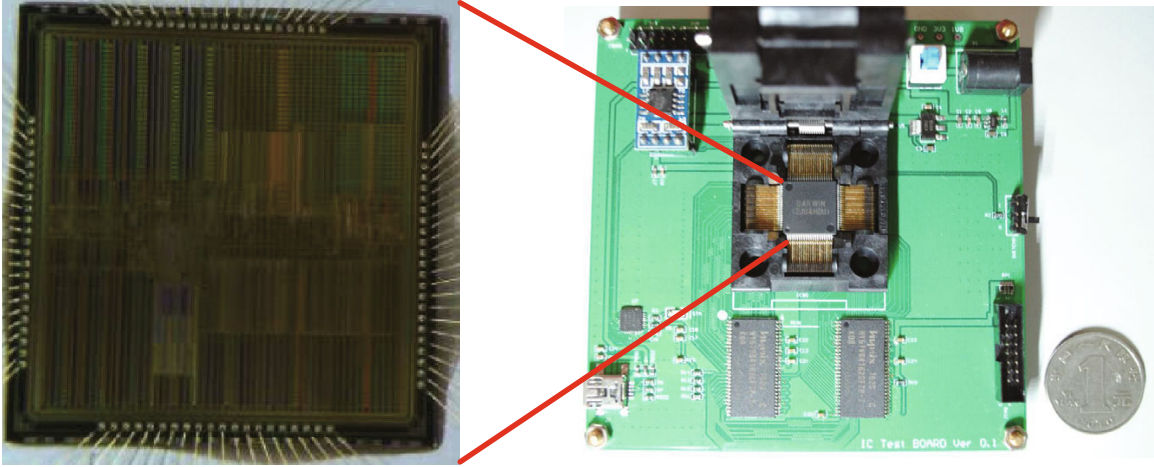


Figure 2.4: Darwin chip die and demonstration PCB board [94].

The complete system also includes a RISC CPU to create a complete neuromorphic System on Chip (SoC). The Darwin chip die and demonstration board is shown in Figure 2.4.

2.1.6 Dynap-SEL

Researchers at the University of Zurich in Switzerland, in the lab of Giacomo Indiveri, have created a novel, mixed-signal, multi-core architecture for neuromorphic processors which combines the advantages of the robustness of asynchronous digital logic for communication with the efficiency and dynamics of analog circuits for computation. The group has designed and fabricated the Dynamic Neuromorphic Asynchronous Processors (DYNAPs) in a 180 nm CMOS process; then they scaled up the design to a 28 nm FDSOI process with Dynamic Neuromorphic Asynchronous Processor with Scalable and Learning (Dynap-SEL) [66, 98].¹ DYNAPs and Dynap-SEL chips both have four neural processing cores. Each core implements 256 analog Adaptive-Exponential Integrate and Fire (AdExp-I&F) neurons arranged in a 16×16 grid. Each neuron has 64 programmable synapses with a max fan-in of 64 connections and a max fan-out of 4000 connections. Each synapse comprises circuitry to model biophysically realistic dynamics, including N-Methyl-D-Aspartate like voltage-gating, leak, spike-frequency adaptation, sodium activation resulting in positive feedback, potassium

¹Note: Per available sources, Dynap-SEL is a second improved version of DYNAPs, implemented in a smaller process with additional features. DYNAPs is discussed in [66]; Dynap-SEL is discussed in reference [98], along with other systems.

channels resulting in negative feedback, and refractory periods, as well as other dynamic models. Dynap-SEL has an additional fifth core which has 1×64 analog neurons with 64×128 plastic synapses with on-chip learning and 64×64 programmable synapses. Dynap-SEL’s fifth core also has increased potential fan-in and fan-out, since up to eight rows of synapses can be merged together, at the expense of decreasing the active neuron count, to achieve a max fan-in of 1000 plastic synapses and 512 non-plastic synapses per neuron, for a network of 8 usable neurons. DYNAPs has been scaled to a PCB board which hosts 9 chips. Dynap-SEL supports integration into a chip array of up to 16×16 chips. DYNAPs and Dynap-SEL use the same routing architecture, so when the routing of DYNAPs is discussed, know that Dynap-SEL functions similarly.

2.2 SNN Packets

The communication patterns found with SNNs are different from the patterns of communication found in traditional computing. With traditional memory transfer, there is a greater emphasis placed on transferring large amounts of data with a high bandwidth to a cache near the CPU, and then hoping that the local cache has all the information needed to perform the computation without a cache miss. With traditional computing, CPU performance has been outpacing memory throughput and speed, resulting in the need for various tricks to guess what information will be needed by the processor ahead of time.

SNN communication is completely different. All of the memory is local in the form of synaptic weights and neuron charge, which allows the memory to be located alongside the computation elements. The information that is communicated in SNNs is simply the presence of a fire event, or spike. Spike information is presented in a streaming and online manner, contrasted with the batch mode used by large, dense transfers. “A biological neuron firing is a pure asynchronous event which carries no information other than that it has happened [107].²” The power of the spiking information lies in the timing of the spike; all information is conveyed by the presence, frequency, and timing of these spiking events. An SNN’s communication is

²There are other neuromorphic systems which represent neurons and synapses as separate hardware elements. In this case, the communication network will need to also be able to send weighted information for postsynaptic events [26].

inherently event-driven and asynchronous. A biological spike only happens as a response to some other event, and the spike is sent to other neurons without the presence of any global clocks. Accumulation of charge in each neuron only happens as a response to an input event occurring. When the accumulated charge exceeds a threshold, an electrical pulse is sent down the axon to the synapse, which communicates via chemical signaling to dendrites of the next neuron [75]. This chemical signaling mechanism is the local memory for the strength of the connection.

This all puts a different strain on communication systems that transmit these types of packets. The emphasis is now on the timing of events that contain a very small amount of information. Luckily, biological neurons only fire at around 10 Hz, CMOS communication occurs much faster than this, operating in the MHz and GHz ranges. This speed discrepancy between the firing rate of the neurons and the signaling rate of CMOS wires enables the use of time multiplexing to bundle many fire events into a single communication channel. Biological neurons have a physical connection for each spiking path in the form of the long axon in the neuron which can stretch as far as a meter in humans [20]. With CMOS technology, having wires that can change their connectivity like the brain is infeasible. Therefore, the connections are made with various communication channels and routing methods that allow for flexible, reconfigurable, virtual connections over a fixed set of wiring.

2.2.1 AER Representation

Spikes are encoded into packets through Address-Event Representation (AER). With traditional AER, the only information included in a packet is the address of the firing element [34] and optionally the time that the event occurs. These streams of address information are then multiplexed onto an asynchronous digital bus; routing of the packets is next performed based on the address of the source. When the time is not included in the packet, the time of the event is implicit based on when the packet arrives at the destination. The delay in communication in this case is considered inconsequential, since it is much lower than the firing rate of the neurons. Both SpiNNaker and Darwin use this simple AER representation for events. This simple AER format is advantageous since communication boards can be developed to route, analyze, record, and insert AER packets into a network

of multiple neuromorphic devices [11]. Alternatively, the ID of the destination, or routing information can be sent instead of the ID of the source. This is done when the structure is set up so that the fire travels to a destination axon, such as in the TrueNorth system. This variation of the traditional AER format comes down to how the packets are routed, in addition to how connectivity is handled.

2.2.2 Routing Method

As alluded to in the previous section, there are two main routing methods employed by the neuromorphic hardware: source routing, where the packets are routed based on the source of the fire event, and destination routing, where the packets are routed based on the destination to which the fire event is traveling. There are two main packet types, multicast and point-to-point. With multicast, a single packet is delivered to multiple destinations; with point-to-point, a single packet is sent to a single destination. If the router supports multicast packets, then neurons with a high fan-out can be handled efficiently by the router. If instead the routing fabric only supports point-to-point packets, then a neuron with fan-out will have to send a new packet for each destination.

Source routing is advantageous since multicast routing can be easily implemented by allowing each router to steer and duplicate a packet based on its own routing table stored at each router. Neurogrid, SpiNNaker, BrainScaleS, and Darwin all use source-based routing with the routing tables for each router stored in a large off-chip memory [12, 107, 82, 53]. The router looks up the destinations the packet should be sent to and forwards the packet to the correct neighbors. The average access frequency for a particular memory location is low since a specific synapse in a large SNN has a very low probability of being activated at any given time [53]. SpiNNaker uses a special ternary Content Addressable Memory (CAM) for route lookup [107]. The identifier is compared under a mask to all the keys in the lookup table to get hit or miss information about which output links the packet should be sent. There is also a ‘default’ route which sends the packet along the same direction it was traveling to help save on the memory required for the routing. The BrainScaleS also has a timestamp-based prioritization mechanism based on the time the events occur [82]. This allows BrainScaleS to

achieve lower latency and higher bandwidth for pulse routing as well as allowing configurable event delays.

The downside of source routing is that a large off-chip memory is needed to store the routing information for each source ID and the number of synaptic connections is limited by the amount of memory available for these routing tables. Furthermore, accessing external memory is slow, and doing so greatly decreases the speed at which packets can be routed. An alternative approach is to route the packets by destination. In this case, the source neuron knows the destination of a limited number of destination elements and encodes the destination as part of the packet. This allows the connectivity information to be stored with the sending neurons and does not require a large off-chip memory. Both TrueNorth and Loihi use destination-based routing to route packets [1, 23]. This allows the routers to be simple, dimension-order routers, which direct packets based on information found in their headers. TrueNorth only supports unidirectional messages sent from a source neuron to a receiving axon. This method works since the receiving axon can then connect to any of the 256 local neurons within the same neurosynaptic core [78]. TrueNorth’s routers connect with its own core and its four neighboring cores, creating a two-dimensional mesh network. Each packet then carries a delta-x and delta-y address of the destination core, a destination axon index, and a destination time for the spike to be integrated. The packets are routed first in the horizontal direction until delta-x is 0. Then the delta-y field is dropped from the packet, and the packet is routed along the vertical direction until it arrives at the destination core. Loihi also uses dimension-order routing on a two-dimensional grid, but supports additional features to relax the connectivity constraints placed on the programmer [23]. These features include sparse network compression, core-to-core multicast, variable synaptic formats, and population-based hierarchical connectivity. Loihi’s Network on Chip (NoC) only supports unicast distributions; however, multicast spikes can be sent by sending multiple unicast packets. The routing of packets is still done by using memory resources local to the core, without depending on off-chip memory. This results in various network mapping constraints, including the max number of neurons per core, and the max number of fan-in and fan-out connections.

DYNAPs uses a mixture of different routing methods with its mixed, tag-based, shared-addressing scheme. The main idea of this scheme, is “to introduce different clusters with independent address spaces, so that it is possible to re-use the same tag ids for connecting neurons among each other, without loss of generality [66].” This scheme divides neurons into clusters, and the communication between the source and destination neurons is divided into two phases. In the first phase, the packets use the destination-address of an intermediate node to route the packet to that intermediate node via point-to-point routing. Then in the second phase, the intermediate node broadcasts the tag stored in the packet to every neuron in the cluster. Each neuron then checks the tag against the tags stored in a CAM to see if the neuron is connected to the source neuron with the same tag. This use of tags allows the tag addresses to be shared among source and destination neurons from different clusters, which results in there being less memory required to store the connection information. This two-stage, tag-based routing scheme is shown in [Figure 2.5](#). In the hardware implementation of this scheme, the clusters are conveniently set to be the neurons contained in a single core of the DYNAPs chip. The first phase routing is done with R1, R2, and R3 routers with the intermediate nodes being the R1 routers at the destination core. The second phase is carried out by the R1 routers at the destination core broadcasting the packets to all the neurons in the destination core. So when an event packet is sent from a neuron, it goes to the local core’s R1 router, where it is either routed to the same core, to a core on the same chip via R2 routers, or to a core on a different chip via both R2 and R3 routers. The R1 router either passes the packet to an R2 router or uses source-address routing to broadcast a multicast packet to all the neurons in the core. R2 routers use absolute destination-address routing to route the packet to the correct core. R3 routers use relative destination-address routing to route the packet to a destination chip at position $(\Delta x, \Delta y)$ via dimension order routing. Once on the correct chip, the R2 routers are used to route to the correct core. The tag-based addressing scheme reduces memory requirements enough to allow the memory used to store connection information to be distributed across the cores and routers in embedded, asynchronous SRAM and CAM memory cells. The SRAM cells are located in the R1 routers and store the source memory required to define the point-to-point connections in phase one of routing. The lines in the source memory contain the tag of the source neuron and the

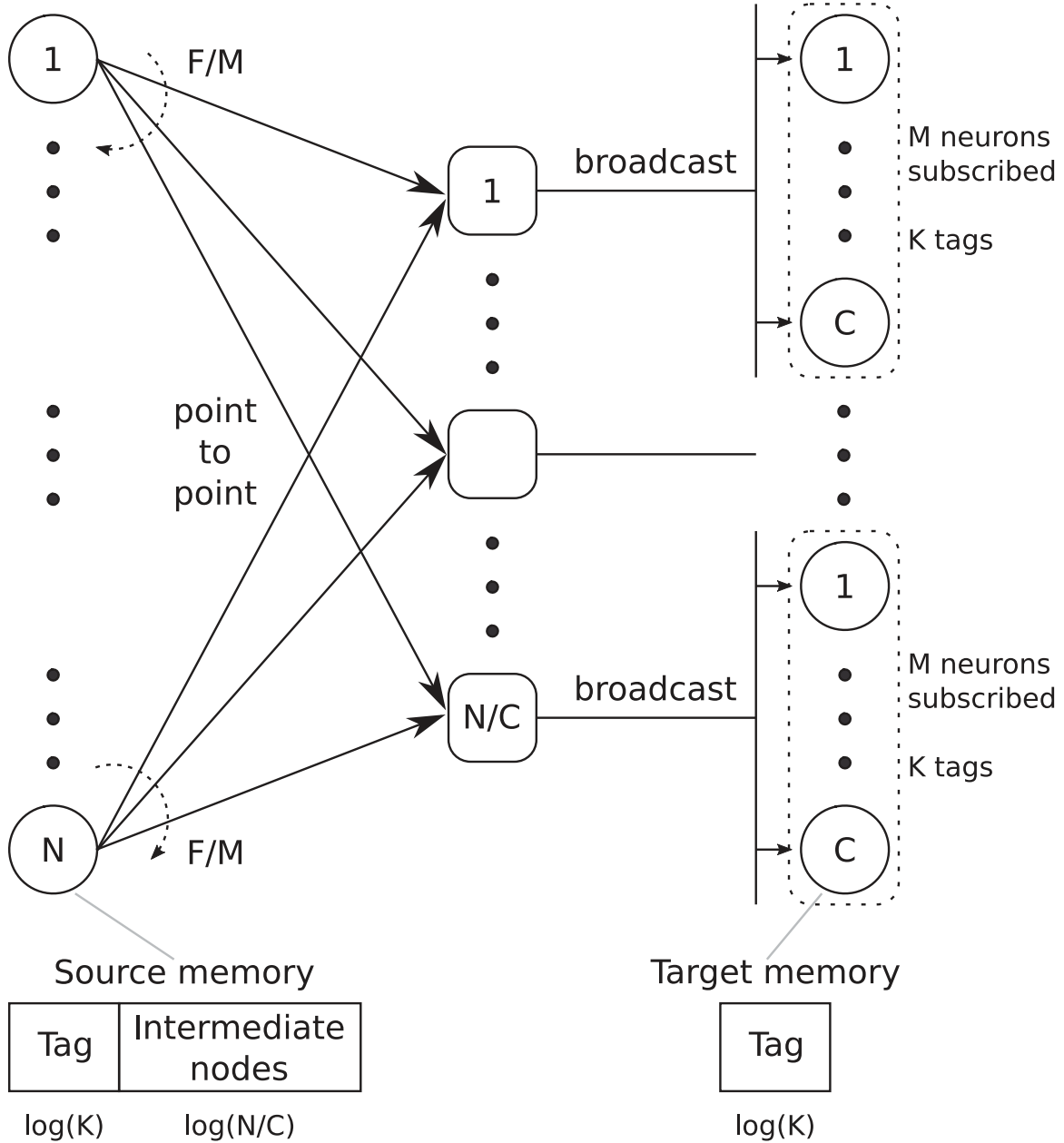


Figure 2.5: The two-stage routing scheme used by DYNAPs. The connections of N neurons, each with a fanout of F , is implemented by first using point-to-point communication to send the source tag to N/C intermediate nodes. This reduces the point-to-point fan-out to F/M . The intermediate nodes then broadcast the source's tag to C neurons within a cluster, where M of these neurons are subscribed to the key. The number of unique keys used in a cluster is K . Note that the neurons on the right side are the same as the ones on the left, but are shown grouped into N/C clusters [66].

address of the intermediate node. The CAM cells are located in the destination synapses and store the target memory that is used for phase two of the routing. The lines in the target memory store the tags of the neurons that these synapses are subscribed to.

Since all the neuromorphic systems have a limitation on the maximum communication bandwidth they can supply, many of the systems use a network compiler to reduce the amount of communication traffic by mapping neurons that are connected to each other to the same core, thus reducing the distance most of the packets have to travel. TrueNorth has created a mapping algorithm which uses a modified very large scale integration (VLSI) placement algorithm to place neurons onto cores and minimize the distances packets have to travel [1]. This greatly reduces the total number of hops that are necessary for the packets to reach their destination. Similarly, SpiNNaker has a PArtition and Configuration MANager (PACMAN) which provides utilities for SNN partitioning, placement, and routing. PACMAN is able to use a variety of partitioning and placement algorithms. One recent SNN spectral analysis based partitioning and placement algorithm is the Graph Optimizer SpiNNaker Tool (GHOST). Essentially, GHOST creates an expanded neuron graph, uses clustering to group highly connected neurons, uses sub-clustering and fusion to reduce the neuron count groups that will fit on a single core, and uses Sammon mapping to place the high dimensional groupings onto the 2D mesh with legalization to fine tune the placement to make it valid [102]. Two organizational principles found in the brain are used to reduce network traffic: local dendritic trees within a pool of neurons and hierarchical axonal arbors between pools of neurons [12]. In the brain, there are cortical columns and regions with many dense connections and these dense regions are connected with long-range cortical connections [78].

2.2.3 Asynchronous Circuits

A main feature of SNNs is their asynchronous and event-driven nature. Both of these features allow the brain to be very energy efficient, as energy is only expended when a spike arrives and no extra energy is expended for a synchronous clock. The neuromorphic components and packet routers are also designed using asynchronous VLSI circuit techniques. These circuit techniques remove the challenges of routing a synchronous clock across a very large chip or across multiple chips in a system, and eliminate the power lost as a result of applying

a clock to idle components. Martin et al. state that, in the future, SoCs will no longer be able to operate under a single clock [57]. The variations across such a large chip will make it prohibitively expensive to attempt to manage the delays in a clock and other global signals. The solution is to use asynchronous circuit techniques which are delay-insensitive and pass information using quasi-delay-insensitive (QDI) circuits. These circuits communicate with asynchronous handshake protocols without a shared global clock. Martin et al. predict that future systems will be made entirely with asynchronous logic, or that they, at the very least, will have to be designed Globally Asynchronous and Locally Synchronous (GALS). Asynchronous VLSI circuits are defined with a high-level description language, for example the Communicating Hardware Processes (CHP) language, and then compiled into a circuit design using semantic-preserving program transformations. All the neuromorphic devices use asynchronous circuits designed in QDI design style based on Martin’s synthesis procedure [58, 59, 57]. These asynchronous circuits are used to implement the NoC structures and routers, enabling them to conserve power when not in use and also handling the varying delays in the wires connecting the elements together.

Additionally, the number of pins available on the chips for external connections is greatly limited. This requires that the multiple asynchronous parallel buses found in the hardware must be serialized into a narrower stream, which uses fewer pins. In the case of SpiNNaker, inter-chip links convert 1-of-5 RTZ (return-to-zero) codes to 2-of-7 NRZ (non-return-to-zero) codes to reduce the number of I/O pins used and to save energy by reducing the amount of signal transitions required to transmit information [37]. TrueNorth uses asynchronous arbiters and merge-split blocks to combine multiple streams of packets into a single stream for sending the packets off-chip [1]. The single stream is then transmitted to neighboring chips via bundled-data asynchronous circuits to minimize the number of interface circuits and pins needed.

2.2.4 Packet Structure

Another important consideration when designing the AER communication is the packet structure of the fire events and if the packets should be fixed length or variable length. Neurogrid supports variable length packets [10]. A Neurogrid packet is a sequence of 12-bit

words that specify a route, address, arbitrarily long payload, and tailword, sent in that order. The tailword specifies the end of the packet. SpiNNaker packets are fixed width with an 8-bit header, a 32-bit content field, and an optional 32-bit data payload [107]. The content field is typically a key to identify the source neuron. TrueNorth packets contain a 9-bit delta-x, 9-bit delta-y, 4-bit delivery time, 8-bit destination axon index, and 2 debugging bits [1]. BrainScaleS's Packets are fixed width, with varying widths depending on the level in the communication hierarchy, but each packet is made up of a multiple of 24-bit pulse events [83]. Darwin packets contain the ID of the neuron that generated the spike and the timestamp of when the spike was generated [53]. DYNAPs packets are created by copying the line from the source SRAM that corresponds to the neuron which fired. Each 20-bit word is made up of a 10-bit address, a 6-bit routing header, and a 4-bit destination core (intermediate node) address. The routing header contains a 2-bit delta-x with 1-bit sign and a 2-bit delta-y with a 1-bit sign [66].

2.2.5 Event-Based Sensors and Actuators

Neuromorphic event-based sensors have been designed based on biological sensors. They seek to replicate the efficiency, robustness, and low-power consumption of their biological inspirations. Among these sensors are different implementations of a silicon retina [21] and a silicon cochlea [19]. Both of these sensors are event-based and only send information when there is a change in the environment. This allows these sensors to require less bandwidth than their traditional counterparts. Neuromorphic sensors send their information as a continuous stream of asynchronous spikes using AER packets. For example, the Asynchronous Time-based Imaging Sensor (ATIS) silicon retina sends illumination change packets with the location of the pixel, the polarity of the change in illumination, and the time when the event occurred [21]. Since both the sensors and the neuromorphic computers use AER encoded asynchronous packets, the neuromorphic processors can directly use information from the sensors for computation without needing an extra conversion step. If event-based actuators are used, then the output from the neuromorphic processor can also be used directly for control applications. An example of this is the AER-Robot, which uses event-based encoders for input and Pulse-Frequency Modulation (PFM) to power DC motors, to create a closed-loop,

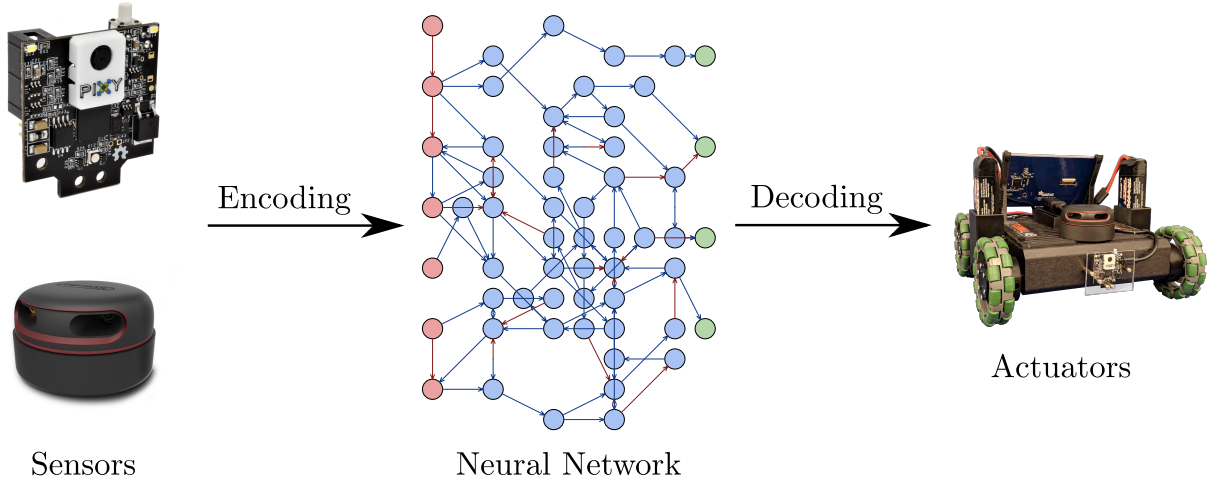


Figure 2.6: Flow of information from sensors through a neural network to actuators. The sensors shown are a Pixy2 camera [69] and a RPLIDAR-A2 [95]. The neural network and robot images come from the GRANT project (see Section 8.6.1).

neuromorphic control system [50]. Figure 2.6 shows the flow of information from sensors that is encoded into spikes, sent to the neuromorphic array, decoded, and then used to control actuators to perform a task.

2.3 Local Communication Challenges

Communication can be viewed from two different levels of organization: local on-chip communication within a neuromorphic core and global system communication across many cores. With both levels, one feature that helps with the design of the communication channels is that synapses can share wires among a group of neurons, since wires propagate signals much faster than biological axons [12].

Neurogrid uses a shared dendrite hybrid model to handle local connections [10]. With this model, each shared-synapse circuit is connected to neighboring neurons, mirroring the structure of overlapping dendritic trees found in biological neural networks. This structure allows Neurogrid to work well for modeling networks which require many neurons with mostly local connections, for example, modeling the neocortex.

In TrueNorth, a 1024×256 bit SRAM crossbar memory is used to define the synaptic connections from 1024 axon wires to 256 dendrite wires [60]. Spike packets are routed to a

particular axon, which can then be connected to up to 256 dendrites. The dendrites are then connected to the neurons for computation. The downside to a crossbar design is that if the synaptic connections are sparse, resources are wasted [12].

BrainScaleS handles local communication with their L1 routing interface. L1 routing is an asynchronous, serial, event-driven protocol, which operates at up to 2 Gbps and is used to interconnect Analog Network Chips (ANC) [80, 82, 79]. The ANCs are connected physically with a dense layer of horizontal and vertical wires that are added as post-processed metal interlinks on top of the wafer. These wires form a high-density pulse routing grid. Temporal multiplexing is used to allow each of the 256 Low-Voltage Differential Signaling (LVDS) bus lanes to carry events from 64 presynaptic neurons through the serial transmitting of 6-bit neuron IDs. Sparse crossbar switches and repeaters are used to propagate the spikes across an arbitrary number of HICANN chips. Figure 2.7 shows what this L1 routing looks like.

SpiNNaker uses the ARM AMBA (Advanced Microcontroller Bus Architecture) protocol for communication within a local clock domain [38].

DYNAPs handles local communication in the second phase of routing by using the R1 routers to broadcast the incoming packet to all of the neurons in the core. The neurons then use the tag to determine if they are sensitive to the fire event [66].

2.4 Global Communication Challenges

With global packet routing there are two routing schemes used: mesh and tree. With mesh routing, routers are connected to neighboring routers; the most common meshes are two-dimensional grids with connections in the four cardinal directions. Tree routing structures the routers in a hierarchy with a root node that repeatedly branches to connect to routers in the lower levels. Mesh routing is advantageous since it has a larger channel bisection, with more links in the network, resulting in higher throughput of packets [10]. However, mesh routing usually has a longer latency due to a larger number of hops for the packet to reach its destination. With a mesh, dimension-order routing is typically employed to prevent deadlock. Tree routing is advantageous when shorter latency is needed, since fewer hops are required to reach the destination; however, there are fewer connection paths, resulting in

lower bandwidth. Deadlock-free multicast routing is easier to implement with tree routing by using up-down routing. With up-down routing, packets are first sent to a common root node, and then the packets are able to be duplicated to multiple child nodes on the downward routing phase. If the efficient multicast capability of tree routing can be utilized, then tree routing offers lower latency and higher effective throughput than mesh routing, and uses roughly two-thirds of the resources [10].

2.4.1 Tree

Neurogrid uses a multicast tree router connected in a binary tree. Up-down routing is used to prevent deadlock and support multicast packets. Figure 2.8 shows a diagram of the Neurogrid chips and the tree routing structure. Braindrop uses a fractal H-tree for routing [35]. BrainScaleS’s Layer 2 routing uses a hierarchical packet-based routing tree constructed with digital network chips (DNC). Eight High-Input-Count Analog Neural Networks (HICANNs) are grouped into one reticle. The reticle is then connected to one DNC. Four DNCs are connected to an FPGA-AER board. The FPGA-AER boards are then connected to each other using the Aurora protocol with 10 Gbps data rate on four parallel multi-gigabit lanes. Communication for the entire wafer requires 12 FPGA-AER boards. The FPGA-AER boards are connected via 1 or 10 Gbps Ethernet links to handle wafer-to-wafer communication. This hierarchical structure is shown in Figure 2.9.

2.4.2 Large Grid

SpiNNaker connects the ARM chip multiprocessors with a two-dimensional toroidal mesh. Each CMP has six connections to neighboring chips forming triangular facets which support ‘emergency routing’ around a failed or congested link [38]. SpiNNaker’s system network connectivity is shown in Figure 2.10 The TrueNorth chip tiles its 4,096 neurosynaptic cores into a two-dimensional array [1]. Dimension-order routing is used to prevent network deadlock. Packets are routed on a first-come-first-served basis, where arbitration is used for packets that arrive at the same time. To guarantee that no packets will be dropped, back pressure is used to prevent new packets from arriving when the router is waiting to send outgoing

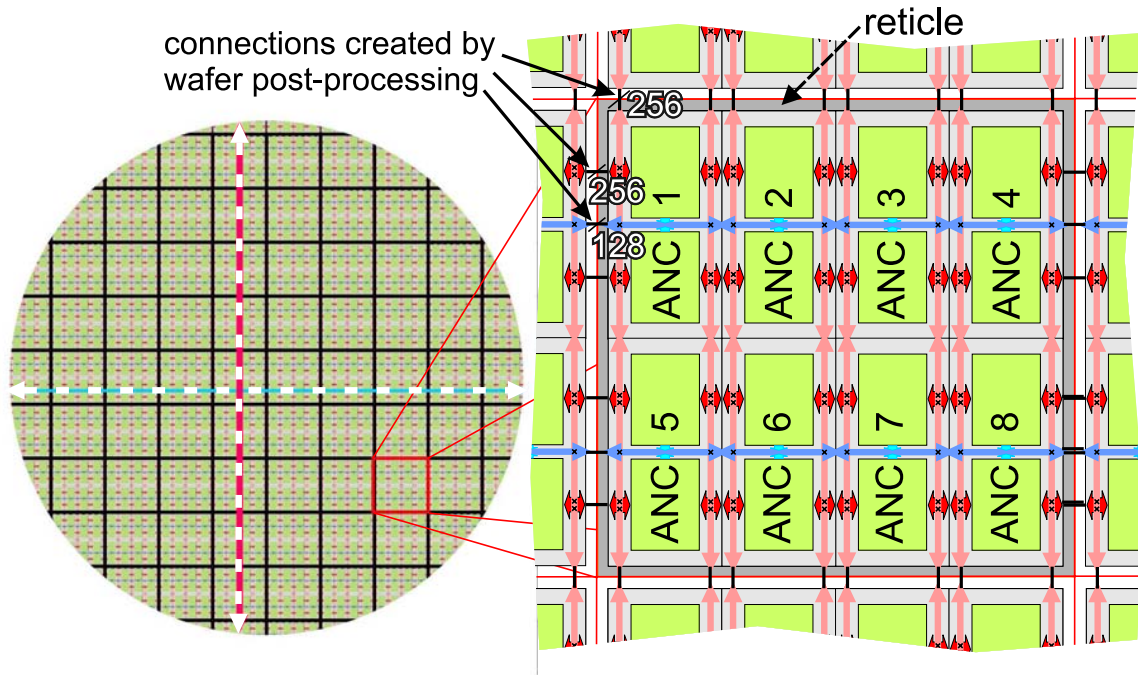


Figure 2.7: A BrainScaleS wafer with 56 complete reticles. The dashed arrows depict one vertical and horizontal bundle of inter-neuron connections. A single reticle is enlarged to show the arrangement of ANCs [79].

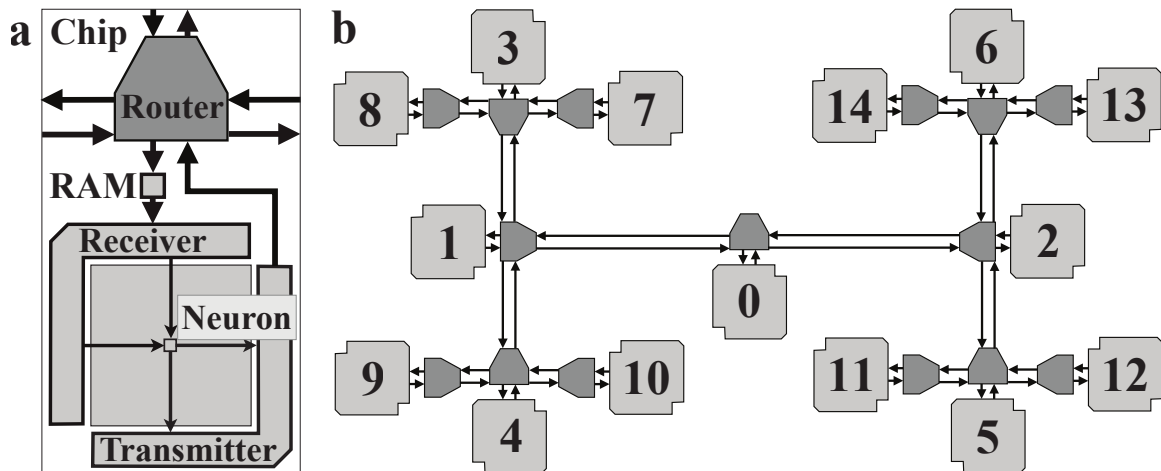


Figure 2.8: Neurogrid: a) Neuromorphic chip with integrated silicon neuron array, receiver, transmitter, RAM, and router. b) Fifteen-node binary tree. Each neuromorphic chip communicates with the others through on-chip routers and interchip links [61].

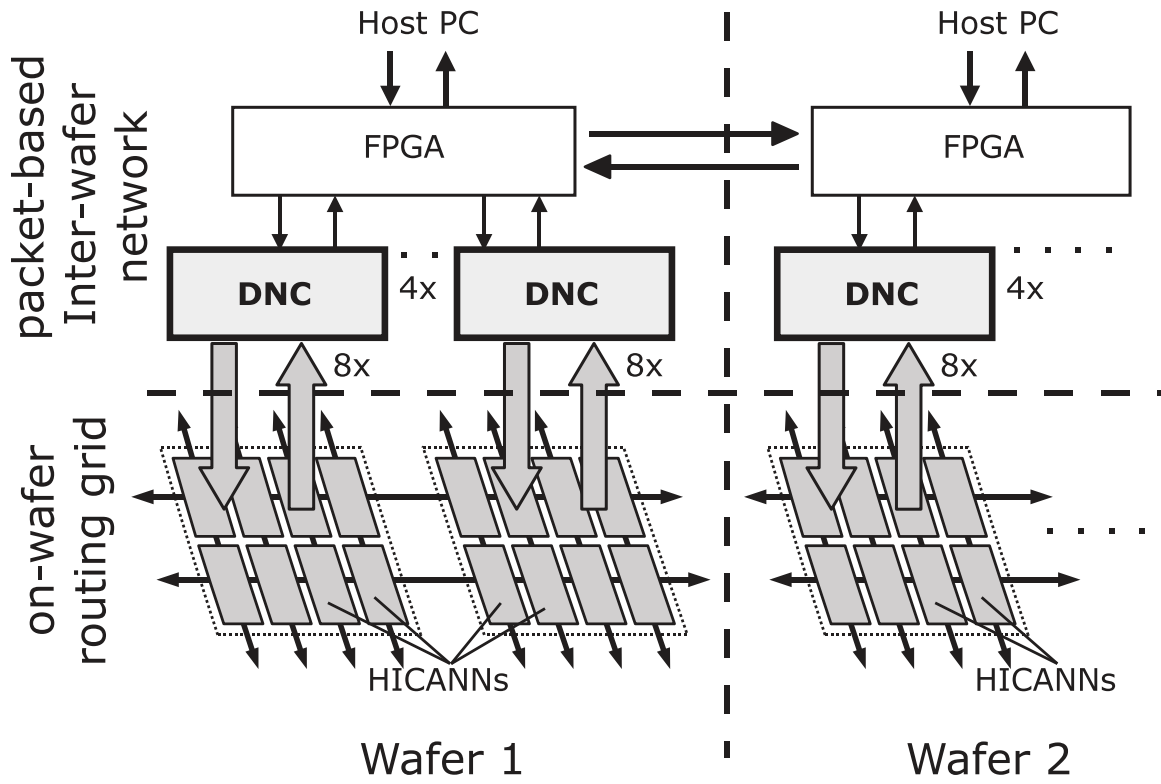


Figure 2.9: Diagram of BrainScaleS's hierarchical L2 routing [82].

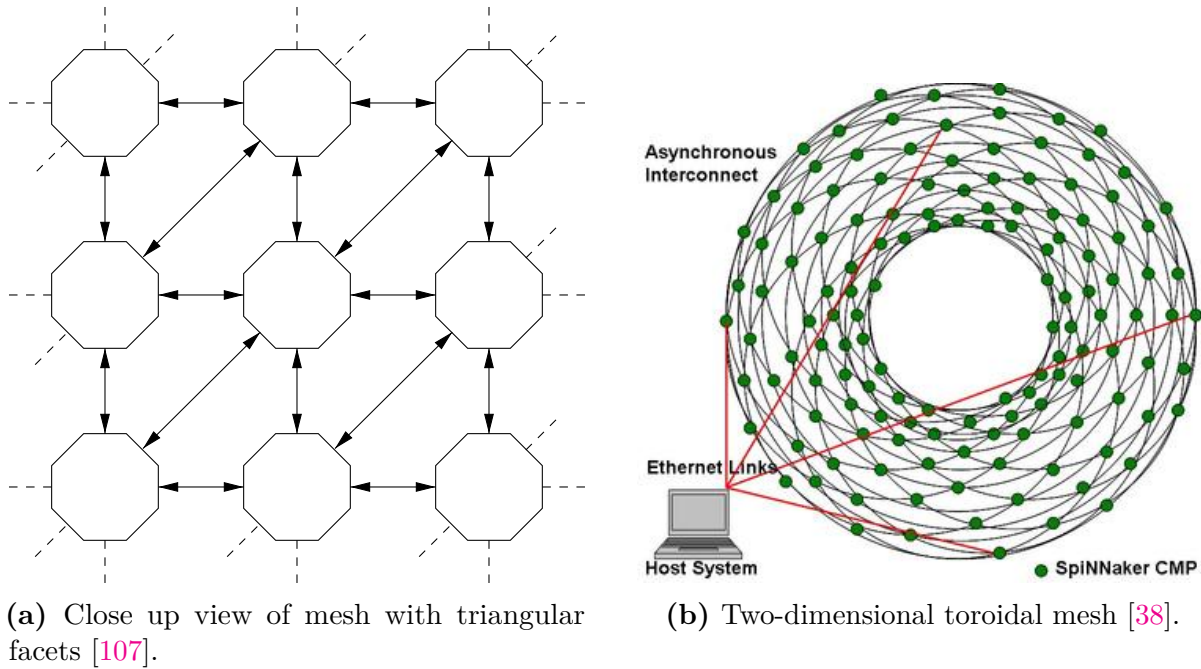


Figure 2.10: The SpiNNaker system network connectivity.

packets. TrueNorth does allow hierarchical communication by sending a spike globally through the network using a single packet. The packet fans out to multiple neurons locally in the destination core. TrueNorth scales to beyond grid boundaries by combining the grid boundaries with native event-driven serializer/deserializer links [78]. Figure 2.11 shows a diagram of TrueNorth chips connecting across chip boundaries. Loihi connects all the cores and processors together in a many-core mesh [23, 105]. The edges of the chip have off-chip communication interfaces to allow Loihi to scale out to many other chips along the four planar directions.

2.4.3 Hybrid

Hybrid designs look to combine the benefits of the previous two approaches. DYNAPs combines hierarchical tree routing with a 2D-mesh in order to minimize the memory resources needed and maximize the network programmability and flexibility. In phase one of packet routing, point-to-point destination-address routing is conducted by the R2 and R3 routers. The R2 routers use a hierarchical tree to route packets within a chip. There can be multiple levels of R2 routers making up the tree. The R3 routers direct the packets among different chips arranged in a 2D mesh with relative destination-address routing [66]. A diagram of this mixed-mode hierarchical-mesh used by DYNAPs is shown in Figure 2.12.

2.4.4 Robustness in Larger Communication

With any global communication network on the same scale as the ones found in neuromorphic processors, robustness of the system is of key importance as failures are almost guaranteed to occur. SpiNNaker has emergency packet re-routing, which allows packets to be sent along an alternative route when a link is detected as failed or congested [38]. SpiNNaker uses acknowledgment packets between monitoring processors to verify that the packet was successfully sent and the communication channel is reliable [9]. BrainScaleS uses a cyclic redundancy check to find corrupted data in the packets [82]. TrueNorth is able to disable and route around faulty cores in the chip so that defects in the chip are hidden at runtime[78].

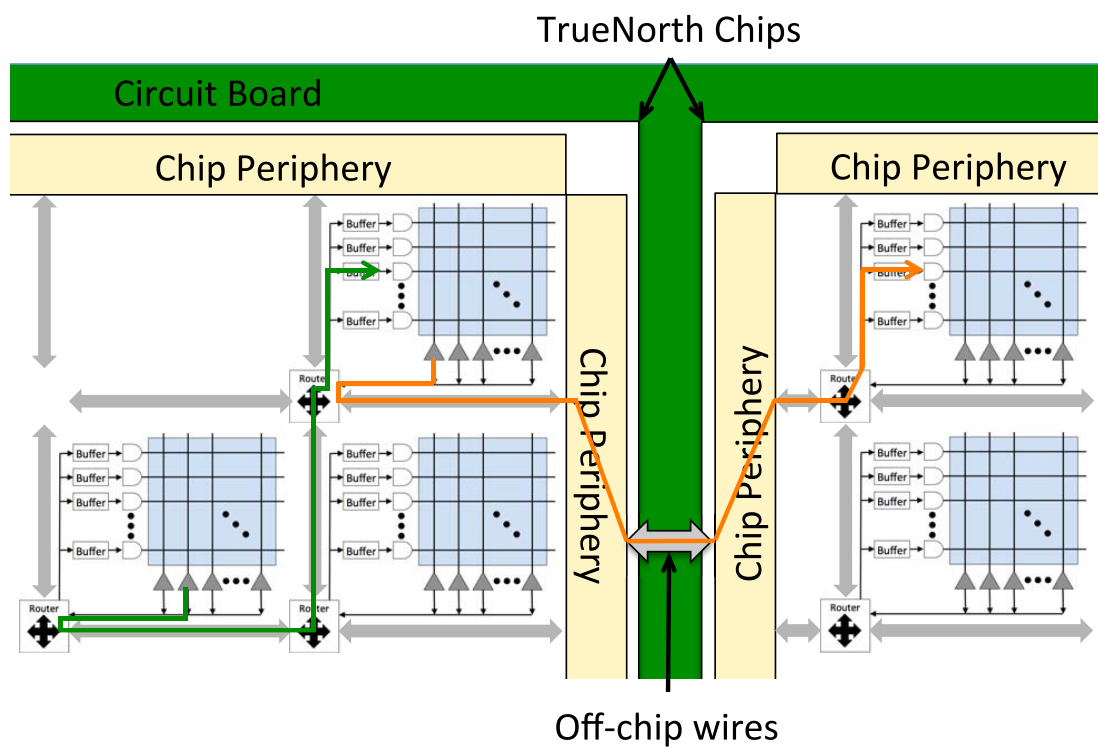


Figure 2.11: TrueNorth cross-chip connectivity [1].

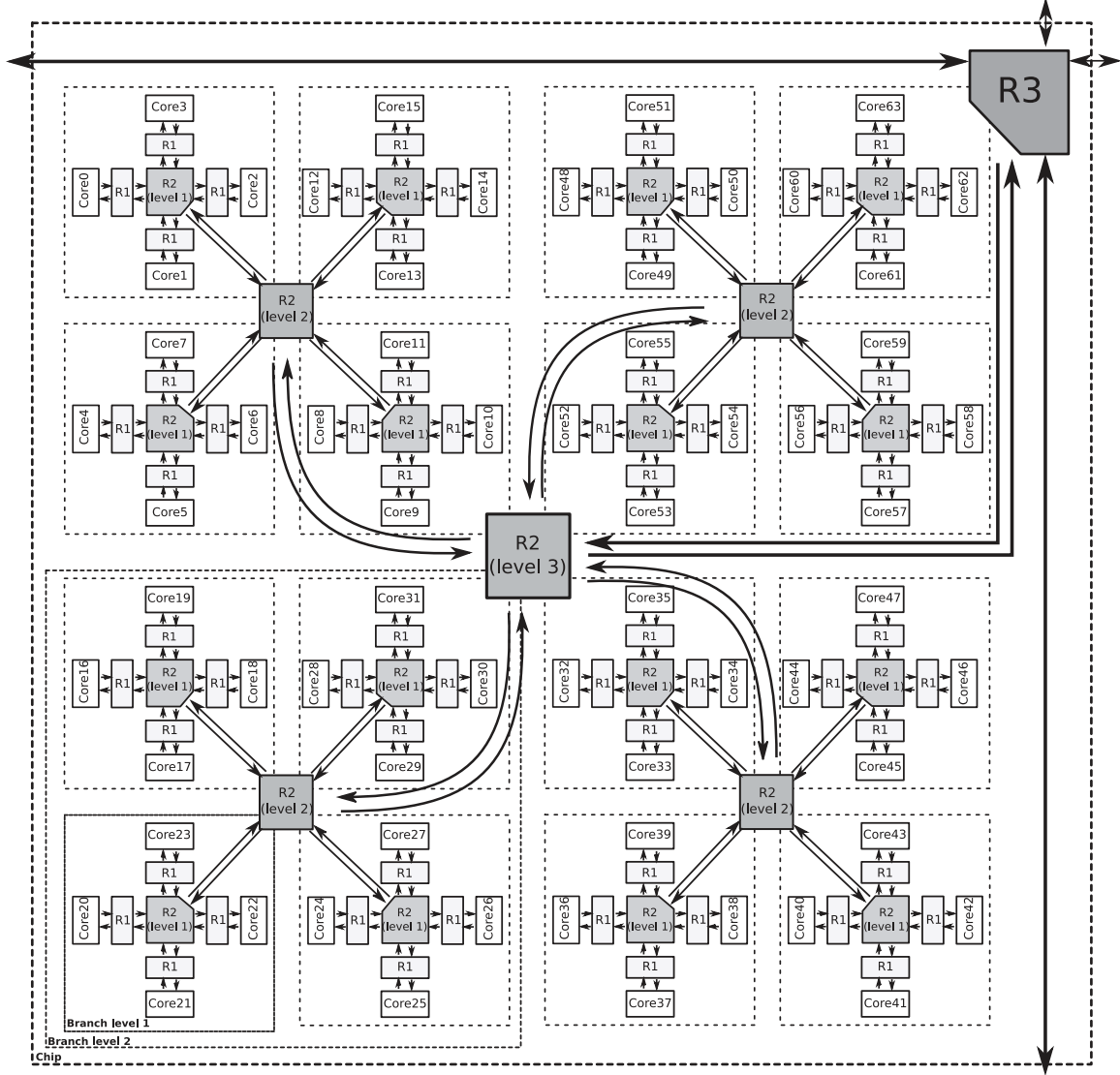


Figure 2.12: DYNAPs hybrid hierarchical-mesh routing scheme example. Individual cores communicate via broadcast operations through the R1 routers. Groups of 4 cores are connected together via a level-1 R2 router. To communicate with cores in different groups, but on the same chip, level-2 or higher level R2 routers are used following a tree-based hierarchical routing scheme. R3 routers are used to communicate to different chips along the four cardinal directions using a 2D-mesh routing strategy [66].

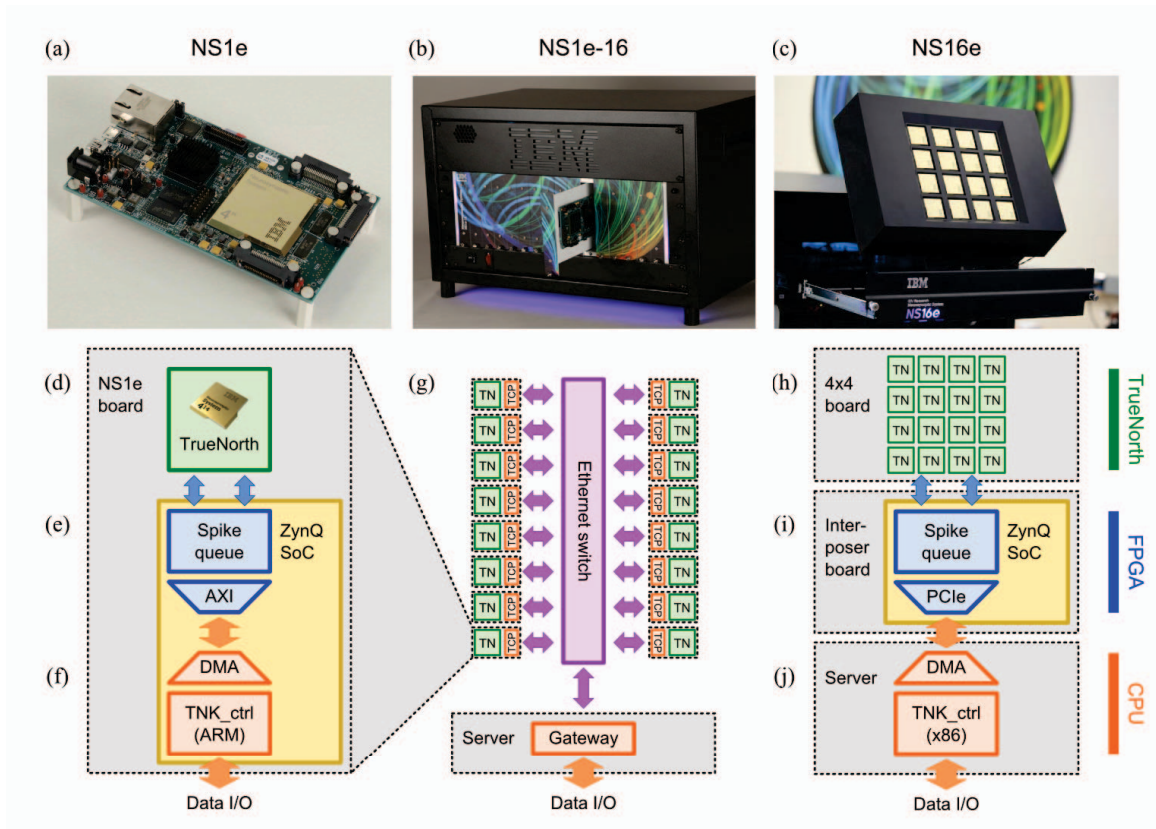


Figure 2.13: TrueNorth NS1e, NS1e-16, and NS16e systems [78].

2.4.5 Host Connection

All the neuromorphic systems are able to connect to a traditional computer to enable the configuration, monitoring, and external signaling. Neurogrid connects to a host computer with USB via the help of a Cypress EZ-USB FX2LP. Neurogrid’s software allows the user to specify the neuronal modules, control the simulation, and visualize the results from running the neural model in real-time [10]. Darwin connects to an off-chip host PC using a USB to UART interface [94]. DYNAPs communicates with a host via an FPGA [66]. SpiNNaker uses multiple 100 Mbps or 1 Gbps Ethernet interfaces to connect to the host, and the monitoring cores on the SpiNNaker chips are used for application support and system monitoring [99, 100]. The smaller TrueNorth NS1e connects to an on-board SoC which functions as the host computer via an AMBA AXI interconnect [78]. The larger NS16e connects to a host computer via a single lane PCIe 2.0. There is also a NS1e-16 system which consists of 16 NS1e boards and a host server networked together over an Ethernet switch. These three TrueNorth systems are shown in Figure 2.13.

2.4.6 Synchronization

Another important consideration is how to handle the synchronization of the network. As mentioned before, the brain is purely asynchronous, and spikes propagate in real-time. This is the approach taken by Neurogrid, Braindrop, SpiNNaker, and BrainScaleS. The downside to not having an explicit synchronization method is that it is impossible to create a cycle-accurate, deterministic simulator of the neuromorphic hardware. This in turn makes it harder to construct and debug Neural Network designs. The alternative is to synchronize the cycle time for the elements, which makes it possible to define a deterministic behavior per cycle and support a cycle-accurate simulator.

There are two main approaches for synchronization. One is to have a fixed cycle time. The other is to allow cycles to have a variable length cycle time and employ other synchronization methods to progress to the next cycle. TrueNorth uses a fixed cycle time, where operation of TrueNorth occurs in two phases [60]. In the first phase, AER packets are routed among the cores. When the packets arrive, they modify the membrane potential of the connected

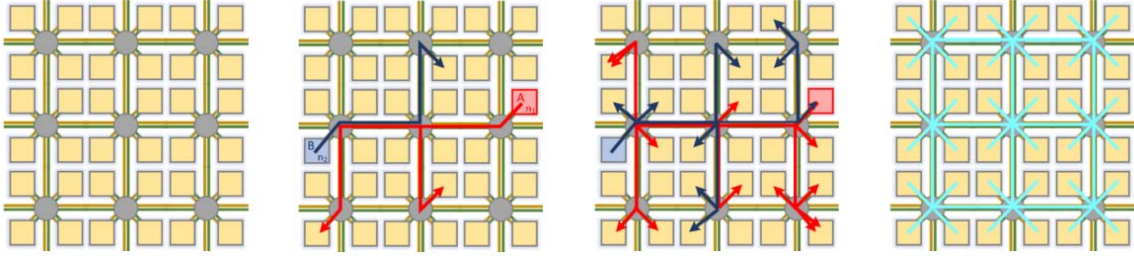


Figure 2.14: Barrier sync operation of Loihi. First box: initial idle state for time-step t (each square is a neurocore). Second box: neurons n_1 and n_2 in cores A and B fire and generate spike messages. Third box: all other spikes from time-step t are distributed to their destination cores. Fourth box: each core exchanges a barrier synchronization message with its neighbors and each core advances its algorithmic time-step to $t + 1$ [23].

neurons. In the second phase, a synchronization event (sync), which occurs every millisecond, is sent to all the cores. Upon receiving the sync signal, the neurons check to see if they should fire, and if so, they send their fire packets to the network. The downside to using a fixed cycle length is that the cycle length has to be longer than the time it takes to send all the packets, resulting in times of no chip activity. If the packets cannot reach their destination in time, then a global error flag is set and the operation of the chip is no longer deterministic.

Loihi and Darwin both have a variable length cycle time. With this method, the timestamp of the network is algorithmic time, and is unrelated to real-time. Loihi uses a mesh-level barrier sync to signify when all the packets have reached their destinations and the timestep can be advanced to the next cycle [23]. This asynchronous handshake provides a significant performance advantage since it eliminates needless idle time in the network and allows the network to run at the fastest speed possible. Figure 2.14 shows a diagram of Loihi’s mesh operation with barrier sync. The speed of the networks is now variable, and how long each cycle takes is set by the slowest component, which bottlenecks the performance of the system. In the case of Loihi, the max speed of the network is limited by the bandwidth of the on-chip router and how long it takes to propagate the fire packets through the network. Darwin also has variable cycle time and uses a time-multiplexing controller to progress the cycle once the previous cycle is finished [53]. The downside to variable length cycle times is that it becomes harder to interface the network with real-time signals and perform real-time operations since real-time and simulation time are separated. The advantages, however, are

that the simulation can run faster than real-time, or at least as fast as possible given the network activity, and they are deterministic in their operation.

2.5 Scalability

Neuromorphic systems are generally designed to have great scaling potential. This allows the systems to either be scaled up to biologically realistic sizes or scaled up based on the complexity of the application being deployed. Other neuromorphic systems target low-power or embedded systems and are purposely designed with less scaling in mind to save resources, since supporting large scaling sizes results in extra overhead in both packet size and storage required.

Neurogrid is designed to scale to 16 interconnected Neurocores on a single board [12]. Braindrop, which will be a single core in a larger scaled up Brainstorm chip, is architected to support a million-neuron multicore system [67].

The BrainScaleS system (NM-PM-1) was scaled up to contain twenty 8-inch full wafer systems [46, 24]. The system is stored in seven 19-inch racks. Five of the racks are used to store the neuromorphic wafer modules and the other two racks are used to store the power supplies and conventional control cluster.

The SpiNNaker system is designed to scale to very large sizes. The SpiNNaker chips are mounted on a board in a 48-node hexagonal array. Then 24 boards are assembled into a crate, with five crates stored in a single rack. High-speed serial cables are used to interconnect the racks [46]. “Virtually any number of racks may be interconnected to form a system of arbitrary scalability [36].” SpiNNaker machines are classified by the approximate number of processing cores in the system. A $10N$ machine has approximately 10^N processing cores. The current largest SpiNNaker machine is the 106 machine with ten 19-inch rack cabinets, each storing five 24-board crates [43, 99]. It has a total of 1,036,800 ARM processing cores, 921,600 of which are for application processing. The machine requires a 100kW (approximate) 240V supply. The other machine sizes are subsets of this large system. The 102 machine is a single PCB with four SpiNNaker nodes. The 103 machine is one PCB with 48 SpiNNaker nodes. Twelve of these 48-node cards are combined into a crate to create the 104 machine.

Five crates are used in a single 19-inch rack cabinet for the 105 machine. Ten rack cabinets are combined to make the 106 machine.

The SpiNNaker 2 project will further scale the system up by switching to a 22FDX process and embedding 144 ARM MF4 cores per chip [45].

Loihi is designed to support up to 4,096 on-chip cores and 16,384 chips. The design is thus constrained because of the design of the mesh protocol and hierarchical addressing scheme [23]. Intel recently announced the largest Loihi system yet—called Pohoiki Beach—which is comprised of 64 Loihi chips with over 8 million neurons. This chip is another milestone towards Intel’s goal of scaling Loihi to 100 million neurons later this year with Pohoiki Springs, which is planned to contain 768 Loihi chips [48]. Loihi also is available in a small USB stick form factor, called Kapoho Bay, which incorporates 1 or 2 Loihi chips. Wolf Mountain is a research board with 4 Loihi chips. Nahuku is an FPGA expansion card which contains 8 to 32 Loihi chips [105].

TrueNorth chips support two kinds of scaling: scale-up and scale-out. Scale-up corresponds to integrating multiple TrueNorth chips onto a single PCB and using TrueNorth’s native interchip asynchronous communication interface for the chips to communicate with each other. This allows the chips to communicate natively, forming a larger unified array of neuromorphic cores. Scale-out corresponds to connecting multiple boards together via standard networking hardware to form a neuromorphic computing cluster. The naming scheme of TrueNorth systems indicates how the system was scaled up from a base single chip design. They are named $NSAe-B$ where A is the scale-up factor and B is the scale-out factor. Note that when B is one, it is left off. The largest TrueNorth system to date is the NS16e-4. This system has a scale-up factor of 16 and a scale-out factor of four. The system is constructed by connecting four NS16e systems together with optical PCIe links within a 4-U rack-mounted standard drawer [28]. A picture of this system is shown in Figure 2.15. Other TrueNorth chips using this same scaling convention as was shown previously in Figure 2.13. Additionally, IBM has deployed internally a NS1e-80, which is a cluster of 80 NS1e boards [28].

Darwin NPU is currently a smaller single chip system targeted for embedded applications; however, the NPU could also be used as a processing element for a NoC architecture. As a

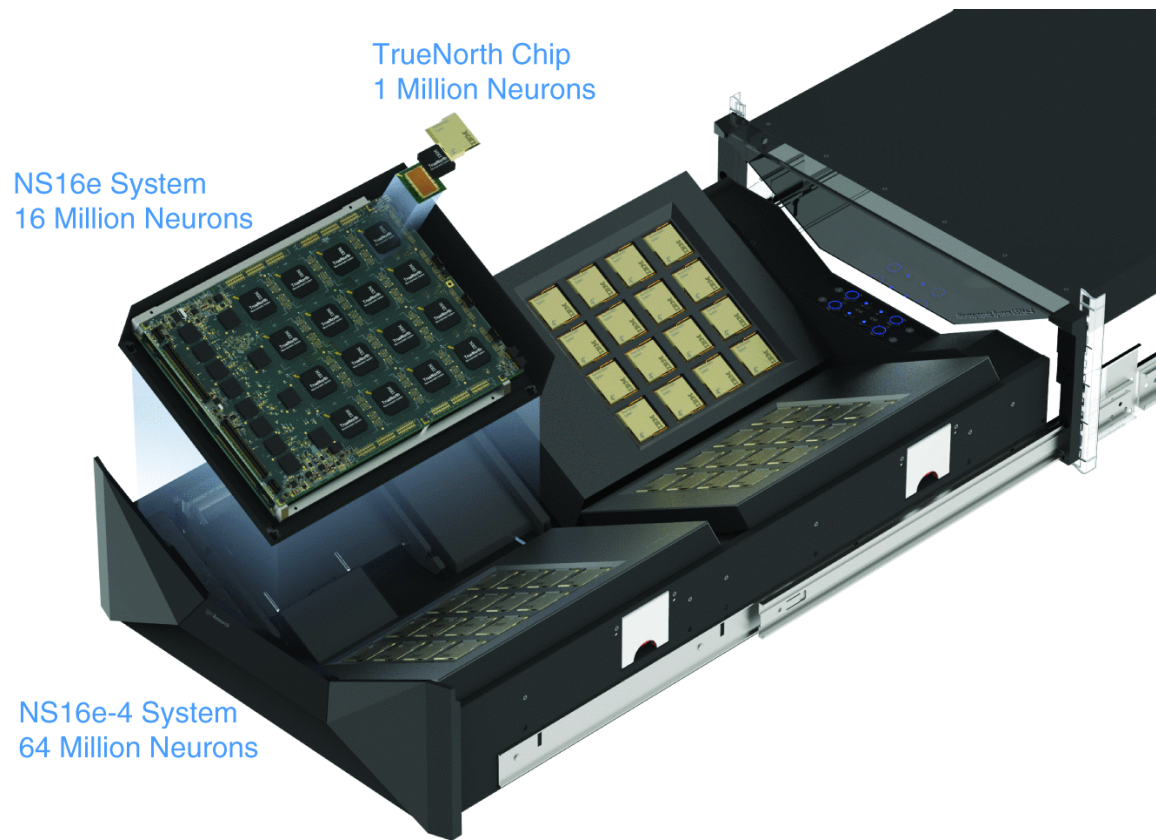


Figure 2.15: The NS16e-4 scaled-up TrueNorth evaluation system [28].

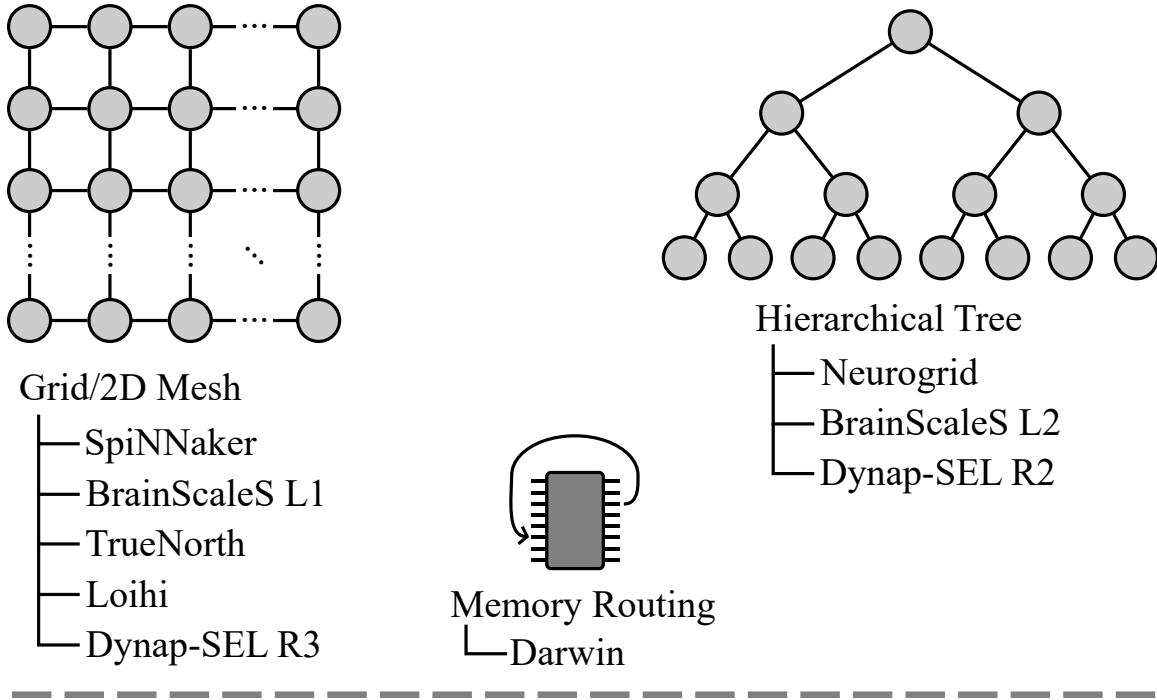
NoC core, Darwin could potentially scale up to millions of neurons on a chip, instead of the few thousand in the current single chip system [53].

Dynap-SEL is designed to be able to be integrated in an array of up to 16×16 chips with all-to-all connectivity among the neurons [98]. Part of the scaling potential of DYNAPs/Dynap-SEL comes from its communication scheme, which allows memory requirements to scale with the number of neurons in a way drastically lower than other standard routing schemes [66].

2.6 Summary

One of the most challenging parts of designing a large scale neuromorphic system is designing a scalable spiking communication network, which is able to keep up with the massive connectivity requirements found in these systems. Table 2.1 and Figure 2.16 summarize the different communication systems found in neuromorphic hardware. These neuromorphic systems are able to efficiently scale up to larger sizes than von Neumann computers can, since they store information with the computation element, which eliminates the von Neumann bottleneck. Neuromorphic systems can be scaled-out with a loose coupling of boards together. For example, the NS1e-16 system from TrueNorth loosely couples 16 single chip boards over an Ethernet network [78]. They can alternatively be scaled-up by tightly integrating multiple systems together, such as the NS16e TrueNorth system, which tightly integrates 16 chips into a 4×4 grid using native tiling. The continued scaling potential of neuromorphic systems is made possible by the exponential decay in hop distance and bandwidth observed in biological neurons [78]. Biologically realistic network topologies have dense clusters of connectivity that are connected together by fewer long range connections [66]. These long range connections also typically have longer signal delay, since the spike must travel a further distance. These two components make it possible to design scalable, large neuromorphic systems with similar topologies. One of the major uses of neuromorphic systems is to use them as co-processors or accelerators to perform computations that are difficult or inefficient to evaluate on a traditional system. Sawada et al. predict that future neuromorphic systems will become a key component of exascale systems [78].

Routing Schemes



Routing Methods

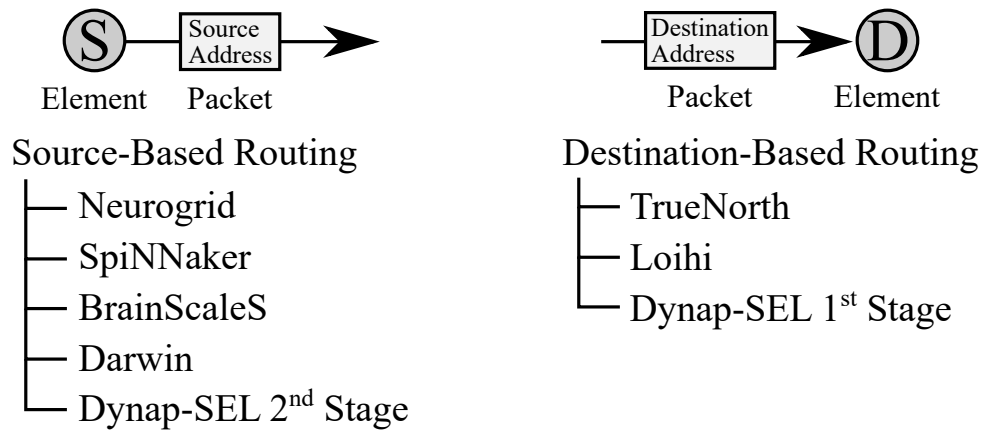


Figure 2.16: Graphical summary of neuromorphic hardware communication systems. The top half of this figure summarizes the routing schemes used by the neuromorphic systems, and the bottom half summarizes the routing methods.

Table 2.1: Summary of neuromorphic hardware communication systems

Neuromorphic Chip	Routing	Max Communication Bandwidth	Connection with Host	Packet Type and Size	Drop Packets?	Simulation Time
Human Brain	Long distance connections between dense clusters of connectivity.			- Electrical pulses - Synaptic cleft	Yes	Real-time
Neurogrid	Multicast Tree Router	- Transmitter 43.4 Mspike/s - Receiver 62.5 Mspike/s - Router 1.17 Gword/s	USB via FX2	- Variable length packets with tailword at the end. - Sequence of 12 bit words that specify route, address, and arbitrarily long payload.	No	Real-time
SpiNNaker	2D triangular mesh wrapped into a torus	5 billion packets/s for the system - Each communication NoC can send at max 7.4 Gbit/s	Ethernet 100Mbps or 1Gbps	8 bit header 32 bit content 32 bit optional data payload - Passed as 4-bit ‘flits’	Yes (configurable)	Real-time
BrainScaleS	- L1 Intra-wafer routing — Asynchronous serial pulse routing grid - L2 Inter-wafer routing — Hierarchical routing tree	- On wafer routing — 32Gbit/s - Inter wafer routing — 2.8 Gevent/s	Ethernet	- L1 — 6 bit neuron numbers (no timestamp) - L2 — 24 bit pulse events (with timestamp)	Yes packets can be discarded if they arrive at the same time or there is network congestion.	10 ⁴ x to 10 ⁵ x
TrueNorth	- Synaptic crossbar for communication within a core - Asynchronous grid routing between cores and between chips - Between chips the buses are multiplexed onto a serial communication bus	- Spike bandwidth is over 640 times lower at chip boundaries than internal to the chip. 160 million spikes per second (5.44 Gbits/sec)	- AXI Bus to SoC - PCIe 2.0 single lane	9 bit dx 9 bit dy 4 bit delivery tick 8 bit axon index 2 bit debug bits	Yes (Global Error Flag when a packet is dropped)	- Meets or exceeds real-time (1x to 21x) - Sync occurs every millisecond to move time to the next time-step
Loihi	- Asynchronous NoC layed out in a 2D mesh - NoC extends in four planer directions to other chips	3.44 Gspike/s cross-sectional spike bandwidth per tile	- Ethernet - USB	- Write - Read request - Read response - Spike message - Barrier message	No (uses barrier sync resulting in variable length network cycle time.)	- Variable cycle time based on load. - Faster than real-time
Darwin	- Routing connections and weights are stored in off-chip SDRAM. - Topology and Weigh-Delay read and used to update Weigh-Sum Queue in Neuron.	Unspecified	UART to USB	- Fixed length with each packet containing the ID of the source neuron. - Time stamp of when the packet was generated.	No (Progresses time once all packets are sent for the previous time step)	70 MHz clock
Dynap-SEL	- Mixed tag-based shared-addressing scheme. - Two stage routing, the first stage is point-to-point. The second stage is a local broadcast. - Point-to-point routers are hierarchical on chip and 2D grid between chips.	27 ns broadcast time 15.4 ns lantency across chips 27 ns broadcast time 15.4 ns latency across chips 30 M events/s (input) 21 M events/s (output)	via FPGA	10 bit tag 6 bit header with dx, dy 4 bit destination	No	Real-time

Chapter 3

Previous Work

The guiding philosophy is not to reiterate or simulate the brain in complete detail, but to search for organizing principles that can be applied in practical devices.

– Don Monroe [65]

3.1 TENNLab Overview

SNACC builds on previous work conducted by the Laboratory of Tennesseans Exploring Neural Networks (TENNLab) research group at the University of Tennessee and Oak Ridge National Laboratory (ORNL) [96]. TENNLab approaches neuromorphic computing design by focusing in three main areas: Applications, Learning, and Processors. The TENNLab software stack is shown in [Figure 3.1](#)

3.1.1 Applications

The area of applications focuses on the tasks that are being solved by the neural networks. They are written in a generic way so that any spiking system could run the application. Many different applications have been written in the TENNLab framework, and more applications are continuously being added [71]. Some of the applications fall into the control category. These applications include an agent in the world that is scored on how it interacts with

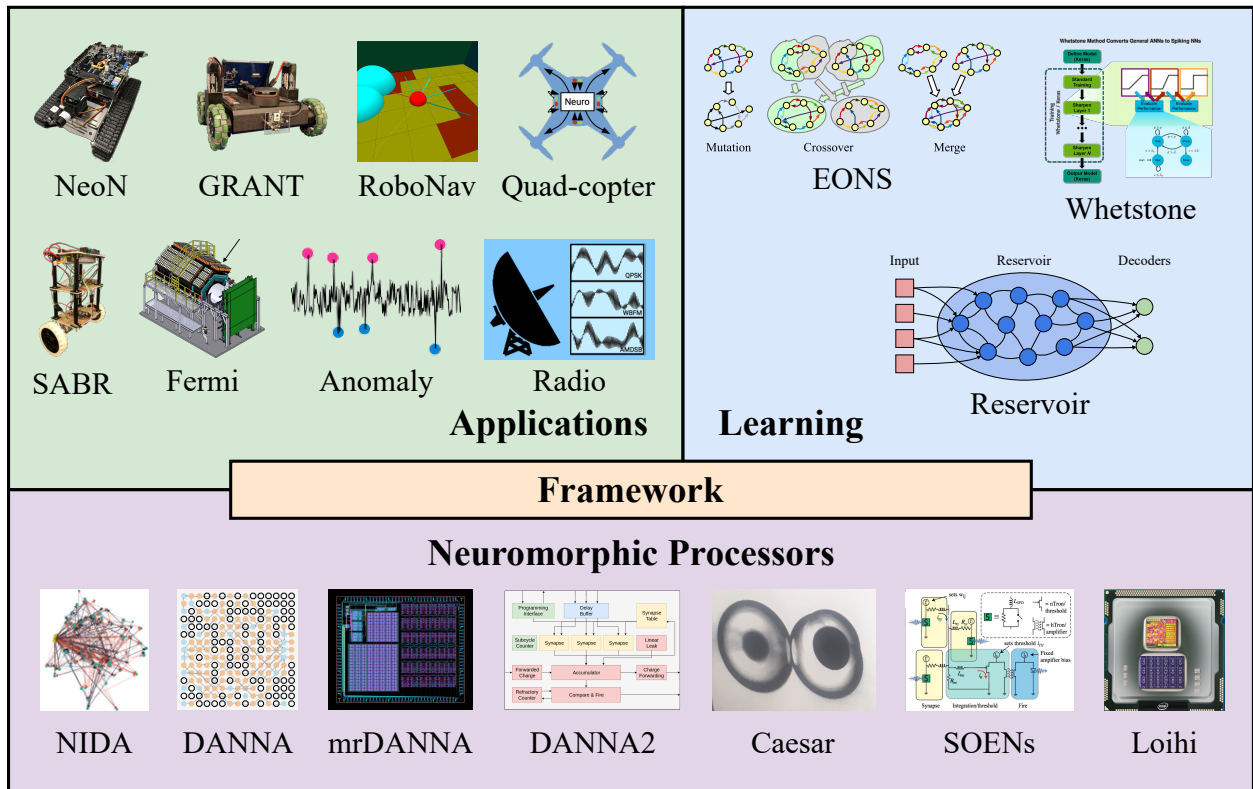


Figure 3.1: TENNLab software stack. With a graphical summary of the different applications, learning algorithms, and neuromorphic processors implemented in the framework.

the world. Applications in this category include the classic pole-balancing control problem, various Atari-like games, and robotics control applications. In order to test the control applications in the real world, three robotics platforms have been designed, built, and used to evaluate networks running on hardware with real-world input. These robotics platforms are NeoN, GRANT, and SABR [64, 2, 119]. We have found that using LIDAR-like sensors for input to the spiking networks yields good results with control applications, and many of our control applications are setup to be able to provide LIDAR-like input to the networks [72]. LIDAR stands for Light Detection and Ranging and these sensors provide a distance measurement from the sensor to the first object the light beam collides with. The LIDAR-like input in these applications measures the distance from the network controlled agent to the other objects in the environment, and these distance values are translated into spiking information for input into the networks.

Another large category of the applications are the classification applications. These applications take as input some number of features, and then produce as output the class or category that the input belongs to. This can be as simple as a binary classifier, where the result is a true/false to indicate belonging to the class, to more complex classification, where the object must be placed in the correct class with multiple possible categories. Some examples of our applications in this category include our general-purpose classification application, blackjack, and anomaly detection. The classification application has tested the neuromorphic systems with many datasets including Iris, Wisconsin Breast Cancer, Pima Indian Diabetes, and Wine [89].

The last category is what I will refer to as logic applications. These applications perform a logical operation and are designed so that they can be composed to form larger applications using the simpler logic operations as the building blocks. Our XOR and comparator applications are some examples that fall into this category.

3.1.2 Learning

The area of learning focuses on how to train networks to solve different applications. Currently TENNLab employs 3 main learning methods. The first and main method used for training by TENNLab is through the use of the Evolutionary Optimization of Neuromorphic Systems

(EONS) [71]. EONS trains networks using a method inspired by survival of the fittest seen in biological evolution. First a population of random networks is created. This population can have a configurable number of networks within it. The random networks are then evaluated with a fitness function to determine how well the network is able to solve the particular problem. The application developer is in charge of implementing the fitness function as part of writing an application. Once all of the networks in the population are evaluated to determine their fitness score, the networks are sorted based on their fitness score. Then a selection algorithm is used to determine which networks “survive” until the next generation. The surviving networks are then used to create the next generation of networks. EONS uses three main genetic operations, generating new graphs from the previous graphs. These operations are mutation, crossover, and merge. The mutation operation randomly changes values or connections in the network. Crossover takes two networks and produces two new networks with components mixed from each. Merge takes two networks and creates a network which is a combination of the previous two networks. The best networks from the previous generation are used as seeds for the genetic operations to generate the next generation. First the merge, crossover, and duplication of networks is performed, and then, the networks may be additionally mutated. Once the new generation has been randomly generated, it is then ranked based on the fitness function and used to generate the next generation. This cycle continues until the target fitness function is reached, compute time runs out, or the target number of generation cycles is reached. [Figure 3.2](#) illustrates this algorithm.

Recently, two new learning methods have been added to TENNLab: Whetstone [92] and reservoir computing [77]. The Whetstone algorithm, from Sandia National Laboratories, works by training a deep neural network and then uses an iterative modification of the network to “sharpen” the activation function into a binary decision. Once the decision is binary, the network can be mapped onto a spiking neuromorphic system.

Reservoir computing uses the neuromorphic networks as the reservoir for transforming input. The network used for a reservoir can either be randomly generated or, alternatively, EONS can be used to evolve a network that has better properties for a reservoir. Once input is supplied to the reservoir, the reservoir will have a non-linear response to the input and

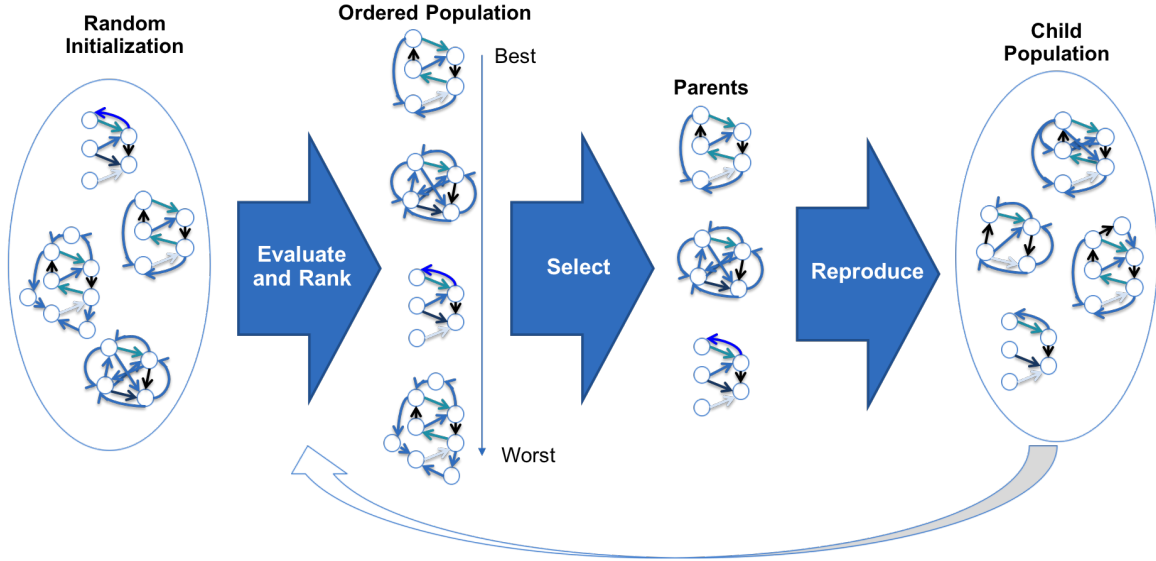


Figure 3.2: An overview of the EONS algorithm.

generate output. The reservoir output is then used as a feature set to train a decoder to produce the desired output.

3.1.3 Neuromorphic Processors

The area of neuromorphic processors focuses on implementing different designs of systems that can run or simulate spiking neuromorphic networks. The framework uses a general graph model called GenGraph [71]. The GenGraph is then used to allow the learning algorithm to implement graph algorithms on a single graph object. The neuromorphic processors will implement functions to convert the GenGraph representation to the internal representation used by the particular processor. The processor can then evaluate the network given the input fire information. The processor then returns the modified network as a GenGraph as well as the fire activity.

Neuroscience-Inspired Dynamic Architecture (NIDA) was the first neuromorphic processor designed by TENNLab [86, 85, 91]. NIDA is a software only, dynamic, spiking neuromorphic model. Neurons are arranged in three-dimensions with synapses connecting the neurons together. The model is continuous, using floating-point numbers to represent coordinates and model parameters. The synaptic delay is the distance the pulses have to travel along

the synapses. Therefore, the location of the neurons plays a large role in the behavior of the system since the location of the neuron influences the timing of the spikes along the synapse. NIDA is also used to help answer the question of which components of the biological brain are needed to perform meaningful computation. Two such mechanisms are leak within the neurons and simple potentiation/depression weight-change mechanisms within the synapses, which are explored by Schuman in [84].

Dynamic Adaptive Neural Network Array (DANNA) is TENNLab’s first neuromorphic processor designed to map well onto digital circuits [26, 31]. DANNA has been used in both FPGA and VLSI Designs [25]. The FPGA implementation is written in VHDL and is designed for Xilinx FPGAs. DANNA arranges elements in a two-dimensional array with location values and parameters represented as integers. One thing that makes DANNA unique is that each element can be configured as either a neuron or a synapse. The neuron elements accumulate charge and compares the charge value to a threshold to determine when to fire. The synapse element conceptually functions like the edges to the directed network graph. The synapse elements only have one input from a neuron element and add a delay to the spike before sending it to an output neuron. In a normal configuration, the neuron elements should only be connected to a single synapse element. The synapse element is then connected to a neuron element. The neuron elements can listen from any element within the 16 nearest-neighboring elements. This element connectivity pattern is shown in Figure 3.3. DANNA supports neuron leak and simple synaptic potentiation and depression. DANNA also supports monitoring the state of the internal network. As potentiation and depression occur, the weight values of the synapses changes. Certain algorithms require the ability to read out the new values of these weights. Therefore, the capture and shift commands are used to retrieve information from the hardware. The capture command tells each element to store its run statistics and current state information inside its shift register. The shift command is then used to read out values from the shift register one row at a time. The number of shift commands required to read the entire array depends on the number of rows in the network. Figure 3.4 illustrates the capture-shift process. Previous work done for my master’s thesis looking at improving the way the host machine connects to a DANNA array loaded onto an FPGA. Section 3.4 discusses this previous work.

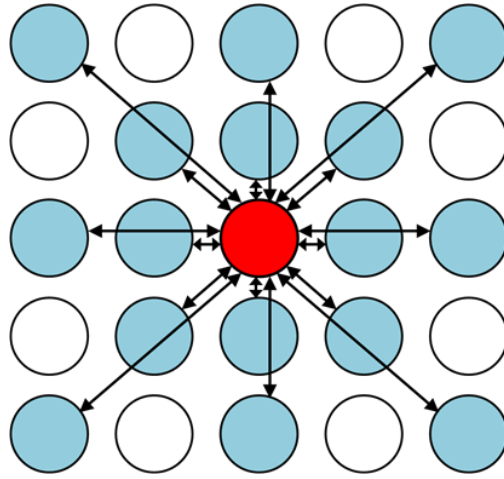


Figure 3.3: DANNA element connectivity [18].

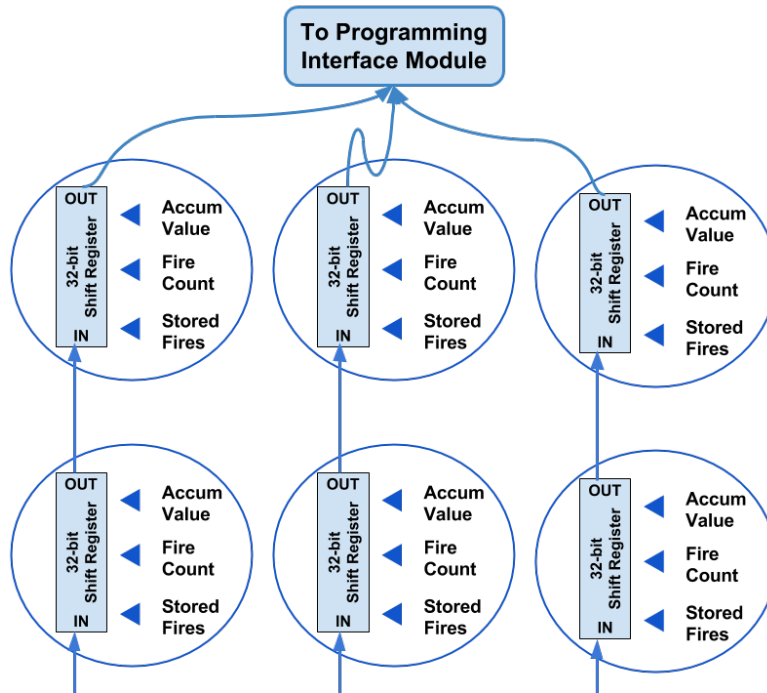


Figure 3.4: Monitoring capture-shift diagram [18].

The memristor-based Dynamic Adaptive Neural Network array (mrDANNA) is TENN-Lab’s neuromorphic processor which uses emerging memristive devices [17, 16, 68]. The mrDANNA architecture is based on NIDA. Like DANNA, mrDANNA arranges elements within a two-dimensional grid with integer values for the element parameters. Each element in mrDANNA contains eight synapses and one analog integrate-and-fire neuron. The mrDANNA architecture is mixed-analog-digital. The control circuitry is digital, and the synapses and neurons are analog. Two memristors per synapse are used to store weight values in the synapse design. The neuron design uses capacitors and operational amplifiers to implement a leaky integrate-and-fire neuron.

DANNA and mrDANNA have recently been fabricated in a test chip using a hybrid memristor-CMOS process flow in SUNY Polytechnic Institute’s 300mm research foundry [16, 15]. The design is implemented in 65nm CMOS/ReRAM using the 300mm wafer platform. The research chip has memristor-CMOS hybrid neural network components, a 512×512 addressable ReRAM block, memristive reservoir computing circuits, multiple individual ReRAM and transistor-ReRAM test circuits, and a fully-digital DANNA.

DANNA2 is the second iteration of DANNA which incorporates many new design changes and features based on what was learned from DANNA and mrDANNA [62, 63]. Just like DANNA, DANNA2 arranges elements in a two-dimensional grid. One major change is that 24 synapses are now incorporated along with one leaky integrate-and-fire neuron in each element. This reduces the complexity and confusion that results from having each element either implement a neuron or a synapse. It also reduces the complexity in the implementation for Spike-Timing-Dependent Plasticity (STDP) by looking for causal fires on a global timestep granularity instead of a sub-cycle granularity. This makes the model easier to implement and also easier to reason about since there no longer needs to be a random number generator to determine the fire checking order on sub-cycles. DANNA2 also introduced the concept of grid and sparse networks. A grid network contains every element within a grid of a fixed size and each element may be connected to its 24 nearest neighboring elements. This nearest neighbor connection pattern is shown in Figure 3.5. A sparse network lifts the nearest neighbor restrictions and these 24 synaptic inputs can come from any other neuron in the network. Example grid and sparse networks trained with EONS are shown in Figure 3.6.

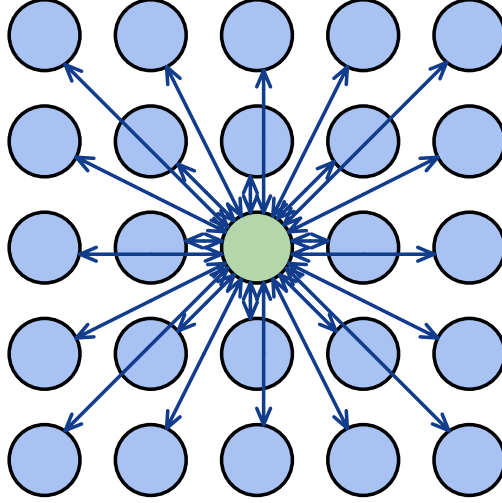


Figure 3.5: DANNA2 element connectivity.

DANNA2 is the model used in the SNACC implementation and will be discussed further in [Section 3.3](#).

Our group also works closely with other researchers to add additional neuromorphic processors to our framework that were not designed by TENNLab. Among the currently included neuromorphic processors are the Caesar, SOENs, and Loihi processors. Caesar uses water and lipid “bubbles” built by the Mechanical Engineering Department at the University of Tennessee to implement the memristor used in the synapse design [42, 104]. A pulse rate is used instead of a charge to represent activity. The bubbles consist of an alamethicin-doped synthetic biomembrane which functions as a soft, two-terminal biomolecular memristor device. This closely mimics biological synapses in ionic transport modality, structure, and switching mechanism. The bubbles operate with very low power and are able to reproduce the behavior of cellular membranes and that of a memristor, which is part of the reason the bubbles are interesting to study. The SOENs model is developed by the National Institute of Standards and Technology and uses super-conducting optoelectronics to create the neuromorphic processor [14]. Intel’s Loihi model discussed in [Section 2.1.4](#) is also supported in the TENNLab framework.

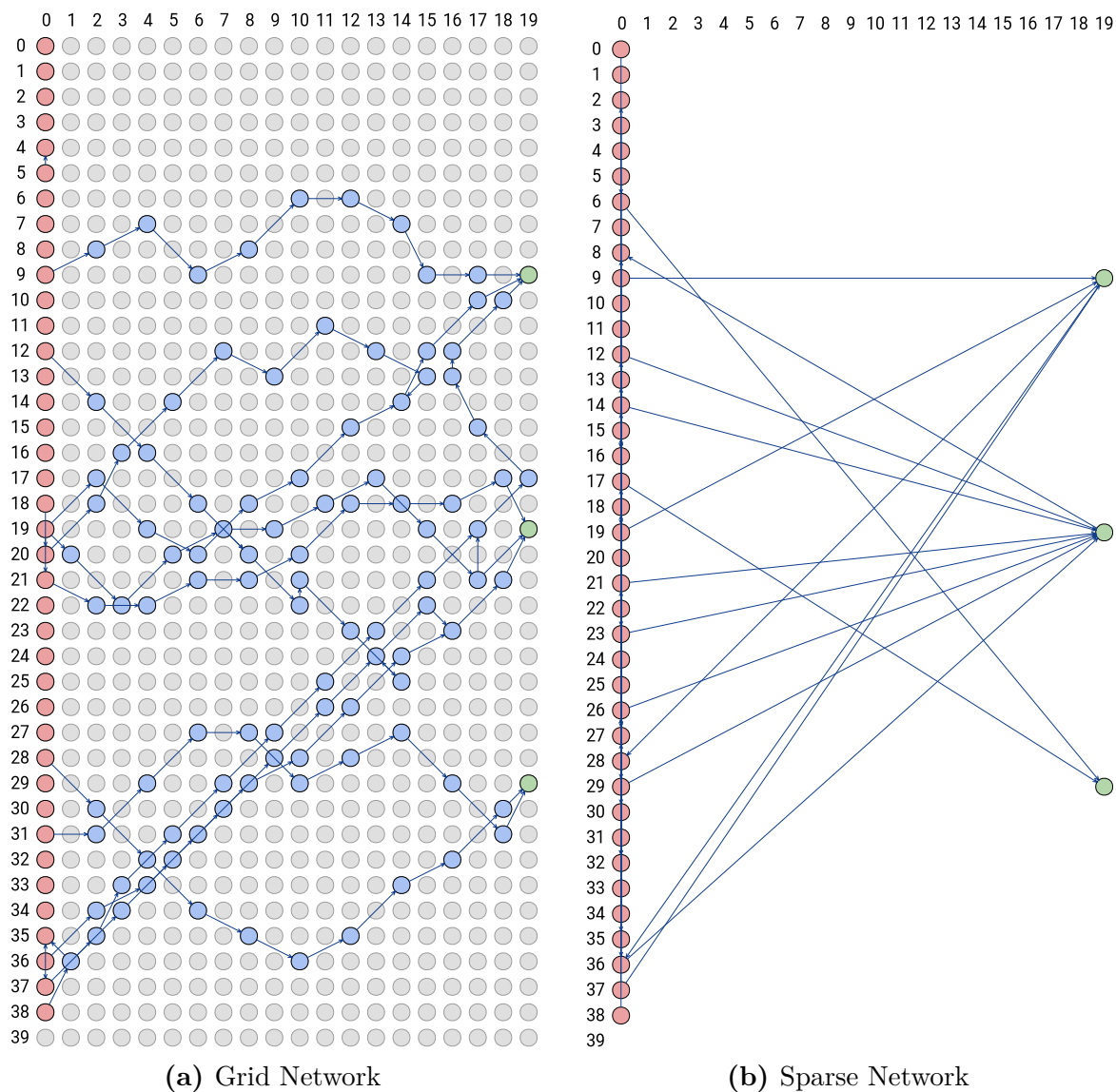
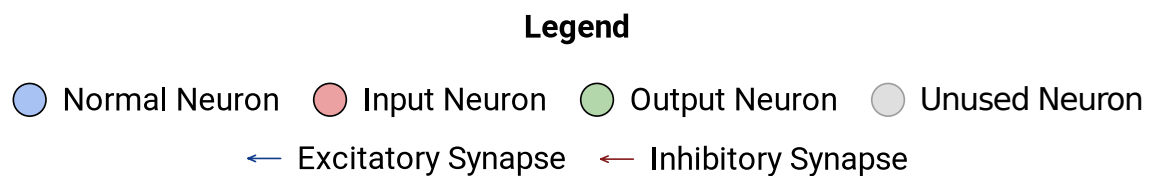


Figure 3.6: DANNA2 example networks.

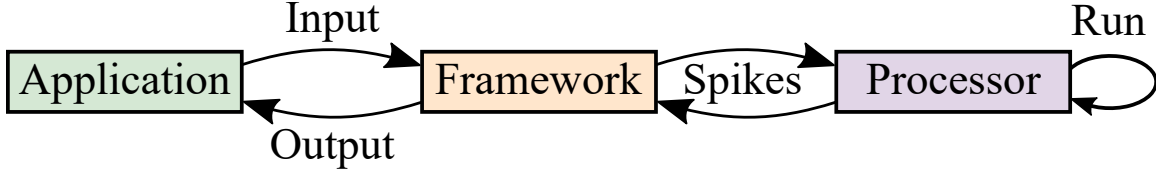


Figure 3.7: The main loop of an application running in the TENNLab framework on a neuromorphic processor. Applications express their input and output as values, whereas the processors handle input and output as spikes.

3.1.4 Summary

The goal of TENNLab is to bring the three areas of applications, learning, and processors together into a unified software stack that enables research into nearly every facet of neuromorphic computing. The power of the software stack is that it allows for interchanging any component to conduct research. For example, any application, learning algorithm, or processor can be chosen for a training run. If you hold two components constant then you can compare the affects of changing the third. This allows for different learning algorithms, processors, or applications to be compared [70].

The ability to make comparisons between the different applications, learning algorithms, and neuromorphic processors is all made possible by the design of the TENNLab framework [71]. Figure 3.7 shows the main loop of application running on a neuromorphic processor. The application keeps track of its state. The application then sends input values to the framework. The framework converts the values into spikes. The spikes are created so that they are appropriate for the selected neuromorphic processor. The neuromorphic processor then evaluates the loaded network with the provided spiking input. Then, the resulting spiking output is sent back to the framework. The framework converts the spikes back into values and sends the data to the application as the network’s response. The application then uses the response to update its state; after which, it provides the next set of input values to the framework.

3.2 Biologically-Realistic Versus Biologically-Inspired Computers

There are two main approaches to neuromorphic computing. The first is to mimic the brain as precisely as possible and to use this model to further our understanding of the brain. The second approach realizes that the brain evolved specifically to be a very efficient biological system, the brain does not need to be modeled precisely to be a powerful computing device. One goal of this second approach is to determine what is possible with certain biological ideas and what components are necessary to build powerful computing systems. Our group leans heavily towards the second approach. We prioritize researching the potential of abstracted neuromorphic systems which take inspiration from biological systems, but are not designed to be precise simulators of biological systems. Our neuromorphic processors implement various Spiking Recurrent Neural Networks (SRNNs) that can be designed with elements using various technologies.

3.3 DANNA2

DANNA2 is used as the neuromorphic architecture in the SNACC system [63, 62]. This section will cover the design and components of DANNA2.

3.3.1 Model

DANNA2 follows a simple SRNN design inspired from the group’s previous work on DANNA and mrDANNA. DANNA2 uses neurons and synapses as the network primitives used to build the neural networks. When the network is viewed as a general graph, the neurons are the nodes of the graph and the synapses are the edges of the graph.

Neurons

The neurons follow a discretized, leaky integrate-and-fire model, where each neuron accumulates charge until a configurable activation threshold is reached. The charge is accumulated based on the fire events coming from the input synapses. The amount of change

in the charge is based on the strength or weight of the synapse. Once the threshold is reached, the neuron will emit a spike to each of the downstream synapses. After the neuron fires, it enters a refractory period where it may not fire again until some configurable amount of time has passed.

The neurons have multiple parameters used to configure their behavior. Activation thresholds are stored as 10-bit unsigned integers up to 1023, and the refractory period may be up to 7 network cycles after the neuron’s last fire. DANNA2 optionally supports linear charge leak, where the charge stored in a neuron may decay back to a resting state following a programmable linear rate.

The neuron charge function for DANNA2 may be expressed as the function [63]:

$$H_{kj}(t) = \sum_{i=1}^N w_i(t)x_i(t) + H_{kj}(t-1) - L_{kj}(t-1) \quad (3.1)$$

The neuron activation function may be expressed as [63]:

$$a_{kj}(t) = f(H_{kj}(t)) = \begin{cases} 1 & \text{if } H_{kj}(t) \geq \theta(t) \\ 0 & \text{if } H_{kj}(t) < \theta(t) \end{cases} \quad (3.2)$$

Synapses

Synapses form the connections between the neurons. They transmit the spike from the presynaptic neuron to the postsynaptic neuron. The synapse controls the time the spike takes to arrive and the weight of the pulse. The weight determines the change in charge the postsynaptic neuron will accumulate.

DANNA2’s synapses are designed to represent weights with a signed 9-bit integer, which allows for weight values between -256 and 255. The delay for the synapses can be any delay up to 15 additional network cycles. Each neuron supports a maximum of 24 incoming synapses; however, this number may be increased by using charge forwarding with some of the neurons configured to act as fan-in elements.

Synapses also may leverage a simple Spike Timing Dependent Plasticity (STDP) model to adjust the synaptic weights during runtime based on the firing patterns of the neurons.

The DANNA2 STDP model uses a discretized lookup table keyed from the time difference between the synapse fire and the postsynaptic neuron fire to determine how much the weight should change.

$$\Delta t = t_{\text{synapse_fire}} - t_{\text{neuron_fire}} \quad (3.3)$$

Using a lookup table allows different weight change functions to be approximated. The table contains potentiation values located from $\Delta t = -N + 1$ to $\Delta t = 0$ and depression values from $\Delta t = 1$ to $\Delta t = N$, where $2N$ is the temporal window for STDP. When $\Delta t = 0$, the synapse fired in the same cycle that the neuron's charge reached its threshold.

Networks

Neuromorphic networks are formed by constructing a sparse, directed graph where the nodes are neurons and the edges are synapses. A valid network must follow the constraints of the system. For example, a valid graph that can run on hardware is restricted to 24 synapses per neuron, but this limitation can be relaxed in the simulator to experiment with different constraints. Networks have point-to-point connections, and two synapses can be used to create a bidirectional connection between two neurons. These networks also commonly have recurrent cyclical patterns to store information from previous cycles within the network. The networks take spiking input via neurons that are labeled as input neurons, and the network produces spiking output from neurons labeled as output neurons. The input spikes have a scaled weight range from 0 to 255. This range corresponds to the positive weight range of a single synapse and gives more flexibility with how values are encoded as spiking input. The output spikes are binary events which just indicate the presence of a spike. This setup mimics biology; each neuron accumulates weighted values transmitted across the synaptic cleft and produces a weightless fire event, which mimics an electric pulse sent down the axon. Having the input and output mimic the brain makes it possible to compose networks together to build larger, more capable networks using the output from one network to drive another network. It is possible to have every neuron be an input and an output neuron; however, design limitations make this infeasible for all but the smallest networks in hardware. The

typical convention is to have input neurons located on the left side of the network and to have output neurons located on the right side of the network.

3.3.2 Hardware Architecture

DANNA2’s hardware architecture is designed to map well onto Xilinx’s Virtex-7 FPGA architecture [108]. As such, any hardware implementation choices were made to effectively utilize the FPGA’s resources.

Element Design

Each DANNA2 element consists of 24 synapses with a postsynaptic neuron tightly integrated into a single element. The digital implementation of this element is shown in [Figure 3.8](#). A collection of registers store the input fire values for each incoming synapse. During each element cycle, three synapses are read into the synapse unit from the corresponding distance register and with the appropriate parameters from the synapse table. The synapse units send the synapse fire information with the correct weight to the accumulator to be summed together. The synapse units also handle STDP weight changes in response to a fire. The accumulated weights are compared in the compare-and-fire unit to determine if the neuron fires. The leak configuration sets up the leak amount, slowly causing the charge on the accumulator to return to its resting state. The linear leak rate can be set on a per-neuron basis and is added in the accumulator as a negative weight. The programming interface is in charge of loading the element’s configuration when a network is loaded onto the hardware. The hardware implementation of DANNA2 uses element cycles and network cycles where a network cycle occurs every 10 element cycles. The 10 element cycles serve to reuse the components of the element so less logic is required. Each cycle uses the element hardware to perform part of the computation task. It takes eight cycles to process each of the 24 synapses, one cycle to accumulate and latch output, and one cycle to latch input.

The element can also be used in a fan-in mode to increase the effective number of input synapses available to a neuron. The 24 synapse values can be forwarded to another neuron with the addition of one network cycle of delay. Using this charge-forwarding method, a

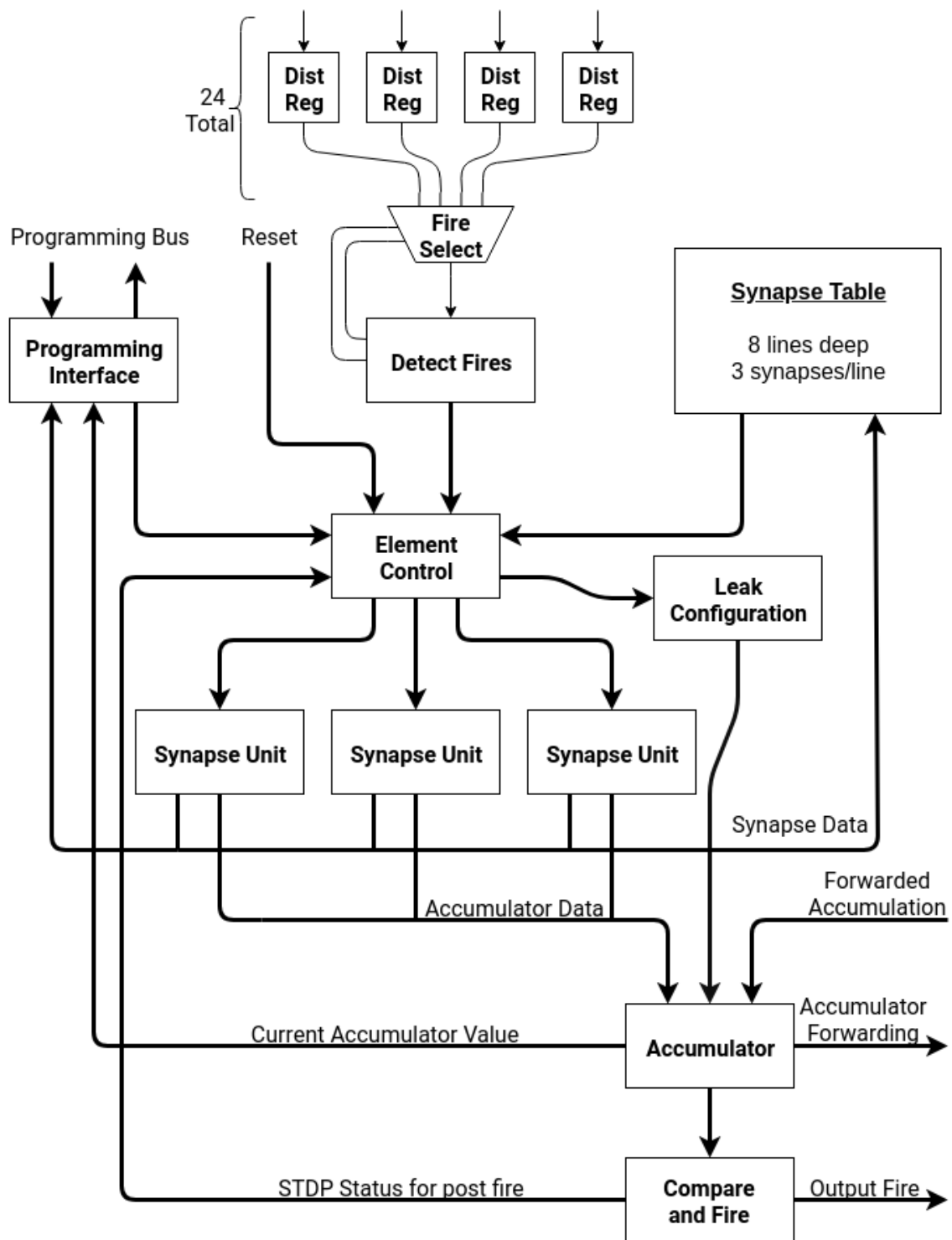


Figure 3.8: Diagram of a DANNA2 element.

fan-in tree can be constructed to increase the synaptic fan-in to any amount. Each element has a maximum of four fan-in ports available to receive the accumulated charge from a fan-in element.

Array Design

DANNA2 neural networks are loaded onto an array of elements where they may then be executed. Two primary array types are each supported with different constraint, complexity, and efficiency trade-offs.

Reconfigurable Grid Arrays A reconfigurable grid array is a rectangular grid with fixed dimensions and a fixed number of elements. Connections between elements are constrained to the 24 nearest neighbors within a five-by-five grid around the element. Self connections are not supported. Limiting connections to spatially local neighbors allows the grids to be able to scale to millions of elements. This local connection scales infinitely in theory. The 24 nearest neighbors of a element in a grid array were previously shown in [Figure 3.5](#). The wiring of the connections are fixed, but each synapse can be configured in order to enable the synapse and specify its weight and delay. This architecture supports the creation of programmable VLSI and FPGA designs, where any grid array of the corresponding size can be loaded onto the hardware design. Thus, the fixed hardware design has the ability to be reconfigured to support any valid network configuration, allowing the hardware to run many varying applications.

Sparse Arrays A sparse array is still a rectangular grid with fixed integer positions; however, only the elements and connections needed for a given network are implemented. There are two main advantages for this. First, connections are no longer limited to the 24 nearest neighbors. The maximum number of connections is still 24, but these synapses can be connected to any other element. Second, the amount of resources required to implement a network is typically less since only the elements that are used are created. The downside with sparse arrays is that they are no longer reconfigurable. Once the hardware design is created, the network cannot be changed. For VLSI designs, this is a prohibitive limitation since

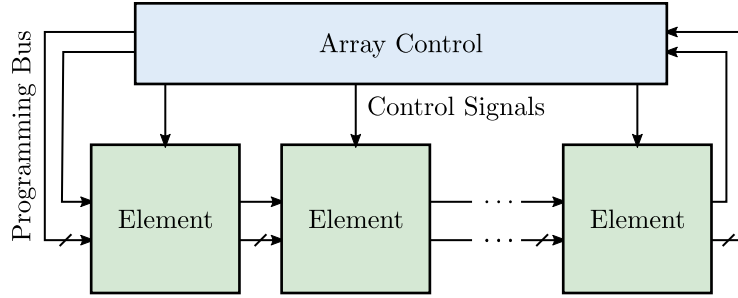


Figure 3.9: Depiction of DANNA2’s element control.

the hardware would be special-purpose and would no longer be able to be reconfigured for different tasks. For FPGA designs, this means that a new bitfile will have to be synthesized for any change in the network design. Sparse arrays still have their uses; the connectivity patterns are much more flexible, and larger sparse networks can be implemented using less space than is possible with a configurable grid array, which would have unused elements.

3.3.3 Element Control

Each element receives three globally routed signals: a 100 MHz clock¹, a run enable signal, and a global reset. In addition to these global signals, each element is connected together via a simple programming bus. The programming bus connects adjacent elements in a chain, which allows an element to receive packets from its previous neighbor and send packets to the next element. The programming bus is used to configure each element in the array. One programming packet is sent per element to configure. [Figure 3.9](#) shows a diagram of DANNA2’s element control scheme. The programming bus is implemented as a communication chain, where programming packets are passed from element to element along each link of the chain.

The programming bus has a 48-bit data bus and a packet start signal. The packet start signal is asserted to indicate the start of a programming packet. The first 48-bit word of the programming packet contains the packet’s destination address. The next nine words on the data bus make up the complete programming packet and contain configuration information. [Figure 3.10](#) shows this packet. Address 0 is reserved and element addresses start at 1.

¹100 MHz is a constraint of the FPGAs used and could be slower or faster on other hardware.

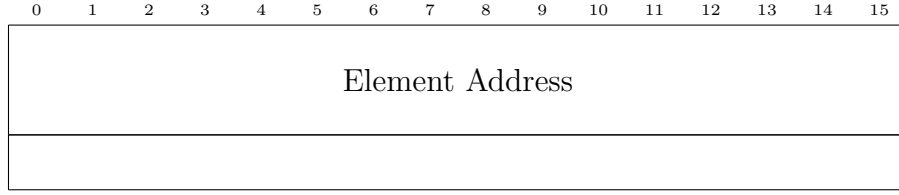


Figure 3.10: Layout of the initial addressing packet sent over the programming bus.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Weight									Delay						
Weight									Delay						
Weight									Delay						

Figure 3.11: Layout of synapse programming information sent over programming bus.

If the first word’s address matches the current element’s address, the element loads the configuration found in the following words. Eight sets of synapse data words are sent with the data layout shown in [Figure 3.11](#). The tenth and final word (shown in [Figure 3.12](#)) contains the configuration data for the neuron as well as a bit to enable or disable STDP. If the addresses do not match, the element forwards the programming packets on to the next element in the chain.

DANNA2 has multiple programmable parameters which are listed in [Table 3.1](#). These values are configured using the programming bus according to the network specification.

3.3.4 Array Control & Communication Interface

DANNA2 uses AXI4-Stream and PCIe in a manner very similar to the DANNA communication interface. These methods are further discussed in [Section 3.4](#) and [Chapter 6](#). DANNA2 has a different packet structure than DANNA for packets sent between the host and the neuromorphic array. The following sections discuss the structure of DANNA2 packets.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Threshold										Refractory			Leak		
S	F														

Figure 3.12: Layout of neuron programming information sent over programming bus. The ‘S’ bit enables or disables STDP and the ‘F’ bit enables or disables fan-in.

Host to Array Packets

There are three types of input packets the host can send to a DANNA2 array. The first type is a configuration packet which is used to program a single element. [Figure 3.13](#) shows this type of packet. In order to load a new network into the hardware array, one programming packet will have to be sent to each element in the array. The packet specifies the threshold, refractory period, and STDP-enable parameters to configure the neuron, as well as the weight and delay parameters used to configure the synapses. An element-type bit is used to configure the element as a fan-in element.

The second type of packet is used to send step-and-fire commands into the array. This command is also used to tell the hardware to run to a timestep in the future. These packets must appear in order of increasing time, and receiving a packet for time t lets the hardware know that it can run to time t . If the packet has no fire events, then it only lets the hardware know that it can run up to the timestep and that there is no input before that time step. This packet layout is shown in [Figure 3.14](#). Since each input fire weight is encoded as a 4-bit value, the input packet structure supports a maximum of 104 input neurons.

The final type of input packet is the reset packet. This packet causes all the activity within the network to be cleared, and does not change the configuration of the network. To clear the configuration of the network, a new network has to be programmed in. The reset packet is shown in [Figure 3.15](#).

Array to Host Packets

An output packet is used to send output fires from the array to the host. [Figure 3.16](#) shows

Table 3.1: Table of Programmable Element Parameters

Parameter	Bits	Range
Neuron Threshold	10	[0, 1023]
Refractory Period	3	[0, 7]
Charge Leak	3	[0, 7]
Synapse Weight	9	[-256, 255]
Synaptic Delay	4	[0, 15]
STDP Learning	1	Enable/Disable
Element Type	1	Fan-in/Regular

0	3	7	11	15	19	23	27	31
Packet Type								
Element Address (32 bits)								
Threshold						F	R	L S
Weight 0			Delay	Weight 1				Delay
Weight 2			Delay	Weight 3				Delay
Weight 4			Delay	Weight 5				Delay
Weight 6			Delay	Weight 7				Delay
Weight 8			Delay	Weight 9				Delay
Weight 10			Delay	Weight 11				Delay
Weight 12			Delay	Weight 13				Delay
Weight 14			Delay	Weight 15				Delay
Weight 16			Delay	Weight 17				Delay
Weight 18			Delay	Weight 19				Delay
Weight 20			Delay	Weight 21				Delay
Weight 22			Delay	Weight 23				Delay

Figure 3.13: Configuration Packet Description. The ‘F’ bit signifies if the element is a fan-in element. The ‘R’ bits set the refractory period. The ‘L’ bits specify the charge leak. The ‘S’ bit enables or disables STDP.

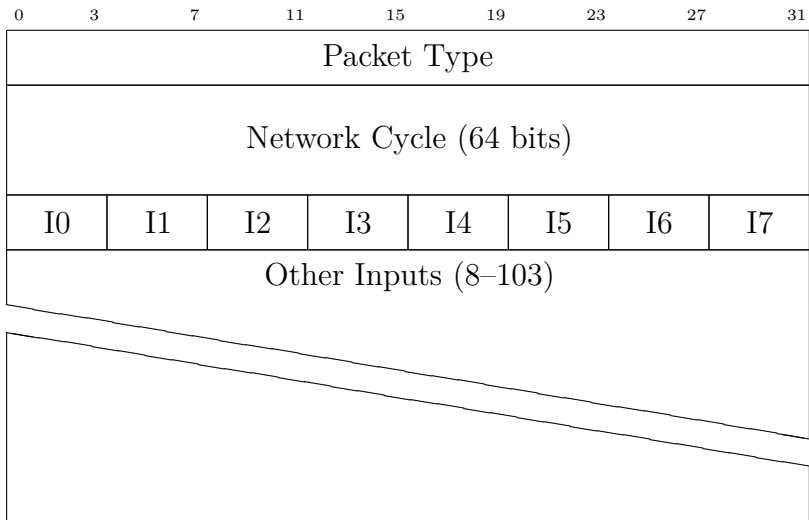


Figure 3.14: Step and Fire Packet Description.

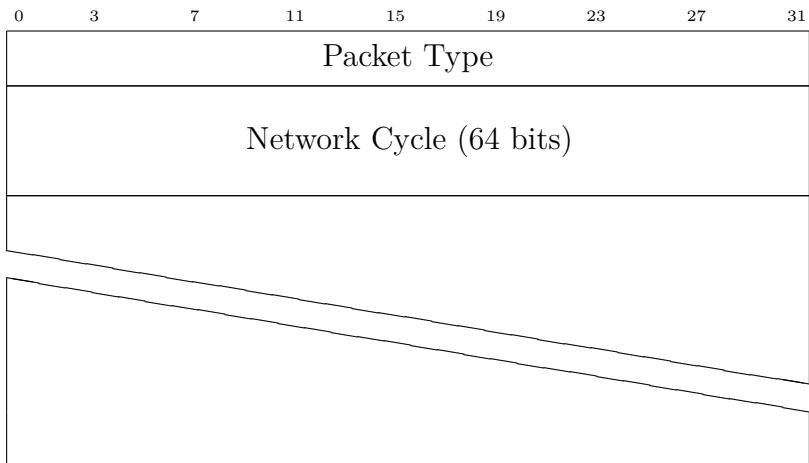


Figure 3.15: Reset Packet Description.

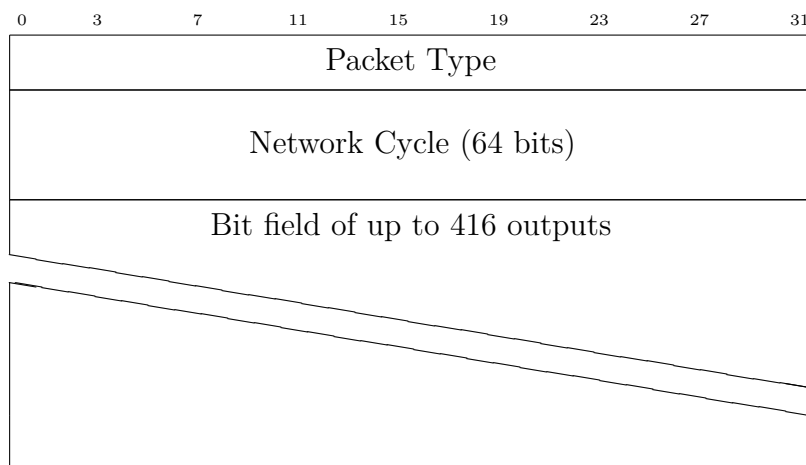


Figure 3.16: Output Fire Packet Description.

the layout of this packet. Each output fire is represented as a single bit in a bit field which can hold fires from 416 output neurons. The output packet also contains the network cycle in which the output appeared. Output packets are sent on timestamps when an input packet is received or when there is a fire from the array. If neither of these cases apply, then no output packet is sent. Sending an output packet the same timestep that an input packet is received allows the host to know that the last timestep has been reached and the hardware is waiting for more input to continue.

3.3.5 Simulator

DANNA2 also has an event-based simulator which can be run on a traditional CPU. The simulator can be used as a library of classes and functions to load and simulate networks, or it can be used as a standalone program with a network and commands provided to it via text files. The network file specifies the parameters and connections of a neural network. The command files provide input and operational commands to the simulator. The input spikes are scheduled for a specific network cycle and the simulator is told how many cycles to run. The output of the simulator is the output spike events as well as the updated network state. [Figure 3.17](#) shows a flow chart with the steps of running the simulator. Network input is applied to the input queue. Once the run command is given, fires will continue to be

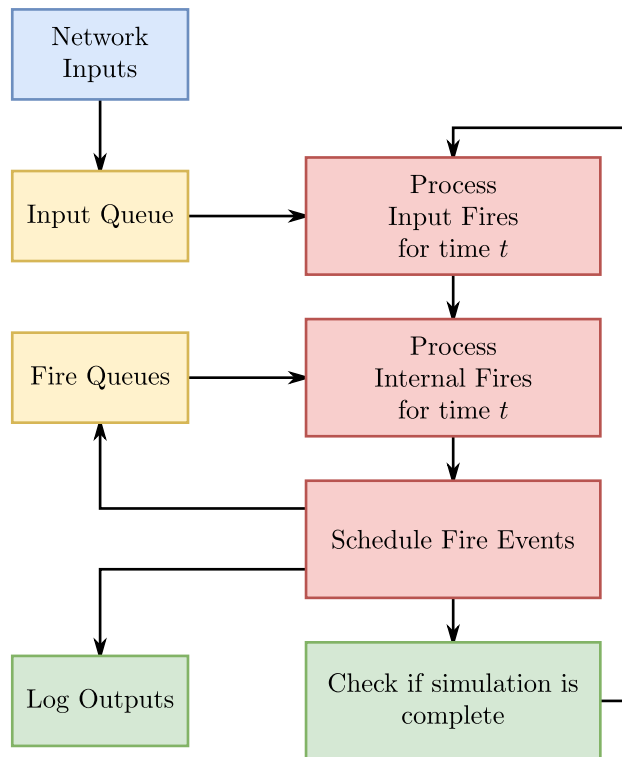


Figure 3.17: DANNA2 simulation flow chart.

processed, new fires will be scheduled, and a check will be made to see if the simulation is finished. The outputs of the simulator will be logged as output. Finally, the simulator logs its output.

The simulator is designed to run with very high performance, which allows EONS to quickly simulate many different networks for training. The network graph is stored in a hash table with an element’s coordinates as the key and the element as the value. Each fire event is stored in a circular buffer with the maximum delay values equal to the number of queues. An event is scheduled by inserting the fire into the queue whose index corresponds to the correct delay value away from the current network time. Fire events that occur within the same network cycle are unordered and do not have any dependencies with other cycles.

3.3.6 FPGA Implementation

The DANNA2 VHDL code must be synthesized and implemented in order to deploy the design to an FPGA. Previous work has been done to deploy DANNA2 to a Xilinx KCU1500 development board with a Xilinx Kintex Ultrascale FPGA using Xilinx Vivado 2017.2 to perform the synthesis and implementation. The element has a hierarchical design, so the top level of the schematic has a clear view of the subcomponents of the array. The subcomponents are further broken down into smaller subcomponents. [Figure 3.18](#) shows the top-level interface for the DANNA2 array. [Figure 3.19](#) shows a high-level RTL view of a single DANNA element. The array uses AXI4-Stream to send packets to and from the array. The array also has a global clock to drive the array, a halt signal to pause execution, and a reset signal to reset all the logic in the component. This reset signal is different from a reset packet and is used to reset the communication logic and ensure that the component is in a known state on power-up.

3.4 Communication with DANNA

This work builds upon previous methods of communication used with DANNA.

The first method of connecting with a DANNA array was to implement the DANNA array onto a single FPGA card connected to the host machine over PCIe [\[26, 106\]](#). This method of

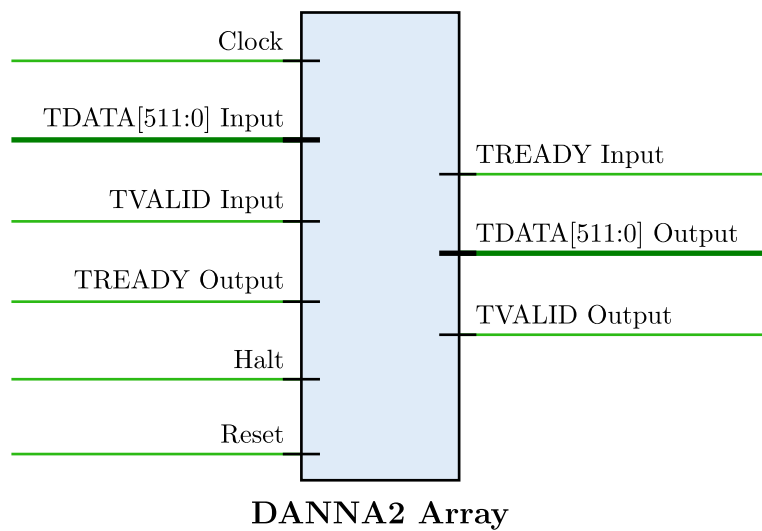


Figure 3.18: Top-level interface for the DANNA2 array.

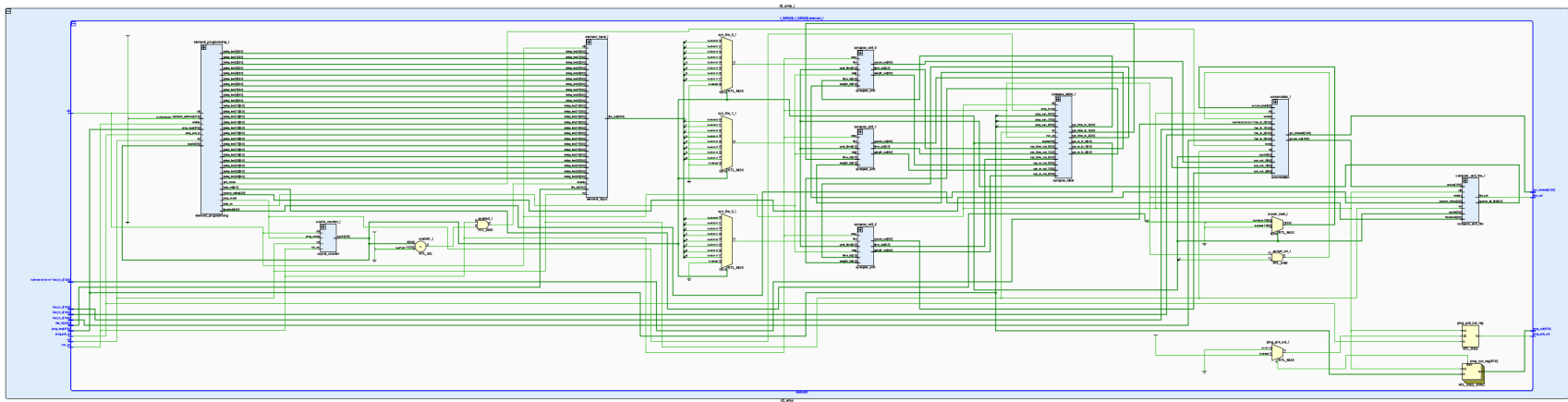


Figure 3.19: Schematic of DANNA2 element generated by Xilinx Vivado [62].

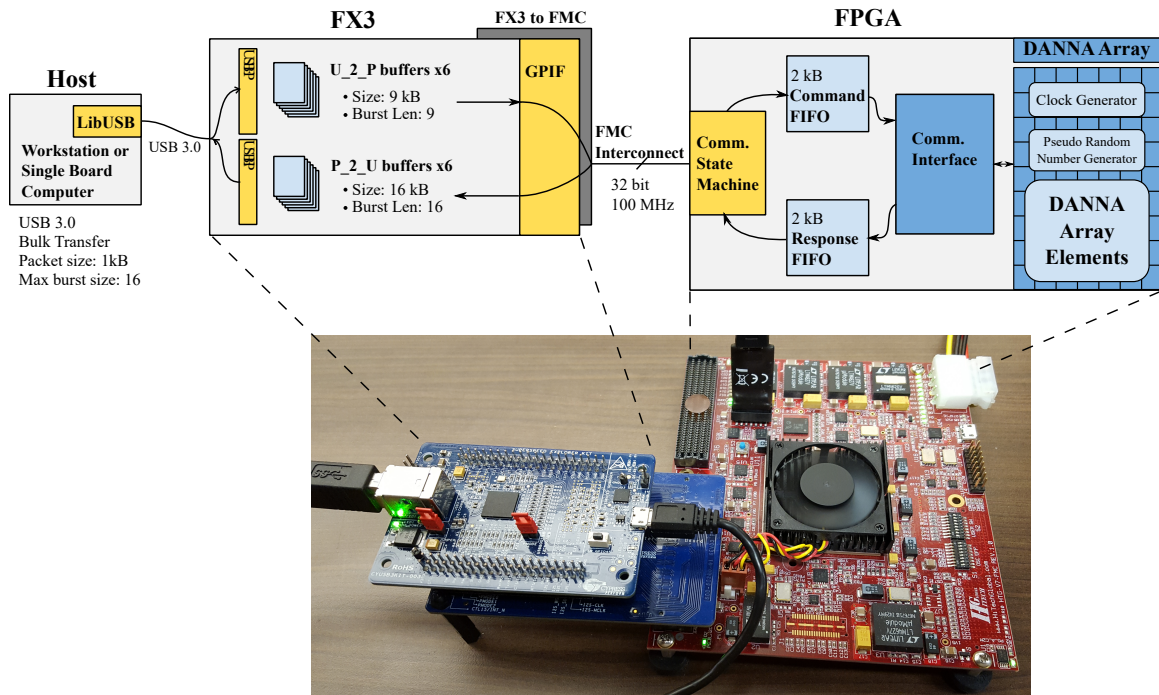


Figure 3.20: Diagram of the FX3 communication setup.

connecting to a DANNA array worked well but required a desktop machine with a free PCIe slot. Furthermore, only one FPGA was connected, which limited the maximum size of the array. The PCIe logic also used up a significant portion of the available space on the FPGA.

The next communication method aimed to reduce the cost of a complete DANNA kit. The kit was completed with a single board computer to act as the host and an FPGA to implement DANNA [27]. In order to make the DANNA array easy to connect to, a Cypress USB 3.0 peripheral controller was used to allow the host machine to connect to the DANNA array via USB 3.0 [106, 18]. This communication design is shown in Figure 3.20. Although this method was able to allow communication between the host and the array, it had performance issues. The DANNA hardware was able to run much faster than the communication could keep up.

In order to resolve the performance and inflexibility issues in connecting to many different hosts and hardware devices, my master's thesis work looked at improving the method of communicating between a host and a DANNA hardware array by introducing a custom communication controller to replace the FX3 and allow flexible, high-performance communication between the systems [122, 121]. Figure 3.21 shows a diagram of this improved

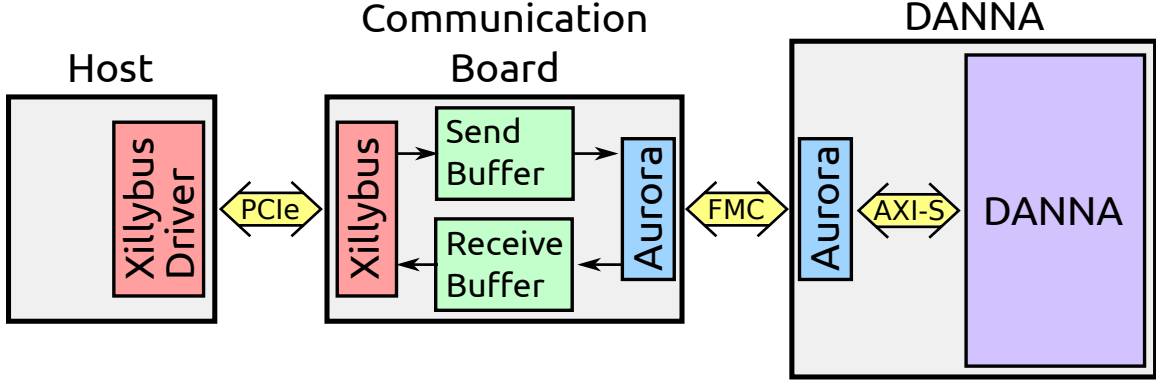


Figure 3.21: Diagram of the communications board.

communication system. As shown in the figure, the host connects to the communications controller using PCIe in a way similar to the original DANNA communication. The communication controller then connects to the DANNA hardware array using the Aurora high-speed 8B/10B protocol over an FMC connection. Details on both of these interfaces are provided in the following sections, since both are used in SNACC.

3.4.1 PCIe with Xillybus

PCIe is a complex protocol with many optional features that must be implemented correctly to meet the specification. The protocol must be implemented in both the hardware and in the drivers for the host operating system. Because of these reasons, Xillybus is used to transfer the packets from the host to the FPGA using the Xillybus driver and hardware design. “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 [117].”

Figure 3.22 shows a simplified block diagram of how Xillybus is able to transfer information between the communication board and the host. The host box shows the Xillybus components which are in the host system. This includes a userspace application which opens, reads, writes, and closes the Xillybus device. By operating on the device files, the system calls use the Xillybus driver to communicate information to and from the FPGA. The FPGA in this case is the communication board. This board has Xilinx’s PCIe interface IP core connected to the Xillybus IP core. Xilinx’s IP core handles the low-level PCIe operations and is configured to

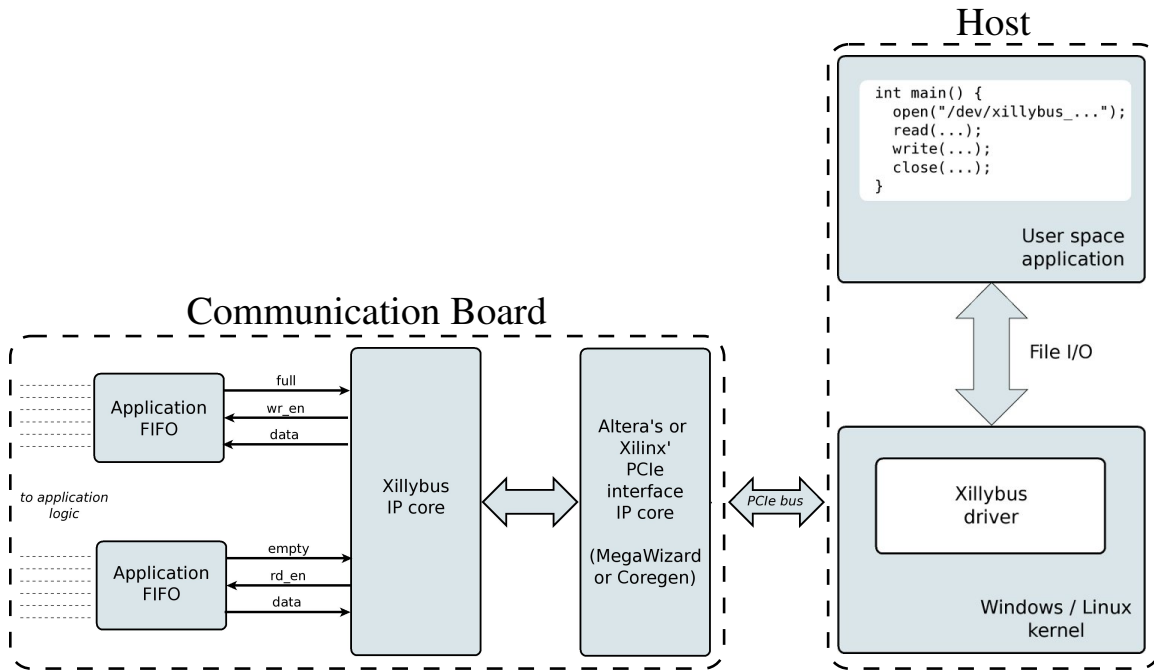


Figure 3.22: Simplified FPGA block diagram of Xillybus using PCIe transport with host interface block [117].

work properly with the Xillybus core. The Xillybus IP core interfaces between the low-level PCIe interface and a simpler First-In, First-Out (FIFO) memory interface. These FIFOs are connected to the application logic and are used for sending and receiving information.

In order to get the best performance for many different workloads, Xillybus has a Custom Intellectual Property (IP) Core factory where an IP core is configured based on the parameters you provide it. This tool was used to get the best possible performance from the PCIe connection. The details of the performance of the PCIe connection and the configuration can be found in [122].

Table 3.2 shows an updated table of the differences between the Xillybus revisions. Revision XXL is recently added and further increases the potential communication bandwidth available via a PCIe connection.

²On Xilinx series-7 FPGAs. For other FPGAs, the connection with the PCIe block is 32 or 64 bits wide.

Table 3.2: Xillybus revision summary [118]

Revision	Related demo bundle	Bandwidth multiplier	Maximal bandwidth	Internal data width	Width of PCIe block bus	Allowed user interface data widths
Revision A	Baseline	$\times 1$ (baseline)	800 MB/s	32	64 ²	8, 16, 32
Revision B	Baseline	$\times 2$	1700 MB/s	64	64	8, 16, 32, 64, 128, 256
Revision XL	XL bundle	$\times 4$	3500 MB/s	128	128	8, 16, 32, 64, 128, 256
Revision XXL	XXL bundle	$\times 8$	6600 MB/s	256	256	8, 16, 32, 64, 128, 256

3.4.2 Aurora

“The Xilinx LogiCORE IP Aurora 64B/66B core is a scalable, lightweight, high data rate, link-layer protocol for high-speed serial communication [110].” Xilinx provides the Aurora LogiCORE IP to establish high-speed serial links using on-chip Xilinx GTX, GTH, GTP transceivers to transmit the data. The Aurora LogiCORE IP is provided to developers by Xilinx and is used to establish multi-gigabit serial links. The IP core supports the use of both 8B/10B and 64B/66B line encoding schemes [111, 110]. 64B/66B offers theoretically improved performance over 8B/10B since it has a very low transmission overhead (3%), compared to the 25% transmission overhead for 8B/10B. Up to 16 transceivers can be used together in parallel, each running at a link speed of up to 6.6 Gb/s. Each transceiver uses separate transmit and receive lines, allowing the data channel to operate in full-duplex or simple mode. The Aurora IP core follows the Aurora 8B/10B Specification v2.2 (SP002) [112] or the Aurora 64B/66B Specification v1.3 (SP011) [109] depending on the encoding scheme. The IP Core has many built-in features, including framing, flow control, and Cyclic Redundancy Checks (CRC). The core will also set up and maintain the communication channel by sending maintenance packets which do not contain any data.

Figure 3.23 shows a diagram of an Aurora channel with all the components labeled with the proper terminology. The user application connects to the Aurora core using a user interface to transmit the user’s data to the Aurora core. DANNA uses the AXI4-Stream protocol for this user interface. See Section 3.4.3 for more information on this protocol. Two Aurora cores are connected together, making them Aurora channel partners. Data is transmitted between these partners over n Aurora lanes. This data is encoded for transmission following

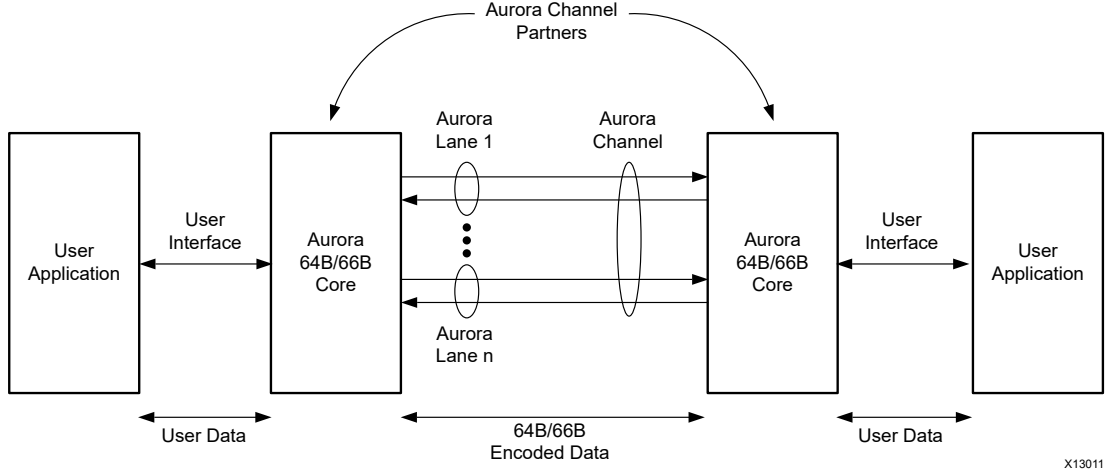


Figure 3.23: Aurora 64B/66B channel overview [110].

the 8B/10B or 64B/66B encoding scheme. Each lane corresponds to each transceiver lane. All of the lanes together form a single Aurora channel.

3.4.3 AXI4-Stream

AXI4-Stream protocol is used by both DANNA and DANNA2 to interface with the neuromorphic system. AXI4-Stream is one of the protocols defined by the Advanced eXtensible Interface 4 (AXI4) protocols, which are part of Advanced RISC Machine (ARM) Advanced Microcontroller Bus Architecture 4 (AMBA4). AXI4-Stream is supported by many Xilinx IP blocks, making it a well-supported protocol with existing components to perform many of the desired operations, like converting bus sizes and interfacing with FIFOs. Aurora also supports using AXI4-Stream for interfacing with the Aurora core. AXI4-Stream also supports framing. Framing is necessary for Aurora to perform CRC checks. Framing is also useful for supporting variable length packets and for using various bus widths to transmit the frame. The next section provides a brief introduction to the AXI4-Stream protocol.

AXI4-Stream uses the signals shown in Table 3.3, which is reproduced from the specification document [5]. Not all of these signals are required. The only required signals are ACLK, TVALID, TREADY, and TDATA. TKEEP and TLAST are both used to support framing. Figure 3.24 shows how the AXI4-Stream master and slave are connected together using these

Table 3.3: AXI4-Stream interface signals [5]. Note: There are more signals than the ones reproduced here.

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.

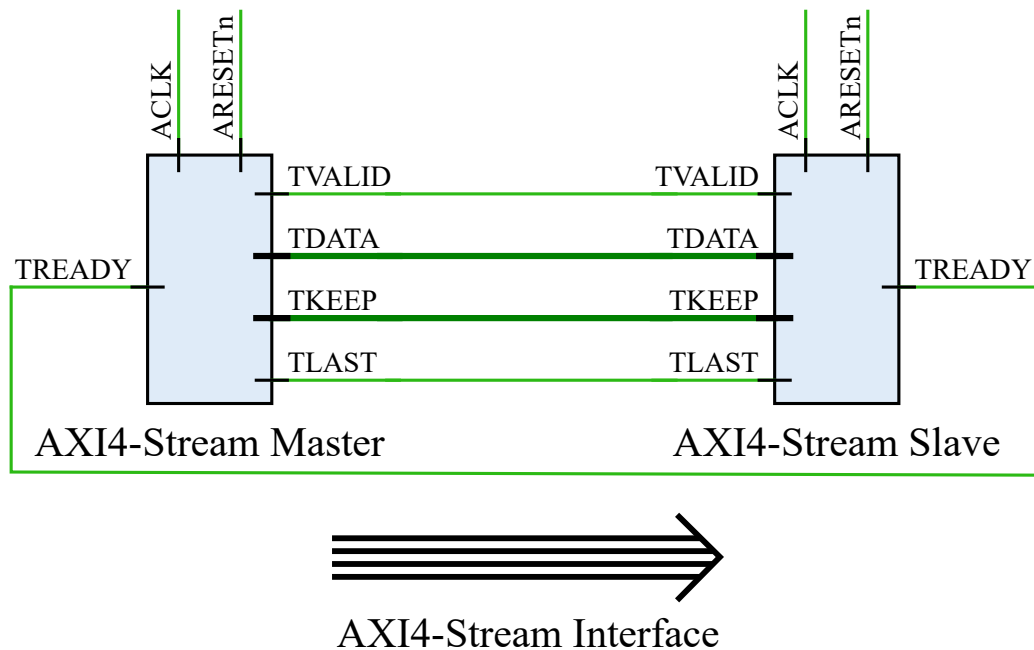
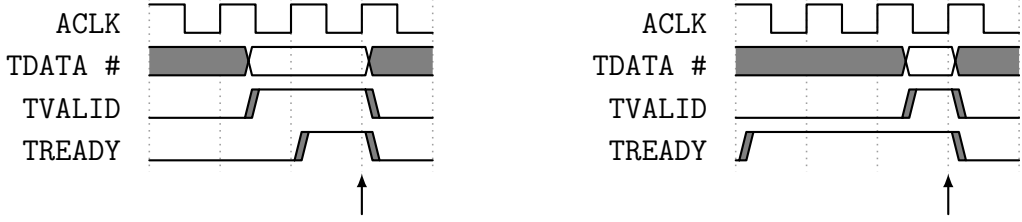


Figure 3.24: AXI4-Stream interface wiring diagram.

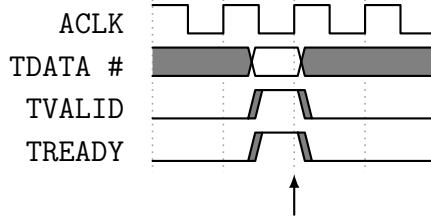
signals. ACLK is the global clock used to sample the signals. ARESETn is the global reset signal. TVALID is used to specify that the data on the bus is valid. TREADY is asserted when the slave is ready for data. TLAST is asserted to indicate the last word or a frame. TKEEP specifies the valid bytes of a word that make up the last frame. Data is passed from the master to the slave using a simple handshake with the valid and ready lines. For bidirectional communication, two AXI4-Stream interfaces are needed since each interface only supports data in one direction. TDATA is transferred from the master to the slave when both TVALID and TREADY are asserted. There are a few other important details about the handshake protocol. First, the master is not allowed to wait for TREADY to be high before asserting TVALID. Also, if data is put on the bus and TVALID is high, TVALID must remain high until the handshake is complete (i.e. TREADY is high). Second, the slave is permitted to wait for TVALID to be asserted before asserting TREADY. Third, if the slave asserts TREADY, it may de-assert TREADY before TVALID is asserted. These three requirements guarantee that the handshake will work correctly. [Figures 3.25a, 3.25b, and 3.25c](#) are reproduced from [5] and show examples of the handshake sequence. In each figure the arrow indicates where the transfer of data occurs. A valid handshake occurs regardless of whether either the TVALID or TREADY signal is asserted first or if they are asserted at the same time. This chapter provides only a brief introduction to AXI4-Stream, which focuses on the components of the protocol used by SNACC. The full AXI4-Stream interface specification can be found in [5].

3.5 Tiled DANNA

A previous TENNLab project also looked at scaling a spiking neuromorphic processor across multiple chips. Eckhart’s master’s thesis titled “Tiled DANNA: Dynamic Adaptive Neural Network Array Scaled Across Multiple Chips” explored using multiple FPGAs to implement a larger DANNA array [33]. My work builds upon what was learned from this project to build SNACC. Tiled DANNA was an important step as we learned how to build larger spiking neuromorphic systems which make use of multiple chips. [Figure 3.26](#) shows the basic design idea of Tiled DANNA. The larger DANNA array is broken up into sub-tiles for



(a) TVALID before TREADY handshake. (b) TREADY before TVALID handshake.



(c) TVALID with TREADY handshake.

Figure 3.25: Examples of valid AXI4-Stream handshakes. The arrow indicates where the data transfer takes place.

implementation. Tiled DANNA behaves the same as one large DANNA array would behave. The Tiling method allows the implementation to be split into smaller instances which work together to function as a large array. The Tiled DANNA implementation chose to have a single FPGA implement a DANNA tile and then to connected the FPGAs together to build the Tiled DANNA system.

Part of the major design challenges with building Tiled DANNA was the convoluted sub-clocking structure used by DANNA. With DANNA, the inputs to a synapse are sampled one sub-cycle at a time. The neurons are also able to fire during any of the sub-cycles, with STDP occurring on the first synapse to cause the neuron to fire. In order to prevent a bias caused by always sampling the inputs in the same order, a pseudorandom number generator is used to shuffle the order in which the inputs are sampled each global network cycle. This makes the sub-cycle timing between fires critical for deterministic operation which matches the simulator and single board designs. Because of this, Tiled DANNA was designed to use synchronous clocks between all the tiles. A master tile was chosen to be the tile to generate the synchronous clocks and provide the clocks to the other tiles. Tiled DANNA requires 6

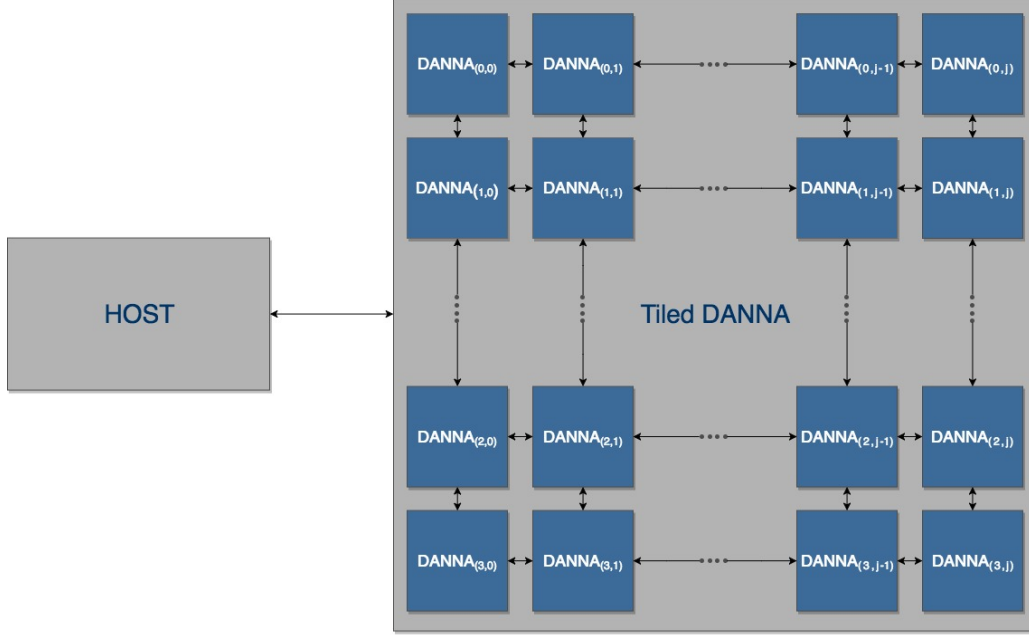


Figure 3.26: Tiled DANNA Structure [33].

clocks to operate. These clocks and their functions are shown in Table 3.4. Many challenges originated from having 6 synchronous clocks which must be passed from the master tile to the other tiles. The global synchronous clock also greatly limits the maximum size of tiled DANNA since the clock skew between the components can grow too great.

Tiled DANNA was initially prototyped using a Prodesign ProFPGA prototyping system, which is a multi-FPGA motherboard with the capability of interconnecting four Xilinx 690T or 2000T FPGAs. Figure 3.27 shows a picture of the ProFPGA system currently set up to support tiled DANNA with two FPGAs.

Tiled DANNA communicates with the host through the use of multiple FX3 communication boards, one FX3 per DANNA tile, as shown in Figure 3.28. The host communication for each tile largely uses the same design as single-board host communication with an FPGA, as was previously shown in Figure 3.20. The host-to-array communication is point-to-point, with the host deciding which communication controller to send each packet to. Each tile has its own communication interface to the host, and the host handled the routing of the messages to the correct tile.

Table 3.4: Tiled DANNA Clocks

Clock Function	Clock Frequency
Accumulator Clock	32 MHz
Acquire Fire Clock	16 MHz
Accumulator Enable Clock	16 MHz 90° phase shift
Global Net Clock	1 MHz
System Communication Clock	100 MHz
Communication Bus Clock	100 MHz 180° phase shift

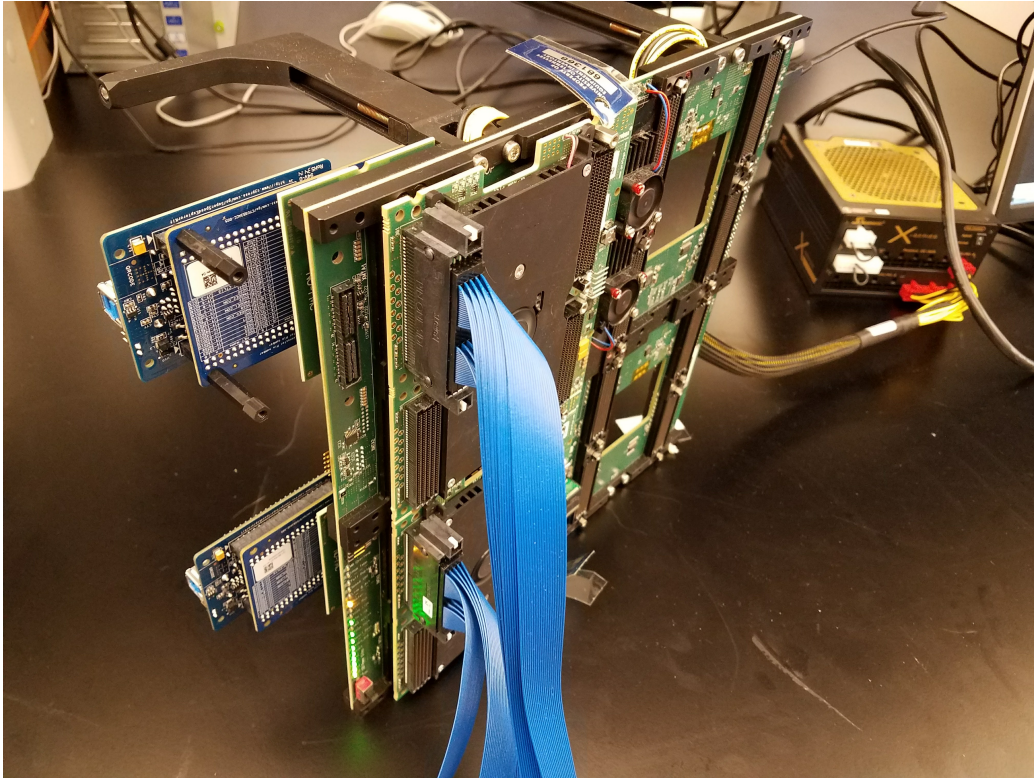


Figure 3.27: Picture of the Tiled DANNA prototype [33].

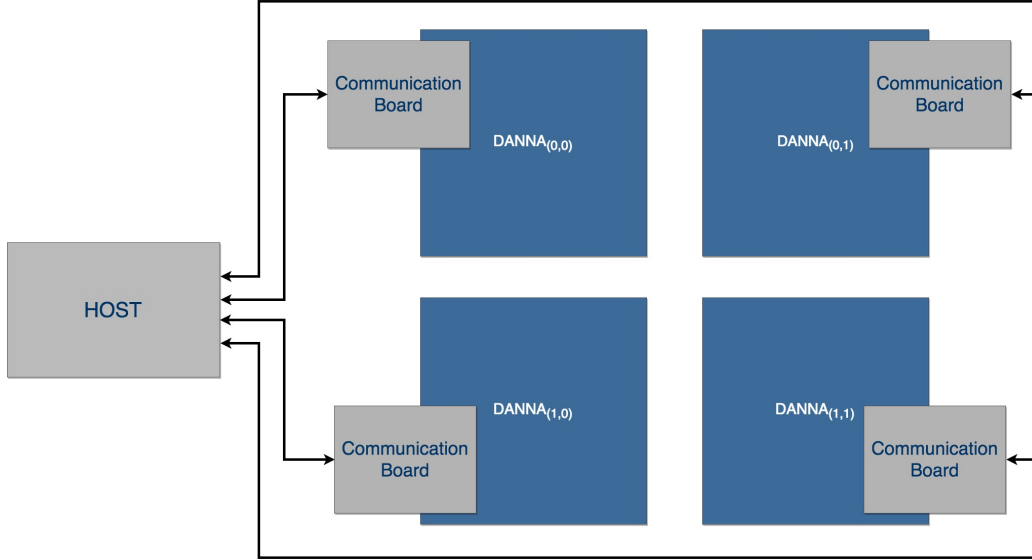


Figure 3.28: Host to Tiled DANNA Communication Interface [33].

The tiles are connected together via a parallel interface bus. This interface transmits the synchronous clocks, as well as connecting the bordering elements together. The connected elements and tiles remain in lockstep across all the tiles, allowing the multiple tiles to operate in unison as one massive DANNA array. The tile-to-tile interface uses a synchronous communication bus. The bus uses 72 unidirectional data lines to implement a 36-bit bidirectional bus. The bus is used to transmit 36 bytes over the 36-bit interface in each direction at a 16 MHz clock rate. Figure 3.29 shows the tile-to-tile communication interface between the DANNA tiles. Elements at the tile boundaries are known as boundary elements. These elements have a one-to-one connection with the boundary elements in the neighboring tile. This limits the connectivity pattern of boundary elements and causes one extra cycle of delay as the fire events cross the tile divide. Figure 3.30 shows the limited nearest-neighbor connections possible with a DANNA element on the boundary. The size of the tile-to-tile interface limits the number of boundary elements each edge can contain to 32 elements. The communication of boundary elements must be completed within one network cycle so that the elements can stay within lockstep. Many additional challenges of designing this tile-to-tile interface came from the fact that the elements could be programmed as a synapse or a neuron, and all the possible connections have to work correctly over this interface. The interface also

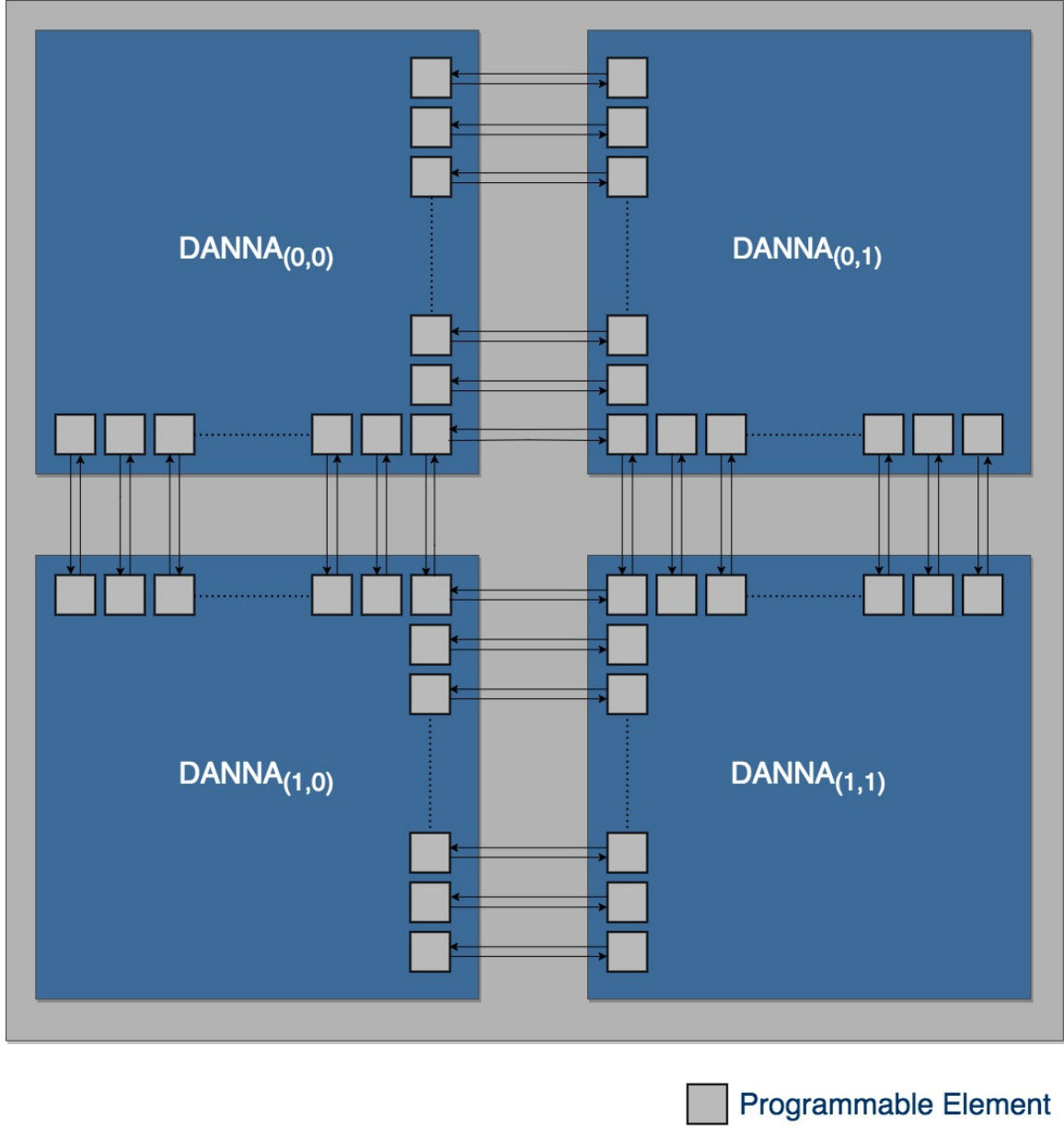


Figure 3.29: Tiled DANNA tile-to-tile communication interface [33].

has to support STDP weight changes. In order to send the element events over the tile-to-tile interface, multiplexers and demultiplexers were used as shown in [Figure 3.31](#).

The master tile is also used to control the execution of the entire system. It is used to synchronously start and stop the execution of the DANNA networks across all of the tiles. The master tile sends the element's clock-enable signal at the correct time to start the operation of the elements in the other tiles at the correct sub-cycle count. The initial testing of Tiled DANNA focused on a one-by-two tile structure and tested the arrays working separately and as a combined array. Further details can be found in [\[33\]](#).

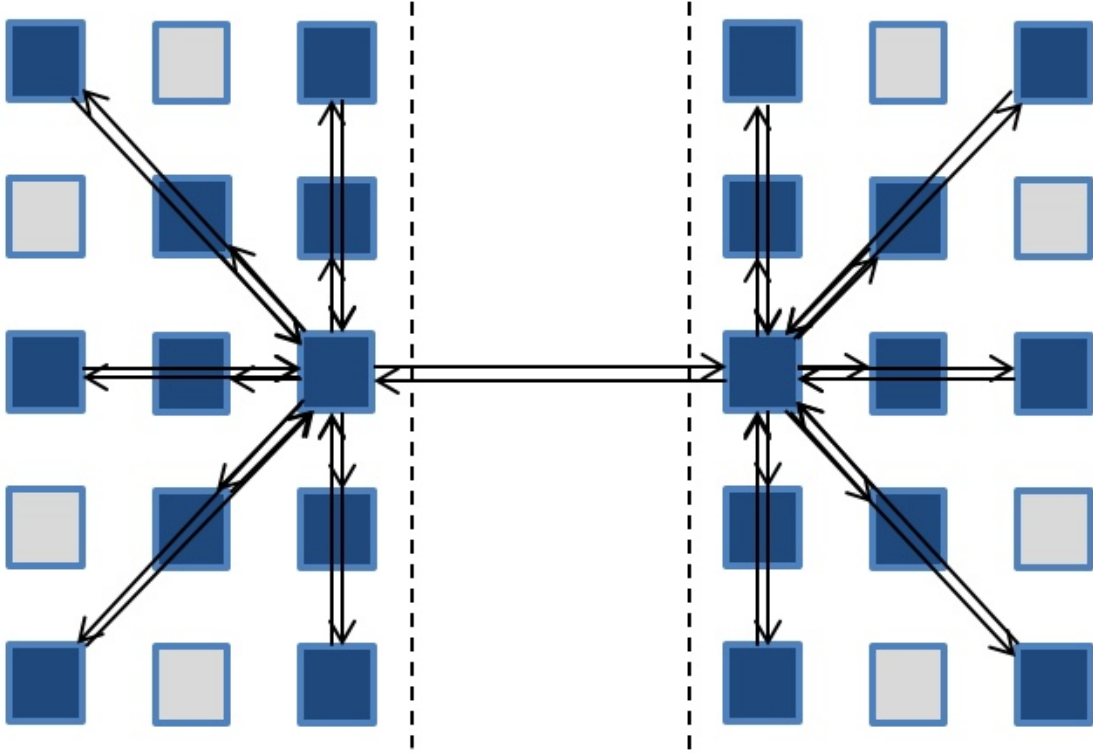


Figure 3.30: Tiled DANNA Boundary Edge Element Nearest-Neighbor Connections [33].

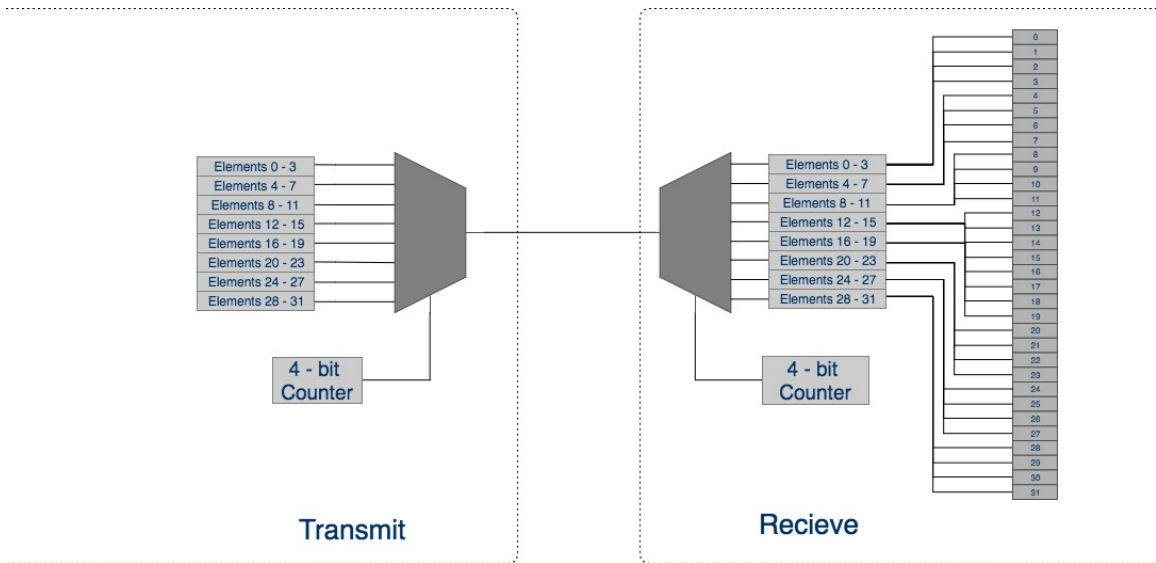


Figure 3.31: Fire event multiplexing and demultiplexing across the tile boundary [33].

Chapter 4

Goals

Building a vast digital simulation of the brain could transform neuroscience and medicine and reveal new ways of making more powerful computers.

– Henry Markram [\[54\]](#)

This project builds on and is a continuation of the previous work conducted by TENNLab. My prior Master’s work looked at building an improved communication setup to better connect a neuromorphic array to a host machine, so that the connection to the host would no longer be a performance-limiting bottleneck. When researching a better connection strategy, it became apparent that using an intermediate board would greatly open up the possible communication protocols that would be available for the transfer. An FPGA was then selected as an intermediary between the host machine and the neuromorphic processor. This allowed high-performance links to be selected for communication between the host processor and the communications board, and for communication between the communications board and the array processor. The impact of the intermediate board was minimal, and this setup allowed for a wide choice of protocols for connecting with the host and for connecting to the neuromorphic processor. In [\[122\]](#), the protocols that would result in the best performance for our systems were chosen. Since we now have the intermediate board and the ability to set up multiple neuromorphic processors, the next question is how to connect them together to build a larger system.

Additionally, since we already used an intermediate board for communication, how can such an intermediate board be used to facilitate the scaling-up of the neuromorphic design,

and what features does such a communication board provide that would not be possible otherwise? From this thought process, the goals for the dissertation emerged. The main goals are to set up a multi-neuromorphic processor system using TENNLab's DANNA2 architecture to demonstrate the scaling capabilities of our neuromorphic systems and to evaluate the use of a separate communications controller to facilitate the scaling of individual neuromorphic processors. The scaled neuromorphic system should place minimal restrictions on the usage of the system and should ideally behave the same as a single system, but with a much greater capacity. Many of the more detailed goals and decisions were driven by the designs of our previous systems and our group's design philosophies. For instance, we believe in the usefulness of creating a deterministic system which can be accurately modeled in software, so because of this, the system was given the extra goal of being cycle-accurate with the DANNA2 simulator. The next goals after building and testing the scaled-up neuromorphic system are to run and evaluate different neuromorphic workloads on the system. Different tasks have different network structures and communication patterns, so this work will also evaluate how different tasks run on the hardware system and discuss the performance characteristics of each.

Chapter 5

Tools

Divide each difficulty into as many parts as is feasible and necessary to resolve it.

– Rene Descartes

This chapter covers the programs, utilities, and hardware that were used to design and build SNACC.

5.1 Vivado

The Vivado Design Suite by Xilinx is used to develop the hardware designs for the FPGAs [113]. This design suite is used to both simulate the design with testbenches and to synthesize and implement the design into a bitfile, which can be loaded onto the FPGA. Multiple versions of Vivado were used to design for different FPGAs. Vivado 2017.2 was used to create the project for the KCU1500. This version included a newer version of the Aurora IP Block; however, this newer design would not work correctly for the older 690T boards. Therefore, Vivado 2014.4 was used to create the projects for the 690T and the VC707, which are used in the SNACC design.

5.2 Xilinx

The FPGAs used in this dissertation are all designed by Xilinx, but using a Xilinx FPGA is not a requirement and other FPGAs could be used. The majority of the VHDL design files are written so that they can port to any design. However, the Xillybus code and the Aurora IP blocks are specific to Xilinx FPGA's, so some changes to the design would be required to port it to a different FPGA manufacturer. For single FPGA testing, the Xilinx Kintex UltraScale FPGA KCU1500 Acceleration Development Kit (KCU1500) with a Xilinx Kintex UltraScale XCKU115 is used [114]. For the SNACC system the Xilinx Virtex-7 FPGA VC707 Evaluation Kit (VC707) with a Xilinx Virtex7 X485T is used for the communication board [116]. The HiTech Global HTG-777 Stackable Development Platform with a Xilinx Virtex7 X690T (690T) is used to implement the neuromorphic arrays in SNACC [44]. Figure 5.1 shows an image of each of these FPGAs.

5.3 Texas Instruments

The Texas Instruments LMK03328EVM Ultra-Low-Jitter Clock Generator EVM With 2 PLLs, 8 differential outputs, and 2 inputs is used to provide a clock signal to the Aurora channels [97]. Figure 5.2 shows a picture of the LMK03328EVM.

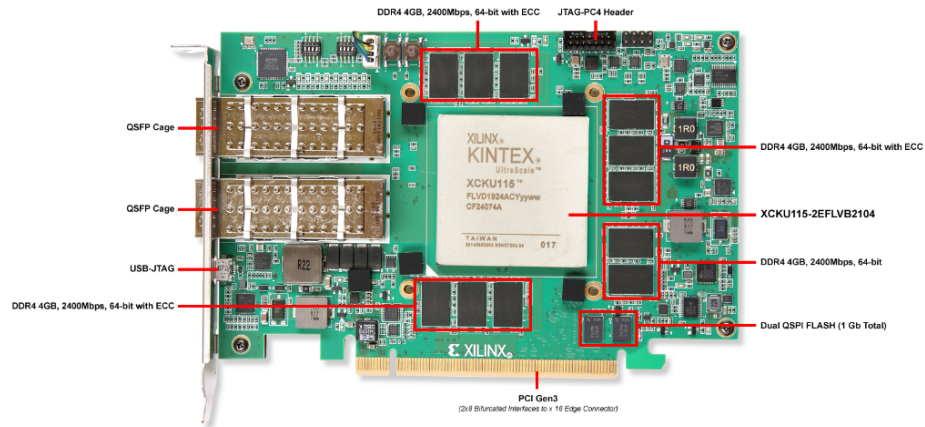
5.4 Xillybus

Xillybus was previously discussed in Section 3.4.1 and is used to implement the Direct Memory Access (DMA) between the FPGA and the host machine over PCIe with Linux.

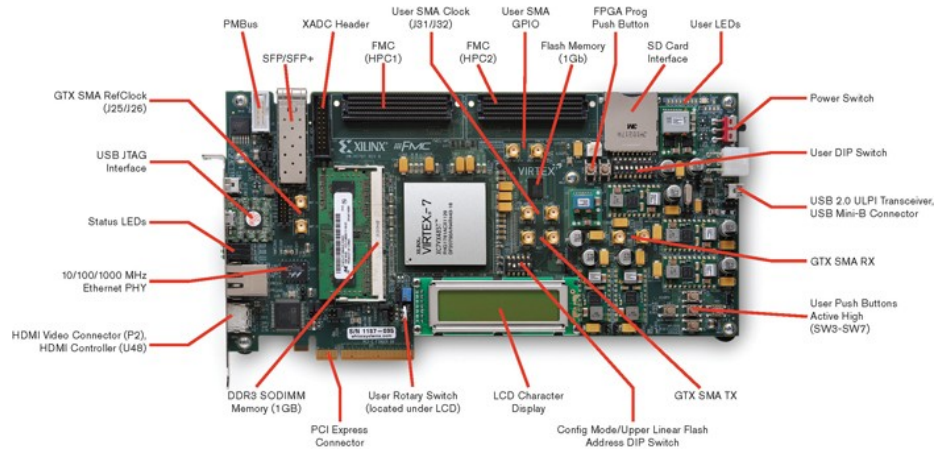
5.5 GHDL

GHDL¹ is an open-source analyzer, compiler, and simulator for VHDL [40]. GHDL is used with VUnit to automate component testing and high-level testing for the DANNA2 core, the acknowledgment logic, and the complete SNACC design. GHDL compiles the VHDL code

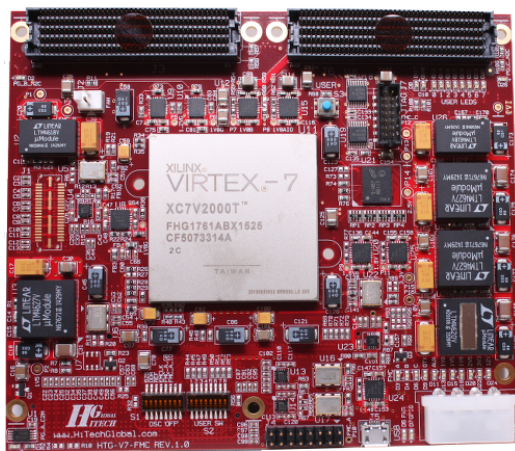
¹GHDL is a shorthand for G Hardware Design Language (currently, G has no meaning) [39].



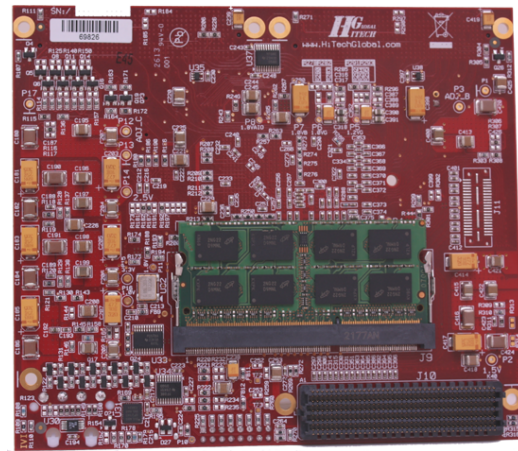
(a) KCU1500



(b) VC707



(c) 690T Front



(d) 690T Back

Figure 5.1: Images of the Xilinx FPGA Boards used.

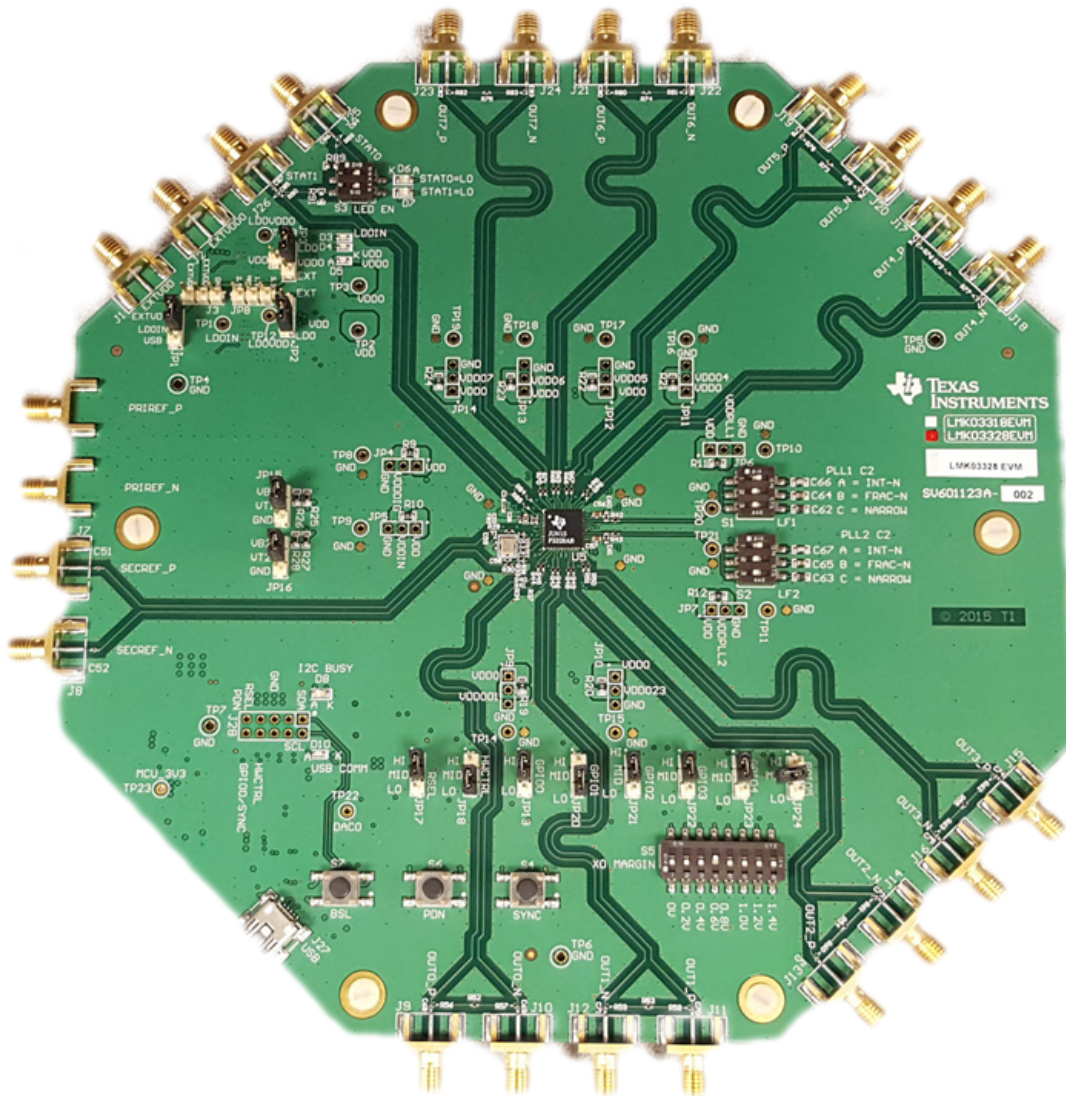


Figure 5.2: LMK03328EVM Ultra-Low-Jitter Clock Generator EVM with 2 PLLs, 8 differential outputs, and 2 inputs

into machine code, which allows the simulation to run quickly, making it possible to simulate large designs. GHDL is able to output waveforms that can be used to analyze the behavior of the system, in addition to being able to write assert statements to verify expected behavior.

5.6 VUnit

VUnit is an open-source unit testing framework for VHDL and System Verilog [6]. VUnit uses GHDL to compile and run VHDL code. VUnit is used to write the unit tests and high level tests for DANNA2 and SNACC.

5.7 DANNA2 Network Visualizer

In order to better visualize DANNA2 networks in a way that correlates with how the networks are organized in grid coordinates, a new DANNA2 visualization tool (DANNA2 Viz) was developed to support the work of this dissertation. The main purpose of the DANNA2 Viz is to visualize the connections and elements of a DANNA2 network file. Figure 5.3 shows a graphical representation of a DANNA2 network using the visualizer. A label file can also be included with the DANNA2 network file in order to label the input and output neurons of the network. Figure 5.4 shows a legend of what the different element and connection colors represent. Details of an element are displayed when the element is clicked on, as shown in Figure 5.5. The visualizer also supports pruning networks, auto-reloading networks when the file changes, saving network images, video recording of networks as they evolve with evolutionary optimization, and video recording of the activity of the fire events as a network is simulated with the DANNA2 simulator.

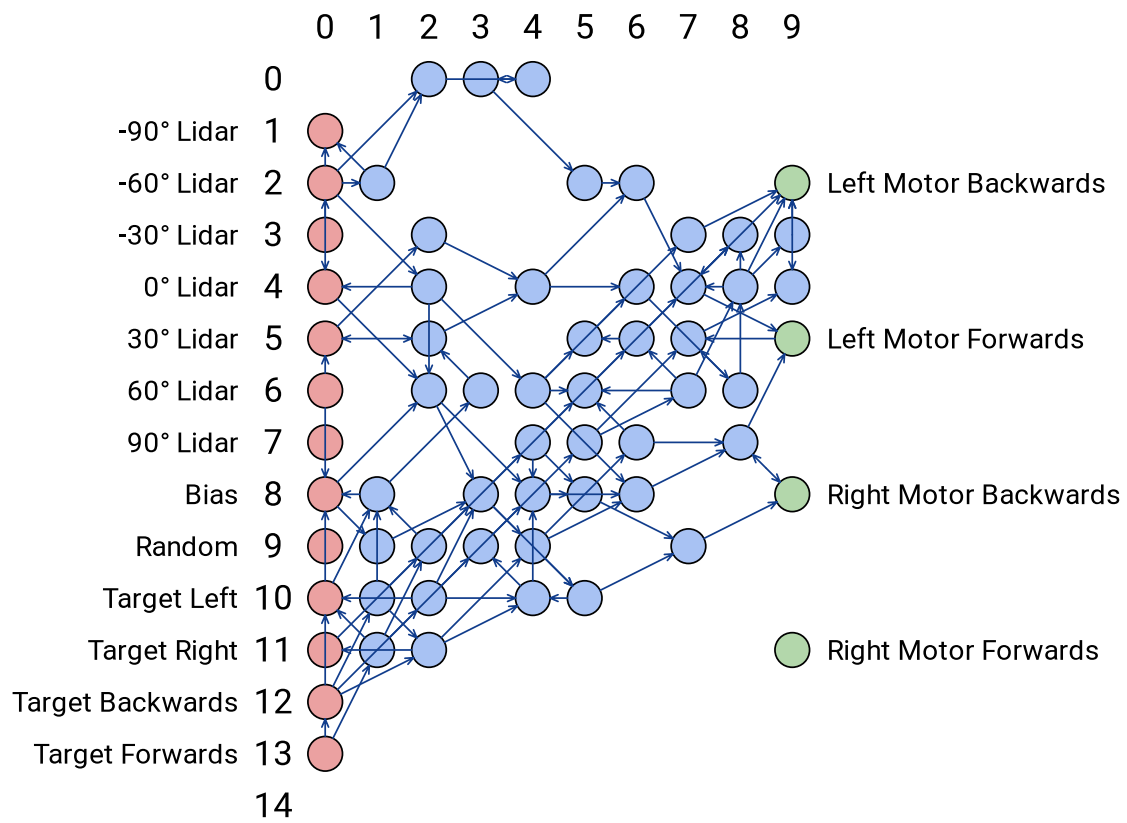


Figure 5.3: An example network with labels displayed by the DANNA2 Viz.

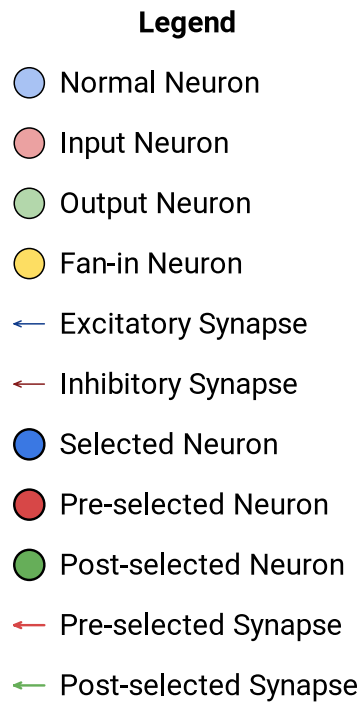


Figure 5.4: Legend of the DANNA2 Viz.

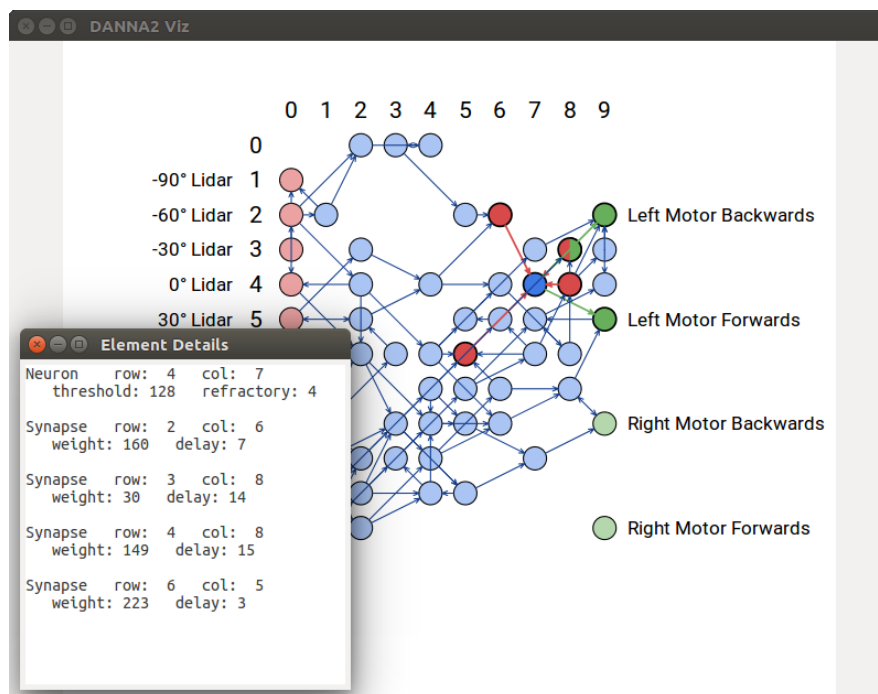


Figure 5.5: DANNA2 Viz with a selected element.

Chapter 6

DANNA2 on a Single FPGA

If the human brain were so simple that we could understand it, we would be so simple that we couldn't.

– Emerson M. Pugh¹[51]

As discussed in [Section 3.3.6](#), DANNA2 has been previously synthesized and implemented for an FPGA as part of Mitchell’s master’s thesis work [62]. The prior work included a working DANNA2 simulator and a working DANNA2 hardware array, but some work was still required to integrate the hardware design into the TENNLab software stack. The communication design needed to be updated from the reused DANNA design in order to get better performance with the new FPGA and new packet structure. Additionally, the hardware was only able to be tested by sending raw binary packets to and from the hardware device. There was no prior software support to enable easy communication with the hardware device. This chapter details work done to update the communication design by connecting the DANNA2 hardware to the host machine using a KCU1500 and to integrate the DANNA2 hardware processor into the TENNLab framework.

6.1 Existing Work

As previously discussed, the DANNA2 array design was finished with the top-level AXI4-Stream interface used to send and receive packets from the array. Also completed was the

¹Pugh mentioned that his father said this.

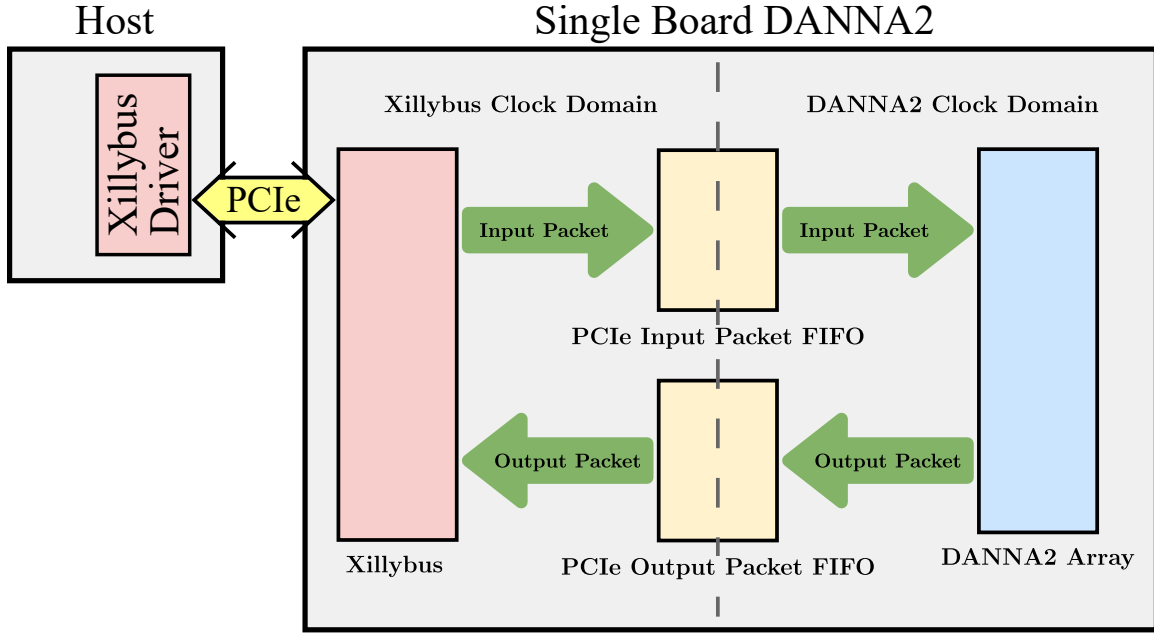


Figure 6.1: Top-level design for DANNA2 single board communication.

array logic, used to tile DANNA2 elements together, and the array control logic, used to parse and create DANNA2 communication packets. Because these designs were changed in the creation of SNACC, I will cover them in greater detail here. [Figure 6.1](#) shows a simplified diagram of the top-level design for implementing DANNA2 on a single FPGA. The Xillybus IP component receives packets from the host and stores the packets into the PCIe read FIFO. The Xillybus IP component also reads packets from the PCIe write FIFO and sends these packets back to the host. [Figure 6.2](#) shows a picture of the single board communication setup.

These FIFOs are used as buffers to store incoming and outgoing packets for the DANNA2 array. These buffers are also used to make full use of the available bandwidth. PCIe communication has higher latency but greater throughput. In other words, PCIe is deeply pipelined and can send many messages quickly, but a single message takes awhile to make it through the pipeline. DANNA2, on the other hand, only works with one message at a time but can generate and consume them at a high rate. To get the best performance from the system, the DANNA2 array needs to be continuously receiving the next packet to be processed. The array will pause its execution if it ever runs out of instructions. Additionally,



Figure 6.2: Picture of the single FPGA DANNA2 hardware.

the array also needs a place to put output packets. The array will also pause if the PCIe write FIFO fills up.

The FIFOs are also used to cross clock domains. The Xillybus IP component operates at the same frequency as the UltraScale FPGA Gen3 Integrated Block for PCI Express's AXI4-Stream interface which operates at 250 MHz. The DANNA2 array is provided a different clock signal to clock the DANNA2 elements. A DANNA2 network cycle occurs every 10 element cycles. The DANNA2 element clock frequency is chosen at 100 MHz, which means that a network cycle occurs every 10 MHz. The maximum frequency at which the DANNA2 elements can run depends on the target FPGA. The element frequency of 100 MHz was chosen since it allows the design to easily map to Xilinx Virtex7 FPGAs. Newer UltraSCALE and UltraSCALE+ FPGAs can operate at higher frequencies and support faster PCIe protocols.

Figure 6.3 shows a simplified schematic of the DANNA2 array component. The DANNA2 array contains an array control component and multiple DANNA2 elements. In this figure a 2×2 grid array is shown. The array controller handles the parsing of input packets and the creation of output packets. The array controller also supplies input fires to the input elements and reads output fires from the output elements. The host input spikes, coming from the array controller, are routed to fan-in port 0 on the elements. This port is used for host input as well as a fan-in port for non-input elements. The wiring of the fire signals

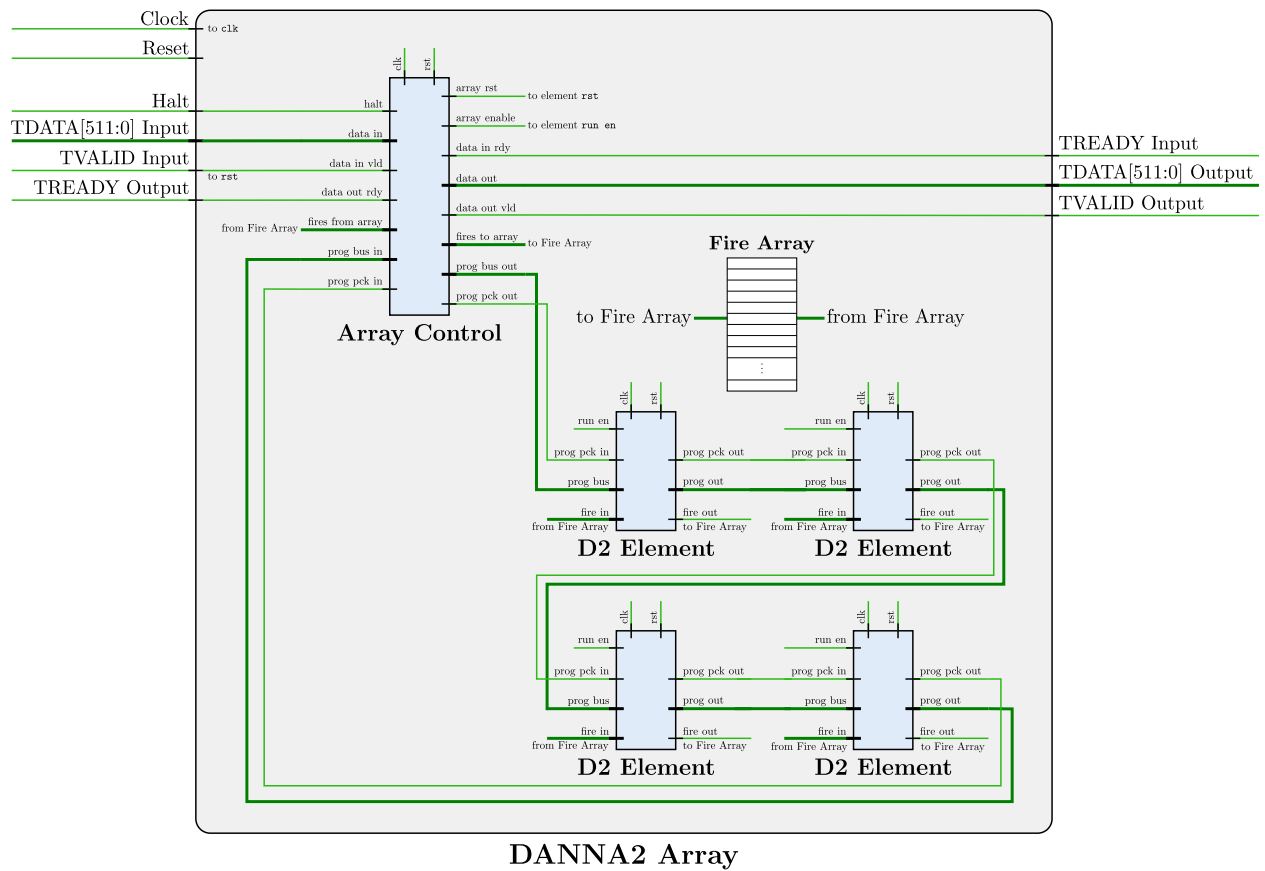


Figure 6.3: DANNA2 grid array design for a 2×2 grid.

is done through the use of a fire array. The fire array is an array of standard logic vectors. Each entry in the array corresponds to the element number and each vector stores the fire information for each of the 24 input synapses in a bitmap. Since each fire event only encodes the presence of an event, each event is represented by a single bit. Although not shown in the figure, the input fires from the host to the array have a magnitude, and charge forwarding circuitry is used to apply the input weights to the input elements. The array controller is also in charge of configuring the elements, and the programming bus previously mentioned in [Section 3.3.3](#) can be seen in the figure.

[Figure 6.4](#) shows a simplified block diagram of each of the elements. The entire design was built hierarchically and the element is broken down into various stages of operation. The execution of the element is also broken down into sub-cycles where the different components are active for different sub-cycle counts. One of the main reasons for using sub-cycles is so the three synapse units can be used to evaluate all 24 input synapses. See [Section 3.3](#) for more detail on the DANNA2 element operation.

6.2 New Design

By building upon existing work, the PCIe connection design was updated to operate at peak performance, and a software driver was designed that integrated the hardware system into the TENNLab framework. This driver allows the hardware system to be a target for evaluating networks and for training new networks using EONS.

6.2.1 Hardware Design

On the hardware side, the communication design was reworked to use Xillybus Revision XL with a 128-bit data bus. The previously used data bus was 64 bits. This switch to 128 bits almost doubled the throughput of the communication between the host and the FPGA. Since the XL design uses a 128-bit bus internally, the use of a 128-bit data bus results in the best performance. [Table 6.1](#) shows the IP Core factory settings used to create the Xillybus Revision XL IP core for the KCU1500. The core is configured to have two streams to transmit the data upstream and downstream.

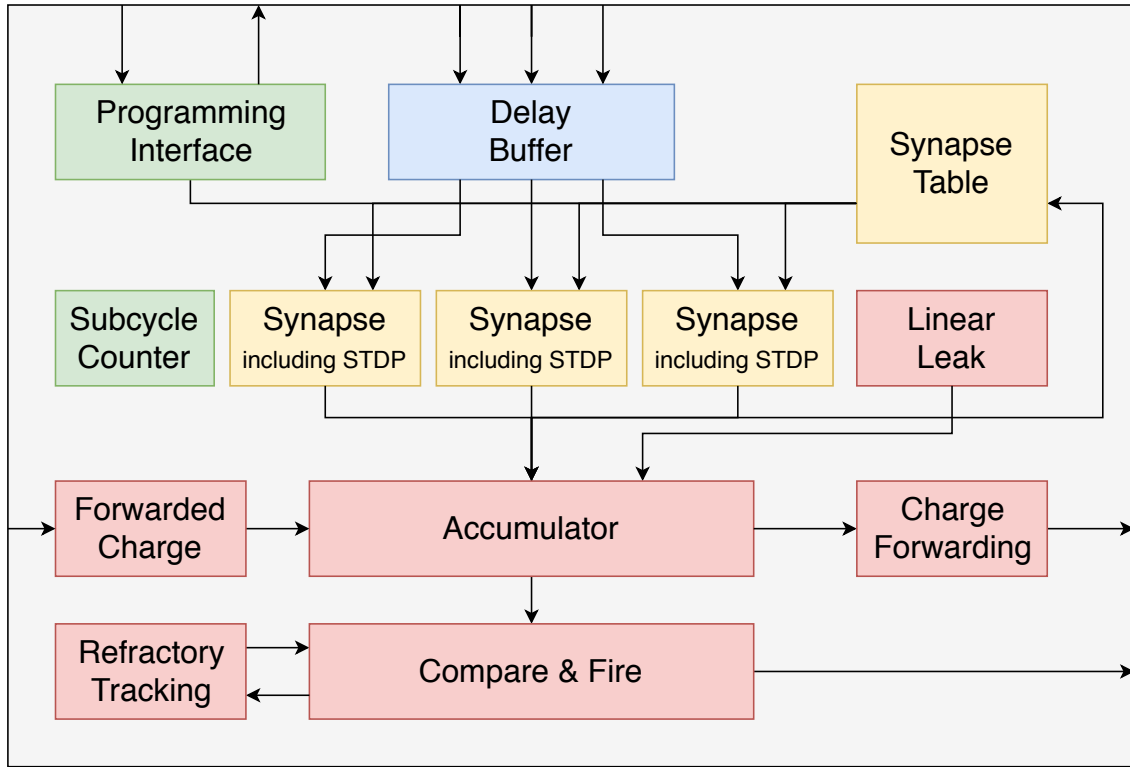


Figure 6.4: Simplified schematic of a DANNA2 Element.

Table 6.1: IP Core Factory Device Files for Xillybus core Revision XL for Xilinx Kintex Ultrascale

Name	Direction	Data Width	Expected BW	Autoset	Details
xillybus_read	Upstream (FPGA to Host)	128 bits	3500 MB/s	Yes	Data exchange with coprocessor
xillybus_write	Downstream (Host to FPGA)	128 bits	3500 MB/s	Yes	Data exchange with coprocessor

6.2.2 Build System Design

In order to make compiling the hardware design easier, multiple scripts were written around Xilinx Vivado in order to easily store the project in Git, and to build the design using both a project-based workflow and a Tool Command Language (Tcl) script-based workflow. The script method of building the project proved very beneficial; it allows customization of the build parameters and also enables Continuous Integration (CI) to be used to automatically generate bitfiles when the source code has changed.

[Figure 6.5](#) shows the directory structure used for the build system. The build script `danna2_build` and the testing script `run_vunit` are in the `DANNA2_HW` folder. They are used to build the design for FPGA hardware and to test the design using VUnit, respectively. The `projects` and `runs` folders do not exist in the repository but are created to hold the built projects after running `danna2_build`. `danna2_build` can be used to create a Vivado project which is stored in `projects` or to create a bitfile with Vivado using the script-based workflow. The output files for running the script-based workflow are stored in `runs`.

Different build parameters can be passed to the script to generate different DANNA2 networks. Grid networks can be generated by passing a width and height to the script. Leak, fan-in, and STDP can optionally be turned on or off. Grid network parameters can either be set by specifying each flag or by providing a ‘specification string’. The ‘specification string’ specifies all the grid parameters in a single string. Sparse networks can be created by providing a DANNA2 network file. The network will then be built into a custom hardware array which contains only the elements specified in the network file and does not contain any of the reconfiguration logic. Using sparse arrays allows larger networks to be built with fewer resources since only the elements used are included in the design. When the bitfiles are generated, different configurations are given different names, so that multiple bitfiles can be generated simultaneously without interfering with each other. The script can also be used to build a Vivado project so that the Vivado GUI can be used to design and build the bitfile. This approach also allows the creation of debug probes to be used to diagnose the behavior of the design by probing the values of signals and busses on the active FPGA hardware. [Figure 6.6](#) shows the command line arguments for the DANNA2 build script.

DANNA2_HW

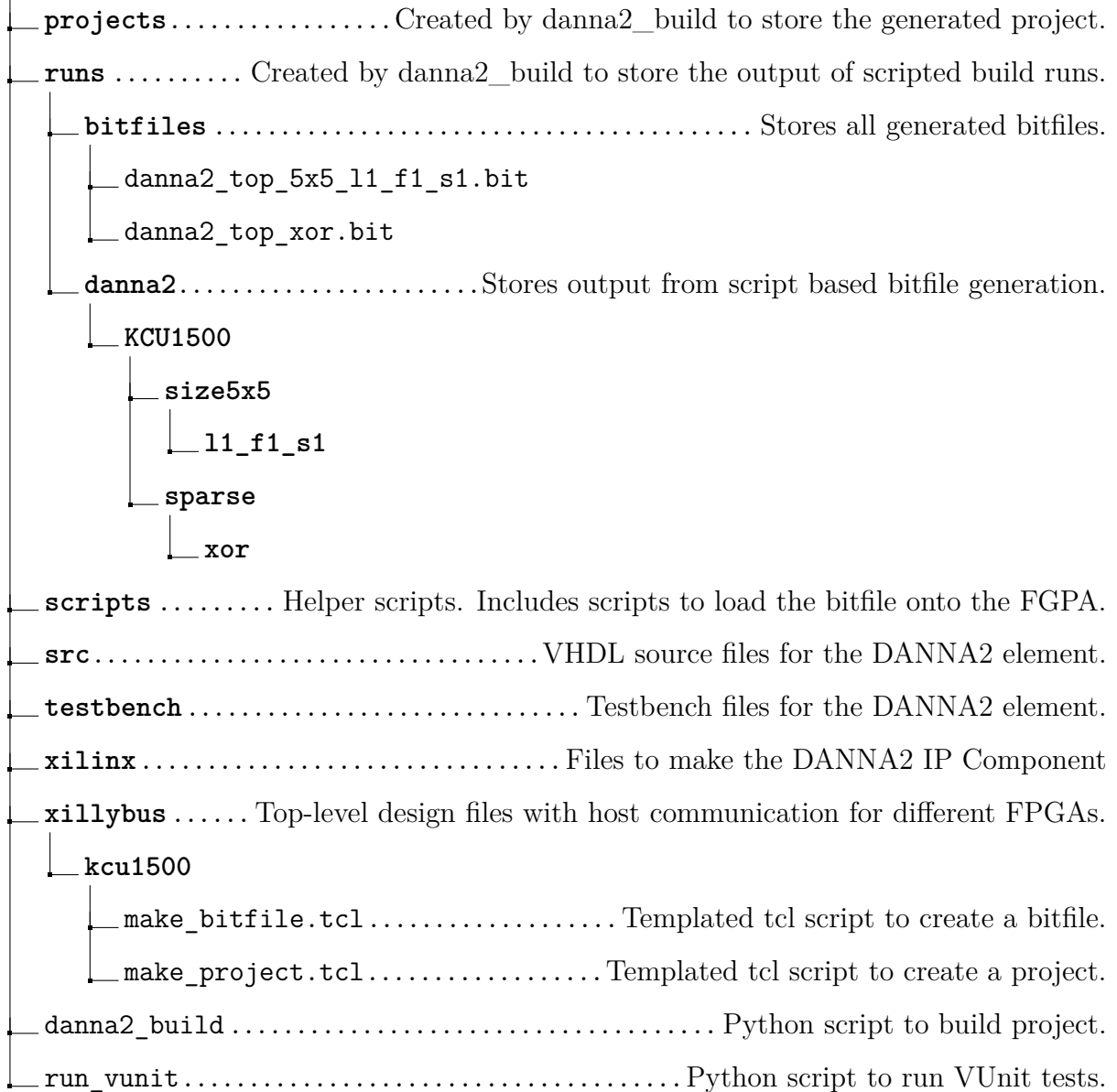


Figure 6.5: Directory structure of the DANNA2 build system.

```
usage: danna2_build [-h] [-p] [-s SPEC_STRING] [-c WIDTH] [-r HEIGHT] [-L]
                  [-l] [-F] [-f] [-T] [-t] [-n DANNA2_NETWORK] [--sweep]

Script to build a DANNA2 bitfile

optional arguments:
  -h, --help                show this help message and exit
  -p, --project              Generate Vivado Project.
  -s SPEC_STRING, --spec-string SPEC_STRING
                           String Specifying the Network Settings.
  -c WIDTH, --width WIDTH   Network width.
  -r HEIGHT, --height HEIGHT
                           Network height.
  -L, --leak                Leak enabled
  -l, --no-leak             Leak disabled
  -F, --fanin               Fanin enabled
  -f, --no-fanin            Fanin disabled
  -T, --stdp                STDP enabled
  -t, --no-stdp             STDP disabled
  -n DANNA2_NETWORK, --sparse-network DANNA2_NETWORK
                           Create a sparse network using the file.
  --sweep                   Generate bitfiles for a sweep of delay values.
```

Figure 6.6: DANNA2 build script settings.

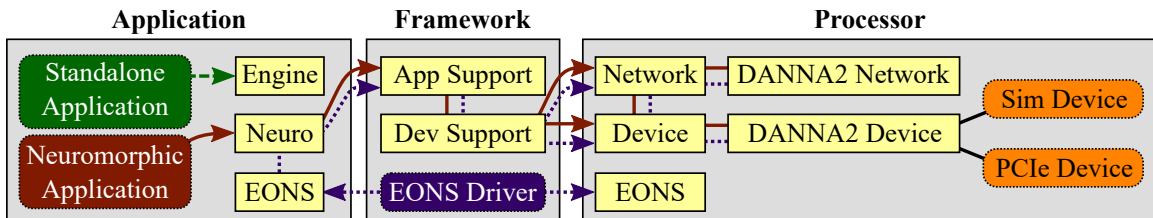


Figure 6.7: Diagram of the TENNNLab Framework with the DANNA2 Processor.

6.2.3 Driver Design

The DANNA2 simulator was already a part of the TENNLab framework. [Figure 6.7](#) shows an overview of the framework structure with the DANNA2 processor included. The TENNLab framework provides a definition of the `NeuroNetwork` and `NeuroDevice` classes. The processor then implements these classes so that they can be compiled as part of the TENNLab framework. The DANNA2 processor implements these classes and also creates its own DANNA2 network and DANNA2 device classes with a specification specific to the DANNA2 processor. The DANNA2 processor can be used with its own device and network classes or through the general processor and network classes defined by TENNLab. The DANNA2 device class is a virtual class defining the DANNA2 interface. This class must be implemented by a child class in order to be used. One implementation of the DANNA2 device virtual class is the DANNA2 simulator device. The DANNA2 simulator device implements the functionality of DANNA2 using a software emulator of the device. It operates by using an event queue to keep track of the activity within the DANNA2 elements. When a spike occurs, the simulator evaluates the spike, which potentially adds more events to the queue.

In order to have the DANNA2 hardware work within this framework, a new DANNA2 device implementation was created called the DANNA2 PCIe device. The PCIe device implements the interface by creating packets to send to the hardware implementation. The hardware will then run the DANNA2 network to determine the resulting output fires. In order for a class to implement the DANNA2 device interface, it must implement the virtual functions shown in [Figure 6.8](#). When the class is constructed, it opens the device files created by Xillybus to send and receive packets to and from the FPGA. The `configure` function creates configuration packets to be sent to the hardware in order to load the elements with the correct settings to match the DANNA2 network file. When packets are created, they are added to the sending FIFO. There are two threads created by the constructor to send and receive packets to and from the FPGA. The write thread takes packets from the sending FIFO and sends them to the write device file. The read thread reads from the read device file and stores fire events. The read thread also updates the current network time. The `apply_input` functions take the input fire events and adds them to a fire queue. The

```

virtual bool configure(Network *network) = 0;
virtual void apply_input(const Coords elm, int16_t weight, uint64_t time) = 0;
virtual void apply_input(int input_id, int16_t w, uint64_t t) = 0;

virtual void monitor_output(int output_id, int start_time = 0,
                           int end_time = -1);
virtual int get_output_count(int output_id);
virtual std::vector<uint32_t> get_output_values(int output_id);

virtual bool simulate(uint64_t steps) = 0;

virtual uint64_t get_timestamp() const = 0;

virtual void pull_network(Network *network) const = 0;

virtual void reset() = 0;
virtual void clear_activity() = 0;

```

Figure 6.8: DANNA2 Device Virtual Functions.

`simulate` function takes the number of steps to simulate as a parameter. `simulate` then reads the fire events which occur within this number of steps from the fire queue, converts them into fire packets, and sends them to the hardware. `simulate` now waits until the hardware timestep matches the last timestep of the simulation before returning. Signals are used between the read thread and the main thread to signal when the last packet is received. The `monitor_output`, `get_output_count`, and `get_output_values` functions are used to mark an output for recording, to get the number of fires from an output, and to get the fire times from an output, respectively. The `get_timestep` function returns the current timestep the hardware is on. The `pull_network` function returns the current state of the network which was loaded onto the hardware. The `reset` and `clear_activity` functions create a reset packet and adds it to the queue to be sent to the hardware. Additionally, `clear_activity` clears out the input and output fire queues.

The PCIe driver allows the FPGA hardware to be used as an implementation for DANNA2 within the TENNLab framework. This allows existing networks to be evaluated and new networks to be trained. Any of the existing TENNLab applications or training methods can now be run on the DANNA2 hardware implementation. TENNLab also has a Neuromorphic Device Factory class. This class allows the initialization of multiple different neuromorphic

devices to be used within a multiple device run. Each of these devices needs to implement the same neuromorphic processor type, but this class allows multiple DANNA2 hardware instances to be run with multiple software DANNA2 instances. The device factory defines the settings for each processor instance. Then when an application requests a neuromorphic device, the device factory will instantiate one and return it to the application.

6.3 Verification

The single FPGA DANNA2 was verified using two main approaches. First the Xillybus communication was setup with a simple loopback implemented on the FPGA. With the loopback, any packet added to the incoming FIFO would be directly sent to the outgoing FIFO. This setup was designed to test the performance of the Xillybus component of the design. The next testing was over the entire system. The DANNA2 network was added onto the FPGA, and various networks and tasks were run on the FPGA and compared against the software simulator. Since DANNA2 is a deterministic neuromorphic design, the output from the simulator should match that of the hardware exactly. A `single_run` utility was written to load a network, provide input, and show the output. The `single_run` program was used to test the output of the simulator versus that of the hardware for multiple test networks. After the output from a single run was verified to match, EONS and reservoir learning algorithms were used to further test the design. Both of these learning algorithms are difficult tests, since they make extensive use of the different commands and generate random networks to test. If the learning algorithms are seeded with the same random seed, then the networks and fitness values should match for both hardware and software. This test is particularly difficult since a single wrong output might steer the populations in the wrong direction, resulting in vastly different fitness scores. Although the extensive verification uncovered many unaccounted for edge cases and hidden bugs, the issues were discovered and corrected such that the hardware was verified to match the simulator for all of the different tests.

Chapter 7

SNACC

Neuromorphic computing people got steamrolled for decades because Moore's Law just kept getting better and better . . . so they could just never catch up. . . That's not the case anymore.

– Todd Hylton [65]

The primary focus of this dissertation is the design and implementation of the Scaled-up Neuromorphic Array Communications Controller (SNACC). SNACC is designed around what was learned from the previous Tiled DANNA project as well as the improvements introduced with DANNA2. Like Tiled DANNA, SNACC's goal is to expand the capabilities of neuromorphic processors by combining multiple neuromorphic processors together to extend their capacity and functionality. As discussed in the introduction and shown in [Figure 7.1](#), SNACC is comprised of three major components. First, there are the neuromorphic arrays, which are tiled together to form local grid networks. Then, there is the Neuromorphic Array Communications Controller (NACC), which is used to send packets from the host to the neuromorphic array and also to connect to other NACCs further away. Finally, there is the host machine, which is a traditional computer used to control the operations of the network, to provide input to the network, and to interpret the output from the network. In between these components is the crucial communications interconnections, which allow the different components to communicate with each other. This chapter will cover each of these major components in detail, discussing the design decisions, implementation, and testing. However, the overall design goals and principles of SNACC will be covered first.

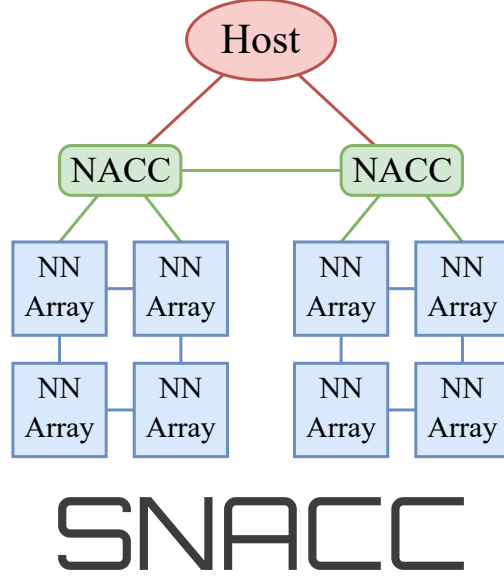


Figure 7.1: High-level overview of SNACC.

SNACC took a different design approach from Tiled DANNA. Instead of using a synchronous clock for the entire system, SNACC takes a globally asynchronous, locally synchronous (GALS) approach. The neuromorphic arrays within a single FPGA are synchronous; however, the communication channels, the different FPGAs, and NACCs all operate asynchronously to each other. A GALS approach was chosen to avoid the problems of providing a synchronous clock to a large system and to allow the system to scale to larger sizes than a synchronous system would permit. SNACC also tries to keep the design of the communication system simplified through the use of a special communications controller. The communication setup is made using point-to-point connections. Local point-to-point connections are used to connect the neuromorphic processors implementing the sub-arrays. Additionally, the devices implementing the sub-arrays connect to the host via point-to-point connections to NACC. The local connections are what allow the system to be able to scale to utilize many tiled neuromorphic processors. The use of NACC additionally simplifies the complexity of connecting to the host. Instead of requiring one communications board per processor to connect to the host, one NACC can facilitate the communication for multiple neuromorphic processors.

Another main design principle of SNACC is to make the array function as closely as possible to a single-processor array. SNACC should function like a large neuromorphic array without having complex, special rules for board boundary conditions. Although there are a few limitations for inter-board communications, a goal was to keep these to a minimum so that the arrays are easier to design and have fewer constraints.

One of SNACCs main design goals is to make it as high performance as possible. DANNA2 networks are capable of running at high speeds with a global clock cycle occurring at a rate of 10 MHz or faster. Therefore, the communication system has to be sufficiently high speed to not bottleneck the system. Because of this, the communication interconnects were chosen based on having high throughput and low latency.

7.1 DANNA2

One of the main components of the neuromorphic system is the neuromorphic processor design used. For SNACC, the DANNA2 processor is used to implement the neuromorphic arrays. DANNA2 is the logical choice for this design since DANNA2 is a digital neuromorphic processor design which can be prototyped on an FPGA or fabricated in VLSI. DANNA2 has many improvements over the first DANNA design.

DANNA2 has a simplified element scheme, where each element incorporates 24 synapses and one neuron. This change eliminates the confusing and counter-intuitive interactions which could occur in DANNA. For example, in DANNA2, the nonsensical connections of a neuron element connecting to a neuron element, or a synapse element connecting to a synapse element cannot occur.

DANNA2 also greatly simplifies the clocking structure. Instead of needing six clocks, DANNA2 only requires one system clock. This single system clock is used to drive each sub-cycle. A network cycle takes ten sub-cycles to complete. Having only one clock signal greatly simplifies the routing of the clock trees and makes it easier for large designs to meet timing requirements during place and route.

The sampling of synapse activity is also greatly simplified in DANNA2. In DANNA the synapses were sampled one at a time in a pseudorandom order and only the first synapse

to cause the fire was potentiated in STDP. With DANNA2 all the synapses within a single timestep are evaluated for that timestep as though the events occurred at the same time. This means there is no sub-cycle resolution between the events, that it simplifies the logic of how synapses work, and that it removes the need for a random number generator to randomize the sampling pattern of the synapses.

Another major change with DANNA2 is that the output from each element is from a neuron, and a neuron's fire event is a single binary signal that indicates if a fire occurred during the current timestep. Since DANNA could have elements configured as a synapse or a neuron, the inter-element signals had to support transmitting a magnitude with the event. Because DANNA2 only has binary events, the communication between elements on different chips is greatly simplified; the fire event is represented as a single bit instead of requiring a byte to represent it.

DANNA2 also has a simplified command packet structure which makes operating the array easier. DANNA has the following nine different input packet types: Run, Halt, Capture, Shift, Reset, Null, Load Element, Step for N cycles, and Fire. DANNA2, on the other hand, only has three input packet types: step&fire, reset, and configure. The run, halt, null, step, and fire commands from DANNA are all replaced by the DANNA2 step&fire command. The step&fire command works by sending the inputs for a single timestep along with the network cycle on which they occur. One constraint of the input packets is that they must be sent in ascending order. This lets DANNA2 know that the network can be evaluated up to the last network cycle for which input has arrived. A step&fire packet with a network cycle and no fire activity causes the array to run to the sent network cycle. The step&fire command introduced with DANNA2 greatly simplifies the operation of the processor and still performs all the same tasks as DANNA but with a fraction of the command types.

All of these changes with DANNA2 make the task of building a large multiprocessor array easier and also makes DANNA2 a great neuromorphic processor design for building a large-scale interconnected array. Because of the hierarchical design of DANNA2, only certain portions of the design had to be updated to be used in a tiled 2D array, and the majority of the DANNA2 source files could be used unchanged from the single-board DANNA2 design. The only files which needed changing were the element array and the array control files. All

of the necessary changes relate to the design of the local communication between sub-arrays, which will be discussed in detail in [Section 7.3.2](#).

The DANNA2 design has been extensively tested with per component testbenches and thorough system-level testing on a single FPGA. High-level VUnit tests were also written to test the new changes made to support sub-array communications. With this test, a new quad-DANNA2 design was created and the communication between the two-by-two grid was tested.

7.2 NACC

The Neuromorphic Array Communications Controller (NACC) sits between the neuromorphic array and the host computer. The primary goal of NACC is to support the communications between the host and the different neuromorphic boards. With the Tiled DANNA project there was a separate communications card for each neuromorphic array. This led to the additional challenge of trying to synchronize communication over asynchronous channels when the operation of the array is synchronous. With NACC, a single communication board is used to support the communication for all of the neuromorphic processors. This design greatly simplifies the complexity of the communication system and opens up some interesting future possibilities. NACC enables communication by allowing the use of separate communication protocols between the host and NACC and between NACC and the neuromorphic processors. NACC opens up the options for the communication interfaces used since there is a conversion board in between. On the host side, PCIe was chosen since it is a high-performance communication bus typically used for coprocessor applications. This allows NACC to have the most direct connection to the host as possible. For connecting to the neuromorphic processors, the connection has to be split to multiple different end points. In order to get the necessary performance, a high-speed, multi-gigabit serial connection was chosen. Multi-Gigabit Transceivers (MGT) are used to send packets quickly to their destination using a point-to-point connection.

The second reason to use a communications controller is that it enables further scaling of the system. If one communications controller is not enough to handle the number of

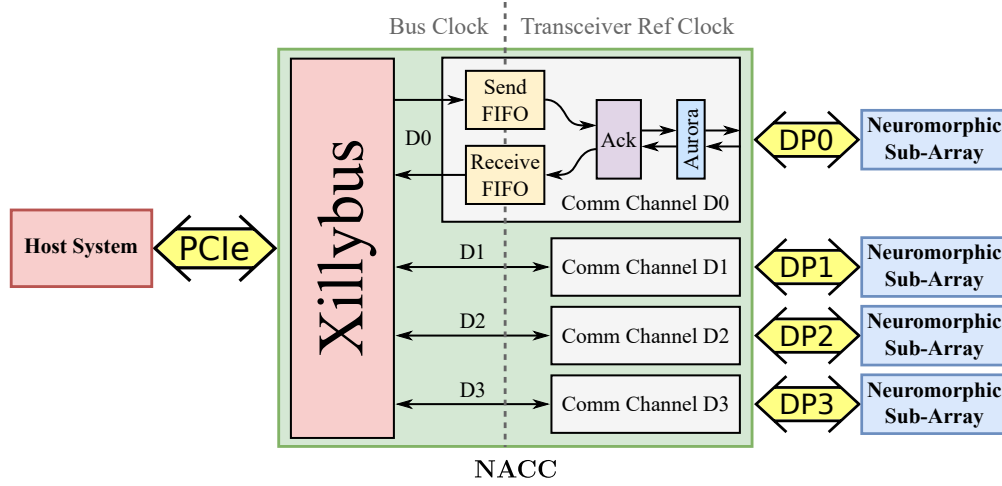


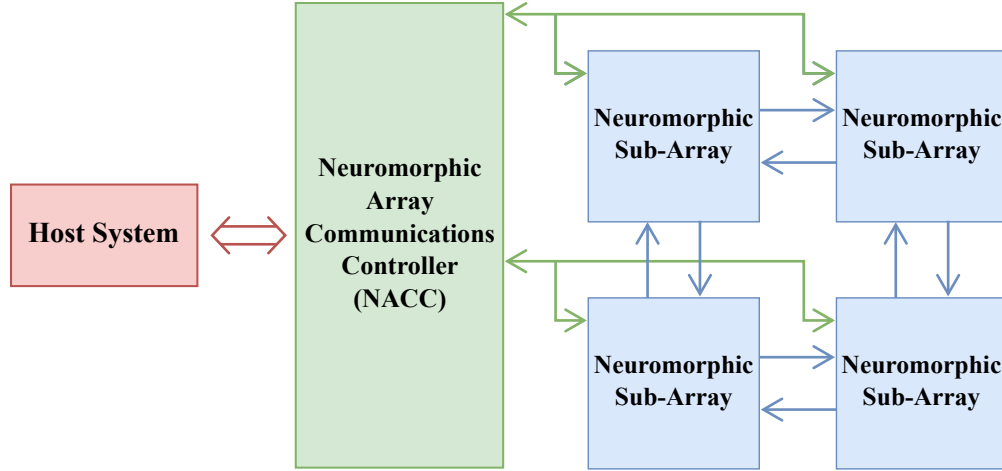
Figure 7.2: NACC Design.

neuromorphic processors, then another NACC can be used to allow for additional scaling. A second NACC could be placed in a host with multiple PCIe slots, which would increase the communication bandwidth to support additional scaling. Further scaling could also be achieved by arranging multiple NACCs in a tree.

The NACC can also be used to relieve the tasks that must be performed on the host. Currently, the driver on the host directs packets to the correct neuromorphic processor; however, this task could be offloaded to the neuromorphic controller. Additionally, a NACC could be used to operate the neuromorphic array without a host. If an input source is connected directly to a NACC, it could be used to create packets from that input source and provided input to the neuromorphic array. Output from the array could then be post-processed by a NACC and sent to the output destination to enable host-free operation of the array.

Essentially, the NACC board is a required component of SNACC that facilitates communication between the host and the neuromorphic processors and could be leveraged in future work to enable further scaling of the system. By adding more functionality to the NACC, it could be used to offset or replace some of the functionality provided by the host computer and even make it possible to run the network without a separate host system.

Figure 7.2 shows an overview of the current design of the NACC. The NACC is designed to convert packets from the PCIe interface from the host to Xilinx high-speed transceivers. The



Connection/Packet Types

↔ = Host Packets via PCIe Bus

↔ = Communication Board Packets via Aurora

↔ = Sub-Array Communication via Aurora

Figure 7.3: SNACC Communication Types.

transceivers use the Aurora protocol to encode packets and are connected to the neuromorphic processors using SMA cables. More details of the communication design can be found in [Section 7.3](#). Inside the NACC design, there is a Xillybus IP block which handles the PCIe protocol and provides eight streams, one read and one write stream for each of the four connected neuromorphic processors. If more neuromorphic processors are used in the design, the number of data streams would be increased. Presently the routing is conducted by the host, and the NACC is used to transfer the packets to the neuromorphic processors. If needed, the design could be changed to have only one PCIe stream transferring packets to the NACC and to have the NACC route the packets to the correct neuromorphic processor.

7.3 Communications

The most challenging part of designing SNACC was the communication system. SNACC has three major communication types within the design. These types are shown in [Figure 7.3](#). This section will cover the design and testing of each of these communication types. However,

the first thing that will be discussed is the custom acknowledgement logic that was designed to work across the Aurora channels. A drawback to using Aurora is that there is no built-in method of guaranteeing that transmitted packets will arrive. If logic is added to resend the missing packets, they will no longer be in their original order. The custom acknowledgement logic has high performance and will retransmit packets when they are lost or the packet is corrupted, while making sure the packets do not get reordered.

7.3.1 ACK

Since Aurora is designed as a link-level protocol, it does not have any provision for handling transmission errors. Although the likelihood of a transmission error in an Aurora channel is unlikely, they do occur. As the transceivers transmit at higher frequencies, the likelihood of an error increases. Corrupt or missing packets will cause the neuromorphic array to operate incorrectly. If the missing packet is a load command, then an element will not be properly configured. If the packet is an input or output packet, then a fire event will be missed by the array or by the host. If the packet happens to be the last packet, which marks the end of the simulation, then the host will think that the array is still running and the system will hang. All of these scenarios are undesirable, which means a retransmission protocol must be put into place. This protocol guarantees that packets will be delivered in the correct order, at their destination, free of errors. The Aurora ACK component detailed in this section was designed for this purpose.

Comparison of Solutions

Many different approaches could have been chosen for reducing or eliminating the chance of transmission errors. One approach is paying for an existing solution that implements an error correction or retransmission design. Three paid options include using a Reed-Solomon Decoder and Encoder, a Forward Error Correction (FEC), or an Enhanced Forward Error Correction (EFEC) IP block, all available for purchase from Xilinx. Another possible solution included with Vivado is to use the AXI Chip2Chip IP core. This core uses per-lane Hamming EEC codes to mitigate bit errors caused by single-bit corruption. The Hamming ECC module

used by AXI Chip2Chip implements Single-bit Error Correction and multiple (Double) bit Error Detection (SECD) functions. The downside to this IP core is that it is designed for AXI4 and AXI4-Lite, not AXI4-Stream.

After surveying the potential solutions, two major approaches to the problem became apparent. The first option is to use an Error Correction Code (ECC) to recover data lost from a bad transmission. The other approach is to use an error-detecting code, such as a Cyclic Redundancy Check (CRC), and then use an Automatic Repeat Request (ARQ) protocol to resend the data until the data is received correctly. Between the two solutions, using a ARQ protocol seemed like the best option. An ARQ protocol is able to successfully retransmit the data eventually, as long as the errors in the packet are detected. This means that the solution is more robust and is not limited to only being able to correct certain errors. In theory, both methods could be used together. The ECC could correct easier-to-solve errors, and retransmission could be used to correct more corrupted or missing packets. In practice, errors occur infrequently and the errors that do occur are a result of the line losing stability for a brief amount of time, resulting in total packet loss instead of bit errors. The addition of a light-weight, fast ARQ protocol was shown to be sufficient to prevent packet loss without a noticeable performance penalty.

Many types and variations of ARQ protocols are available; of these, many use some form of the sliding window protocol. A sliding window protocol is used in packet-based data transmission protocols when reliable, in-order packet delivery is required. It is used in many transport layer protocols, including the Transmission Control Protocol (TCP). The sliding window protocol works by assigning each packet of data a unique, consecutive sequence number. This number is used to place packets in the correct order and to identify duplicate or missing packets. However, there is a limit to the range of numbers that can be used for this sequence number. By placing limits on the number of packets which can be transmitted or received at a given time, an unlimited number of packets can be represented by a fixed-size sequence number, which is able to wrap around to the minimum value once the maximum value is reached. The window is the logical range of packet numbers that the sender or receiver will send or accept. As packets are successfully sent and received, the window boundary slides up to include higher sequence numbers. This is why the protocol is named “sliding

window.” A maximum size for the window is half the size of the largest sequence number. In practice the window size is much less than this value. Some protocols even support changing window sizes, such as the TCP protocol.

Sliding Window Protocol

The basic sliding window protocol works as follows [22]. The transmitter and receiver both store the current sequence number, n_t for the transmitter and n_r for the receiver. These numbers initially start at the first sequence number, zero. The transmitter and receiver also have a window size w_t and w_r , respectively. These sizes may be held constant or could vary throughout the transfer. The window size must be one or greater. n_t is the sequence number of the next packet to be transmitted, and n_r is the sequence number of the first packet not yet received. Both sequence numbers are monotonically increasing with time.¹ The receiver also tracks the highest sequence number received; n_s is the one-more-than-the-highest sequence number received. When the receiver only accepts packets in order ($w_r = 1$), n_r is the same value as n_s . All packets with a sequence number less than n_r have been received and no packets with a sequence number greater than n_s have been received. Between these two numbers, some number of the packets have been received. After receiving a packet within its window size, the receiver updates the variables appropriately and transmits an acknowledgment to the sender with the value of the new n_r . The transmitter monitors the highest acknowledgment received, n_a , knowing that all the packets up to n_a have been received. However, it does not know if the packets between n_a and n_s have been received; i.e. $n_a \leq n_r \leq n_s$. The sequence numbers stored in the variables must obey the rule that $n_a \leq n_r \leq n_s \leq n_t \leq n_a + w_t$. That is:

- $n_a \leq n_r$: The highest acknowledgment received by the transmitter cannot be higher than the highest packet acknowledged by the receiver.
- $n_r \leq n_s$: The span of received packets cannot pass the end of the partially received packets.

¹The sequence number wraparound is not shown in the following relations, which assume infinitely large numbers are used to represent the sequence numbers.

- $n_s \leq n_t$: The highest received packet cannot be higher than the highest sent packet.
- $n_t \leq n_a + w_t$: The highest sent packet is limited by the highest acknowledged packet and the size of the transmission window.

The transmitter operates by sending data up to w_t packets ahead of the latest acknowledgment n_a . In other words, it may transmit packet n_t as long as $n_t < n_a + w_t$. Without any communication errors, all the packets will be acknowledged and n_a will become equal to n_t . If an error occurs and the acknowledgments do not arrive after a reasonable delay, the transmitter may retransmit all the packets between n_a and n_t .

The receiver receives packets with sequence number x . The receiver then checks if x falls within the receive window, $n_r \leq x < n_r + w_r$. If x falls within the window, the receiver accepts it. If $x = n_r$ then n_r is increased by 1, which makes it possible to receive further packets. Packets less than n_r are released in sequence order to the application. If $x > n_r$ it is stored until all the proceeding packets have been received. If $x \geq n_s$ then n_s is updated to $n_s = x + 1$. If x falls outside of the receive window, the packet is discarded and n_r and n_s are not modified. Regardless of whether the packet is accepted or not, the receiver still must transmit an acknowledgment containing the current n_r . However, there is no point in having a larger receive window w_r , than the size of the transmit window w_t , since it cannot accept a packet which has not been transmitted. The useful range for w_r is $1 \leq w_r \leq w_t$.

With many implementations, the sequence number is a fixed number of bits. This means the sequence numbers must wraparound when the maximum value is reached, which allows the same range of numbers to be reused multiple times as sequence numbers. The key to this working is to correctly handle the detection and usage of numbers when the wraparound takes place. A wraparound is detected when the difference in the sequence numbers is more than half of the sequence range away from the expected number. With unsigned values on most hardware implementations, the math and the comparisons both work properly when the number overflows. Therefore, the main requirement is to treat sequence numbers whose sequence number is more than half the sequence number range away as old sequence numbers.

Variations of the Sliding Window Protocol

There are some common variations to the sliding window protocol which have their own names. The simplest version is the stop-and-wait protocol. With the stop-and-wait protocol both the transmit and receive window are only one packet in size. Therefore, once a packet is sent, the sender stops and waits for the packet to be acknowledged before sending the next packet. With this simple protocol only two sequence numbers are required to implement the protocol.

Another form of the sliding window protocol is the Go-Back- N ARQ. With Go-Back- N , the transmit window is greater than one, $w_t > 1$, but the receive window is fixed at one, $w_r = 1$. This means that the receiver will only receive packets in order. If a packet is not the next packet in the sequence it is dropped. In order to start receiving packets again, the sender has to start over with the last packet acknowledged and resume sending from there. In other words, the transmitter goes back n packets and starts transmitting again. In this case $n = n_t - n_a$.

Selective repeat is another variation of the sliding window protocol. With this variation, the acknowledge packets include information about the missing packets and the sender only selectively resends the missing packet to the receiver. Packets which are received correctly at the receiver but are received out of order are stored until the missing packets are received.

Chosen Variation

For Aurora ACK, two variations of the sliding window protocol were tried. The first version is an implementation of the stop-and-wait protocol, used as a comparison point since it was not expected to perform well. The second version implemented is a modified version of the Go-Back- N protocol. The basic algorithm for Go-Back- N is found in [Appendix B.1](#). This version of Go-Back- N added aggressive retransmission and flood control. The aggressive retransmission starts to resend unacknowledged packets once all of the newly added packets have been sent once. Flood control allows a fixed number of resends and then waits for a certain number of cycles before sending again. The implemented Go-Back- N protocol only sends acknowledgment packets, no negative acknowledgment packets are sent. Through testing

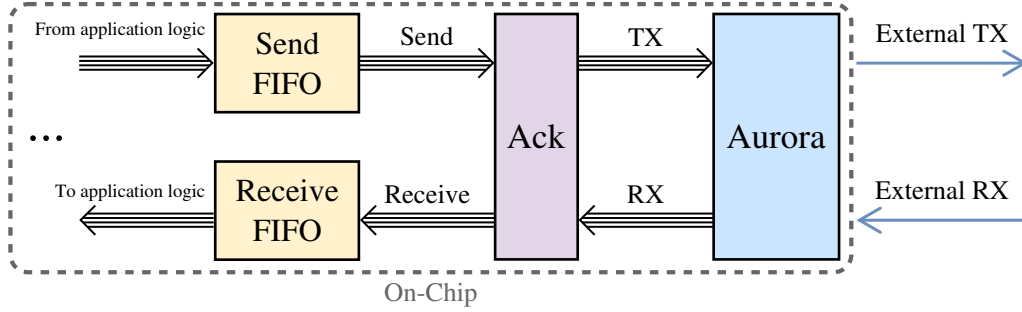


Figure 7.4: Diagram showing how Aurora ACK is used.

and tuning, the performance of this modified Go-Back- N protocol met all the requirements of the system and was chosen to be used in SNACC.

In summary, a custom, lightweight, Go-Back- N retransmission protocol was developed to work directly with AXI4-Stream, since an existing off-the-shelf solution, which met the needs of the project, did not already exist. The custom protocol is specially designed with aggressive retransmission and flood control for low-latency and high-bandwidth, and to work well with the application of neuromorphic computing. The protocol was developed and added on top of Aurora to ensure that packets sent over the Aurora channel are received correctly.

Design

Aurora ACK is used to add an ARQ protocol to an AXI4-Stream for use over an Aurora channel. [Figure 7.4](#) shows how Aurora ACK is connected to surrounding components for use. Aurora ACK module implements the ARQ logic and also has several testing architectures: passthrough, loopback, and dev. Passthrough connects the send to TX and RX to receive without modifying the signals. Loopback connects send to receive and does not use TX and RX. Dev implements connecting send to `/dev/null` and receive to `/dev/zero`. These testing architectures are shown in [Figure 7.5](#).

The design of the architecture for performing the ARQ logic is broken into two halves, one for sending and the other for receiving packets. The send and receive AXI4-Stream buses operate on complete packets at a time, with the TX and RX buses operating on one 64-bit word² at a time with each frame containing the data for one packet. This design allows the

²Aurora is configured to operate with 64-bit words.

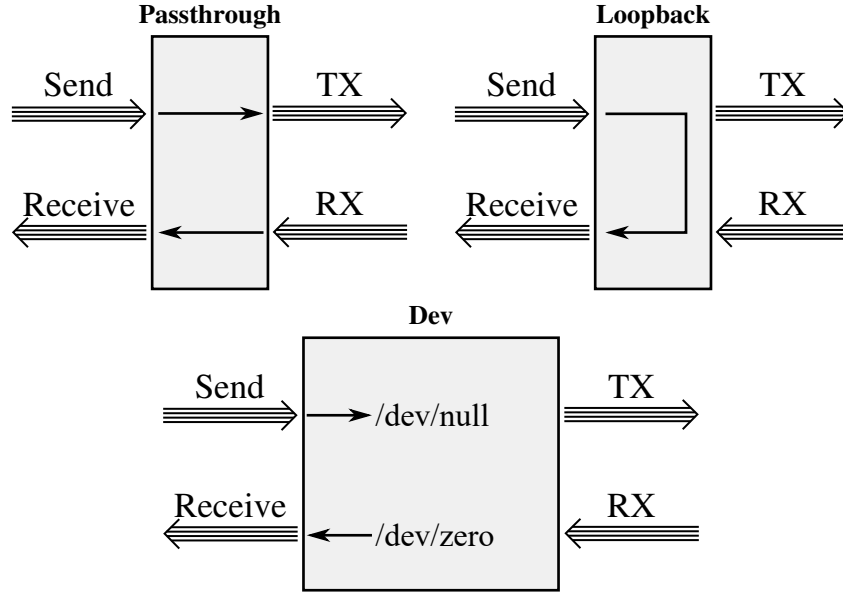


Figure 7.5: Diagram showing the testing architectures for Aurora ACK.

user logic on the FPGA to operate with whole packets, while the packets are split up into transmittable words within a frame when the data is sent over the Aurora channel. [Figure 7.6](#) shows a detailed schematic of how the Aurora ACK component is implemented with the Aurora ACK Send and Aurora ACK Receive subcomponents. Three signals are used to pass packet received information from the receiver to the sender. These signals are `ack_num`, `ack_send`, and `ack_set`. `ack_set` indicates that a new acknowledgment packet should be sent with `ack_send` as the acknowledgement number. `ack_num` indicates the number of the last packet sent that has been received at the other side of the channel.

A diagram showing the basic design of Aurora ACK can be found in [Figure 7.7](#). Once a packet arrives on the send bus, the read control process adds the packet, along with its assigned sequence number, to a custom window buffer where it will wait to be sent. The send and drop control is the main process in charge of operating the sender. In the starting state, the Finite-State Machine (FSM) checks to see if it can send a data packet, can drop an acknowledged packet from the window buffer, or send a new acknowledgment packet in this order. The state machine can only perform one of these tasks at a time, and this order was chosen intentionally. One cycle is needed to decide which action to take, then one cycle is used to perform the action. Sending packets is the highest priority in order to reduce

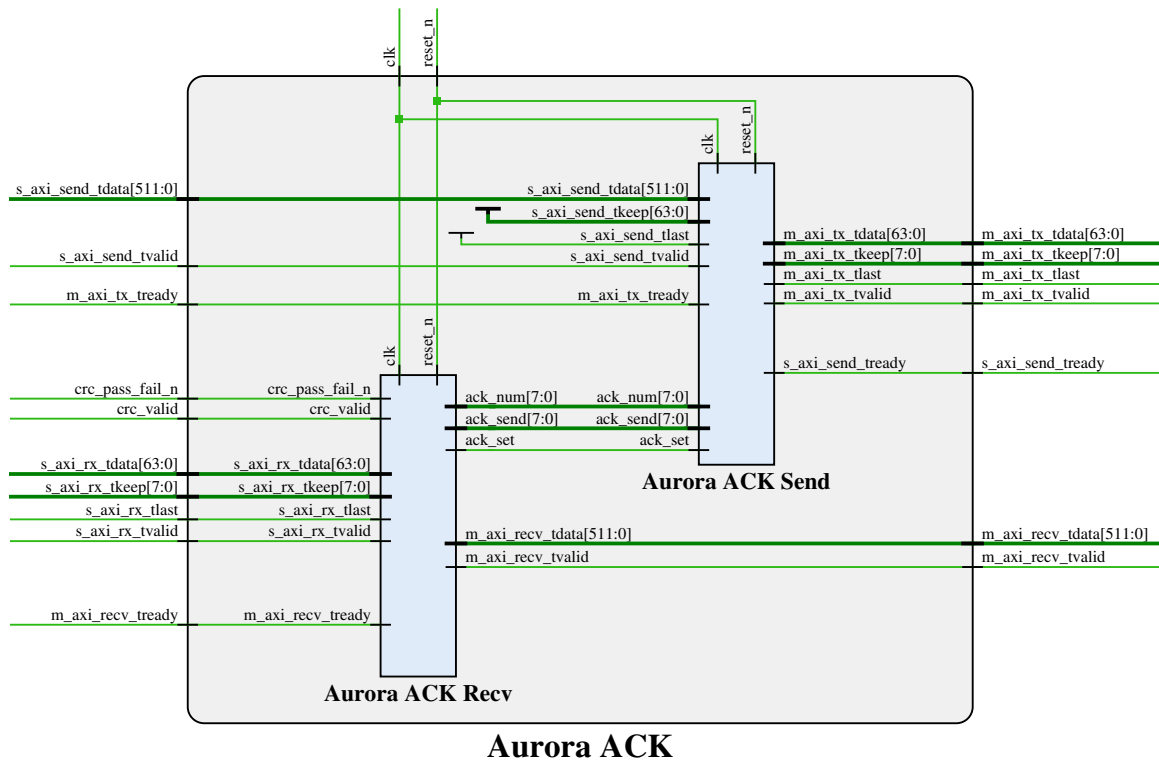


Figure 7.6: Schematic showing the components of Aurora ACK.

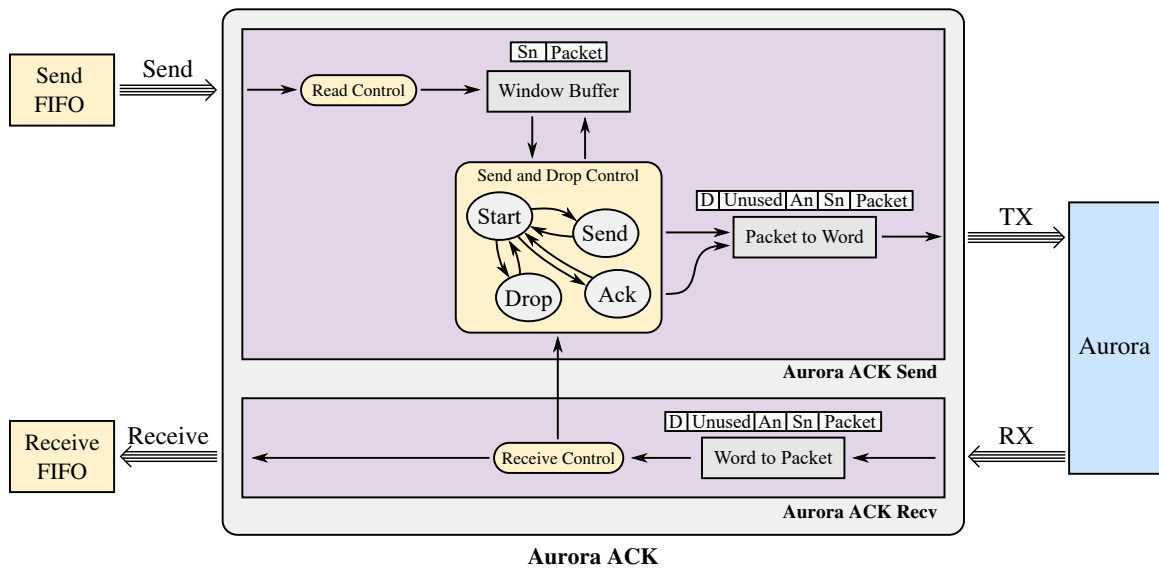


Figure 7.7: Simplified diagram showing the design of Aurora ACK.

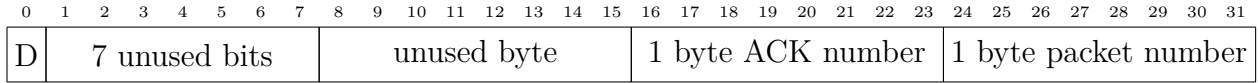


Figure 7.8: Aurora ACK header structure.

the latency of the transmission. A packet is sent anytime the window buffer is not empty, the packet to word component is ready for another packet, and retry count is less than the maximum flood value. The maximum flood value is used to implement flood control, by limiting the number of retries within a short time frame. When the packet is sent, a header is added as the first word of the packet. This header includes a bit, D , to indicate if the packet is a header only acknowledgement packet or a data packet with the data following the header. When $D = 1$ the packet contains data. If the header is for a data packet, it also contains the packet's sequence number, Sn . In both cases the header also includes the sequence number of the latest packet as an acknowledgement that it was received. [Figure 7.8](#) shows the packet structure of the header. The header is shown as a 32-bit word, which is the minimally supported word size. If a larger word size is used, the header packet will contain additional unused bits. The packet to word component takes multiple cycles to send the packet to Aurora. One cycle is required for each word that must be sent plus one word for the header. This fact gives the state machine plenty of opportunity to drop acknowledged packets from the window buffer. Once the retry count reaches the maximum flood value, the send and drop control waits the "flood wait" number of cycles before starting to resend. This waiting period is cut short if a new packet is added to the window buffer. When the new packet is added, it is sent first and followed by the already-sent packets.

The state with the next priority is to drop an already-acknowledged packet from the window buffer. A packet is dropped when the window buffer is not empty and the ACK number is greater than or equal to the oldest packet's sequencing number. If the oldest packet's sequencing number is more than half of the maximum sequence number away, then it is inferred that the sequence numbers have wrapped around and the packet is dropped.

Finally, the state with the lowest priority is to send an acknowledgement packet if there is no data to send and there is a new packet to acknowledge. This state is reached when there is

a new ACK number and the sender is ready to send. This state has the lowest priority since, ideally, the acknowledgement would be sent with a data packet. Also there is a finite number of packets which can be dropped before more can be sent. Typically the number of packets to drop is low, so the receiver is not starved for a new acknowledgment. An acknowledgment packet consists only of a header with $D = 0$ and the acknowledgment number.

The packet to word component is a custom AXI4-Stream width converter which follows the AXI4-Stream protocol and takes a packet with header as the input and outputs words of the packet, one-at-a-time, within a frame. From this component, the packets are sent out of the TX AXI4-Stream bus.

When a word arrives on the RX bus, the word is first sent to the custom word to packet component. This component does the inverse operation of the packet to word. Once a complete packet is received, the packet is sent to the receive control process. This process checks the packet for errors using the CRC provided from Aurora, updates the signals sent to the send component, and if the packet is the next in sequence, sends it out the receive bus. When a new packet is received, the receive control first checks to see if the packet has a CRC and that the packet is the expected length based on its type. If the packet passes this check, the acknowledgement number is read from the packet and used to update the `ack_num` signal sent to the send logic. The next check determines if the packet is a valid data packet and if the receive bus is ready. If so, the packet is sent out of the receive bus. If the packet is not the next packet in sequence, or the receiver is not ready to receive, the packet is dropped. The last check is to determine if the packet is the last received packet in the sequence; if it is, then `ack_set` line is asserted to send another acknowledgment to the sender, since the sender is still trying to send an already received packet. When a packet is successfully sent out of the receive bus, the receive number is increased, the `ack_send` number is updated, and `ack_set` is asserted.

The custom window buffer is a key memory component used to store packets within the send window. The window buffer is a special type of circular buffer with dual read and write ports. Packets are always added at the end, but they can be accessed in any order based on their relative position in the buffer (i.e. packet 0 is the first remaining packet that was added and packet 1 is the packet added after packet 0). [Figure 7.9](#) shows the top-level interface

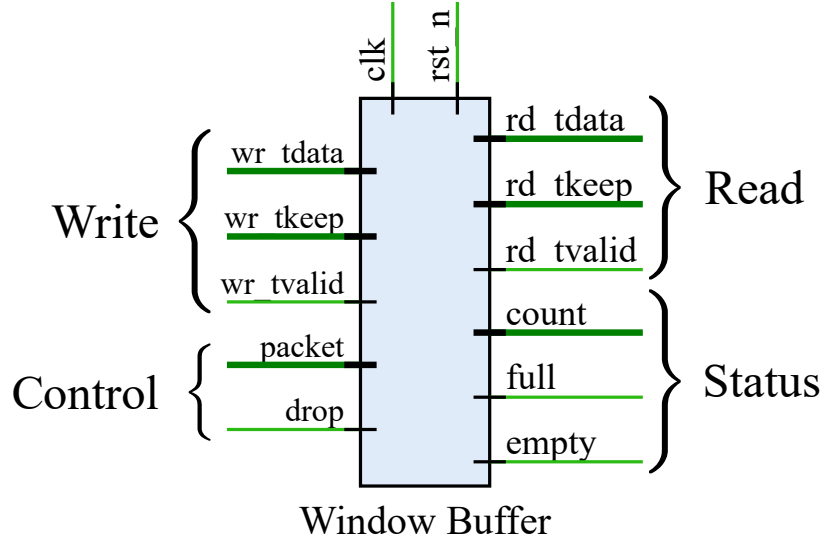
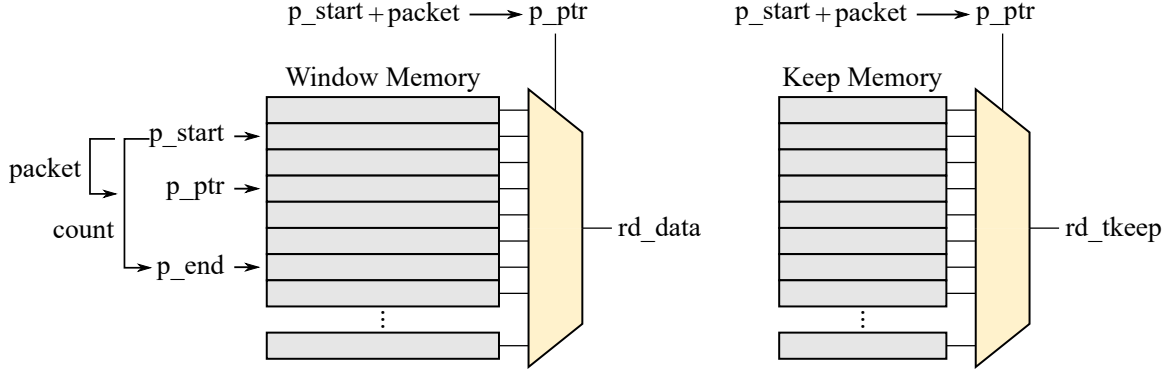


Figure 7.9: Top-level interface for the window buffer.

for the window buffer. Data is added to the buffer using the write ports. The interfaces are compatible with AXI4-Stream if the inverse of the full signal is used as TREADY. The packet line is used to select the packet to read. The read bus is also AXI4-Stream compatible and will be valid when packet is less than count. Count is used to indicate how many packets are stored in the buffer. Full and empty are used to indicate the buffer's current state. [Appendix B.2](#) shows the pseudocode for the window buffer operation.

[Figure 7.10](#) shows how the window buffer is designed conceptually. The window buffer contains two circular buffers. One buffer stores the packet information while the other stores the keep bits for the packet. There are three pointers used to keep track of the buffer. The first is `p_start` and it is used to indicate the location of the first packet stored in the buffer. The `p_end` pointer is used to store the location of the last packet location plus one. The pointer `p_ptr` is equal to `p_start` plus `packet` and is used to read from the buffer. The default frame depth for the window buffer is 16, but this number is configurable. The buffer is full when `p_start` is equal to `p_end` and the count does not equal zero. The buffer is empty when `p_start` is equal to `p_end` and the count is equal to zero. The same pointers which are used to index the packet buffer are also used to index the keep buffer.

Both the window buffer and Aurora ACK components have generics which can be used to customize the components for a specific application. With the window buffer, the frame



$\text{FRAME_DEPTH} = 2^4 = 16$ $\text{rd_tvalid} \leftarrow 1$ when $(\text{packet} < \text{count})$ else 0
 $\text{full} \leftarrow 1$ when $(\text{p_start} = \text{p_end} \text{ and } \text{count} \neq 0)$ else 0
 $\text{empty} \leftarrow 1$ when $(\text{p_start} = \text{p_end} \text{ and } \text{count} == 0)$

Figure 7.10: Design of the window buffer.

width and window size are both configurable. With Aurora ACK, the word size, frame width, maximim flood count, flood wait count, and whether a new packet resets the flood count are all configurable. This makes both components flexible and usable for additional applications. The pseudocode for Aurora ACK send and Aurora ACK receive can be found in [Appendix B.3](#) and [Appendix B.4](#) respectively.

Verification

The content of this section is published in an IJCNN conference paper titled “Neuromorphic Array Communications Controller to Support Large-Scale Neural Networks” by Young et al. © 2018 IEEE [121], reprinted with permission.

Aurora ACK, the window buffer, packet to word, and word to packet have each been tested extensively using testbenches with assert statements verifying the correct functionality of each design. Test vectors were added to test the edge cases of each design, as well as regression test vectors when bugs were identified. They have also been proven to operate correctly with on-board performance testing and through the building and testing of SNACC.

Multiple benchmarking tests were performed to verify the performance of the PCIe and Aurora communication. These tests were designed to test the neuromorphic communications system with a communication board (shown previously in [Figure 3.21](#)). Previous work verified

the PCIe and Aurora portions of the system. New tests were added to test the effect of adding the Aurora ACK ARQ protocol. The complete tests compare the performance of the new communications board against the previously used Cypress FX3 USB based communication. Each test setup is designed to measure either the complete communications path or an individual component of the communications path. Each test implemented a communications loop back. The host sent messages and measured how long the same messages took to be received. The packet size for the messages was chosen to be 64 bytes long. The test setups used to measure performance are as follows.

FX3: Measures the performance of the prior Cypress FX3 communication setup. This is the only test setup not using PCIe.

PCIe: Measures the performance of Xillybus PCIe component of the communication board design.

PCIe 64: Measures the increase in performance of the Xillybus PCIe component when 64-bit buses are used internally instead of 32-bit buses. The larger 64-bit bus forces the smallest transfer size to 64-bits but provides additional performance.

PCIe with FX3 Emulator (PCIe GPIF): Designed to measure the performance of the General Programming Interface (GPIF) used with the FX3 implementation.

Aurora x1: Aurora with one lane of communication.

Aurora x2: Aurora with two lanes of communication. The number of lanes equal the number of high speed transceivers used for the connection.

Aurora x1 Stop-and-Wait ARQ: One lane Aurora with a Stop-and-Wait ARQ. One packet is sent and acknowledged before the next packet is sent. This shows the need for a more complex acknowledgement protocol to guarantee packet delivery.

Aurora x1 Go-Back- N ARQ: One lane Aurora with a Go-Back- N ARQ. Go-Back- N has sufficient performance when the error rate for the data path is low.

Aurora x2 Go-Back- N ARQ: Two lanes Aurora with a Go-Back- N ARQ.

All the tests were conducted from a host system consisting of an Asus P10S-M micro ATX motherboard, an Intel Xeon e3-1275 processor, and 32 GB of DDR4-3333 memory. The computer was running Ubuntu 16.04.2 LTS. Because of a buffer flushing problem with the Xillybus driver packaged with Ubuntu 16.04, the newest Xillybus driver needs to be downloaded and installed. Xillybus Revision B was used as it performs better than Revision A and is a drop-in replacement for Revision A. The FX3 test setup used a Cypress FX3 board (FX3) and a HiTech Global HTG-777 with a Xilinx Virtex7 X690T (690T). The remaining tests all used a Virtex7 Xilinx VC707 evaluation board (VC707) for the communication board. The PCIe GPIF test setup and the Aurora test setups all communicated with a 690T that acted as the neuromorphic array board. The Aurora communication channel used a 156.25 MHz reference clock and operated at a line rate of 6.25 Gbps. The channel also proved to have a very low error rate, with no observed CRC errors or packet errors in a 1.5 TB round trip transfer. The Aurora communication logic uses less than 1% of the FPGA's resources, allowing the vast majority of the FPGA's resources to be used for the neuromorphic array.

The two main metrics measured are round trip latency and round trip throughput. There are two main variables when performing the benchmarks. 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 transferred before the user program has to be involved. This variable is called the transfer size.

The other main variable is the total amount of data that is transferred. This total amount is transferred one transfer size at a time until the total amount is reached. The user program needs to be reentered to make the next transfer call when the total size is bigger than the transfer size. Thus, 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.

Latency Benchmarks The first set of benchmarks are aimed at measuring the latency of one round trip transfer of a 64-byte 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 for all the test setups and the results of the benchmark can be found in [Figure 7.11](#). The FX3 test setup had by far the highest round trip latency, with a latency value of 80.38 μs . All the other test setups have a round trip latency of around 6 μs . The measurements obtained had a low variance, with the standard deviation from the FX3 benchmark being 2 μs and from 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 [29]. The much lower values of the PCIe benchmarks is also logical. Since 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. Taking into account the standard deviation values, the measurements for PCIe are all roughly the same. Although the relatively large latency for the PCIe interface hides the latency of Aurora, the theoretical latency for both the GPIF and Aurora can be calculated. The GPIF latency is calculated as shown in (7.1).

$$\text{GPIF latency} = \text{clock freq.} \times (\text{data cycles} + \text{overhead}) \quad (7.1)$$

Assuming the overhead is around 10 cycles, then the round trip transfer time is 0.4 μs , as calculated in (7.2).

$$\text{round trip time} = 100 \text{ MHz} \times ((16 \times 2) + 10) = 0.4 \mu\text{s} \quad (7.2)$$

The theoretical increase of 0.4 μs is hidden in the experimental results, since the GPIF transfer can start while the PCIe transfer is still in progress.

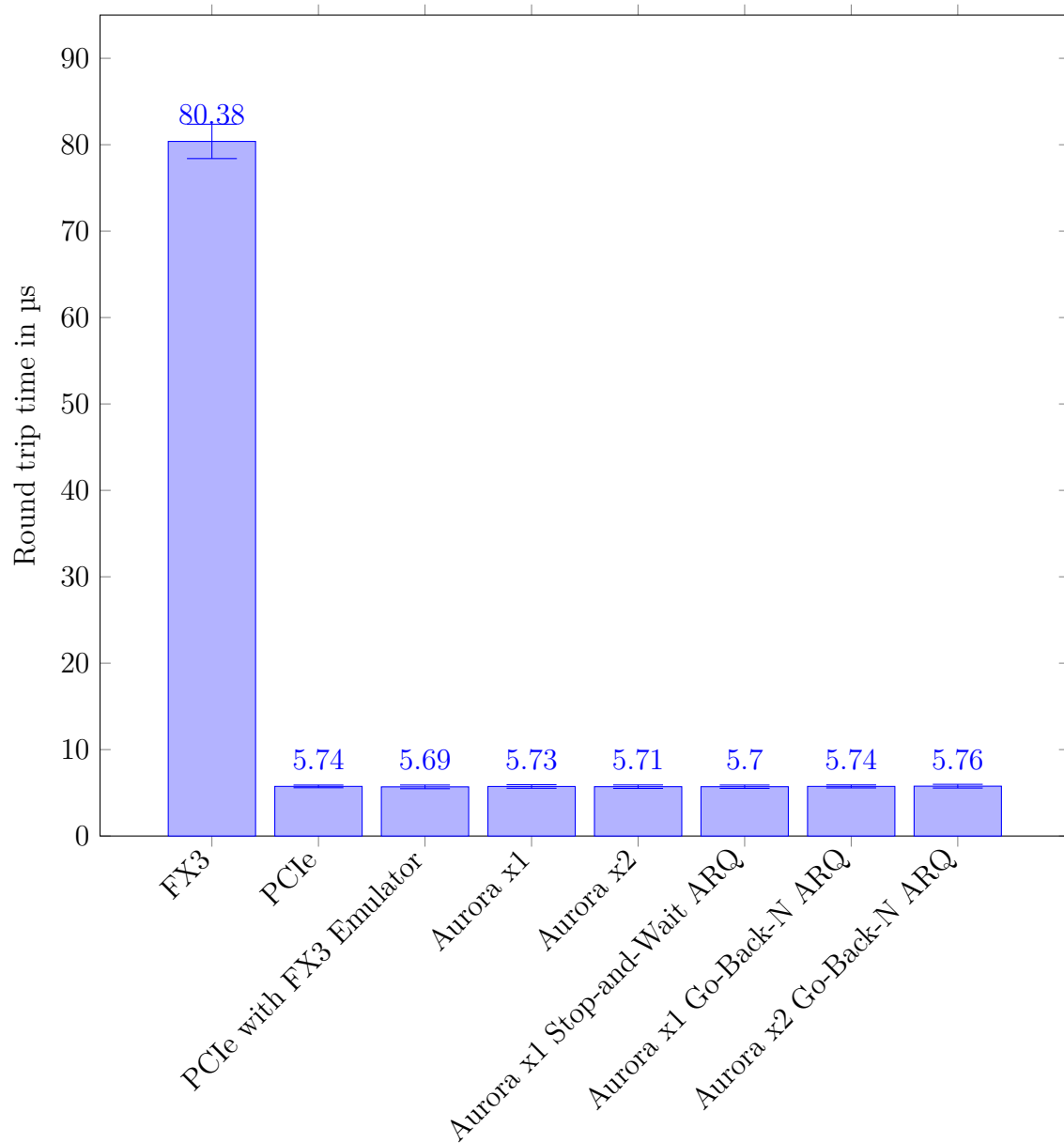


Figure 7.11: Round trip time comparison.

A theoretical calculation for Aurora can similarly be made. The Aurora latency can be calculated as shown in (7.3), which results in a theoretical latency of 0.1638 μ s.

$$\frac{\text{bits to transfer}}{\text{transfer rate}} = \frac{64 \times 16}{6.25 \text{ Gbps}} = 0.1638 \mu\text{s} \quad (7.3)$$

The latency is hidden by the PCIe latency since the Aurora transfer can start while the PCIe transfer is still taking place.

Throughput Benchmarks The second set of benchmarks are aimed at measuring the maximum throughput of each design. Figure 7.12 shows the 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 [29]. The GPIF's maximum throughput is shown by the PCIe with FX3 emulator line. According to "Optimizing USB 3.0 Throughput with EZ-USB" [29], the maximum 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 maximum theoretical throughput of the GPIF interface is $32 \text{ bits} \times 100 \text{ MHz} = 400 \text{ MB/s}$ 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 maximum throughput of the PCIe test setup is 917 MB/s. Since this maximum is much greater than the Aurora x1 or PCIe with FX3 emulator tests, it can be inferred that PCIe was not the bottleneck in these tests. The PCIe test shows the upper throughput limit with the PCIe implementation used in the communication board. If more bandwidth is needed, 64-bit streams can be used. The maximum throughput of PCIe 64 is 1511 MB/s. In order to reach the maximum throughput, a transfer size of 1K and 2K is needed for PCIe and PCIe 64, respectively. All the implementations have the same rate of change in the beginning

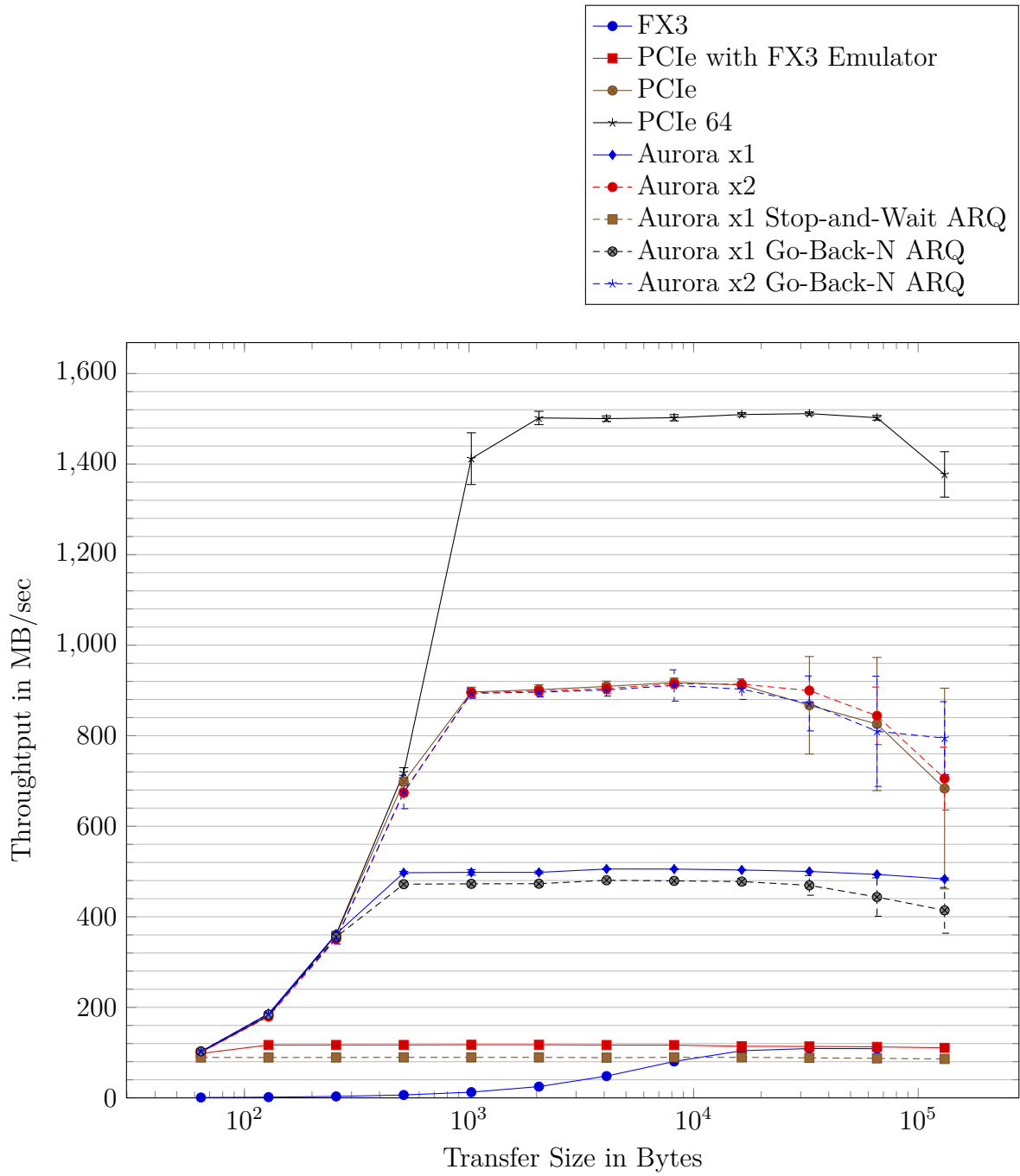


Figure 7.12: Throughput experiment, varying the amount of data transmitted per function call.

region before the maximum is achieved. 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 maximum throughput of 505 MB/s. Moving from one lane to two lanes roughly doubles the maximum throughput. The test is now limited by the 32-bit bus PCIe implementation, which results in a throughput of 914 MB/s. Aurora should continue to double in performance as the number of lanes is doubled. 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 can 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 addition of a Go-Back- N ARQ only added a slight decrease in bandwidth with a maximum throughput of 480 MB/s for one-lane of Aurora. This is only a 5% decrease in throughput caused by the overhead of adding packet numbers and sending acknowledgments. This shows that the addition of the lightweight retransmission protocol proved to have minimal overhead due to the hardware implementation and the lightweight nature of the protocol.

From this graph, many helpful conclusions can be made. First, the new communication board has room to scale. If the single lane Aurora limit is reached, 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, Xillybus Revision XL, which offers a maximum throughput of 3500 MB/s with a 128-bit internal data width, can be used [118]. The new communication setup can scale far beyond the previous communication limits of the FX3 and GPIF interface. Second, the maximum throughput is only reached when large blocks are transferred at a time, with the sweet spot seeming to be 1 KB of data.

Additional tests, which varied the total transfer size and kept the transfer size fixed to a single 64-byte packet, showed that performance is highly dependent on the buffer size used to call the transfer function and not on the total amount of data being transferred. This means that the best performance is achieved by buffering multiple packets together and making large transfer calls to the Xillybus driver.

7.3.2 Local Communications Between Sub-arrays

The different sub-arrays are designed to operate asynchronously to each other following a Globally Asynchronous, Locally Synchronous (GALS) design pattern. This design pattern was chosen in order to enable larger scaling than would be possible if the entire system had to operate under a single, global, synchronous clock. Using multiple asynchronous clock domains also comes with the added benefit of being able to use different clock domains for transferring data than for running the neuromorphic elements. The communications can run on faster clocks than the neuromorphic elements and use high-speed serial communication to transfer information. With a synchronous system, the communication bus would have to be very wide to run at the same frequency as the element clock, or multiple synchronized clocks would have to be used, which are aligned such that the data arrives at the correct cycle of the element clock. Using synchronous clocks would limit the total scaling potential of the system and make the design much more difficult since multiple synchronous clock systems would have to be used, just like the Tiled DANNA project. As Martin et al. noted, future SoCs and large-scale designs will no longer be able to operate under a single clock because the variations across a large chip or multiple large chips make it prohibitively expensive to attempt to manage the delays in a clock and other global signals [57]. My solution to this problem for SNACC, is to make use of multiple Globally Asynchronous, Locally Synchronous clock domains. Each individual neuromorphic array will operate under their own local clock domain. For performance reasons, these clocks should be the same frequency, although the system will still operate if the frequencies differ. Each point-to-point communication channel will also operate on its own local clock domain. The communication clocks will typically run at higher frequencies than the element clocks, in order to lower communication latency and raise communication bandwidth. The communication board will also operate under its own clock, and the host will, of course, also operate under its own local clock.

There are a couple of important considerations with using a GALS design. First, the timing of events are no longer guaranteed. One does not know for certain when a packet or signal in one clock domain will arrive relative to signals within another clock domain. This fact alone could causes issues with deterministic behavior and with processing real-time

signals. In order to ensure deterministic behavior, the operation of all the neuromorphic arrays is based on waiting until all the information needed to evaluate the next timestep of the simulation is obtained before executing the next timestep. This approach causes the sub-arrays to pause as they wait for the necessary packets from the host, or their neighbors. Once the needed packets arrive, the sub-array's time advances to the next timestep. This approach decouples wall-time from simulation-time and makes the hardware operate more like a distributed event simulator. As the previous events arrive, the next event is computed. This data-driven operation guarantees that the logical operation of the array is deterministic. In other words, if the hardware is configured with the same network configuration and receives the same input, the output will always be the same. Furthermore, this output will always match the output from the software-based event simulator.

The first obstacle with a GALS approach is thus resolved, but the issue of processing real-world signals or meeting hard evaluation deadlines still exists. Although the logical behavior is guaranteed to be deterministic, the time necessary for the output to be available after the input is supplied is not guaranteed. The actual time the computation takes depends on many factors, including but not limited to the relative timing of each of the asynchronous components, whether any packets are corrupted and need to be resent, and also whether the Aurora channel is performing channel maintenance. Although a hard guarantee is not possible, it is possible to provide a soft timing guarantee. The actual computation delay is a function of many different probabilities. The actual computation times can be measured and the timing can be represented as a probability of when the result will be available. Since DANNA2 operates at a frequency much faster than biological neurons, the result should always be available faster than these neurons would operate. Because of this, the entire array could be slowed down by only sending in input from the host at a fixed, real-time frequency. As long as this frequency is sufficiently slower than the average unconstrained operating frequency, the likelihood that the result will be available within the slower frequency is extremely high. For example, say that the DANNA2 arrays are able to run on average at a frequency of 9 MHz. Then if input is supplied every 1 MHz, the likelihood that the network will be able to run each network cycle within this 1 MHz window is exceptionally high. Therefore the system is able to make the soft, real-time guarantee of running at 1 MHz.

Since this is only a soft, real-time guarantee, it is of course possible that a failure or sequence of highly unlikely events will result in the timing deadline not being met. In this case, the host can be notified that the timing was not met and make a decision accordingly. This form of handling real-time processing with asynchronous circuits is very similar to the policy used by TrueNorth, as discussed in [Section 2.4.6](#). One difference between TrueNorth’s policy and this one is that the behavior of SNACC is guaranteed regardless of whether the soft, real-time guarantee is kept; only the run time is not guaranteed. TrueNorth, however, will always continue evaluating at a fixed rate, and if the packets fail to arrive on time, TrueNorth’s output will differ but not the timing of when the outputs arrive. For SNACC, the decision was to prioritize the determinism of the logical behavior of the system, at the expense of having a hard timing guarantee for when the output will arrive. This decision makes SNACC similar to Loihi (also discussed in [Section 2.4.6](#)), where the system will compute faster than real-time, based on when the packets arrive, but does not guarantee when the computation will be finished.

In order to flesh out and verify the sub-array communication of SNACC, two projects were started. The first project is a SNACC software simulator, which simulates and visualizes the communication packets being sent around the system. This simulator is used to test different scenarios and verify that an asynchronous design would be able to exhibit good performance. The second project consists of making a fake DANNA2 core, which can be used to test the communication of the system. The fake core sends full-sized packets, keeps track of the current timestep, and waits the correct amount of time before moving to the next timestep. Both of these projects were used to verify the correct functionality of the SNACC communication system before the DANNA2 Neuromorphic core was ever added to the system.

SNACC Simulator

The SNACC simulator is designed to simulate the asynchronous communication patterns of the SNACC system using software. The simulator is an event-based simulator which evaluates events with a one nanosecond time resolution. The SNACC simulator was designed with the

help of Adam Foshie, as his undergraduate research project, and includes both a command line program to simulate different scenarios and a GUI to visualize the transfer of the packets.

The simulator uses a logical view of SNACC to simulate the timing of the communication system and operates on a grid of DANNA2 neuromorphic sub-array processors. (These DANNA2 objects are functionally equivalent to the fake DANNA2 components discussed later.) [Figure 7.13](#) shows the simplified representation of the ports available on a single chip. The chip has an input and output port for communicating with the host, as well as ports for sending and receiving packets to the north, south, east, and west neighboring chips. Each element also has a sub-array clock, which is used to simulate the DANNA2 element clock that is asynchronous to the communication logic. During every cycle of the element clock, the chip checks to see if it has the packets needed to execute the next cycle. In order to run the next cycle, it must have information about the current cycle from the host and each of its neighbors. The element takes ten cycles of the element clock to emulate the time taken to evaluate the element's output for the next timestep. After this, the element sends an output packet to each of the neighbors and to the host. The element represents all the synchronous elements on the chip, and the packets contain all the relevant information for the elements connected across the chip. In order to compute the next timestep $t + 1$, all of the information from t must be collected. This would require one packet to be sent to each neighbor every cycle. The latency of communication between chips is a performance issue. Although the channel can support sending a new packet each cycle, the latency of the packet being sent through the channel takes multiple cycles. This is the main reason that the performance of this asynchronous design is less than the performance of the individual chips. The chips have to sit idle while the next packet is in transit.

This performance issue can be alleviated if there is an inherent delay as part of sending the packet over the channel. For example, if there is an inherent delay of one, the packet can have a latency of two cycles and still arrive on time. The number of cycles available for the transmission is one plus the added delay. The general concept of how the sub-array synchronization works can be viewed as a credit system. Initially each device is given $\text{credits} = 1 + \text{delay cycles}$. In the simple case of no extra delay cycles, this number is one. The credits represent the information needed to run to the next cycle. For the first cycle

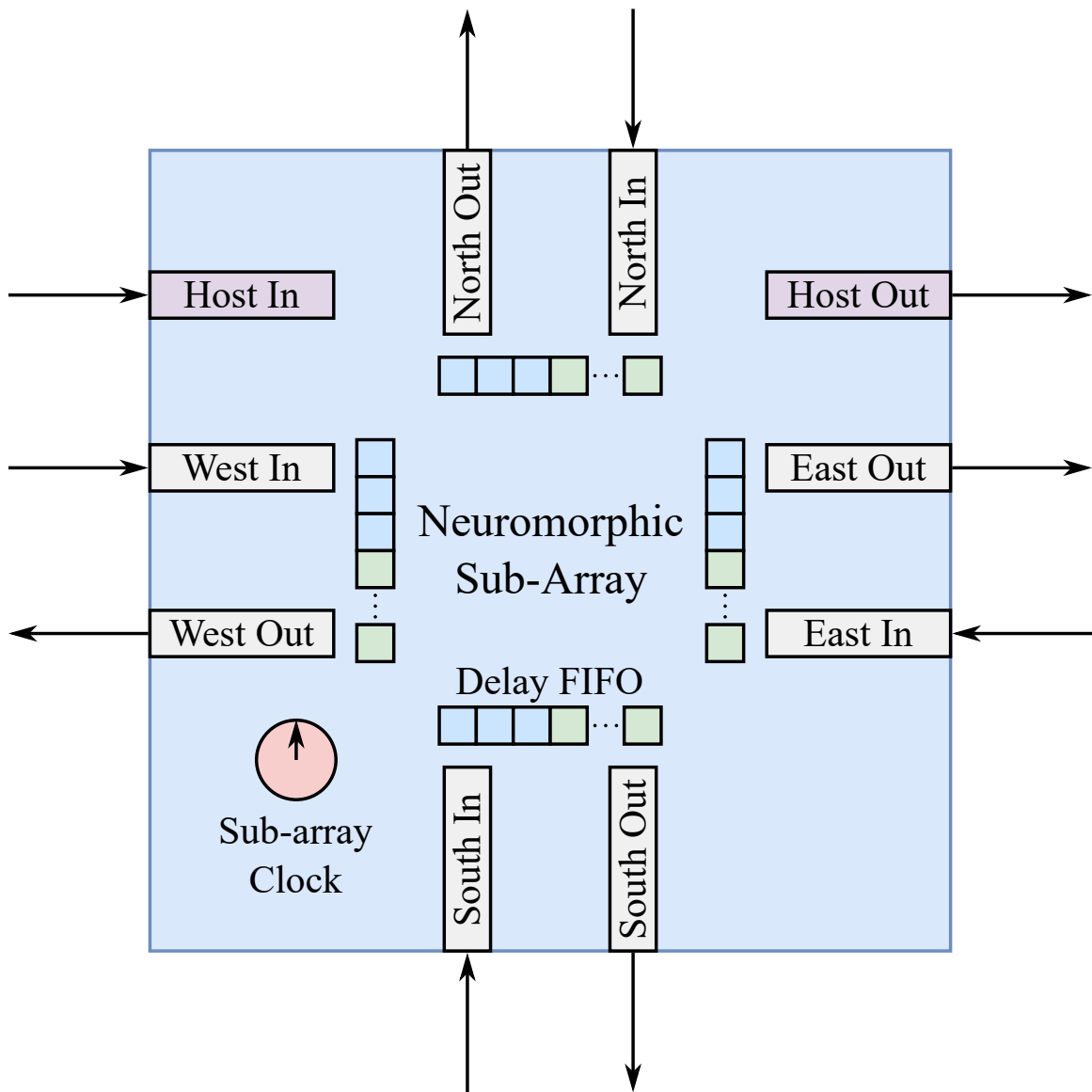


Figure 7.13: Logical view of a simulated fake DANNA2 chip.

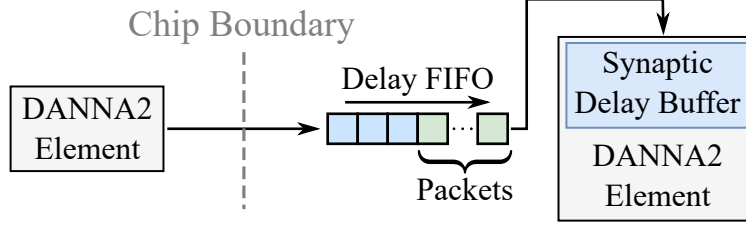


Figure 7.14: Delay FIFO is used to move the synaptic delay from inside the element to make it part of the communications channel, thereby hiding the communication channel’s latency.

$t = 0$, the information needed to run is from cycle $t = -1$. Since this cycle is before the simulation began, each chip starts out with one credit. When the element has a credit from each of its neighbors and fire information from the host, the chip can evaluate the next cycle. Once the evaluation is complete, it sends a credit to each of its neighbors. When a credit arrives at a chip, it is stored in a delay FIFO until it is needed. The number of packets the delay FIFO needs to be able to hold is $2 \times \text{credits}$. This can be easily proven by assuming one chip has sent all of its credits to its neighbor; once all the credits arrive, the neighbor will have its starting credits and its neighbors starting credits.

Another way to view this “credit system” solution is as a way to move the synaptic delay from the element’s synaptic delay buffer to the communication FIFO. If a minimum synaptic delay is enforced, then the communication FIFO can use the delay to hide the sub-array communication latency. The synaptic delay buffer then only adds the delay that is over the minimum delay. [Figure 7.14](#) shows how the delay computation is moved from the synaptic delay to the communications channel, where it can be used to hide the communication latency.

This setup is simple to implement since the inherent delay is a function of the starting number of credits. The elements do not need to know the delay value beyond the starting condition; the element just needs to wait until it has a packet from each of its neighbors before it runs to the next cycle. In the initial state, the chip will send empty packets equal to the number of starting credits. The empty packets represent the fire activity before the simulation started. Once the system is running, the fire packets from each neighbor are applied to the array and the array is advanced one cycle, after which the output from this

cycle is sent to its neighbors. [Figure 7.15](#) shows a diagram of the complete SNACC simulated array. The tiling system is designed so that it can extend to any arbitrary size.

The simulator works by simulating events. Each event occurs at a time as measured from the beginning of the simulation. The time of the simulation is measured in nanoseconds and the resolution of the simulator is one nanosecond. The event simulator works by scheduling each event in an array sorted by the time the event occurs. Then the simulator simulates one nanosecond at a time, and processes each event that occurs at the current time before moving to the next nanosecond. Each chip operates asynchronously from each other and each of their clocks is simulated to occur at different times. The clocks are allowed to be out of phase and also have period drift. Various parameters are able to be changed to customize the behaviour of the simulation. [Table 7.1](#) shows the customizable parameters for the SNACC simulator. As the table shows, the different clock frequencies, error probability, and communication latency can be adjusted to see how they affect the simulation. The input files contain information about when the host releases the next simulate packet into the array. The files have a simple format, with each line containing the time the input is available on the host in ns and the timestep the data should be sent. To make creating these files easier, an input file generator script is also included. This script creates six main types of input files:

- The bottleneck is not the host (NoHostBneck).
- The bottleneck is always the host (ConstBneck).
- The host activity is high at the start and end, but gradually ramps down toward the middle (i.e. The host is the bottleneck at beginning and end.) (BneckRampDown-Up).
- The host activity is high at the start only (i.e. The host is the bottleneck at the beginning but gets less active towards the end.) (HostBneckRampDown).
- The host activity is high for the first half of simulation then drops to no activity for the second half. (BneckStep).
- The host activity is stochastically generated. (Stochastic).

The SNACC simulator outputs various statistics about the run once the simulation is finished. [Figure 7.16](#) shows the statistics that the simulator outputs. The SNACC simulator

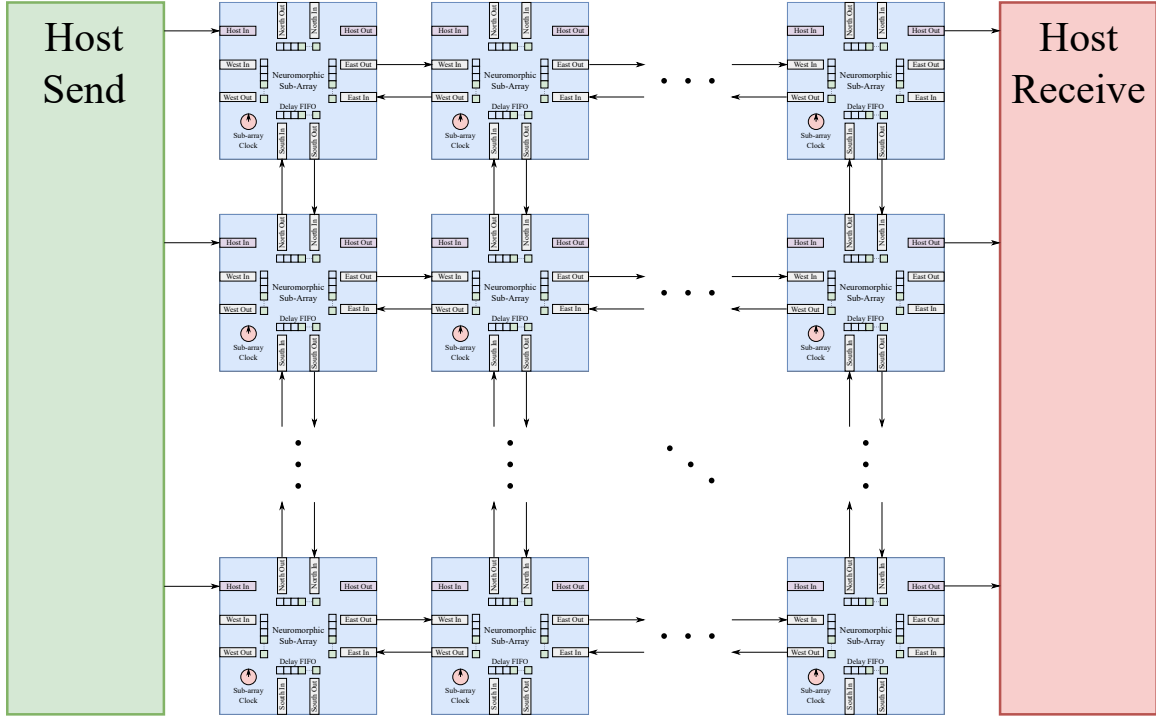


Figure 7.15: Logical view of a simulated SNACC.

```

DANNA[0][0] Final Cycle: 10001
DANNA[0][1] Final Cycle: 10002
DANNA[0][2] Final Cycle: 10002
DANNA[1][0] Final Cycle: 10000
DANNA[1][1] Final Cycle: 10001
DANNA[1][2] Final Cycle: 10001
DANNA[2][0] Final Cycle: 10000
DANNA[2][1] Final Cycle: 10000
DANNA[2][2] Final Cycle: 10001

```

```

Minimum Effective Clock Period = 575 ns
Maximum Effective Clock Period = 5070 ns
AVG Effective Clock Period = 2245.95 ns
Standard Deviation = 1281.47 ns
AVG Effective Clock Frequency = 0.445245 MHz

```

Figure 7.16: SNACC simulator post-simulate statistics.

Table 7.1: Parameters for the SNACC simulator

Parameter	Default	Description
hw	FALSE	Run the test using fake DANNA2 hardware.
columns	2	The number of columns in the grid of DANNA2 chips.
rows	2	The number of rows in the grid of DANNA2 chips.
output_clk	5	The clock period of the host in ns; used to check for output.
base_clk	10	The base clock period for the DANNA2 chips in ns.
slow_clk_mult	10	The factor to multiply the base clock period by to produce the slow clock period.
clk_policy	0	Clock policy for DANNA2 chips. 0 Sets the clock policy to 'fast' which uses the base clock. 1 sets the clock policy to 'slow' which uses (base clock) * (danna_slow_clk_mult).
clock_offset_max	2	Maximum number of nanoseconds each DANNA2 chip's base clock can be offset from each other.
clk_period_drift	0	Enable or disable clock period drift between chips. Allows period to be 9, 10, or 11 ns.
delay_cycles	3	Number of delay cycles to use to hide communication latency. Allows chips to fire without host input for specified number of cycles.
comm_latency_mean	486	Mean value for communication latency in ns. Latency is given a Gaussian distribution when fires are sent.
comm_latency_stddev	5	Standard deviation for communication latency in ns.
error_prob	0.0000002	Probability that communication will fail and another latency time will be added for retransmission.
row_input_delay	2000	Delay between when the host can write to different rows.
host_input	NoHostBneck_10000.txt	Name of file to use for host activity input.
host_fifo_max	5	Maximum FIFO size on the host.
window_size	10	Number of cycles included in window for calculating sliding window average.
seed	-1	Seed for a simulation run. Default is -1 = time(0).
verbose	0	Verbose options. 0: prints stats only. 1: + prints generated offsets/periods. 2: + prints sliding window average period/frequency. 3: + prints fire events with time stamp.
enable_viz	FALSE	Show the visualizer with the simulation.

also has a visualizer which can be used to watch how packets are sent between the various chips. A labeled screenshot of the visualizer is shown in [Figure 7.17](#). The visualizer is animated so the progression of SNACC is clear. The speed of the visualizer can be adjusted so that the details of the simulation can be seen.

There are a few interesting observations found from viewing how SNACC operates with the visualizer. First, since the inputs arrive on the left side, the timesteps form a gradient and update in waves. When the host is at time t , the first column is at $t + 1$, the second column is at $t + 2$, and the output is at $t + \text{column_count}$. [Figure 7.18](#) shows this gradient state. When the next packet arrives from the host, the columns update their timesteps one at a time. Notice, however, that since the later columns are able to run into the future, the output could be ready before the input for that cycle is provided. In this case the output is at $t + \text{column_count}$ when the input is at t . This behaviour hides the startup latency when the array receives new input.

The second observation is that there are three main bottleneck states. The first state was just discussed, and it is when the array is waiting on the host. The second state is when the array is waiting on neighboring packets. The third state is when the system is bottlenecked by the time it takes to process the next time step. The first state is host bound, the second state is communication latency bound, and the third state is neuromorphic array bound. In practice, the first two states determine the effective frequency of the array when the effective frequency is less than the ideal maximum frequency. The effective frequency is the frequency of the average network cycle. When the host and the neighbor communication are not limiting the effective frequency, the effective frequency is equal to $\frac{\text{element frequency}}{\text{number of cycles per network cycle}}$. This frequency is the maximum ideal frequency for SNACC, as it is only bound by the performance of the neuromorphic array.

The SNACC Simulator verified that an asynchronous, credit-synchronized, sub-array communication system would function correctly, and that such a system can have great performance. The performance is almost that of a single chip, as long as the number of additional delay cycles is sufficient to hide the latency of the communication channel between chips. This result was encouraging, as further testing was concurrently being conducted on

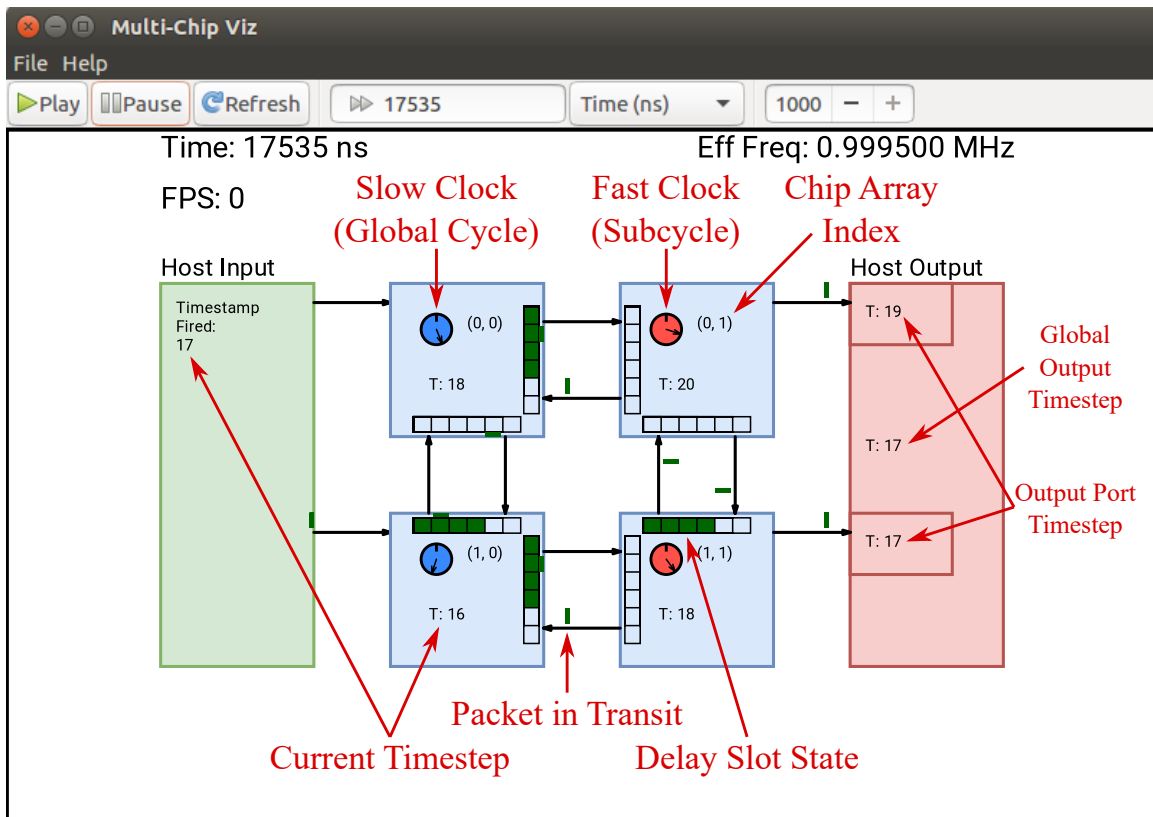


Figure 7.17: SNACC Simulator visualization.

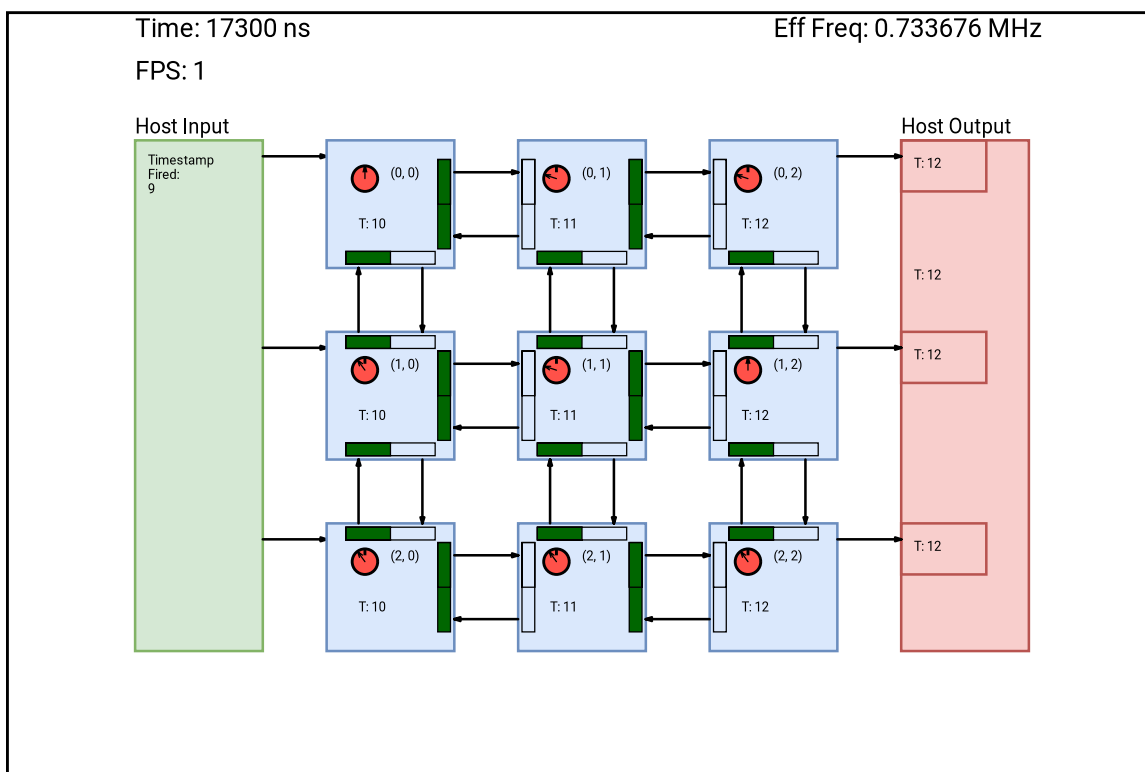


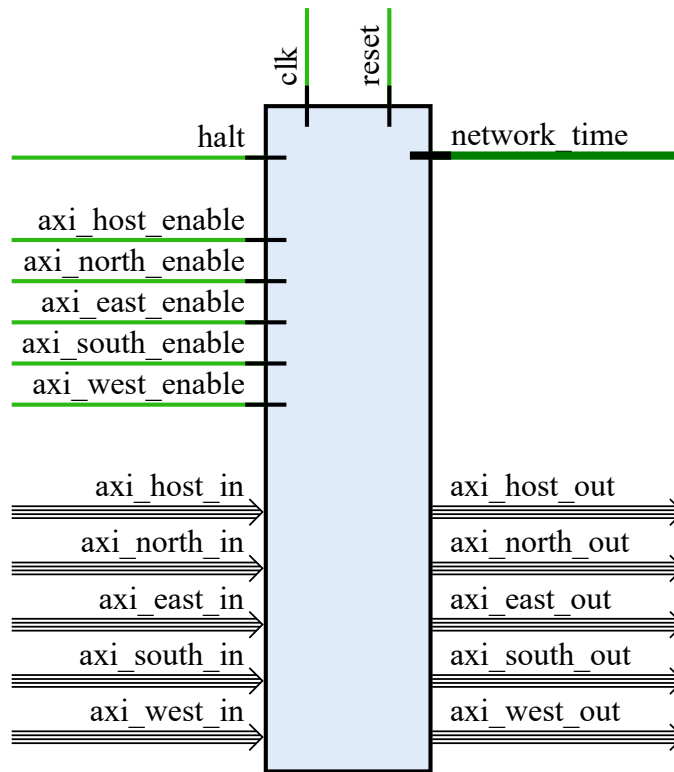
Figure 7.18: Host bottlenecked array state.

hardware using a fake DANNA2 core, which essentially implements the logic of the simulated fake DANNA2 chips.

Fake DANNA2

The next step of designing SNACC was to build the communication system that was prototyped in the SNACC simulator. This first design of SNACC uses a “fake” DANNA2 core for testing. The fake core keeps track of its current timestep, sends and receives full-size packets, and generates and accepts packets at the correct rate. All of the neuromorphic logic is left out to make the testing of the communication system easier. The fake DANNA2 multi-array has the same ports as the real DANNA2 multi-array. [Figure 7.19](#) shows the top-level interface for both the fake DANNA2 multi-array and the real DANNA2 multi-array. One major goal of the sub-array design is to be able to use the same design for each of the sub-array systems. Because of this, the DANNA2 multi-array interface is designed to be the same for each instance. Therefore, each sub-array has an enable bit to turn on and off the different AXI4-Stream connections. The board in the top left of the array will only have the south and east AXI4-Stream ports enabled. The north, south, east, and west enable ports turn on and off the receiving and sending of packets. The host enable port only turns on and off the sending of packets. All of the sub-array components have to receive input from the host for the configuration and running of the arrays; however, only the last column will have fire output to send back. These enable signals allow all of the neuromorphic arrays to have the same design. Switches on the board are used to enable or disable each of these ports based on where the array is in the design. The network time output is a debugging output and is used with debugging probes to verify the correct operation of the array.

The fake DANNA2 multi-array architecture is a finite-state machine (FSM) with two states, wait for input and run. In the wait for input state, fake DANNA2 waits for packets from the host and from all enabled neighbors. Once it has the required input, the packets are consumed, the network time is incremented, and the FSM goes into the run state. In the run state, the FSM counts the correct number of sub-cycles, then broadcasts an output packet to all enabled ports with the new network time. Then the FSM goes back in to the waiting for input state.



DANNA2 Multi-Array

Figure 7.19: Top-level interface for the DANNA2 multi-array. The wide arrows represent AXI4-Stream buses.

The top-level design for the board connects the multi-array DANNA2 to communication channels, which send the data from each AXI4-Stream bus out an Aurora channel with the custom ACK logic. [Figure 7.20](#) shows the design for the multi-array DANNA2 board. The different port-enable lines are directly mapped out to a Dual In-line Package (DIP) switch. This makes it easy to use the same design for each of the sub-arrays and to change the DIP switch to enable the correct ports, based on the location of the sub-array in the large-array. Each input channel consists of an Aurora input side, ACK input side, and packet FIFO input components. Each output channel consists of a packet FIFO output, an ACK output side, and Aurora output side components. The input and output sides of ACK and Aurora are part of the same component. They are shown as separate boxes in the diagram for clarity. The sub-array communication channels also have an additional preload component on the input side. The preload component is designed to output a number of no-op packets equal to the number of credits required for a certain delay between the boundaries of the boards. As discussed previously, the number of credits is one plus value of delay added between the boundaries. The preload component is a simple FSM with three states: Init, Load, and Run. Initially the state machine is in the Init state; in this state, the load count signal is assigned the value of delay plus one. If this value is equal to zero, the state machine moves to the run state. If the value is not zero (the more likely case), the FSM moves to the load state. In the load state, the FSM keeps outputting no-op packets and decrementing load count until the load count signal is zero, at which point the FSM changes to the run state. When preload outputs no-op packets, it makes sure the handshake is complete before moving on to the next packet. In the run state, the preload component directly passes the AXI4-Stream signals through. While preload is not in the run state, it is not ready to receive packets from the AXI4-Stream input.

Initially, the packet FIFO in and packet FIFO out components were implemented with a standard Vivado AXI4-Stream compatible FIFO. This proved, however, to be a major performance bottleneck, since any component that lies in this communication channel is on the system-level critical path for chip-to-chip communication performance. Additionally, any savings in latency for the packet FIFOs is a times four improvement, since the packet FIFO appears four times along the critical sub-array communication path from one neuromorphic

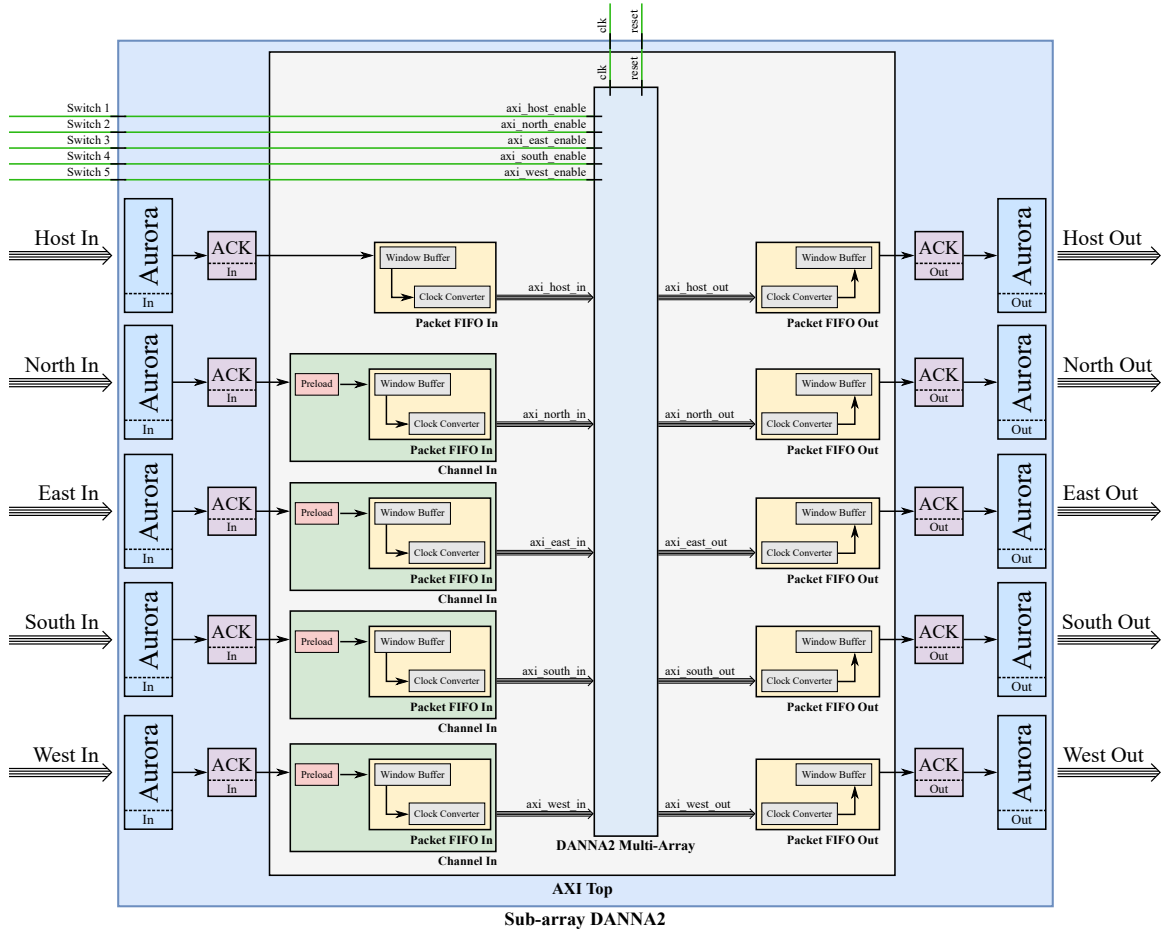


Figure 7.20: Top-level design for multi-array DANNA2.

core to another. This critical path from one core to the other is

$$\begin{aligned} \text{DANNA2} &\rightarrow \text{Packet FIFO} \rightarrow \text{ACK Out} \rightarrow \text{Aurora Out} \\ &\rightarrow \text{Aurora In} \rightarrow \text{ACK In} \rightarrow \text{Packet FIFO} \rightarrow \text{DANNA2}. \end{aligned} \quad (7.4)$$

Therefore, the packet FIFO was replaced by the custom window buffer detailed earlier, and a custom clock converter. This design proved to have lower latency than using a standard two-clock FIFO or using Vivado's AXI4-Stream clock crossing IP block. This design also has the added benefit of requiring no IP blocks other than the Aurora IP block. The design is thus much easier to port to a different FPGA from another manufacturer or to a VLSI workflow. The new critical path is now

$$\begin{aligned} \text{DANNA2} &\rightarrow \text{Clock Converter} \rightarrow \text{Window Buffer} \rightarrow \text{ACK Out} \rightarrow \text{Aurora Out} \\ &\rightarrow \text{Aurora In} \rightarrow \text{ACK In} \rightarrow \text{Window Buffer} \rightarrow \text{Clock Converter} \rightarrow \text{DANNA2}. \end{aligned} \quad (7.5)$$

The custom clock converter is designed to convert an AXI4-Stream from one clock domain into another clock domain. The design of the AXI4-Stream clock converter is shown in [Figure 7.22](#). The clock converter converts a slave AXI4-Stream bus from the s clock domain to a master AXI4-Stream bus clocked in the m clock domain. The design is based on a bi-direction toggle synchronizer with a data line. One direction of the toggle synchronizer is to synchronize the TVALID line. The other direction is to synchronize the completion of the AXI4-Stream handshake. The TDATA line is latched with TVALID, so that it is held stable during the handshake. The AXI4-Stream handshake is setup so that if the master port is ready, the handshake will be completed the same cycle as the toggle. In order to build this design, a custom S-State, Synchronizer, and Set/Clear TFF components were created. The other components in the design are standard flip-flops and logic gates.

S-State is the FSM designed to handle the toggle synchronization in the s clock domain. [Figure 7.21](#) shows the design of this state machine. The FSM has two states, idle and send. The FSM starts in the idle state. In this state, TREADY is held high and it waits for the AXI4-Stream bus to raise TVALID. Once TVALID is asserted, TREADY is lowered and the

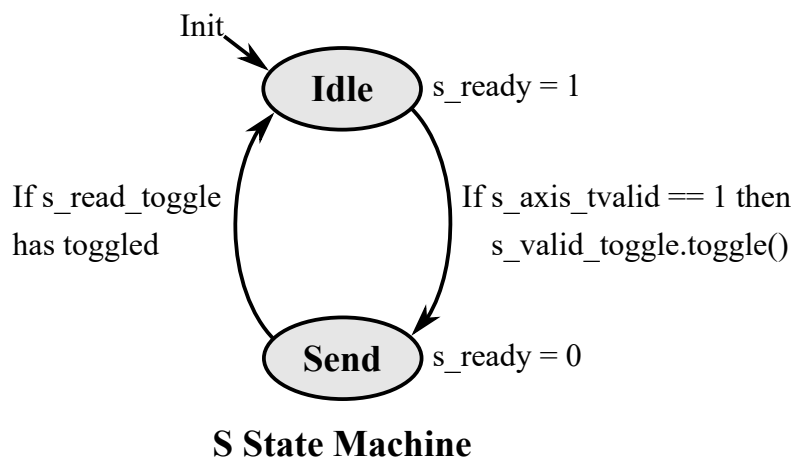
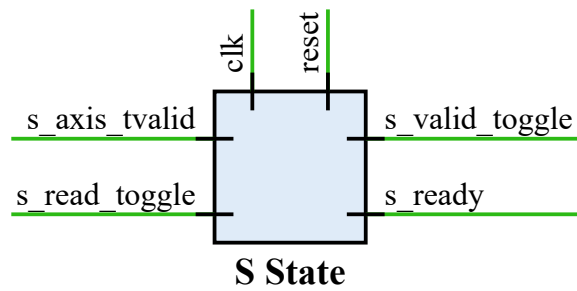


Figure 7.21: S Clock Domain State Machine used to implement the AXI4-Stream Clock Converter.

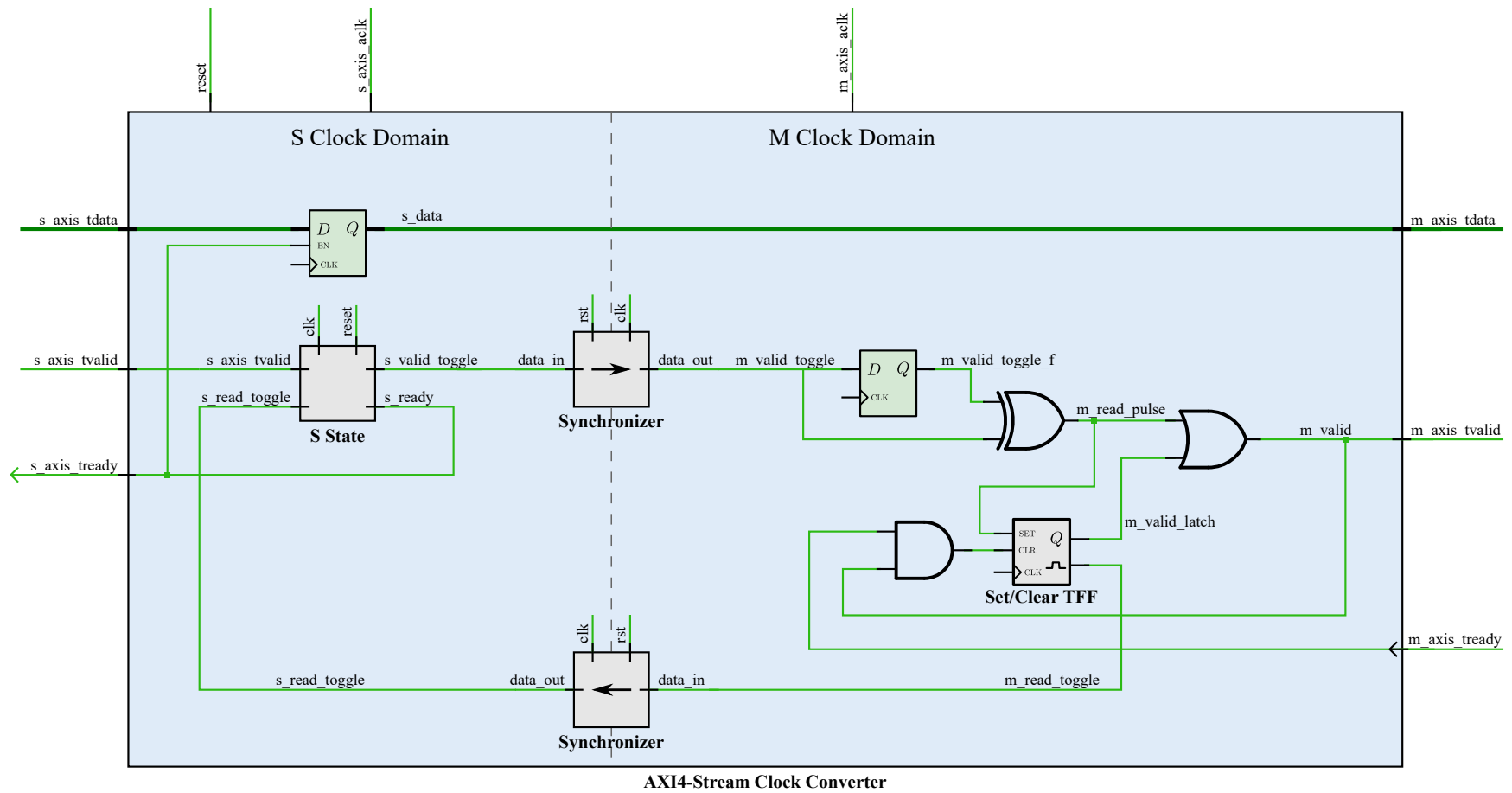


Figure 7.22: Top-level design for the AXI4-Stream Clock Converter.

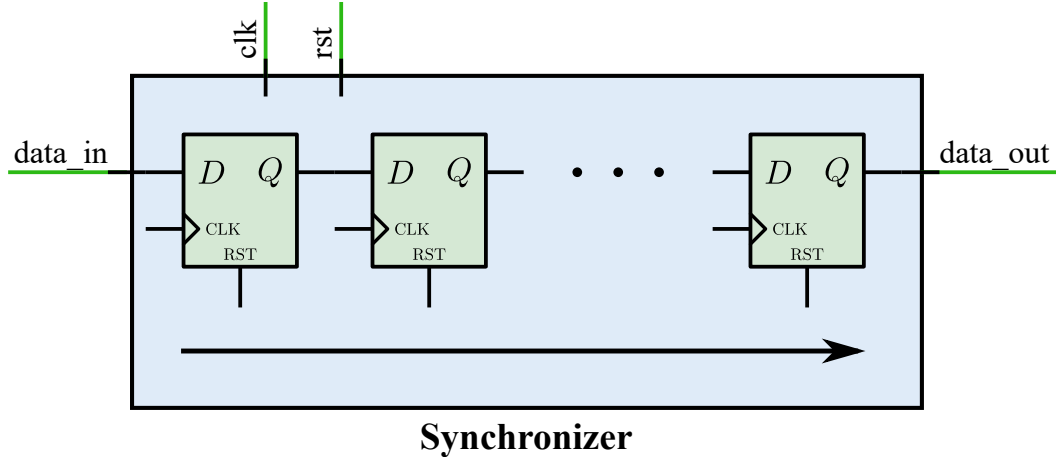


Figure 7.23: Single-bit n -stage synchronizer used in the AXI4-Stream Clock Converter. Each flip-flop is one stage of the synchronizer.

`s_valid_toggle` line is toggled. Once a toggle is received on the read toggle line, the FSM goes back to the idle state, indicating that the bus is ready for another packet.

The synchronizer component is an n -stage shift-register, which is clocked by the target domain's clock. The component is written generically so any number of stages can be used; however, two stages are usually sufficient and are used by default in the clock converter design. The design for the synchronizer is shown in [Figure 7.23](#).

The set-and-clear Toggle Flip-Flop (TFF) design handles the toggle synchronization in the m clock domain. This component has a set and clear port, with clear having the priority when both signals are high. When clear is high, Q is set low and a toggle is sent out the toggle port ($\neg$). When clear is low and set is high, Q is raised high. When neither set or clear are asserted, Q keeps its previous value and $\neg$ does not change. The set-and-clear TFF is shown in [Figure 7.24](#). The design is implemented with a set-and-clear flip-flop and a D-flip-flop.

The fake DANNA2 design proved to be a great test design; it was used to work out any problems with the communication design without having the additional complexity of the neuromorphic core. The next step in the design process was to replace the fake DANNA2 core with a real DANNA2 core.

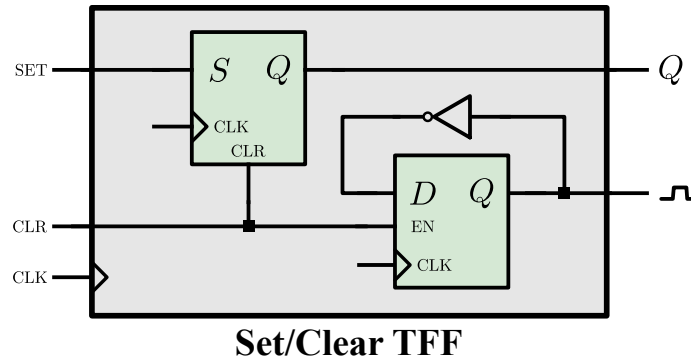


Figure 7.24: Set and Clear Toggle Flip-Flop (TFF) design used in the AXI4-Stream Clock Converter.

Real DANNA2

Fake DANNA2 was designed so that the real DANNA2 could replace the DANNA2 Multi-Array component with the same top-level interface (shown previously in [Figure 7.19](#)). As previously mentioned, two components of DANNA2 had to be updated to support the multi-array design. These components are the element array and the element array control.

The multi-array design was changed to pass the data from the additional AXI4-Streams for the neighboring chips to the new array control logic. This new control logic also includes output ports for fires from neighbors and input ports for fires to neighbors. The array generation logic is mostly the same. There is still a generate loop to create all of the internal DANNA2 elements. The difference is that the fire array, which holds element fires, has been extended to include locations to store fires to and from the neighboring boards as well. Logic is also added to take the fires from neighbors and place them in the correct spot in the array. Similar logic takes the output from the array and sends it to the fire to neighbors port. [Figure 7.25](#) shows a visual diagram of the extended fire array. The array is extended two levels deep; this allows the internal elements to still maintain their full 24 possible synaptic connections. The exception is the corners of the array. Because the system only has connections to the north, south, east, and west, there are no diagonal connections to provide fire information on the corner elements. There are possible ways to get this corner information, but they are too slow or resource intensive. One way would be to add diagonal connections, but this would double the communication channels and the resources needed for

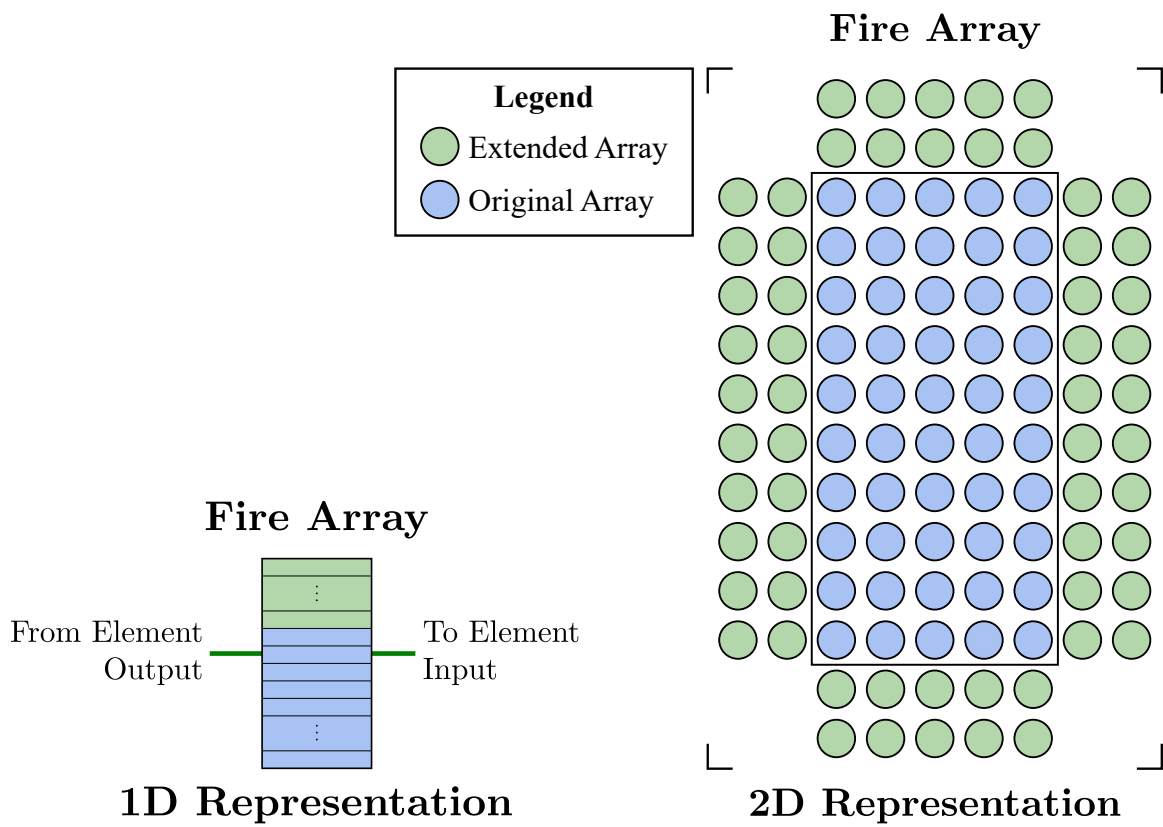


Figure 7.25: Diagram of the extended fire array.

communication. Another way would be to use a two-phase packet transfer. The first phase would pass packets north and south and the second phase would pass packets east and west. With this method, each board would obtain information about the corner boards. Thus this method would double the effective latency of sub-array communication, which would hurt performance too much. In the end, it was decided that the best solution would be to not allow diagonal connections at the borders of the sub-arrays. This has proven not to be a big limitation, and will be discussed further in [Chapter 8](#).

The array control was modified to have additional input and output ports for the neighboring AXI4-Stream buses and for fires to and from the array. Additional signals called neighbor block and neighbor ready were added to make sure each neighbor is ready to receive packets and that each neighbor has provided the previous timestep's fire information. If the previous information has not arrived yet, or the neighbors are not ready to receive a packet, then the array operation will pause, until it is ready to run. When the neighbors have sent the fire information from the previous cycle and are ready to receive a new packet, the array will run until the next timestep. Fire information will be read from the neighbor packets and the fire information will be sent to the array. Fire information from the run will be put into packets and sent to the neighboring boards once the next timestep is reached. Input packets will only be expected from enabled ports; similarly, output packets will only be created for enabled ports. However, input packets are always expected from the host. The reset logic was also modified so that fire information from $\text{delay} + 1$ neighboring packets will be ignored. This is necessary to clear the fire information from before the reset happened. Because the packets are still needed for synchronization, they are not dropped, but their fire information is ignored.

The fire packets sent between the sub-arrays are bit fields with a one if the neuron at that location fired and a zero if no fire occurred. The fires that make up the packets are from the two rows or columns closest to the edge bordering the neighboring sub-array. In [Figure 7.25](#) the two rows of green elements are sent from the neighboring sub-array. The two rows or columns nearest the borders will be sent to the neighboring board. The packets between sub-arrays use the same size as host packets. These packets are 64 bytes (or 512 bits). This means that sub-array sizes up to $512/2 = 256$ by 256 are supported. If sub-arrays

get larger than 256 per edge, then the size of the packets could be increased, or multiple sub-arrays could be implemented per board, so that this number is kept at 256 or lower. For small arrays, the excess bits in the bit field are unused and left at zero. Since the actual size of sub-arrays are currently much lower, there is room to grow. It would also be possible to save a few cycles in the transfer by reducing this packet size to the nearest multiple of 64.

7.3.3 NACC and Sub-array

The NACC and sub-array communicate over an Aurora channel much like the sub-arrays; however, the input from the controller does not need a preload circuit. Within the DANNA2 core, the host input and output is handled the same way as DANNA2 on a single FPGA. On NACC, the communication channels similarly include the Aurora and ACK components. NACC also has larger native FIFOs with different read and write clocks, which are used to buffer packets and change clock domains. FIFO latency here is not as crucial since Xillybus has significantly higher latency. On NACC, it is more important to have large FIFOs to support the large, bursting transfers from PCIe.

In addition to the Aurora channels, there is also a reset signal sent from NACC to the neuromorphic sub-arrays. This is used to reset the sub-arrays to an initial state. The reset is set up so that it will be asserted when the host is not connected to NACC. When the host connects, the reset will be held low, the communication channels will be established, and the sub-arrays will be in a cleared state. This means that in order to reset the system, the host only has to disconnect and reconnect to NACC.

The NACC and sub-array communication is largely the same as in the prior DANNA communication work. The major changes are the addition of multiple communication channels to enable communication to multiple sub-arrays. Additionally, the connection hardware is changed to support multiple connections. The previous setup directly connected the Neuromorphic Array to the NACC using a direct FMC connection as shown in [Figure 7.26 \[122\]](#).

A direct FMC connection was possible because the 690T supports stacking operation. To support connections to multiple sub-arrays, FMC-to-SMA adapters are used. The SMA cables are used to connect NACC to the sub-arrays. Four SMA cables are required for

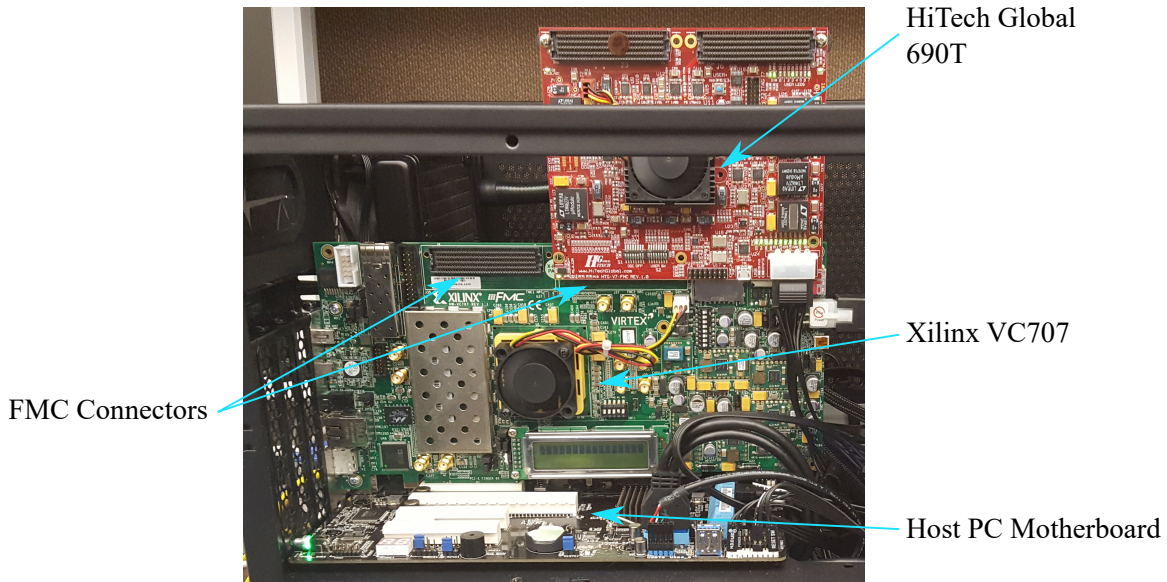


Figure 7.26: Picture of the previous DANNA communication setup hardware.

each bidirectional signal-lane Aurora channel. Two SMA cables are needed to transmit the differential signal and two sets of cables were required to support bidirectional transport. An external clock board is also used to provide reference clocks to each of the Aurora channels. Although the clock board is providing the same clock to each channel, this is not a requirement. The Aurora channels only require a clock of the correct frequency per channel for the system to function properly. An additional reset line is also used to send a reset to each of the sub-arrays. At first, the state of the Aurora channel was used as a reset; however, with long jobs, the Aurora channel would momentarily go down, which would result in a bad sub-array reset. To avoid this issue, a separate reset line is used. The new hardware setup is shown in [Figure 7.27](#).

7.3.4 Host and NACC

The host to NACC connection uses Xillybus, similarly to the previous DANNA2 communication system. The main difference is the configuration of the Xillybus IP block. The Xillybus IP Core Factory is used to configure a Revision B Xillybus core for Xilinx Virtex-7. This core is set up with eight device files. All of the device files are configured with a data width of 64 bits, an expected bandwidth of 1600 MB/s, and no autoset internals. The internals of device

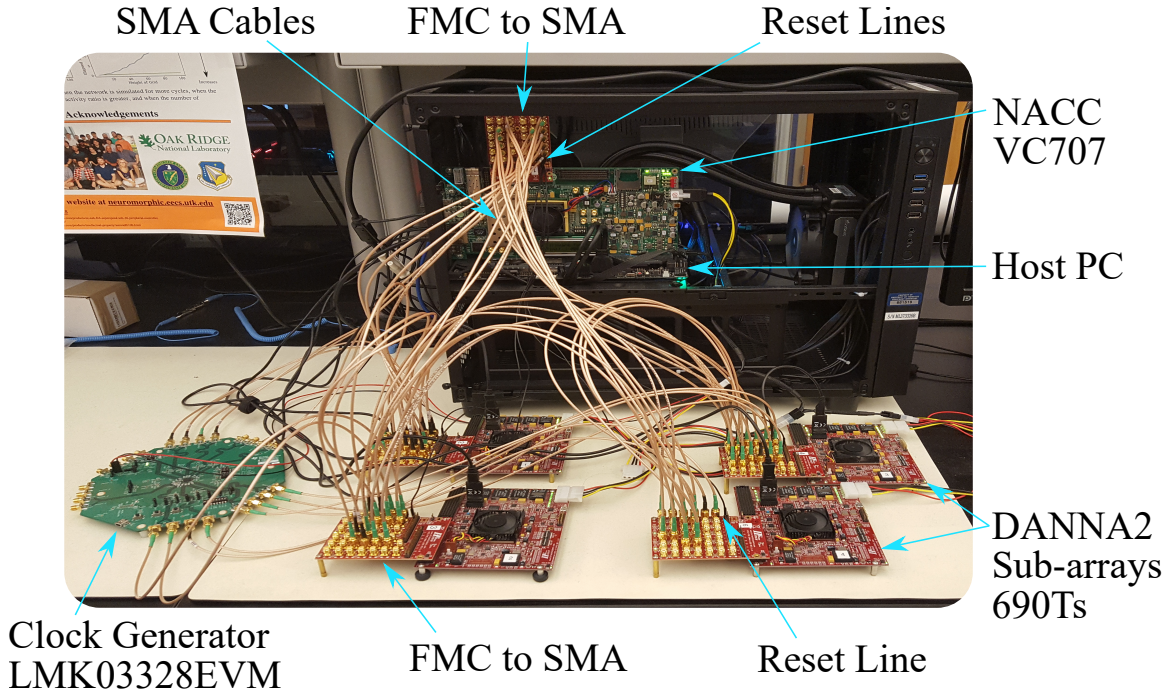


Figure 7.27: Picture of the SNACC hardware setup.

files are configured as asynchronous, with 512 buffers, each with a size of 128 kB (total size of 64 MB), and as the type “data exchange with coprocessor”. The downstream streams are also configured with DMA acceleration with 16 segments of 512 bytes. Two device files, one upstream and one downstream, are configured for each of the neuromorphic sub-arrays in the two-by-two SNACC system. They are labeled as DP0 (Data Pair 0) to DP3.

7.4 Host

The host is a Linux computer, which is used to configure the hardware, to provide input step/fire packets that cause the neuromorphic hardware to execute, and to process the resulting output fires. The host system used for SNACC runs Ubuntu 16.04 with updated Xillybus drivers. The system is powered by an AMD Threadripper 1950X, Asus ROG Zenith Extreme EATX motherboard and 32 GB of DDR4-3600 memory. This computer was custom-built for SNACC and was designed to have sufficient PCIe capabilities to further extend SNACC with multiple NACC boards. The ROG Zenith Extreme motherboard with the

Threadripper processor has 4x PCIe 3.0 x16 slots, which support quad cards at x16/x8/x16/x8. This means that the system could potentially support 4 separate NACC cards. Currently the system has a KCU1500 used for single board DANNA2 development and the SNACC setup shown in [Figure 7.27](#). The case housing the computer is a Thermaltake Core X5 ATX desktop case. This case was chosen because it is spacious and lays the computer motherboard down flat. The computer's power supply is used to power the host, NACC, DANNA2 sub-arrays, and the clock distribution board. The host setup is similar to the single-board design, using Xillybus device files to send and receive packets between the NACC and the host driver. The host driver implements the DANNA2 Device Virtual Functions shown previously in [Figure 6.8](#). SNACC also has a similar build system to make compiling bitfiles easy.

7.4.1 Build System Design

A build system similar to the one designed for DANNA2 on a single FPGA was also set up for SNACC. Three main python scripts are provided to build the NACC, the DANNA2 sub-arrays, and the fake DANNA2 test design. The directory structure for the build system is shown in [Figure 7.28](#). This directory is located within the DANNA2_HW directory and uses the DANNA2 source files from that directory. The three python scripts are found in the build folder. All of the scripts can be used to generate a bitfile using the script-based workflow, or a project file which can then be opened in Vivado for a project-based workflow. The fake DANNA2 build script adds the ability to specify the delay value to add as inherent delay for cross sub-array communication. The DANNA2 build script has the delay parameter in addition to the build flags used for the DANNA2 single board build script. The width and height parameters are for the width and height of the sub-array. The full system will have the combined size of all the sub-arrays. Currently, all the neuromorphic arrays in SNACC need to be the same size. If the number of total elements is not an even multiple of the number of sub-arrays, then the last sub-array can either have unused elements or be generated with a smaller sub-array size. The source files are organized based on the type of file. Since many of the projects reuse the same source files, they are not separated by project. When output products are generated, they are placed in their own path, so that different builds with different parameters will be stored in a different location and will not interfere with

SNACC

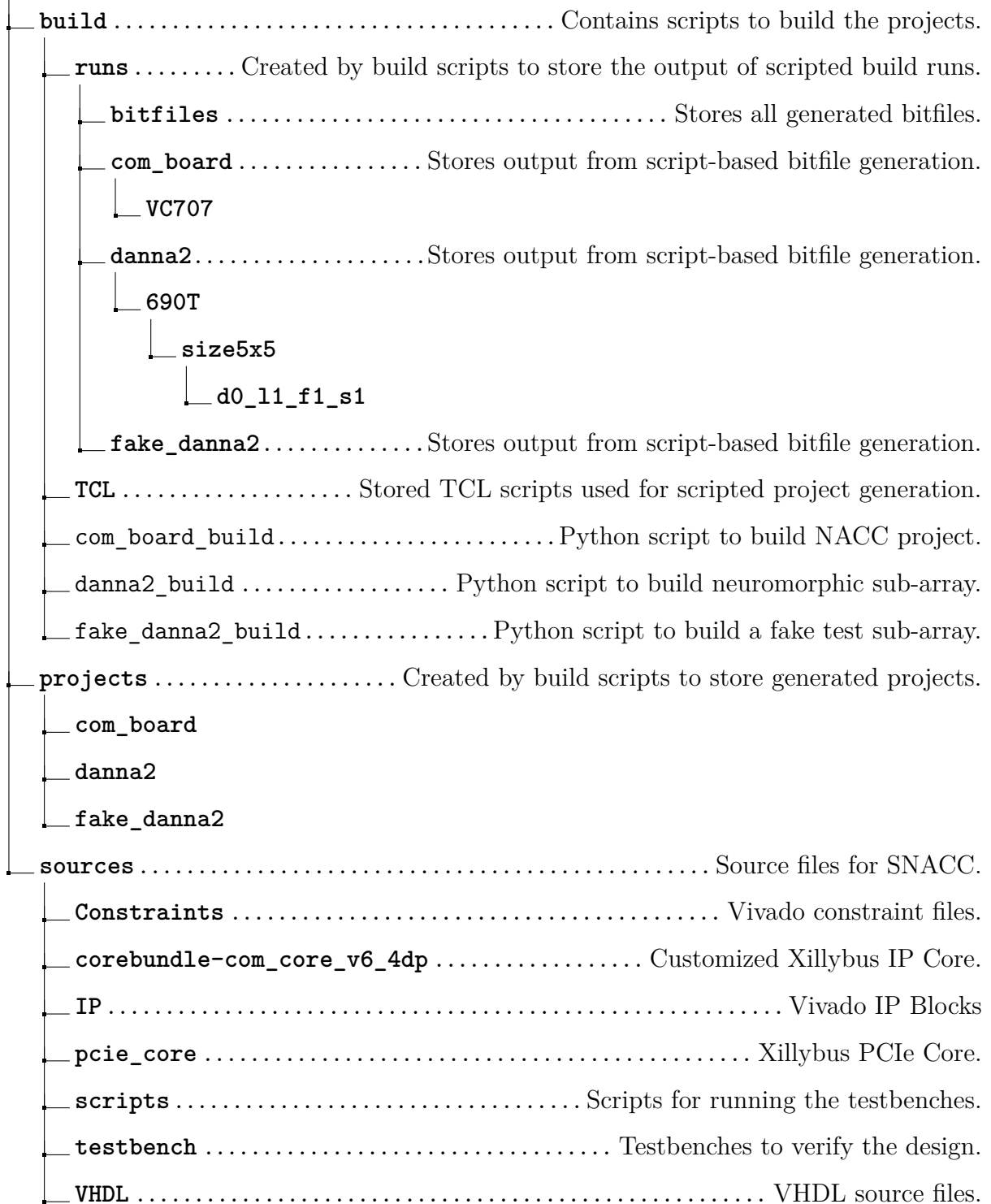


Figure 7.28: Directory structure for the SNACC build system.

each other. The DANNA2 sub-arrays can also be specified with a ‘spec string’. The string is similar to the one used for single board DANNA2 but includes a parameter for the delay value. The VUnit test script in the DANNA2_HW directory also has a high-level test to verify the operation of the SNACC system.

7.4.2 Driver Design

Two separate designs are used to communicate with the fake DANAN2 design and the real DANNA2 design, respectively. For fake DANNA2, hardware testing was added to the SNACC Simulator program. Now in addition to simulating the communications using the software event simulator, the SNACC Simulator is also able to use the fake DANNA2 hardware to test array performance. The comms interface is used to specify the communication functions and the DANNA2 interface specifies the functions for the fake DANNA2 device. These interfaces were previously implemented in software and now they are reimplemented to execute on the hardware. The hardware implementation of the fake DANNA2 interface opens the device files and starts two threads, one for reading and another for writing to the device. This class also defines functions to add packets to the send queue and to parse received packets. The class also stores the cycle the device is currently on. This value is updated by reading the response packets. Messages and locks are used to ensure the correct operation of this multi-thread program.

The comm interface implementation handles the running of the tests. The run function starts a new thread to handle input from the sub-arrays and sends the last cycle of the test to the FPGAs not in the first column. Then those FPGAs are sent input packets when the simulation time is greater than or equal to the time specified in the input file. The test will continue to run until all of the sub-arrays finish running to the ending timestep. This fake DANNA2 hardware test program uses ten threads in total for a two-by-two SNACC test. One main thread is used for sending communication packets from the host to the sub-threads. Another thread is used to collect timestep updates from the sub-arrays to determine when the run was completed. Finally, two threads per sub-array are used to read and write packets to the device file.

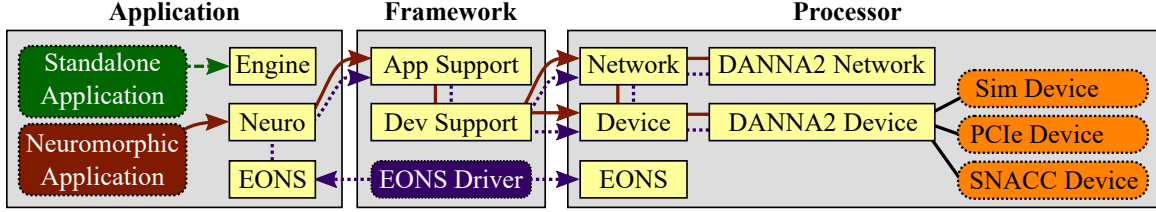


Figure 7.29: Diagram of the TENNLab Framework with the DANNA2 SNACC device.

The real DANNA2 driver was created by adding a new implementation to the DANNA2 device virtual class within the TENNLab framework. [Figure 7.29](#) shows an overview of the TENNLab framework structure with the DANNA2 SNACC device included. This implementation is split over two classes. The first class is called SNACC device and implements the DANNA2 device. The second class is called SNACC single and handles communication with a single sub-array. The SNACC device contains a vector of the correct number of SNACC single objects. When a function of this class is called, it calls the corresponding functions in the correct SNACC single objects. As with fake DANNA2, the SNACC single class uses two threads, one to handle sending and one to handle receiving. When the SNACC device simulate is called, a simulate thread for each SNACC single is created. These threads move the correct packets to the send queue and wait for the simulation to be complete. A simulate call for a two-by-two SNACC array uses 13 threads. These threads are the main thread, four simulate threads, and eight single device threads. All of these threads work together to correctly talk to each of the neuromorphic array boards. This large number of threads is not an issue since they are all used to handle I/O, and they spend much of the time waiting for data to arrive. Locks and messages are used to make sure the multiple threads work together correctly.

The basic thread creation flow is illustrated in [Figure 7.30](#). The read and write threads are initialized when the SNACC hardware class is constructed. These threads will continue to exist until the SNACC hardware is deconstructed. The write thread takes any packets added to the send queue and writes them to `/dev/xillybus_write_dp0`. Conversely, the read thread takes any packets received from `/dev/xillybus_read_dp0` and records the fire information in the recorded fires data structure. When the simulate function is called, a

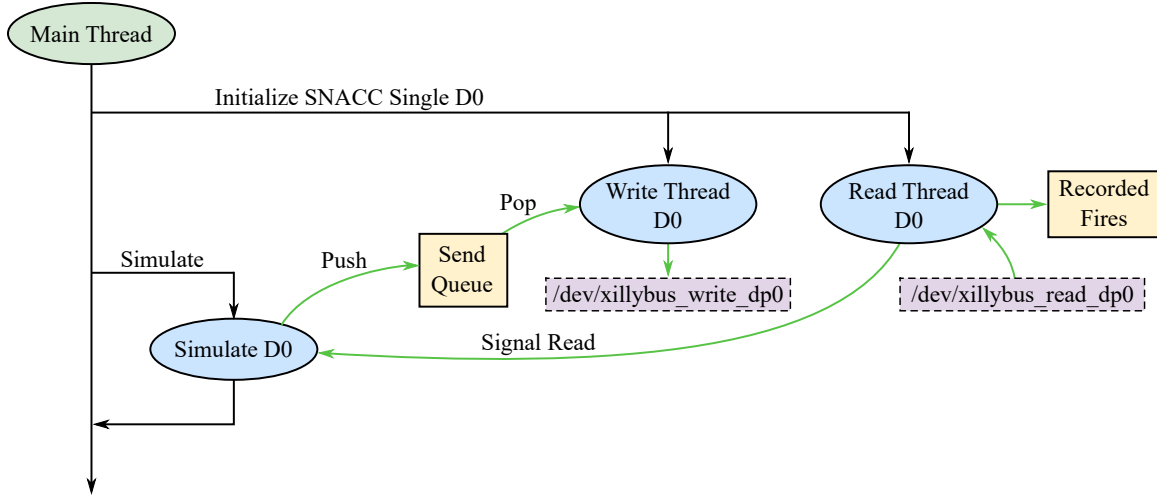


Figure 7.30: Flowchart of SNACC Driver Threads. Note: Only the threads for DANNA2 sub-array 0 (D0) are shown. The threads for the remaining sub-arrays are not shown.

simulate thread is created for each sub-array. This function pushes the fire data to the send queue. When new packets are received, this thread checks to see if the simulation is finished. Once the run is complete, the simulate thread joins back to the main thread.

7.5 Verification

As SNACC was developed, each component was verified both individually and as part of the system. Each component was tested for bugs as well as checked for performance. Some of the components underwent multiple iterations in order to achieve better performance. The first phase of verification was to test each custom component with a testbench. This verified that the components were logically correct individually. Each testbench was written with assert statements so that the correct behavior is checked by running the testbench. The next phase of testing involved combining components together and testing the system at a higher level. Next the design was deployed to hardware and debug probes were used to watch key signals to verify correct behavior. Some of the aspects of the design could only be thoroughly tested on hardware. These components included the clock domain crossings, the asynchronous interaction of components, and the behavior of the Aurora components. The rigors of hardware simulation paid off in helping verify the design.

The most important components are the ones that lie on critical communication paths. These paths include the host to sub-array channels and the sub-array to sub-array channels. The sub-array to sub-array channels were particularly important since they are key to determining how fast the neuromorphic arrays can run. Also, because of the symmetric design of the sub-array to sub-array channels, any reduction of latency in a component could have a two or four times reduction in latency overall.

One of the first improvements made to this critical path was to switch from using the Aurora 8b10b IP core to using the 64b66b Aurora IP core. The 64b66b encoding has lower latency, is able to operate at higher line frequencies, and supports greater throughput. Changing to 64b66b reduced round trip latency from 485 ns to 452 ns, allowed the channel to operate at a higher frequency, and increased throughput. When this array-to-array latency is measured in clock cycles, the base latency for Aurora 64b66b was 58 cycles on average at a 156.25 MHz clock. When adding framing to the Aurora core, the number of clock cycles increased by an average of one cycle. Added CRC checking to the core increased the latency by an average of seven cycles, which means that together the latency is increased by eight cycles over the base latency. The Aurora ACK ARQ component requires both framing and CRC to operate. This means that the Aurora channel has an unavoidable latency of 66 cycles on average. This amount also agrees with the advertised latency in the Aurora documentation.

The initial Aurora ACK implementation added an additional 32 cycles on average to the latency. Multiple improvements were added to lower this value. The receive logic was reworked to save two cycles on average. Creating custom AXI4-Stream width converters saved an additional 20 cycles on average. However, even with these changes, there was still a large difference between the minimum and maximum latency for the sampled latencies. The reason for this difference was because the aggressive retransmission was too aggressive, resulting in the channel being preoccupied with a retransmission when a new packet arrived. Because of this, the flood control feature was added. With a max flood of three, four additional cycles were saved. After fine-tuning this value to two, an additional clock cycle was saved. [Figure 7.31](#) shows debug probes with the correct behavior of flood control. The packet is retransmitted twice, which allows the channel to be idle when a new packet arrives. With all

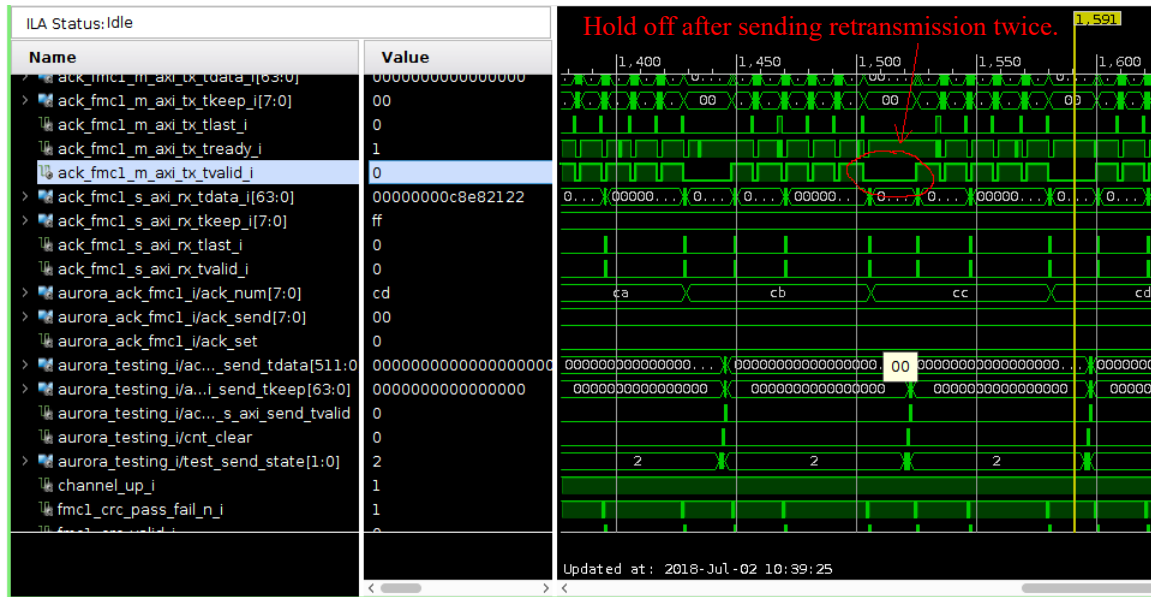


Figure 7.31: Debug probes showing the correct behavior of the max flood control.

of these changes, the average number of cycles added for acknowledgment is five cycles. These five cycles are for the send latch, writing to the window buffer, deciding to send a packet, sending a packet, and the receive latch. These improvements changed the additional latency required for the retransmission logic from 32 additional cycles to only five additional cycles.

Another key area of improvement was with the packet FIFOs, which are used to implement delay slots for packets and for clock domain crossing. The original packet FIFO design added 20 cycles on average for the array-to-array packet latency measured on the 156.25 MHz clock. Changing to the custom clock converter logic saved 11 cycles. Changing the FIFO for the window buffer saved an additional two cycles. This means that on average the redesigned packet FIFO only adds seven cycles instead of the initial 20 cycles.

Why are improvements to the array-to-array communication path so critical? It is because the array-to-array communication path directly effects how many delay cycles are required to hide sub-array communication latency. Since DANNA2 operates at 10 MHz, the maximum latency which is required without any added network delay is 100 ns. With this converted to a 156.25 MHz clock domain, the maximum number of cycles of latency is 15 cycles. However, it is impossible to get this latency down to 15 cycles since the base latency of Aurora is 66 cycles. This is where the inherent board crossing delay cycles come in. In order to avoid any

performance penalty, five delay cycles are required to hide a latency of 66 cycles.

$$\begin{aligned}
\frac{\text{delay}}{10 \text{ MHz}} &\geq \frac{\text{latency}}{156.25 \text{ MHz}} \\
\text{delay} &\geq \left\lceil \frac{\text{latency}}{15.63} \right\rceil \\
\text{delay} &\geq \left\lceil \frac{66}{15.63} \right\rceil \\
\text{delay} &\geq \lceil 4.22 \rceil \\
\text{delay} &\geq 5
\end{aligned} \tag{7.6}$$

Similarly, using (7.6) to calculate delay, for the improved communication channel, with a latency of 78 cycles, a delay of five is enough to hide the communication latency.

$$\text{Total Latency} = \text{Base Latency} + \text{ACK Latency} + \text{Packet FIFO Latency}$$

$$\text{Total Latency} = 66 + 5 + 7$$

$$\text{Total Latency} = 78$$

$$\begin{aligned}
\text{delay} &\geq \left\lceil \frac{\text{latency}}{15.63} \right\rceil \\
\text{delay} &\geq \left\lceil \frac{78}{15.63} \right\rceil \\
\text{delay} &\geq \lceil 4.99 \rceil \\
\text{delay} &\geq 5
\end{aligned}$$

Before the improvements to the communication channel, a delay of eight would have been required to hide the communication latency.

$$\text{Total Latency} = \text{Base Latency} + \text{ACK Latency} + \text{Packet FIFO Latency}$$

$$\text{Total Latency} = 66 + 32 + 20$$

$$\text{Total Latency} = 118$$

$$\text{delay} \geq \left\lceil \frac{\text{latency}}{15.63} \right\rceil$$

$$\text{delay} \geq \left\lceil \frac{118}{15.63} \right\rceil$$

$$\text{delay} \geq \lceil 7.55 \rceil$$

$$\text{delay} \geq 8$$

The improvements to the array-to-array communication channel greatly improved system performance by allowing the system to perform better with lower levels of delay. A delay of five is required to hide the effects of array-to-array communication. A further increase in performance is not possible without a decrease in the base Aurora frequency. In order to only need a delay of four, the latency would have to be under 62 cycles.

The effect that delay cycle variation has on the average element clock cycle frequency was both modeled using the SNACC simulator and measured on the hardware. The results are shown in [Figure 7.32](#). As expected from the calculations, five added delay cycles are required to hide the array-to-array communication latency and allow the multi-DANNA2 array to operate with the same performance as a single board DANNA2 array. The lower average element clock frequency at ten delay cycles added is likely an artifact from the tuning of the flood control parameters.

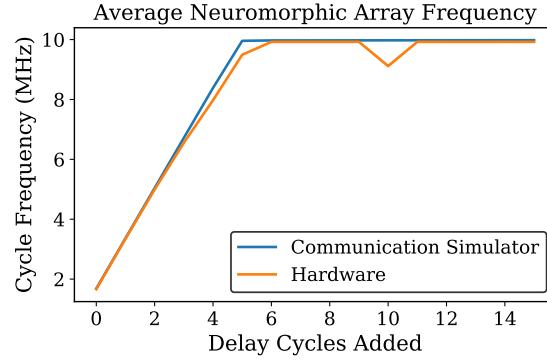


Figure 7.32: Added delay verses average element clock cycle frequency.

7.6 Power

The power usage of SNACC is explored in this section. Power is looked at both empirically from measurements taken from the built SNACC system and also theoretically by analyzing the energy consumed by the communication channels using the Xilinx Power Estimator. The mathematical analysis will discuss scaling potential and serves to isolate the energy used only for SNACC communication.

The built test system consists of one custom-built AMD Threadripper based host PC, one KCU1500 for single-board FPGA testing, one VC707 which is used as the SNACC communication board, and four 690Ts which are used as the SNACC sub-array DANNA2 processors. The total system consumes approximately 168 W to 170 W when sitting idle. This range stays the same regardless of whether the host system is connected to the sub-arrays or whether the sub-arrays are running a neural network. The power usage is greatly affected by the activity of the host system. The energy use of the FPGAs stays fairly consistent in this case since Aurora channel upkeep packets must be transmitted even when there is no active communication. Additionally, the amount of power used by the DANNA2 design on the FPGAs is very little compared to the static power of powering all of the components on the development boards. If the NACC and sub-array FPGAs are removed from the system, the power usage drops down to approximately 93 W. This means that the SNACC FPGAs consume approximately 76 W, both when idle or running. Of this 76 W, the VC707 consumes

~ 27 W, the clock generating board consumes ~ 4 W, and the sub-arrays consume ~ 45 W or ~ 11 W each.

The power required for developing the system using FPGAs on development boards is much higher than what would be necessary if custom boards were built and if ASIC chips were used instead of FPGAs. The power of the total system could be reduced by implementing the NACC and the sub-arrays in VLSI with various power-saving techniques, such as clock gating and asynchronous communication. Additionally, the system could save power by powering down the NACC and the sub-arrays while no jobs are currently being run.

The Xilinx Power Estimator (XPE) was used to perform the mathematical calculation of the energy required for SNACC communication [115]. Using this tool a single Aurora 64B66B communication channel using GTH transceivers on a 690T was calculated to require 0.334 W. Each sub-array uses five Aurora communication channels. One channel is used to connect back to the host and four are used to connect to neighboring sub-arrays. The total power usage of all five channels is calculated to be 1.669 W by XPE. This tool estimates that the communication channel only takes up around 15% of the 11 W measured empirically. This low percentage of power is likely due to the extra unused components on the 690T development board using additional power.

XPE can similarly be used to calculate the power used for communication on the NACC board. One Aurora 64B66B communication using GTX transceivers on a VC707 requires 0.390 W. The NACC has a total of four Aurora connections in the two-by-two SNACC system. This brings the total power usage for Aurora communication up to 1.559 W. With the power needed for the Xillybus PCIe port, the total NACC power used for communication increases to 4.085 W. This means that the total power required for the communication in a two-by-two SNACC setup is 10.761 W. The power usage for communication as SNACC scales is further explored in [Section 7.7](#).

The power required for communication is substantial for high-speed, multi-gigabit communication. With new low-power implementations of neuromorphic circuits, the power required for the spiking communication will become a significant part of the total system's power usage. Since SNACC is a prototype system designed with FPGAs, scalability of the design was prioritized over low-power operation of the design. Once large DANNA2

designs are implemented in VLSI, lower power communication links should be explored. If the communication is limited to within the VLSI design, then low-power asynchronous communication or AXI4-Stream busses could be used for communication. In order to communicate off chip, an FPGA could be used as an intermediary to interface with the host or SNACC system. For low-power external communication, the spikes could be fed in directly to and read from the VLSI chip without being first converted into packets. This would result in the lowest power operation, since a host would not be needed in conjunction with the DANNA2 processor. Once a host system is involved, the power used by an FPGA is not unreasonable for interfacing with the neuromorphic processor.

7.7 Scalability

This section looks at the scaling potential for the SNACC system based on both the limitations for the communication channels selected and on limitations with the current DANNA2 processor design. SNACC is designed to enable large scaling of the DANNA2 array and as such, allows many DANNA2 processors to be networked together to build a larger neural network. Although the SNACC design can be applied to different neuromorphic processors, the scaling analysis is done with the design choices of DANNA2 in mind. This section looks at the size limitations for each major component of SNACC and then builds up to look at the maximum size of the complete system.

7.7.1 Size per Sub-array

Starting closest to the neuromorphic processor, the first component to look at is the size limitations for a particular sub-array. This size is limited by the capacities of the FPGA or VLSI chip used to implement the DANNA2 processor for the sub-array. This analysis will continue the pattern of designing DANNA2 networks such that they are made with dimensions which are a multiple of five and the width of the array is roughly half of the height of the array. With the current DANNA2 design, the Xilinx 690T board can hold a 35×20 array when STDP, leak, and element reading are enabled. This size is increased to 50×25 with STDP, leak, and element reading disabled.

If larger FPGAs are used, then the size of the sub-array can also be increased. The KCU1500 can hold a 50×25 DANNA2 array with STDP, leak, and element reading enabled. If STDP, leak, and element reading are disabled, the maximum array size is increased to 70×35 . Newer FPGAs and VLSI design will further allow for increases in the sub-array size.

So what then is the maximum possible size of a sub-array? One design limitation is the packet structure for the DANNA2 processors. The maximum number of inputs supported by an input packet is 104, so a sub-array grid of 100×50 is the upper limit of what is supported by the input packet structure, keeping to the multiple of five scaling pattern. If every input element does not need to be an input, then the next limitation is the number of outputs supported by the output packets. Since each output is only a single bit, the maximum number of outputs supported is 416. The size of the sub-arrays could be made larger by spacing out the 104 input elements among the rows and still using all of the rows for outputs, up to 416 outputs. The sub-array can continue to be scaled up beyond 416 rows by likewise spacing out the output elements.

The 32-bit address selected to index the elements is not a limiting factor for the sub-arrays. The largest grid which can be indexed by the 32 bit address is 92680×46340 , so the choice of a 32 address will not be a limiting factor. If the entire combined array used a uniform index to index across the whole system, then this choice could start to be a limitation. However, the current implementation addresses the elements in the sub-array individually, so the choice of the 32-bit element address space will not be a limiting factor for SNACC scaling.

The main limitation for sub-array sizes for the current implementation of SNACC comes from the packets used for the sub-array communication. The current packet structure encodes each fire event on the edge of the array as a single bit. Two rows of neuron fires are sent so this packet size limits the number of edge elements to 256 elements. This means the maximum sub-array size supported without changing the sub-array packet structure is 256×256 . This limitation is with one Aurora lane between the sub-arrays. If the Aurora lanes are doubled (which can be done still using one FMC connection) then this number can be doubled without affecting the latency. So with two lanes and 1024 bit packets, a sub-array of 512×512 can be supported. If two Aurora lanes are used east to west and one Aurora lane is used for north to south, then a 512×256 sub-array could be supported, which keeps the one-half

width to height ratio. Of course the packet structure could be changed or more Aurora lanes added to be able to support larger sub-arrays with a similar setup. The key is that the current sub-array communication packet structure can scale to the same size as the host communication packet structure.

Therefore, the maximum size of the sub-arrays is largely constrained by the capacity of the implementation method and not the design of DANNA2 or SNACC. However, as larger sizes are able to be built—either with larger FPGAs or with VLSI—the next constraint hit is a result of the DANNA2 packet designs. To overcome this limitation, either a new packet structure can be used or fewer input and output neurons could be used. Eventually the sub-array communication channels will start to be a limitation; however, this limitation can be mitigated by adding more communication lines or by using one chip to implement multiple sub-arrays.

7.7.2 Size per NACC

The current NACC design is built with a VC707 development board. This board has two FMC ports, which each support eight SMA, bi-directional, differential ports. This means that with the VC707, 16 SMA ports can be used to connect a single NACC to the sub-arrays. Each sub-array uses a one-lane Aurora channel to communicate with the host, which requires one SMA port. Therefore the maximum number of sub-arrays each VC707 board can support is 16.

The number of supported sub-arrays can be increased if a custom NACC board is designed. If the same FPGA is used for the custom board, the number of sub-arrays which can be supported is increased to 48. The FPGA used by the VC707 is the Virtex-7 VX485T (485T) which has a total of 56 GTX transceivers. Since the PCIe endpoint needs eight of these transceivers, a maximum of 48 could be used for sub-arrays.

In addition to looking at the connection from the NACC to the sub-array, the connection from the NACC to the host must also be considered. If each sub-array transmitted at its maximum rate of one packet per network cycle, it would generate and consume 5.12 Gbps of bandwidth. This maximum rate really only occurs in bursts when configuring and reading out the neural networks. During the running of the neural network, the rate in which output

packets are generated is likely less than one packet for every four network cycles. The input packet rate is likely even less with packets only occurring periodically or at the beginning of the run. This would put the average transfer bandwidth at less than 1.28 Gbps. One channel of Aurora 64B66B supports a line rate of 10.3125 Gbps, which is plenty of bandwidth compared to the maximum DANNA2 burst transfer rate of 5.12 Gbps.

The connections to the sub-arrays can only feasibly scale as much as the connection to the host can support. As verified in [122], the maximal performance of Xillybus revision B is 1700 MB/s, which is equivalent to 13.6 Gbps. This means that one NACC PCIe connection can support two sub-arrays at the maximum throughput or ten sub-arrays if a packet only occurs every four cycles on average. If NACC was modified to use the new Xillybus revision XXL, which supports a maximum performance of 6600 MB/s, ten sub-arrays could be supported at the maximum throughput or 41 sub-arrays if a packet only occurs every four cycles on average.

In summary, the current maximum size per NACC on a VC707 is 16 sub-arrays, which supports a maximum throughput of one packet every six cycles. With a custom board using a 485T and Xillybus XXL, this number could be increased to 48 sub-arrays with a maximum throughput of one packet every five cycles.

7.7.3 Size per SNACC

The NACCs must be able to connect to a host to build the SNACC system. The currently used host PC can house four VC707 cards with its motherboard and CPU design. The CPU can support a maximum of 64 PCIe lanes, so a custom design could potentially support a maximum of eight VC707 cards.

So with these restrictions, how far can SNACC scale? With the same components currently in use, SNACC can scale to a total DANNA2 network size of 400×200 with one host, four NACC boards, and 64 sub-array boards implementing a 50×25 DANNA2 array. This scaled-up design implements 80,000 neurons and 1,920,000 synapses and supports a maximum throughput of one DANNA2 packet every six network cycles.

If the DANNA2 sub-array implementation is improved such that 100×50 DANNA2 arrays can be built, then the total network size increases to 800×400 . This size would implement

Table 7.2: Scaling Summary

NACC per Host	Sub-arrays per NACC	Sub-array Height $\times$ Width	Total Array Height $\times$ Width	Number of Neurons	Number of Synapses	Communication Power (W)
4	16	50 $\times$ 25	400 $\times$ 200	80,000	1,920,000	142
4	16	100 $\times$ 50	800 $\times$ 400	320,000	7,680,000	142
4	16	512 $\times$ 256	4,096 $\times$ 2,048	8,388,608	201,326,592	185
4	48	512 $\times$ 256	7,094 $\times$ 3,547	25,165,824	603,979,776	534
8	48	512 $\times$ 256	10,033 $\times$ 5,017	50,331,648	1,207,959,552	1,068
8	48	512 $\times$ 512	10,033 $\times$ 10,033	100,663,296	2,415,919,104	1,325

320,000 neurons and 7,680,000 synapses. If the sub-arrays' size is increased all the way up to 512×256 , then the total network size would be 4096×2048 or 8,388,608 neurons and 201,326,592 synapses.

With a custom NACC design that supports 48 sub-arrays each and a sub-array size of 512×256 , this maximum scaling could increase to 25,165,824 neurons and 603,979,776 synapses implemented across a total of 192 sub-array boards. The maximum supported bandwidth with this setup using Xillybus version XXL would be one packet every five cycles on average. If all 64 of the CPU's PCIe lanes were utilized, eight NACCs could be supported and the maximum scaling potential would be 50,331,648 neurons and 1,207,959,552 synapses implemented across 384 sub-array boards. The scaling potential of SNACC is summarized in [Table 7.2](#). This table also calculates the estimated power needed for communication based on the communication power estimates computed by XPE in [Section 7.6](#).

Although the design of SNACC is able to scale to these large sizes, building the system with SMA cables and individual FPGA development boards is infeasible. To build out the system at this scale, a printed circuit board (PCB) with multiple sockets for neuromorphic processors will need to be created to reduce the cost and size of connecting all of the processors together. One NACC chip and 48 DANNA2 sub-arrays could be placed and routed on the PCB, to build one blade of the SNACC system. Then eight of these blades could be inserted into the host system to build the full SNACC system. [Figure 7.33](#) shows a diagram of this scaled-up SNACC system.

To build larger sub-array sizes, the digital design will need to be implemented using VLSI. The VLSI fabrication will allow for denser packing of the elements, which will enable the

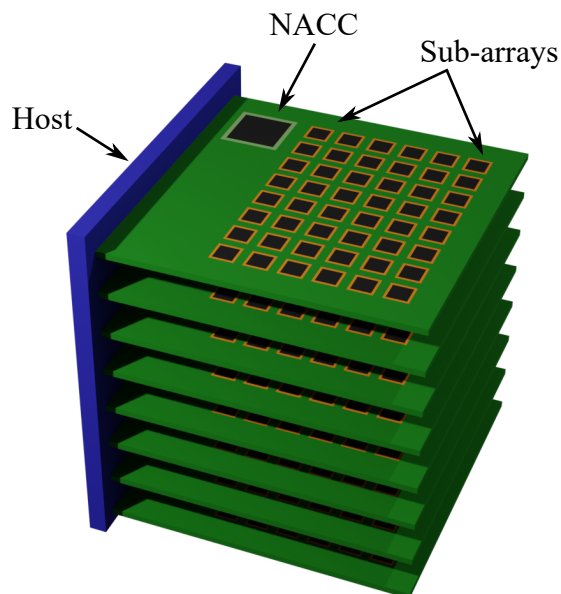


Figure 7.33: Scaled up SNACC System.

building of larger DANNA2 neuromorphic processors for use in the sub-arrays. If a point is reached when more than 512×512 elements can be packaged into a single VLSI chip, then the chip can be subdivided into a multi-core design, where multiple sub-arrays are implemented within a single VLSI chip. Each core of the multicore VLSI design would still need a separate communication channel to NACC, so the total scaling potential of the system would remain the same. However, with one chip implementing multiple sub-arrays, fewer physical chips would be required to reach the same size. Additionally the sub-array connections within the single chip could use the AXI4-Stream protocol directly for communication instead of needing to use the high-speed, serial, multi-gigabit transceivers.

One additional limitation for large DANNA2 networks is the time necessary for a fire to traverse the width of the network. For a small DANNA2 network, which is only ten elements wide, the traversal time for going straight across the array is between 10 and 160 cycles. This correlates to $1\ \mu\text{s}$ to $16\ \mu\text{s}$ with a 10 MHz network clock. For large arrays this delay is longer. For a network which is 160 elements wide, this range is 160 to 2560 cycles ($16\ \mu\text{s}$ to $256\ \mu\text{s}$).

If the largest possible size was built, the width of the network would be around 10,000 elements. This would result in a traversal delay of 10,000 to 160,000 cycles assuming a straight connection, which correlates to 1 ms to 16 ms. At this scale, the traversal time of the

network is the same as the real-time update deadline discussed in [Section 8.6](#) for real-time applications. Networks this large would likely not be useful for real-time applications, since the network is not able to respond to input fires within the real-time update deadline. For large networks of this scale, a different connectivity pattern, perhaps one that allows for longer jumps, will be needed.

In conclusion, the SNACC design has great scaling potential and can be used to scale designs like DANNA2 to very large sizes, while maintaining the high-performance of the processors. This design supports the scaling of DANNA2 grid arrays to a point where the DANNA2 grid connection patterns will limit the usefulness of larger networks. To further scale DANNA2, additional connectivity patterns in addition to the nearest neighbor connections are needed.

Chapter 8

Applications and Performance

Nothing is particularly hard if you divide it into small jobs.

– Henry Ford

The DANNA2 neuromorphic processor has been successfully deployed onto hardware using FPGAs for prototyping the design. DANNA2 has been prototyped on a single FPGA as well as on multiple FPGAs using the SNACC system. The DANNA2 simulator has been used to perform many different neuromorphic tasks. This chapter looks at some of these applications and evaluates the performance of running these tasks on hardware as well as with the software simulator. A wide variety of key application types are explored which cover a wide range of the application space. [Section 8.1](#) looks at quantifying the performance characteristics of the DANNA2 SNACC system on hardware versus the DANNA2 SNACC system in the software simulator by looking at the performance of various patterned test networks. The remaining sections in this chapter look at various applications which can be run on DANNA2 SNACC hardware, and look at the trade-offs of running the applications on hardware versus the software simulator.

8.1 Performance Measurements

In order to evaluate the performance of the different DANNA2 single-chip and SNACC based neuromorphic networks, networks with different test patterns were created. These test

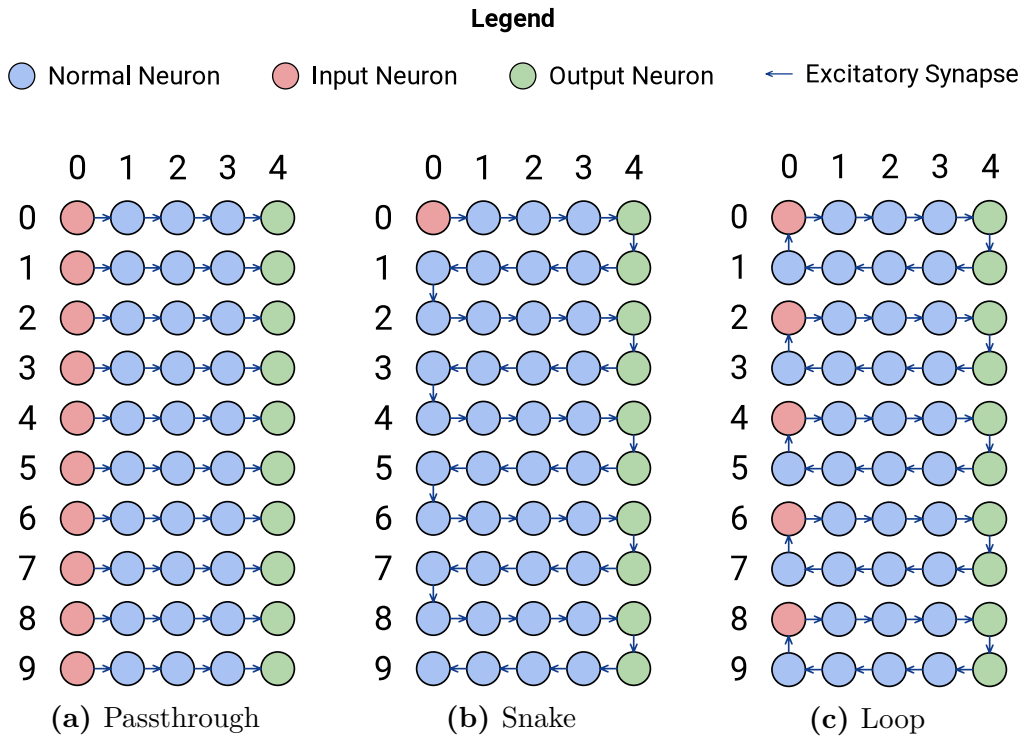


Figure 8.1: Example passthrough, snake, and loop networks with a height of ten. The input elements are shown in red, hidden elements are shown in blue, and output elements are shown in green. All of the synapses are excitatory.

patterns are shown in [Figure 8.1](#). One main reason these test pattern networks are used is that they can be generated for any network dimension. This flexibility allows the testing of DANNA2 processors with various network sizes, all with similar and predictable network activity. These patterns were also crafted to have different ratios of input activity to internal activity. Each neuron in the patterns has a threshold of one and a refractory period of zero. The synapses all have a weight of two. The synapses in the passthrough network have a delay of ten and in the snake and loop networks, they have a delay of seven.

The passthrough network pattern takes input fires from the left edge and sends them to the right edge (shown in [Figure 8.1a](#)). The snake network pattern, shown in [Figure 8.1b](#), has only one input, but every element in the array is used to pass the event from the start of the snake to the output at the end. Every element in the last column is set as an output. Since the hardware will send back all the fires from the last column regardless of the neuron setting, making each of these elements an output element will make the comparison between software and hardware more accurate. As shown in [Figure 8.1c](#), the loop network pattern uses recurrent internal loops to maintain internal activity without any additional external input. With loop, input fires are only sent in at the beginning of the run, and the loops cause these fires to continue circling throughout the network. The networks are loaded onto the SNACC system by generating a network the same size as the large neuromorphic array and then loading equal regions of the network onto each of the sub-arrays. [Figure 8.2](#) shows how the DANNA2 elements are split between the sub-arrays for a 40×20 network.

These three network patterns have varying amounts of input fires compared to internal fires. With passthrough, a fire is required for each row, and each input fire results in a number of internal fires equal to the width of the network. The snake pattern only has one input, and each input fire results in a number of internal fires equal to the area of the network. With the loop pattern, input is only provided at the start of the run, and the recurrent loops maintain the network activity. This means that each input results in a number of fires equal to the number of cycles that the network is run for.

To collect data for this section, a data collection script was written in Python. This script automates the loading of bitfiles as well as the rebooting needed to rescan the PCIe bus. The

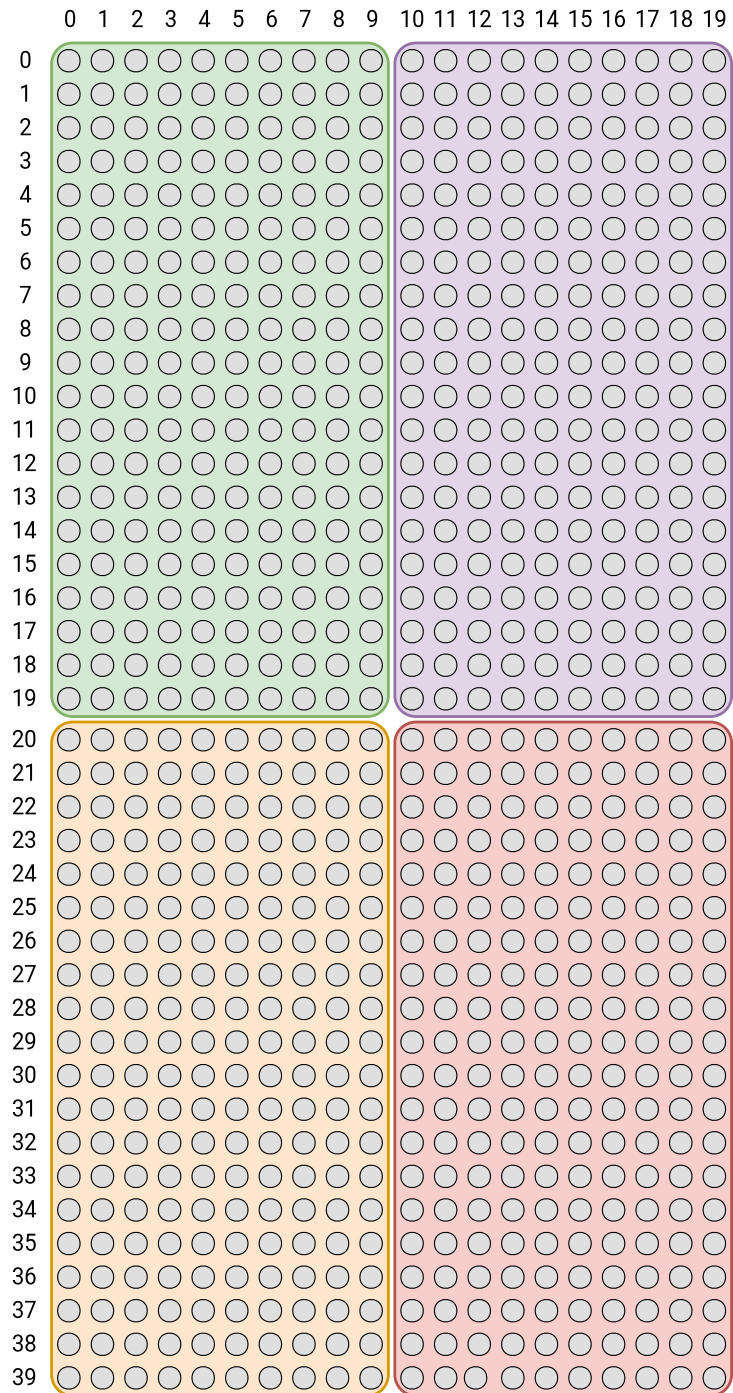
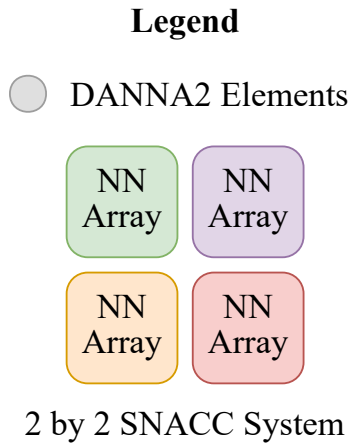


Figure 8.2: Network split to load DANNA2 arrays onto the SNACC system.

script also generated, timed, and collected data from each test run. The final output of this script is a Comma-Separated Values (CSV) file containing all of the collected data.

For the data collection, each network pattern (passthrough, snake, and loop) was evaluated with each device type (software simulator, single FPGA, SNACC with an added communication delay of zero, and SNACC with a delay of six). A delay of zero was tested since it results in the minimal network connectivity restrictions, and a delay of 6 was added since it is large enough to hide the communication latency. For more information on SNACC communication delay, see [Section 7.3.2](#). The number of fires sent into the network was varied from once every cycle to once every 20 cycles, decreasing by one. Each test was run for 1 k, 5 k, 10 k, 50 k, 100 k, and 500 k cycles. Each device was tested with network sizes ranging from 5×5 , with an increase of five rows and maintaining a width of the nearest multiple of five that is half the height. This means the simulator device tested the following sizes: 5×5 , 10×5 , 15×10 , 20×10 , 25×15 , 30×15 , 35×20 , 40×20 , 45×25 , 50×25 , 55×30 , 60×30 , 65×35 , 70×35 , 75×40 , 80×40 , 85×45 , 90×45 , 95×50 , and 100×50 . Single-board FPGA was tested for these sizes: 5×5 , 10×5 , 15×10 , 20×10 , 25×15 , 30×15 , 35×20 , 40×20 , 45×25 , 50×25 , 55×30 , 65×35 , and 70×35 . Additionally, SNACC was tested for the following sizes with a delay of zero and a delay of six: 10×10 , 20×10 , 30×20 , 40×20 , 50×30 , 60×30 , 70×40 , 80×40 , 90×50 , and 100×50 . Each test was run 10 times to get an accurate time. The SNACC system used two-by-two DANNA2 sub-arrays, which means each sub-array was loaded with a DANNA2 network with half of the total number of rows and columns.

The total evaluation time as well as the time for the simulate function call were recorded as the metrics from each test. The total time includes the time spent loading the network and generating the fires. The simulate time for the simulator only measures the time required to process the input and internal events and generate the output events. For the hardware, this is the time required to send the packets to the hardware and receive the response packets back. The simulate function call time is also referred to as the network evaluation time. In all, 190,800 test runs were performed to collect the data presented in this section.

The simulator and the hardware evaluate the neuromorphic networks differently. The simulator uses an event queue and processes each event as it happens. This has the side

affect of allowing the simulator to run incredibly fast when the run has few events. With the simulator, the time required to evaluate a given timestep depends on the number of events which occur at that timestep. This means the simulator is particularly well-suited for evaluating large, sparse networks with relatively few fires. The simulator treats input fires the same as internal fires, so there is no additional penalty for an event being an input versus an internal fire.

The hardware processors, on the other hand, evaluate each timestep at a fixed frequency, but must wait for the required input from the host before continuing execution. Since the array is implemented in hardware, the events of each element are evaluated in parallel, which results in a fixed time for evaluating each timestep regardless of the number of events which occur during the timestep. Array input is not the same as internal fires. Internal fires are generated and consumed by hardware elements and thus do not affect timestep evaluation time. Input fires originate from the host, and the host must convert each of the fire events into a fire packet to be sent to the hardware. If the host is not providing the input packets at the same rate as the timestep evaluation, the input generation will limit the performance of the hardware processor. This extra processing needed by input packets causes the number of input fires to have a large impact on the total run time of the hardware processors. Therefore, the hardware is best suited for evaluating long-running jobs. These utilize dense arrays with a large amount of internal activity compared to the amount of external input, and thus the host is not a performance bottleneck when many fire events must be processed.

Figure 8.3 shows select size sweeps for the different network patterns run for 10 k cycles. These graphs show the expected behavior based on the way the different DANNA2 devices function. The selected graphs are for 10 k cycles, since they show many interesting features. 10 k cycles is long enough to see patterns form, but few enough cycles to see the performance crossover point as network size increases. The graphs in the first column only show the time required for the network evaluation portion of the run which is also the time spent in the `simulate` function call in the DANNA2 API. The graphs in the second column show the total time for the entire test run, which includes the time required to load the network and generate input. For hardware, this means the configuration packets and input packets had to be created.

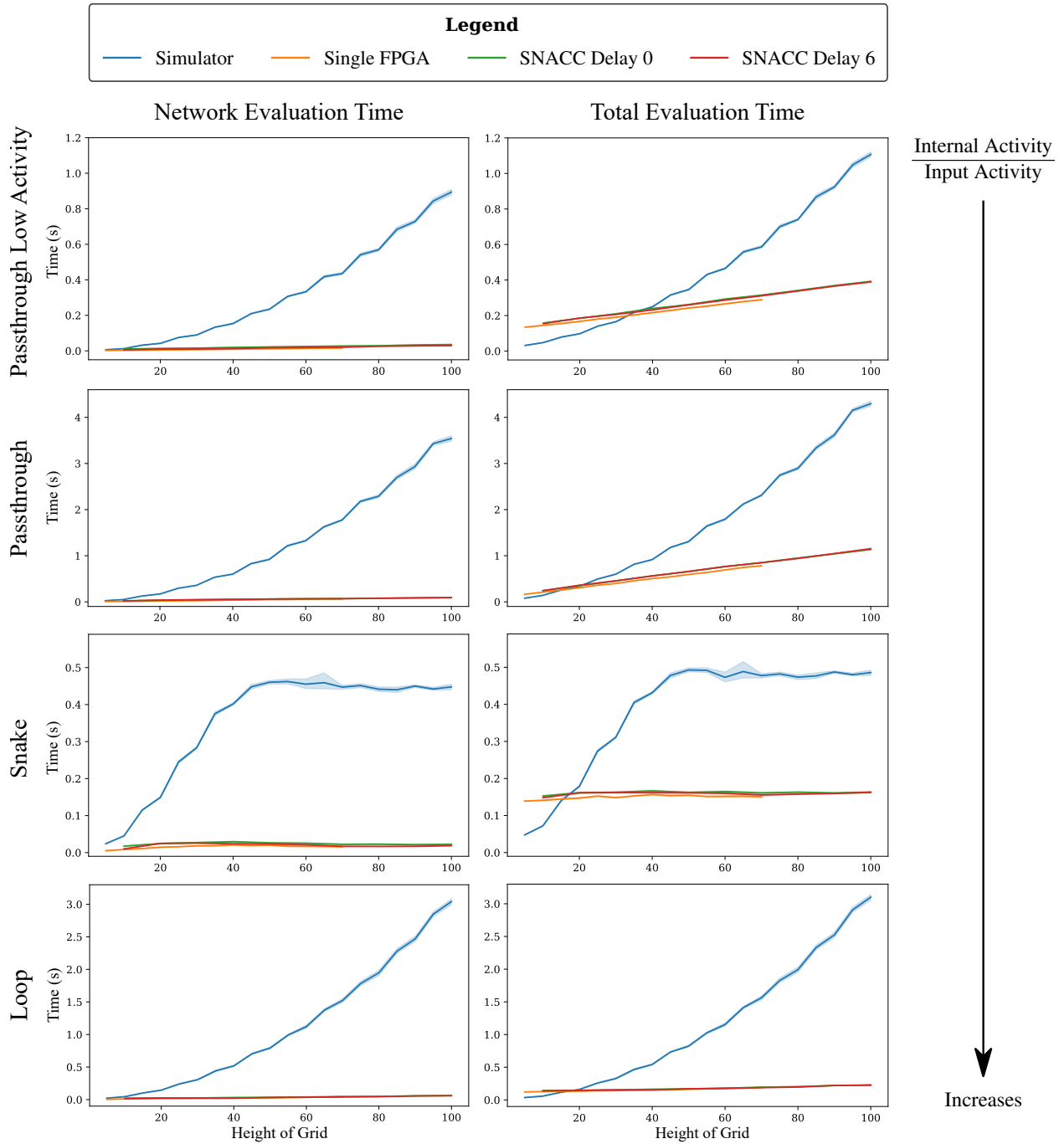


Figure 8.3: Network evaluation time comparison between select size sweeps for hardware and software for 10k cycles.

As seen in the first column of graphs, the time required to evaluate the array in hardware does not change with height, since the network was run for the same number of cycles in each case. The graphs on the top row show the elapsed time for running a passthrough network with an input fire every 4 cycles. In the case of the total runtime, the time required for the hardware to evaluate the network changed linearly with the height, which for passthrough, corresponds to the number of input elements. The software simulator time increased at a rate linearly correlated with the total number of elements in the array¹. The width of the array is the closest multiple of five that is half the height; this explains the wavy increases in the simulation evaluation time in this plot. When plotted with the x-axis showing the number of elements, the simulator trend is a straight line.

Going down the rows of graphs, the relative amount of internal activity per input packet increases. The passthrough low activity run received an input packet every 4 cycles. The passthrough run in the middle row receives an input fire every cycle. This change in internal activity per input packet causes the hardware to outperform the simulator sooner with smaller network sizes. (Lower elapsed time is better.) Since the snake network only has one input per cycle, the elapsed total time for hardware is constant for both the evaluation time and elapsed total time. The simulator time for the snake pattern plateaus since the snake is too long to be filled up with only 10 k fires. When the snake network is run for more cycles, the simulator trend follows that of the other networks. The times for loop networks are also linear for hardware since no input occurred after an initial few cycles.

From these graphs, the simulate time for hardware is always faster than the simulator, but the total elapsed time for evaluation is only faster in hardware some of the time. For the low activity passthrough, the hardware is only faster for networks with a height greater than 35. This number is decreased to around 20 for the other networks with more internal activity.

As mentioned previously, the simulator uses an event queue and must process each event in the network sequentially. This means that the time the simulator takes to evaluate a network should depend on the number of events which must be evaluated. For each of the collected test runs, the number of input fires and the number of fire events were calculated

¹The simulator time for the snake pattern plateaus since the snake is too long to be filled up with only 10 k fires. If the test was run for a larger number of cycles, then the simulator trend for snake would match that of the other networks.

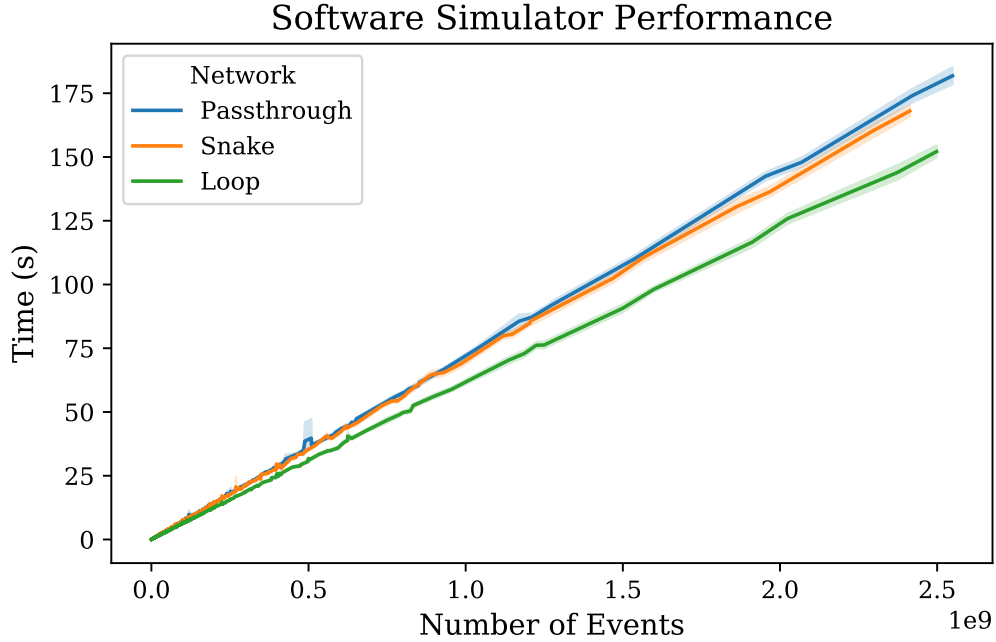


Figure 8.4: Simulator performance for a varying number of events.

based on the pattern of the network and on the input firing pattern. [Figure 8.4](#) graphed the software simulation time for the simulator runs versus the number of events which were simulated. Just as expected, based on how the simulator works, the correlation between the time taken and the number of events is linear. [Figure 8.5](#) shows a graph of the simulation per cycle versus the number of events per cycle. The data shown in this graph is for runs with more than 50k cycles, which is needed to avoid the plateau with the snake runs. This graph also shows a linear relationship between the time per cycle and the number of events per cycle. [Figure 8.6](#) shows that as the number of events increases, the time per event stabilized to a single value. From these graphs, the time the simulator will take to evaluate a network can be estimated based on the number of events, since the average time per event is stable.

The hardware does not have the same strong linear correlation between execution time and the number of events that the software simulator has. [Figure 8.7](#) shows a graph of the hardware evaluation time similar to the previously discussed [Figure 8.4](#), which shows the software evaluation time. With the software simulator, graphing the network evaluation time as a function of the number of events results in a straight line. However, similar graphs of

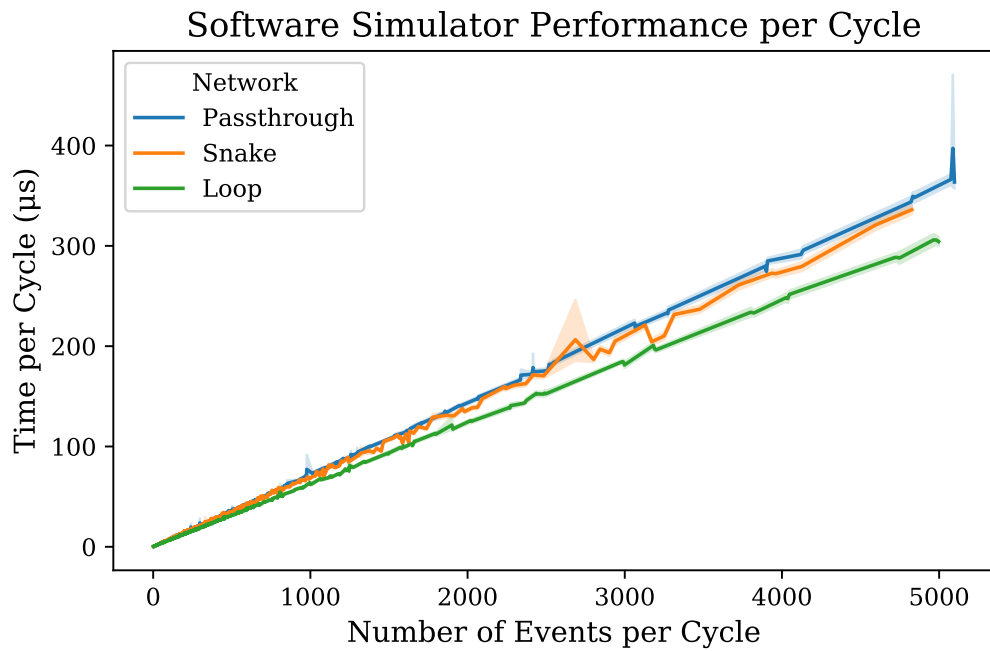


Figure 8.5: Per cycle simulator performance for a varying number of events per cycles.

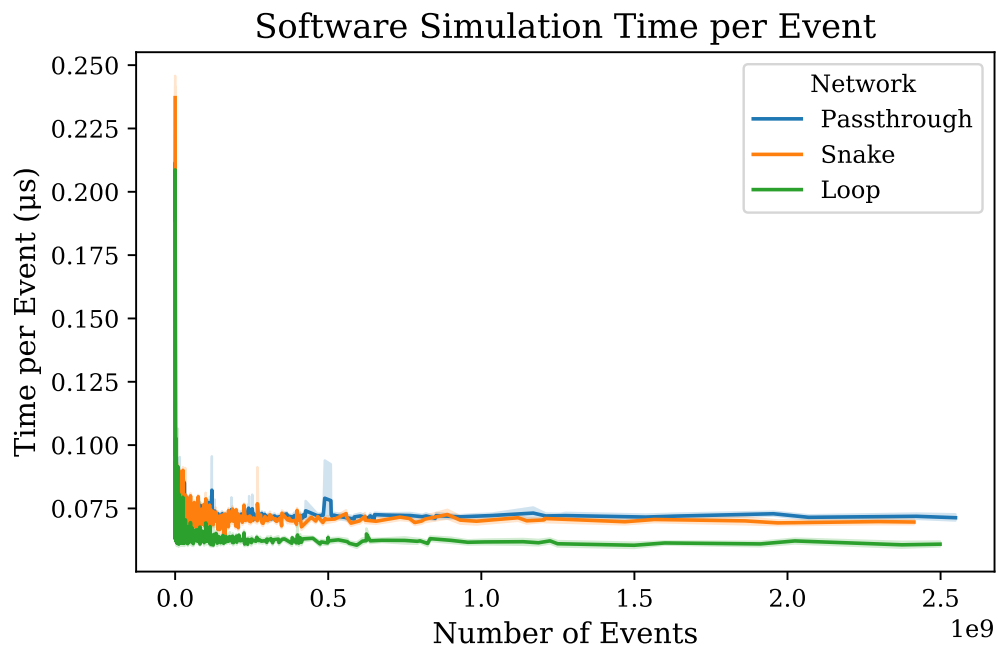


Figure 8.6: Time per simulation event for a varying number of events.

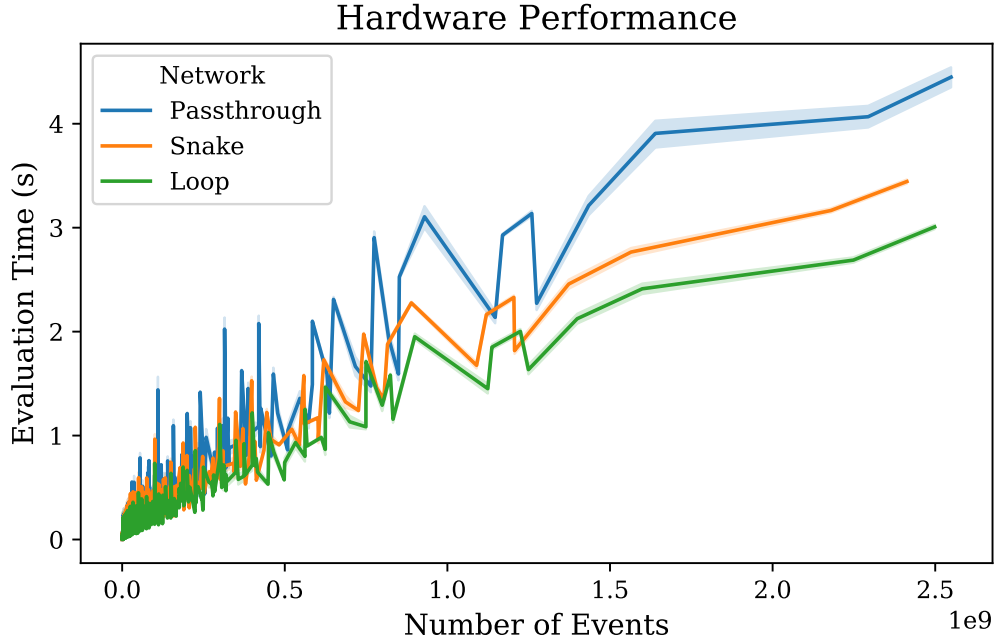


Figure 8.7: Time for the hardware device evaluation for a varying number of events.

evaluation time versus the number of events with hardware results in a jagged upward line. The hardware is able to evaluate multiple events in parallel, so the jagged line is caused by the hardware sizes increasing. The increase in hardware size allows more events to be executed in shorter amounts of time. The line trends upward since the encoding time needed to encoded the input and decode the output events increases along with the network size.

As discussed previously, the hardware has a fixed cycle time of $0.1 \mu\text{s}$, but the performance of the hardware is most limited by the time needed to build, send, and parse network packets.

Figures 8.8 and 8.9 show the time the hardware took to execute based on the number of input fire events for the Single FPGA system and SNACC, respectively. These two graphs only show data from the passthrough runs since these runs have repeated input. As shown by these graphs, there is a roughly linear correlation between the number of input fires and the time required for the evaluation. The ideal line shows that the number of input fires should not be a factor if the inputs could be supplied at the same rate as the hardware is able to run. If the inputs are not supplied as fast as the hardware can run, the hardware has to pause execution and wait for the packets to arrive before evaluation can continue.

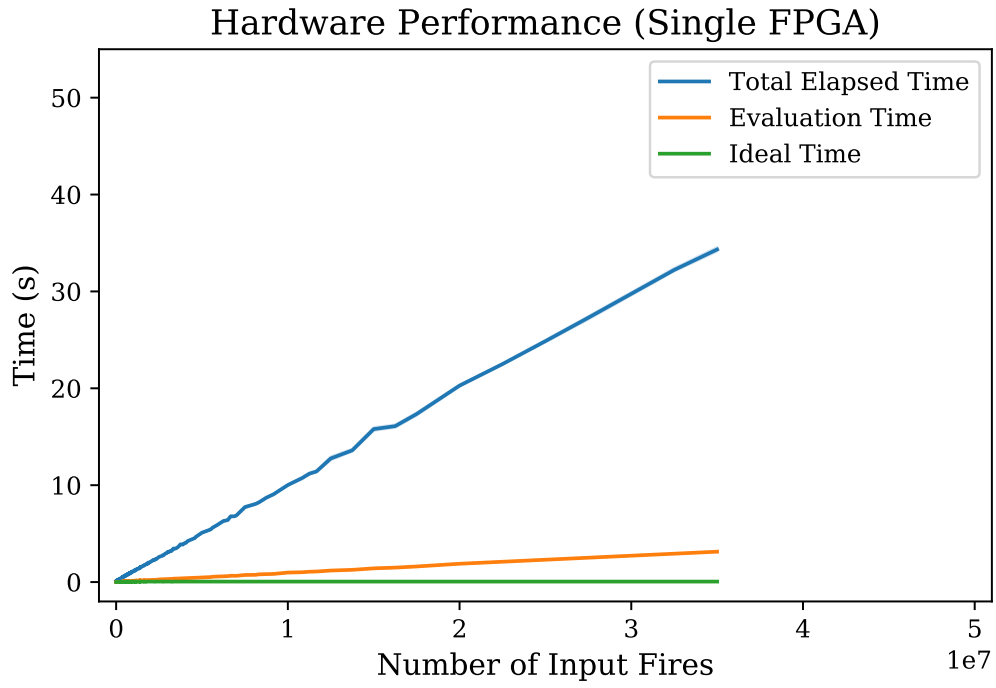


Figure 8.8: Single-board DANNA2 processor evaluation time for a varying number of input fires.

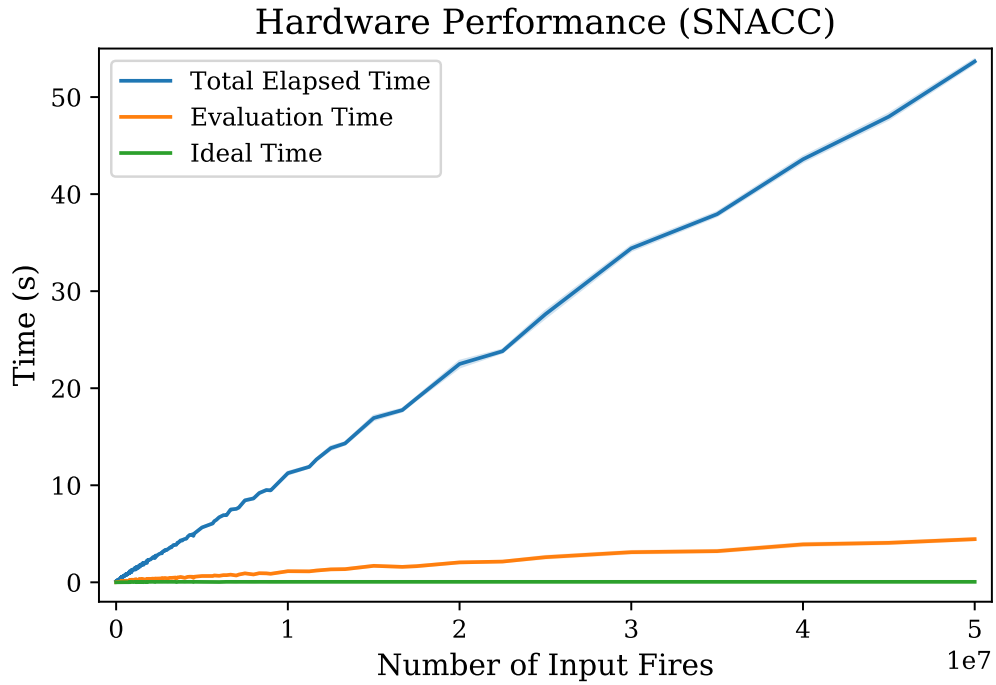


Figure 8.9: SNACC DANNA2 processor evaluation time for a varying number of input fires.

Figure 8.10 shows the average time per cycle for each device when a network is run for a given number of cycles. Figure 8.11 shows the same data, but for only hardware devices and for cycles greater than or equal to 100k. From these graphs, the time per cycle converges to a fixed value for each of the devices. This value of convergence is not the same for every network and input rate, but each does converge, which allows the means of all the runs to converge as shown in these graphs. Although the hardware never reaches the ideal time of $0.1\text{ }\mu\text{s}$, the hardware evaluation time does average to a little under one $1\text{ }\mu\text{s}$. The total hardware average is around $4\text{ }\mu\text{s}$ at 100k cycles and is still decreasing closer to $3\text{ }\mu\text{s}$ at 500k cycles. For short hardware runs, the loading time and packet encoding time dominates the total evaluation time, making the average time per cycle very large. It only takes around 10k cycles for the startup cost to be hidden enough for the hardware's average time per cycle to be lower than the simulator's time per cycle.

As mentioned before, the performance limitation on hardware for networks with a high rate of host input is the time required for the host to provide packets. This limitation can clearly be seen when the total elapsed time is graphed versus the number of cycles, with a line for each of the different numbers of inputs, as shown in Figures 8.12 and 8.13. These figures only show data from the passthrough networks with fires every cycle. Since fire packets arrive every cycle, the total simulation times for the hardware devices are completely driven by the times needed for the host to build and send the input packets. As seen in these graphs, the number of inputs dictates the slope of the line, which in turn, is the evaluation time per cycle for that particular run.

With these performance trends, how does one determine which DANNA2 neuromorphic device will be able to evaluate a given network the fastest? To help answer this question, the mean evaluation time for each of the ten runs per test was computed. Then the mean values were sorted by total evaluation time. Next, for each test run, the DANNA2 device which evaluated the network the fastest on average was declared the winner. Now with the winning networks determined, various graphs could be made showing where the trade-off points are.

Figure 8.14 shows the winner for select parameter sweeps. The graphs are shown as swarm plots so that each data point is displayed. All data points are presented by placing overlapping data points next to each other. This figure shows that there are a few factors

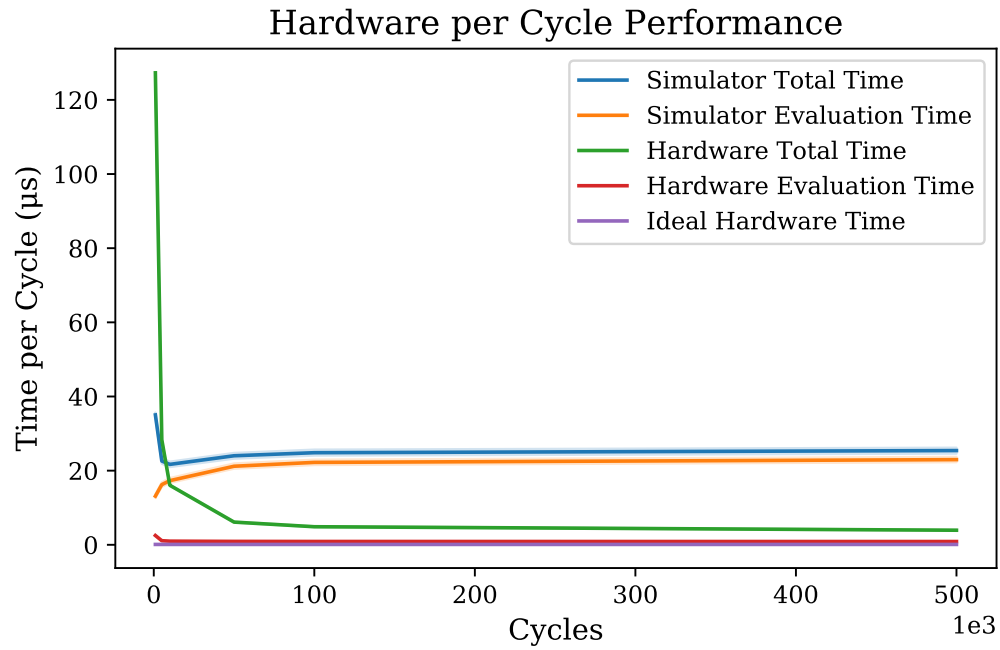


Figure 8.10: The average time per cycle for each device when run for a given number of cycles.

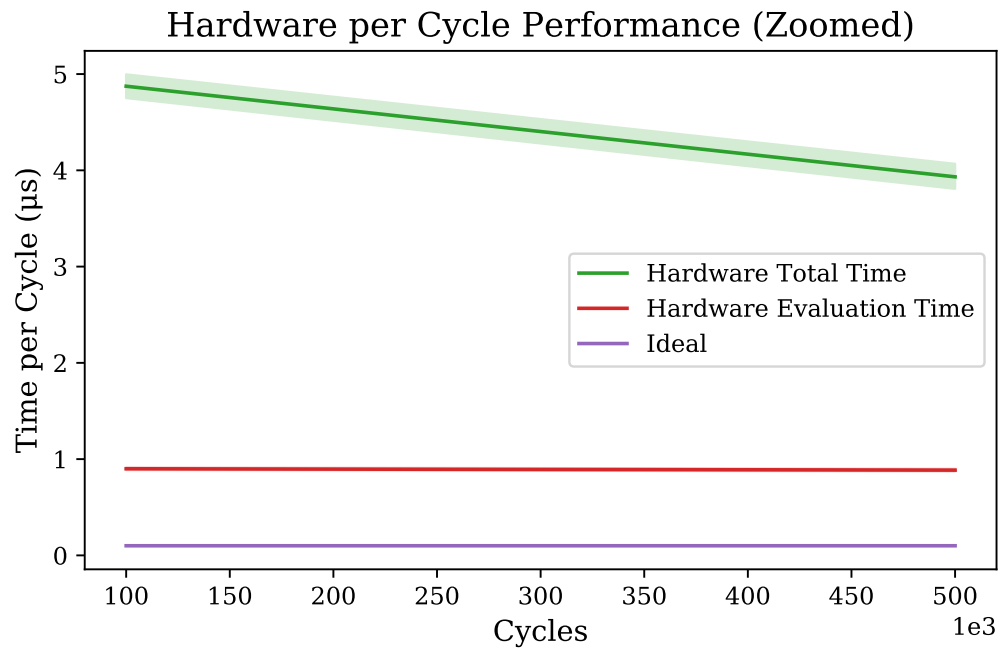


Figure 8.11: The average time per cycle for each hardware device when run for a given number of cycles greater than or equal to 100 k.

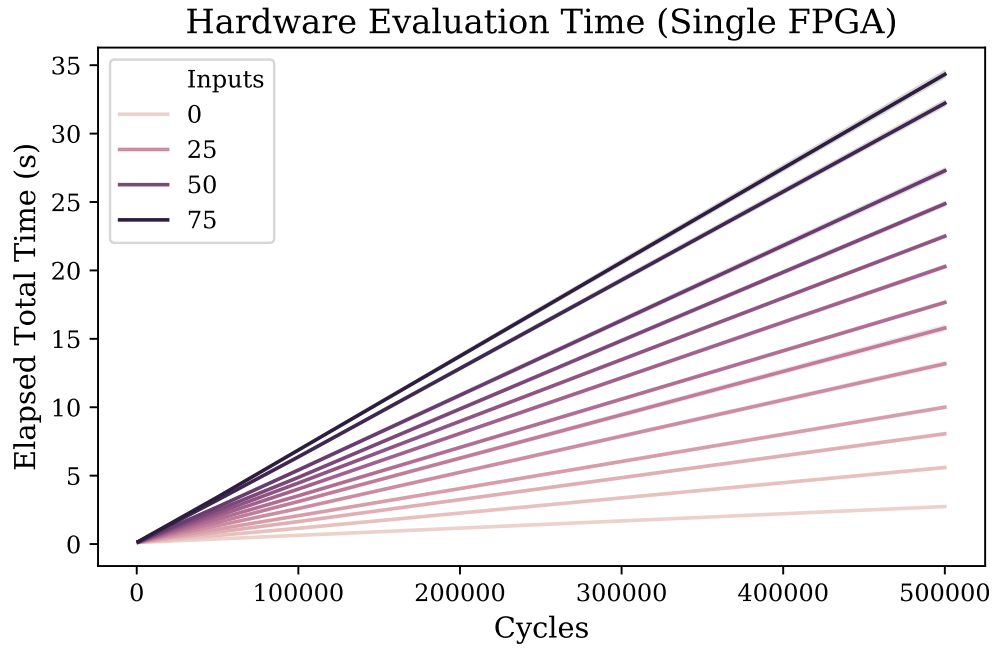


Figure 8.12: The effect the number of inputs has on single-board FPGA evaluation time for passthrough networks with fires every cycle

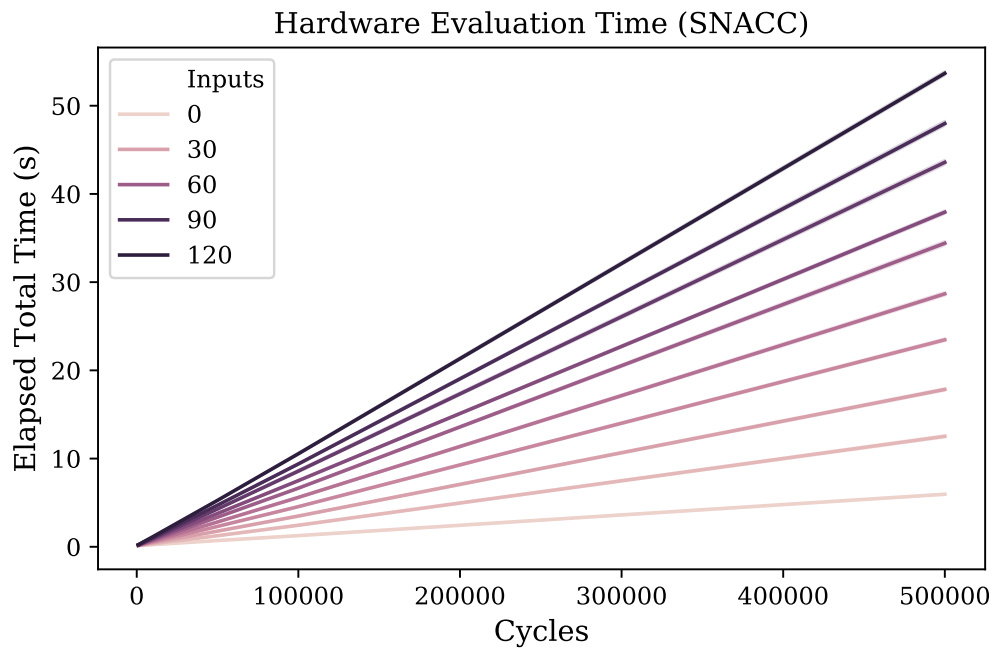


Figure 8.13: The effect the number of inputs has on SNACC evaluation time for passthrough networks with fires every cycle

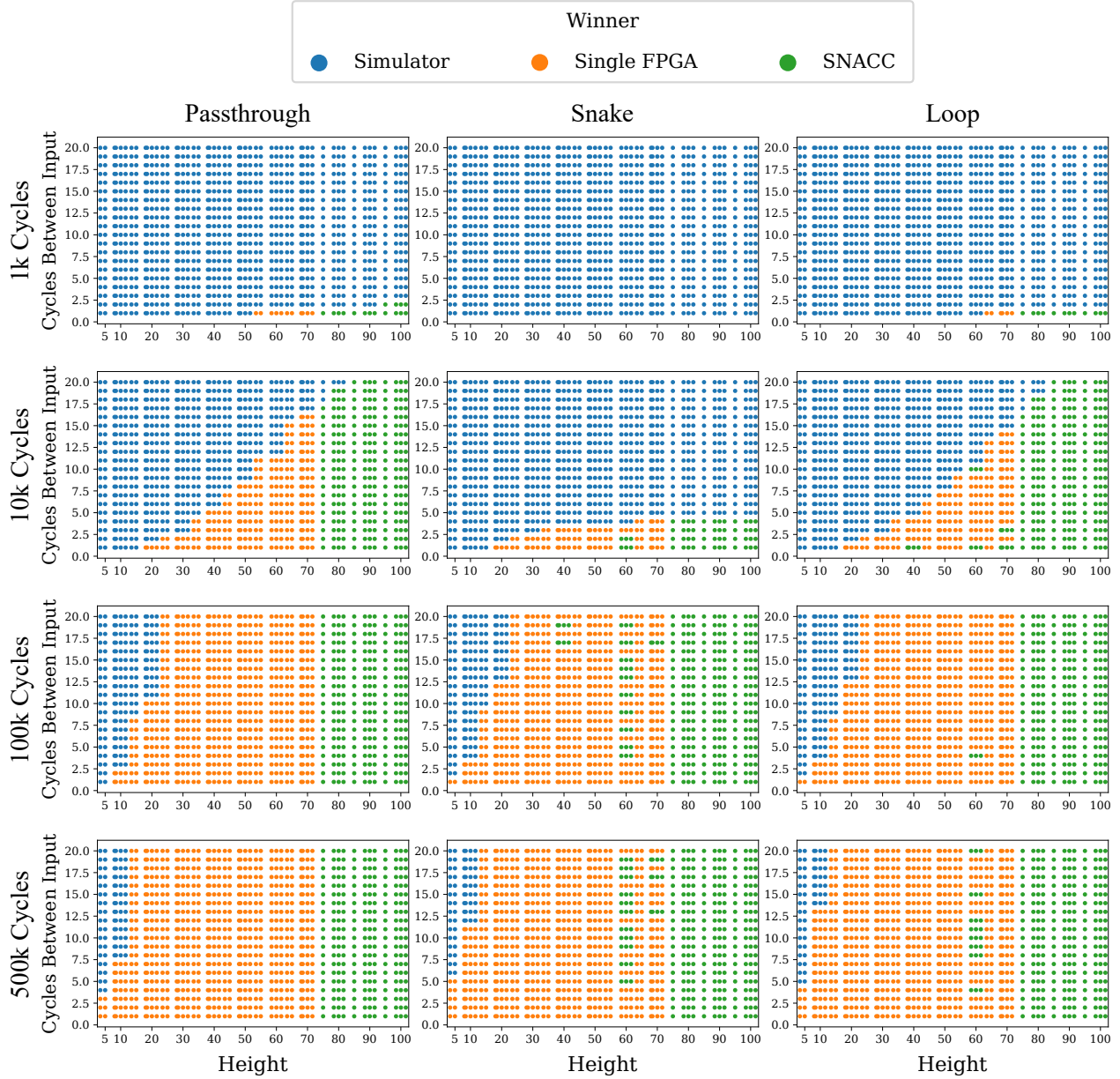


Figure 8.14: The winners for evaluating networks for select network size sweeps for various array sizes and input activities.

which determine the device with the best performance. One main factor is the number of cycles the network was evaluated for. The hardware performs better for more of the test cases as these tests are run for a greater number of cycles. The height of the network plays a key factor in determining if the single-board FPGA or SNACC will perform the best. Typically, the single board system will outperform SNACC, which leaves SNACC to perform the best when the network size is too large for a single FPGA. There is a roughly diagonal boundary separating the software and hardware when the number of cycles is held constant. When the activity in the network is higher, as is the case when there are fewer cycles between inputs, the hardware tends to outperform the software. Likewise, as the size of the neuromorphic array increases, the hardware tends to outperform the software. These graphs also show how many data points were collected for each set of parameters, since the axis corresponds to values which were swept over. The number of dots at each location corresponds to the number of test runs with that value. Since the network sizes tested are different between the simulator, single-board FPGA, and SNACC, some locations have more values than others.

Although height and activity were the parameters specified when creating the tests, they are not the best to use to determine the expected relative performance of the systems. To figure out the best metrics to determine performance, the time estimates for the different devices are explored further. As mentioned before, the simulator's runtime depends on the number of events. More specifically, an equation to find the simulator's runtime is shown in (8.1).

$$\text{Simulator Time} = \sum_{Events} (time_per_event) \quad (8.1)$$

This equation assumes that each event requires the same amount of time. In practice, different types of events could take slightly different times.

Similarly, the equation for the hardware's runtime is shown in (8.2).

$$\text{Hardware Time} = \sum_{Cycles}^{Cycles} (element_clock_period \times input_stall_i) + load_time \quad (8.2)$$

With hardware, the time for each cycle is the element's clock period and any amount of stalling required for the host input to arrive. The hardware also has a large network load time, which occurs at the start of the hardware evaluation.

Since these equations depend on different summations with different variables, setting these equations equal to one another and solving for the trade-off point is difficult. However, multiple observations about the runtime behavior can be made. The hardware will perform better when the number of events is larger compared to the number of cycles. The hardware also must run for enough cycles to overcome the load time penalty. In essence, the hardware will be faster when run for a long number of cycles with a large number of events throughout the run. Input stalls hurt the hardware performance, so recurrent networks where one input fire results in many internal fires will also improve the hardware's relative performance.

In order to create graphs to show the performance trade-off points, the major variables which affect (8.1) and (8.2) must be identified. One key component is the number of events per cycle, which captures the difference between the summations. Another key is the total number of cycles, which needs to be large enough to hide the startup cost of the hardware. The input stall time is also an important factor in (8.2). To capture it, the effective number of firing elements, or the number of internal fires per input fire metrics can be used. The effect that these variables have on the winning device is shown in Figure 8.15. These figures only show the comparison up to 50 k cycles, but the area in which the software is the fastest continues to shrink as the number of cycles continues to increase. The effective number of firing elements is calculated by dividing the number of events by the number of cycles in which there are fires. As seen by this figure, as the number of cycles increases, the initial hardware startup cost is hidden, and hardware starts to outperform the simulator earlier. When only running for 1 k cycles, the software simulator performs the best for networks less than about 1.8 k events per cycle. If the runs are increased to 5 k, then this number decreases to around 400 events per cycle. When run for 10 k cycles, this number is decreased to around 60 events per cycle. At 500 k this number is reduced all the way down to around 14 events per cycle. These performance boundaries are summarized in Table 8.1.

To further explore the trade-off point, Figure 8.16 shows all the data points with the number of internal events on the x-axis and the internal fires per input on the y-axis. Figure 8.17 shows the same graph but zoomed in. Since the x-axis is now events instead of events per cycle, the number of cycles is included in the axis. The maximum internal events in which the simulator is fastest is at 6.3 M events; however, with a larger number of

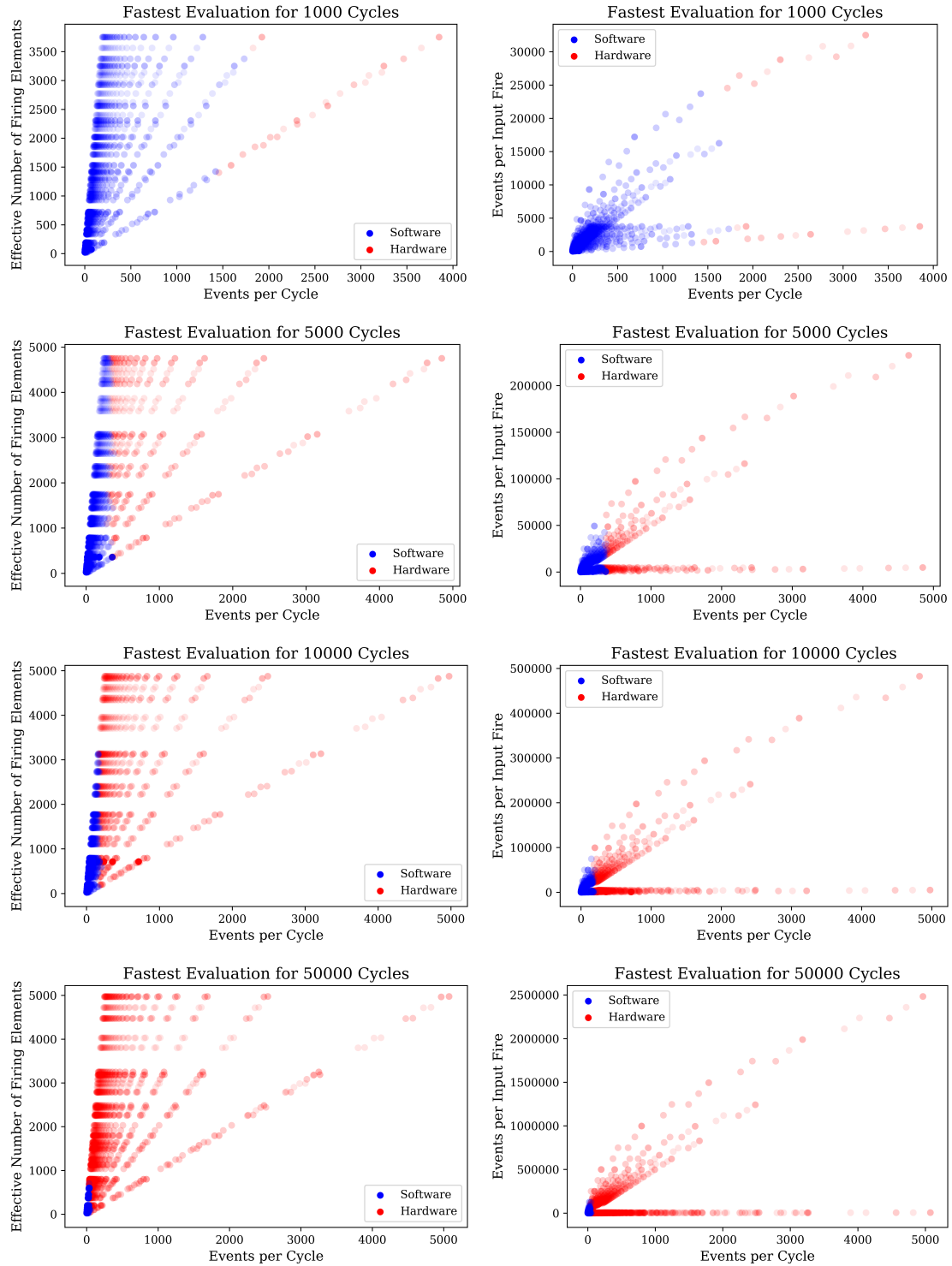


Figure 8.15: Graphs showing the winning device type based on key variables for a varying number of cycles between 1 k and 50 k.

Table 8.1: Maximum Events per Cycle Where the Simulator is Fastest

Cycles	Max events per cycle
1 k	1733
5 k	378
10 k	208
50 k	55
100 k	37
500 k	14

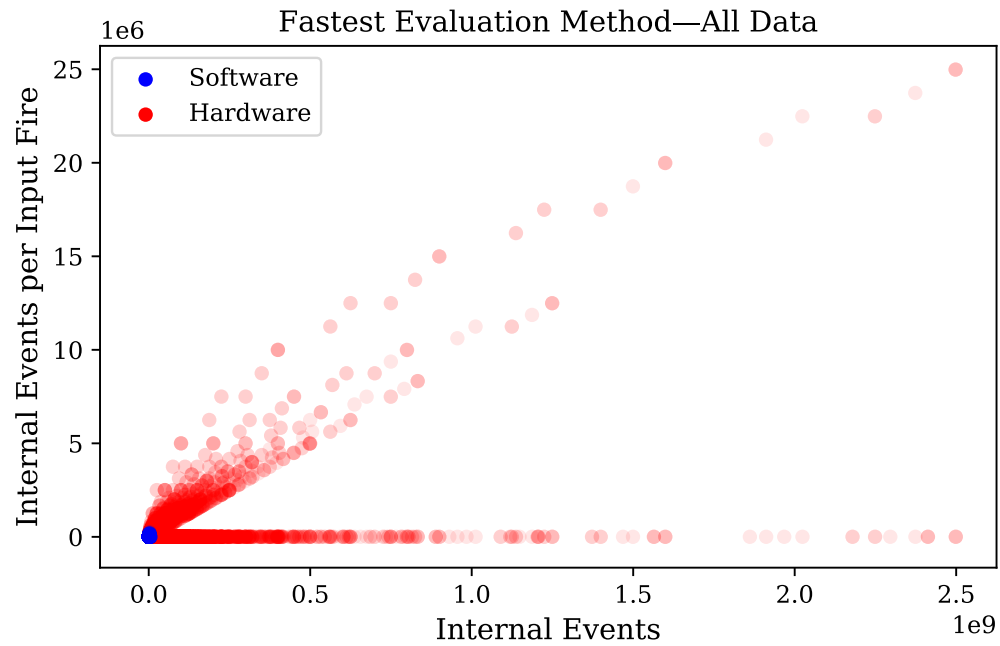


Figure 8.16: Fastest evaluation method for all data.

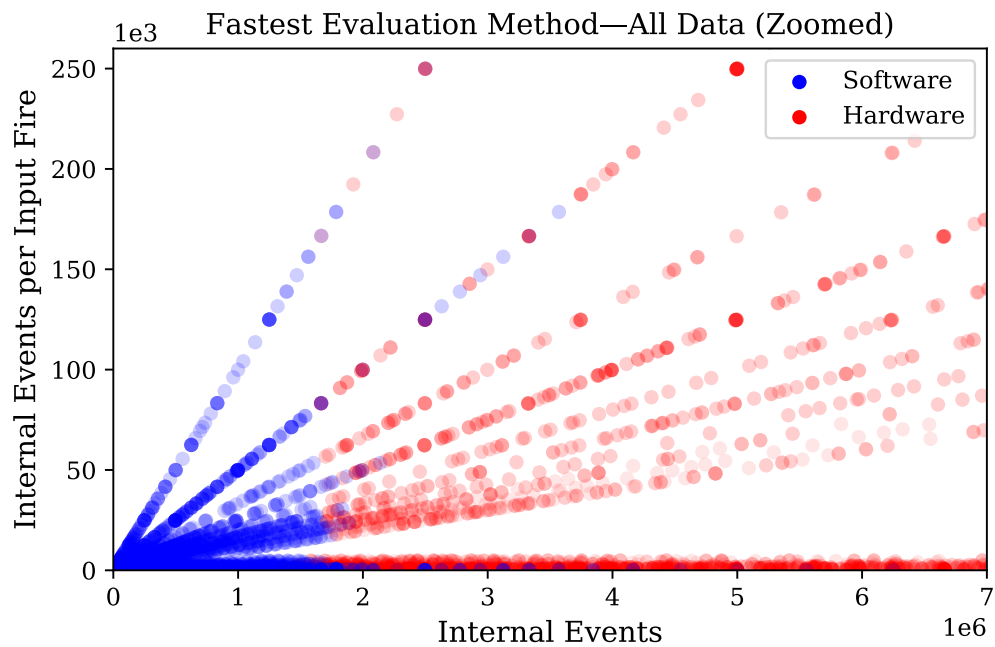


Figure 8.17: Fastest evaluation method for all data (zoomed).

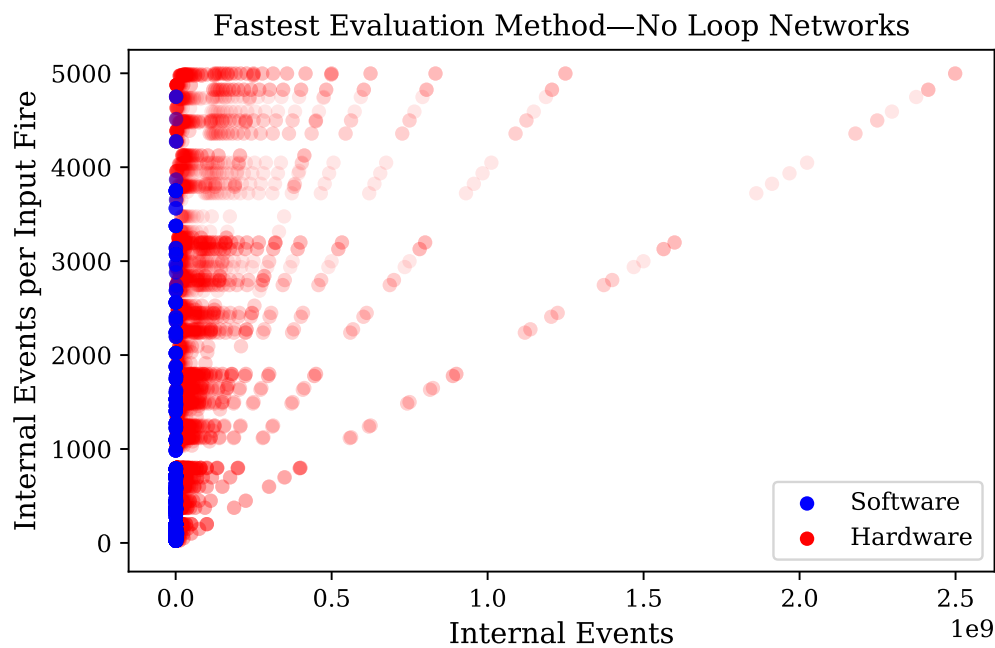


Figure 8.18: Fastest evaluation method for non-loop networks.

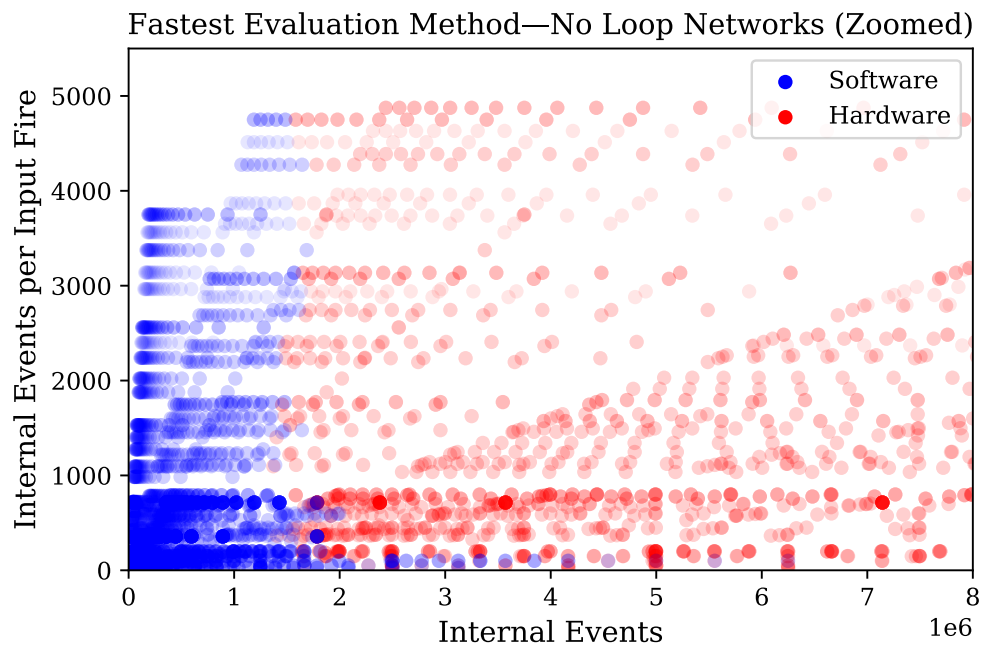


Figure 8.19: Fastest evaluation method for non-loop networks (zoomed).

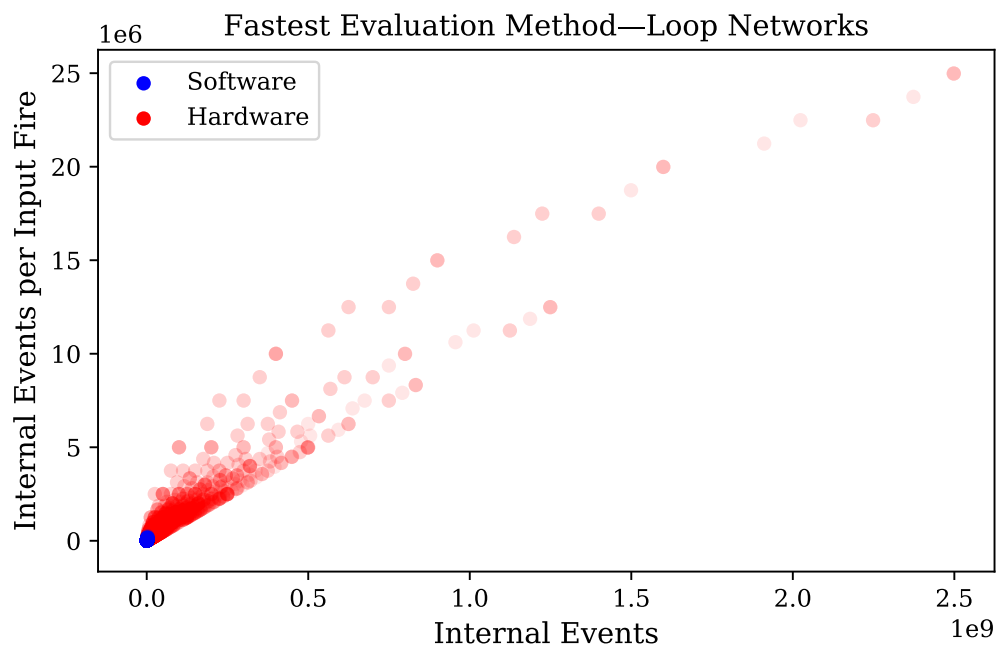


Figure 8.20: Fastest evaluation method for loop networks.

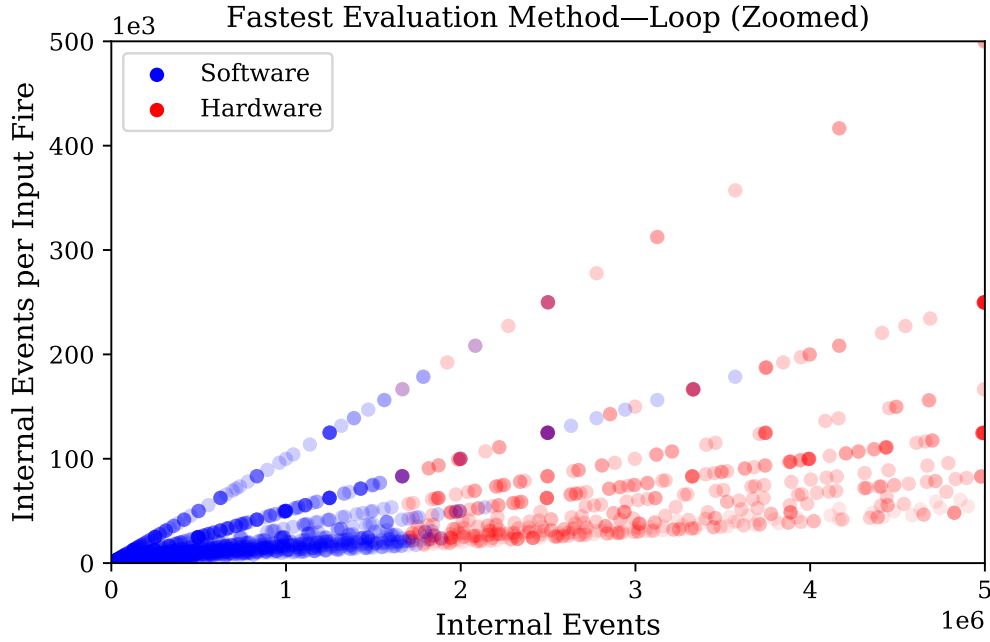


Figure 8.21: Fastest evaluation method for loop networks (zoomed).

internal events per input fire, the hardware starts winning around 2 M events. The full graph shows that the hardware wins over the majority of the search space; however, many of the ways DANNA2 networks are currently used fall within the area where the simulator performs best. Current applications are usually run for under 1 k cycles. This will likely change as the hardware supports running larger network sizes quickly.

The loop network test cases account for the data points with the higher internal events per input fire as shown in [Figures 8.20](#) and [8.21](#). Therefore, it is interesting to look at the non-loop networks by themselves and in more detail, as shown in [Figures 8.18](#) and [8.19](#). The zoomed-in graph shows there is an almost rectangular cutout where the hardware starts to perform better than the software simulator. At around 2 M internal events, the hardware performs best for all the test runs with over 100 internal events per input fire. Below 100 internal events per input fire, the software simulator still performs best until around 6.3 M internal events, after which the hardware always performs best.

In conclusion, this section looked at the performance of both the software simulator and the hardware devices using test networks specifically designed to analyze these systems.

The equations which determine the evaluation time for each device were discussed and experimental data was shown to back up these equations. The key factors which contribute to determining the best performing device were discussed, and the trade-off point between hardware and software was explored using the collected data and these key factors. Although the specific neuromorphic job will need to be benchmarked on each device to determine exact performance numbers given the network design and element activity, this section should give an idea of the type of factors which contribute to the evaluation time of each system.

8.2 EONS

As mentioned in [Section 3.1.2](#), Evolutionary Optimization of Neuromorphic Systems (EONS) is one of the main methods used by TENNLab to train neural networks to perform a wide variety of tasks. This section looks at how EONS was modified to support SNACC and at the performance when hardware is used to perform EONS training.

8.2.1 GenGraph support for SNACC

EONS uses a general graph (GenGraph) representation to perform its genetic algorithms. EONS passes the GenGraph network to the device driver to convert it into a valid network for that device, which in this case is a valid DANNA2 network. Another function will do the reverse by taking a device network and converting it into a GenGraph network. The forward conversion is used to load a new network onto the device, and the reverse operation reads back any changes which occurred to the network as a result of the simulation. With the addition of the SNACC implementation, the conversion function from GenGraph to the DANNA2 network has to be changed to account for the special connectivity restrictions between sub-arrays in SNACC.

Specifically, with SNACC, links to neighboring boards have a minimum of one additional clock cycle to cross the boundary. This added clock cycle can range from one to the maximum synapse delay and is configurable in the hardware design. The lower the delay, the slower the effective frequency of the network cycles becomes, since the FPGAs will be waiting for information from the neighboring boards before continuing. The second connectivity

constraint is that network connections that would need multiple board jumps to make the connection are not allowed. This restriction prohibits diagonal connections at the corners of the local networks. The FPGAs only have inter-board connections in the four cardinal directions, so a diagonal connection at the edge would require two jumps to reach the destination.

In order to make EONS adhere to these connection restrictions, the GenGraph to DANNA2 network converter was modified to enforce these restrictions when the network is converted. Additional DANNA2 configuration parameters were also added to specify the delay amount and the number of sub-arrays that make up the SNACC system. When the GenGraph network is converted to a DANNA2 network, the global network coordinates are now converted to device and local tuples, where the device is the coordinates of the FPGA and local is the location of the element within the grid on that particular FPGA. If the synapse crosses one connection and the synapse delay is less than the delay parameter plus one, then the synapse delay is changed to the delay value minus the synapse delay parameter plus one. If the connection would require more than one device crossing, then the synapse is dropped. Both of these checks guarantee that the resulting DANNA2 network can be mapped onto SNACC.

There should not be any major side effects from using this method to map EONS networks onto SNACC. These changes are made as EONS is creating networks to learn the application. The fitness evaluation is done on the changed version of the network. When EONS reads networks back to GenGraph from DANNA2, the changes made to the network are applied to the GenGraph network and EONS creates additional networks from the modified networks. Additionally, only a few of the synapses are affected in a given network. In one test case, which had 1202 synapses, 44 synapses had their delay changed and three synapses were dropped.

Other options were considered when designing this setup. One option would be to change EONS and the GenGraph specification so that invalid synaptic connections are taken into account and not generated. This approach seems unnecessary and would involve changing the workings of EONS. Another approach would be to reject any potential networks which could not be directly mapped onto SNACC. This option is not ideal since doing so would require multiple generation attempts before a valid network was generated.

8.2.2 Performance

EONS has successfully been run using the event simulator, single-board FPGA, and SNACC. The resulting fitness values from each epoch matched across all the different devices when EONS was configured to generate valid networks with SNACC’s border crossing restrictions. TENNLab’s Neuro Device Factory can specify the use of both the single-board FPGA and SNACC setups in a single EONS run. The Neuro Device Factory can also be used to specify any combination of simulators and hardware devices to use for EONS.

Results were collected by running EONS with two example neuromorphic applications with different network sizes. The first results were collected with the pole balancing application, which generated 15×10 DANNA2 networks. These networks were run for 100 network cycles each to get output spikes to make the next application decision. When training this application on hardware, the hardware took longer to train than when the same training was performed on a Threadripper CPU core. [Figure 8.22](#) shows the elapsed real time it took to run each epoch using the event simulator, single-board FPGA, and SNACC. Performance on the hardware is not terrible, but the hardware performance is poor when compared to the event-based software simulator. In order to understand why the epochs were slower on the hardware than on the simulator, additional data was collected and a histogram was created to show the duration of each simulation call for each implementation. This histogram is shown in [Figure 8.23](#). On average, each simulation call took longer on hardware than in software.

The second application tested was an object targeting task called Robonav. (See [Section 8.6.1](#) for more information on the task.) This training run was scaled up from the usual five Light Detection and Ranging (LIDAR) inputs to include 93 of these inputs and use 100×50 DANNA2 networks. Initially, the networks were only evaluated for 100 cycles and a histogram of these durations is shown in [Figure 8.24](#). This figure looks very similar to the duration histogram for the pole balancing application. Since the hardware run time depends on the number of simulation cycles, its performance is roughly the same. The performance of the simulator is also very similar since the network did not have a substantial increase in the number of events.

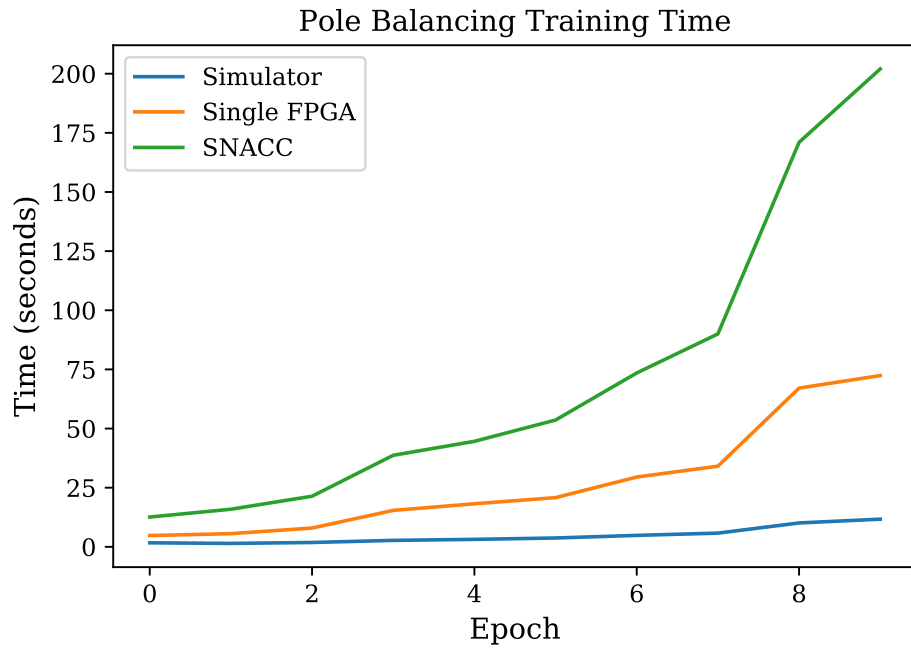


Figure 8.22: Evaluation time versus epoch number for the pole balancing application training.

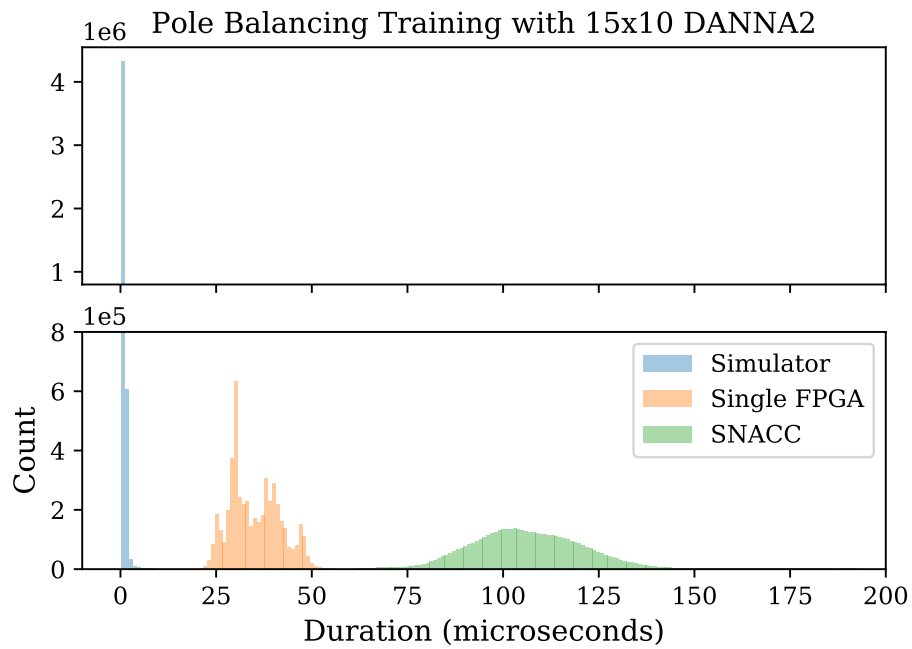


Figure 8.23: Duration histogram for each simulation call for training the pole balancing application. The network was run for 100 cycles for each network evaluation. Each bin in the histogram is 1 microsecond wide.

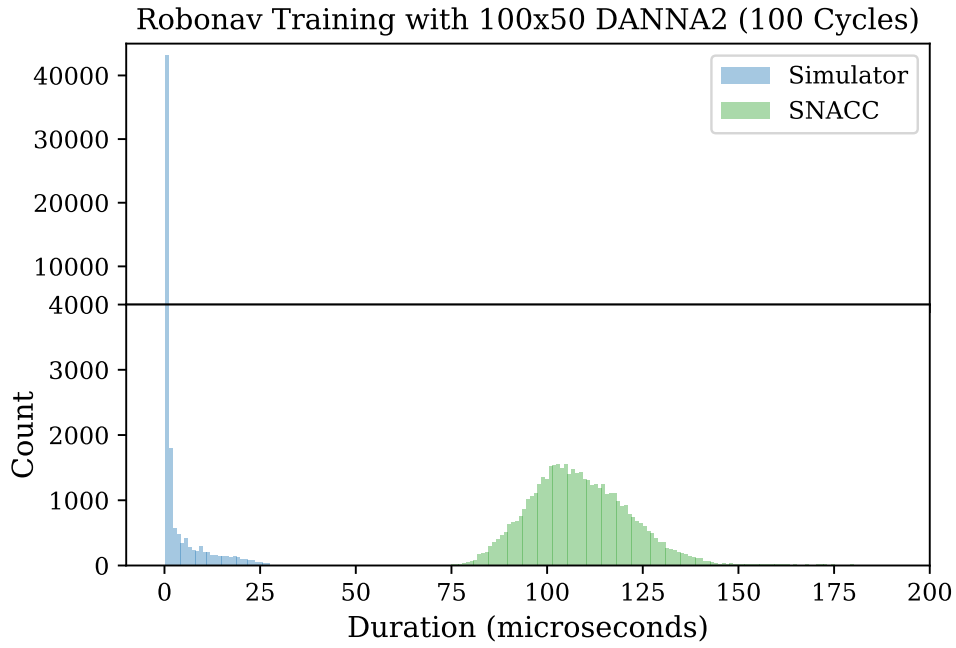


Figure 8.24: Duration histogram for each simulation call for training the Robonav application. The network was run for 100 cycles for each network evaluation. Each bin in the histogram is 1 microsecond wide.

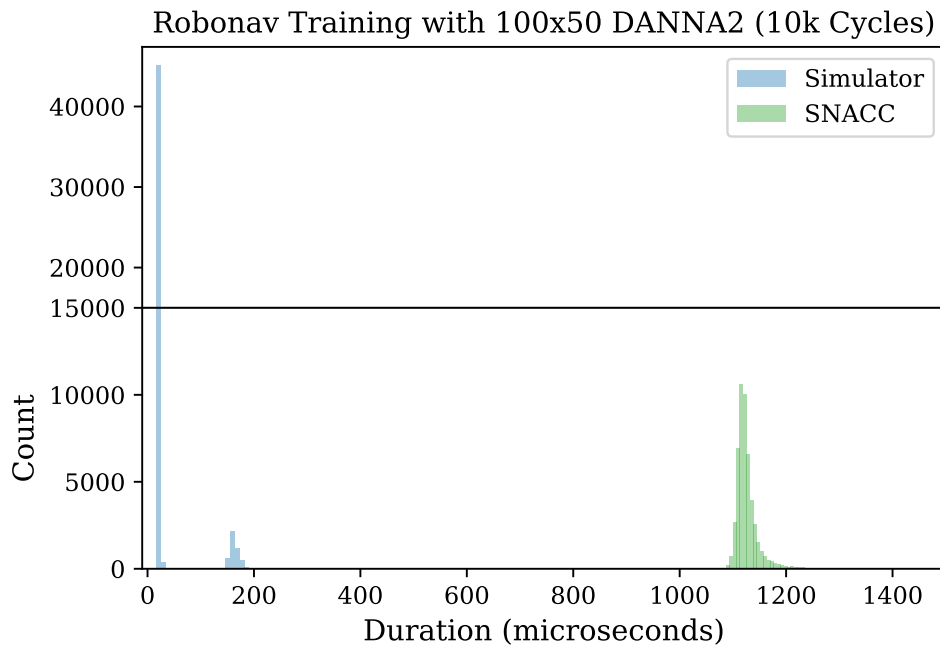


Figure 8.25: Duration histogram for each simulation call for training the Robonav application. The network was run for 10k cycles for each network evaluation. Each bin in the histogram is 10 microseconds wide.

[Figure 8.25](#) shows the durations of a similar training run, but this time the networks were run for 10k cycles instead of only 100 cycles. Once again, the performance of EONS is not as good on the hardware as it is in the software simulator. The next section, [Section 8.2.3](#), discusses this in more detail and proposes a way to improve the hardware performance of EONS.

8.2.3 Streaming Training

As seen in [Section 8.2.2](#), the performance of EONS on the hardware is significantly worse than running EONS on a single CPU core. Typically, computer systems are able to simulate multiple networks in parallel by using multiple CPU cores or large computer clusters to use more compute resources and reduce the training times. Currently, only a few hardware systems are connected to a machine at a time, so many parallel hardware evaluations are not possible because of the limited number of hardware devices available for use. However, even if multiple hardware systems were built together into a neuromorphic cluster, the performance per processor would still be less than that of CPU cores. [Though 8.1](#) shows that the hardware is capable of better performance, low performance of EONS training on hardware is observed. This is related to the types of networks typically built by EONS for applications and the way EONS uses the hardware systems to evaluate networks.

Applications which use EONS generally target network sizes with only around 150 elements. Additionally, these networks are typically only run for hundreds of cycles. Thus, the performance of these networks clearly falls within the region where the software simulator will outperform the hardware. Part of the reason small networks with few elements are used for EONS is to help with convergence to a viable network. The small networks have smaller search spaces than the large networks, which helps the networks converge to a network with a good fitness score. Additionally, smaller networks with shorter runtimes simulate faster in the event simulator, which allows EONS to evaluate more networks and train for more epochs to arrive at a better solution. So part of the reason current applications trained with EONS perform better with the software simulator is because the network sizes were chosen to help EONS converge quickly. Some of the gap in performance will be removed if hardware is used to make training with larger networks that run for longer times feasible. Thus, as

applications start to train larger, longer-running networks, the hardware performance will start to overtake the software simulator performance and allow these more complex networks to be trained.

The other limitation with EONS and hardware devices is how EONS uses the hardware. The current way EONS is run on hardware results in a huge bottleneck in the training process, where the training is ping-ponging back-and-forth between running on the CPU and running on the DANNA2 device. This causes neither the host nor the DANNA2 processor to operate anywhere near their maximum potential. The host is used to generate new populations of networks to evaluate. These networks are then sent to the hardware one at a time. The host sends the network configuration packets to the hardware, after which the host sends the input packets. The host then idles while waiting for the DANNA2 processor to send back the results from the run. After the hardware sends the output packets back to the host, the host then sends the next set of input packets, or sends a new network to load.

With this setup, there is idle time on the host as it waits for the network to run, and also idle time on the network while the host generates new input packets or networks. Additionally, by waiting for a response before sending the next packets, the buffers in the communication pipeline empty between every operation and the long latency of the host-to-array communication is added to every operation. Even if the host execution took no time (which it does not), this use of hardware would still be inefficient. The round trip latency for host communication through Xillybus is relatively large; however, the throughput of the system is high, allowing the system to handle a constant stream of packets. After the output packets are sent to the host, the input buffer is empty and the hardware is just sitting idle, which means the start-up latency cost will have to be paid again when new input packets are sent.

A solution to this problem will be to conduct EONS training so that the hardware is treated as a streaming data pipe instead of as a blocking function call. Since EONS works on populations of networks, there are many networks which need to be evaluated. EONS could generate the network load commands and input fires in parallel, and then add them to a queue to be sent to the neuromorphic hardware. Other EONS threads could read the output from an output queue, and then process the output to create the next network and input

packets. This would allow EONS to use multiple CPU threads to generate a stream of input for the neuromorphic array; the array would then process the stream in order. Additional CPU threads could then process the output and prepare new packets ready to be streamed in. The CPU thread would also have to keep up with the application state for each of the in-process evaluations. This state will need to be matched with the correct output and updated to create the correct new input in order to continue training.

This streaming data pipe setup for training would also allow easy use of multiple hardware devices. Currently, if multiple hardware devices are added, each one is used in the same inefficient blocking method. However, if the efficient streaming method is used, then the EONS threads could split the array input between each of the hardware devices and additional EONS threads could read the output from each device. This setup would result in either the hardware systems being fully utilized as they process the data streams or the host would be fully utilized creating and parsing packets to and from the hardware. Of course both of these options could occur at different points in the training, but ideally the hardware will be running at maximum throughput and the host system will be able to keep the communication streams full. With the hardware systems fully utilized, they will be able to outperform the per-cycle performance of network evaluation with larger, longer-running networks. The exact speed-up possible with streaming training will depend on the application being trained and on the parameters and activity of the networks being evaluated.

The Threadripper host used in SNACC is well-suited for the kinds of tasks required for the host with streaming training, since the Threadripper has many threads at its disposal to create and process the streams to and from the hardware, as well as being capable of maintaining threads to update the application state. Details on the Threadripper host system can be found in [Section 7.4](#). Restructuring EONS to operate in this streaming manner is future work currently being explored by TENNLab.

8.3 Reservoir

Another application for neuromorphic systems is as the reservoir for reservoir computing [\[77\]](#). The basic design of a reservoir computer is shown in [Figure 8.26](#). To perform reservoir

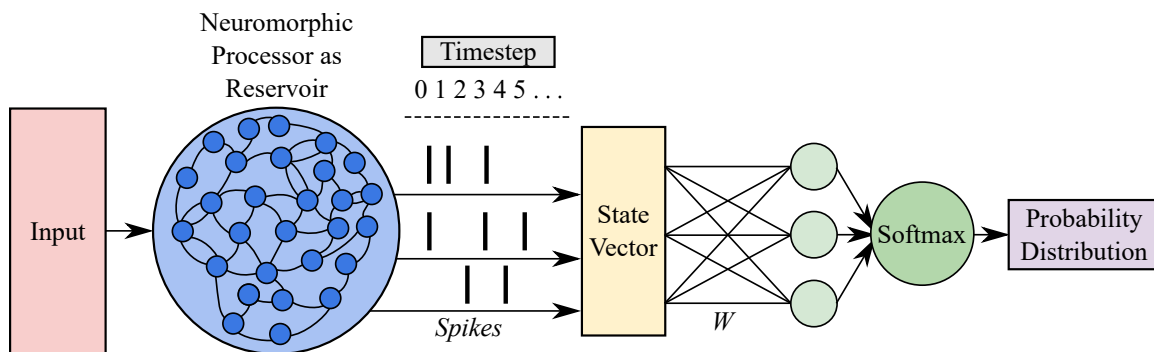


Figure 8.26: Diagram of reservoir computing. Input is passed into the reservoir, then after a predetermined amount of time, the reservoir is sampled and the sample is passed into the readout layer as a feature vector [77].

computing, the first step is to pass the input into the reservoir, which can be an actual liquid reservoir or a recurrent spiking neural network. The reservoir performs a non-linear transformation on the input spikes to generate the output spikes. These output spikes are a feature vector generated by the reservoir in response to the input which can then be used by a readout layer to generate the final output. The readout layer is trained with the feature vectors and the expected output to learn the correct classification. The complete system then works by providing input to the reservoir, the reservoir transforming the input into a feature vector, and the readout layer transforming the feature vector to the final output.

There are two main steps to use DANNA2 as a reservoir in TENNLab’s reservoir learning framework. The first step is to generate a DANNA2 network to be the reservoir. This network can either be generated randomly or EONS can be used to try to generate a reservoir with better features for the task. The next phase involves training the readout layer. In this phase, the reservoir network remains unchanged and is fed the raw input to generate the feature vectors. These feature vectors and the expected output are used with the backpropagation algorithm to train the readout layer.

The DANNA2 hardware systems have great potential to speed up the readout layer training phase. Loading new network configurations onto the hardware is relatively time consuming since it involves sending a load packet for each element in the array. Using an already loaded network to convert inputs to outputs is fast, especially if the inputs can be streamed into the hardware without letting the communication pipes empty. This means that

reservoir can use the hardware in a streaming method as previously discussed in [Section 8.2.3](#). This allows the hardware to be used at its full potential, only having to wait for the input fire packets to arrive. Additionally, the startup cost will only have to be paid once at the start of training. After starting, the hardware will continually transform input spikes into feature spikes.

Initial testing shows that hardware is able to speed up the training of readout layer for reservoir computing. [Figure 8.27](#) shows the training for a reservoir readout layer to learn a classification task. The reservoir network used in this test was a randomly generated 35×20 DANNA2 network. The training was conducted with the software simulator, a single-board FPGA, and the SNACC system with multiple added communication delays. All of the hardware systems, with the exception of SNACC with an added delay of zero, were able to outperform the event-based software simulator. As expected based on the effect of the delay cycle previously discussed in [Section 7.5](#), the SNACC performance increases as the delay increases, up to a maximum delay of six. The single-board FPGA implementation performs the best, as expected based on the analysis in [Section 8.1](#).

The initial testing of DANNA2 hardware to train the readout layer for reservoir computing shows that the hardware implementations can be used to reduce training time, even for relatively small reservoir sizes. Based on the analysis of [Section 8.1](#), many factors go into determining the fastest evaluation method for DANNA2 networks. However, the reservoir use case has great hardware performance potential since networks do not have to be reloaded and input can be streamed to the hardware. Additionally, as the reservoir networks become larger and have more activity, the hardware will have an increasingly greater performance advantage over the software simulator. Reservoir networks also have a potentially greater number of events in the networks, which will result in better relative performance for the hardware compared to the software simulator. DANNA2 hardware can also be used while running the reservoir computer after the readout layer has been trained. The hardware will still perform well since it is doing the same operation it did during the readout layer training.

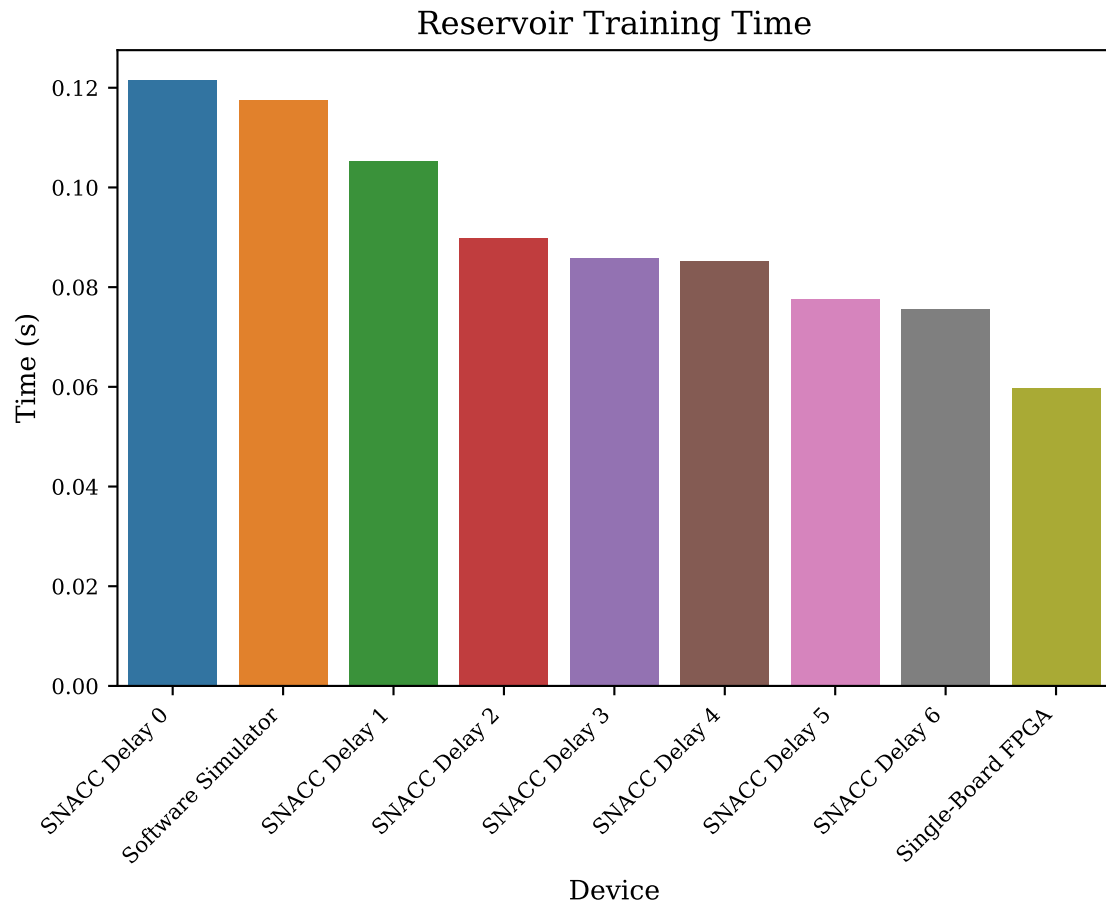


Figure 8.27: Graph of reservoir training time for different DANNA2 devices. The DANNA2 reservoir used for testing was 35×20 elements.

8.4 New DANNA2 Features to Support Graph Algorithms

Because neural networks essentially evaluate a graph of neurons and synapses, it is possible to use neuromorphic hardware to implement other more traditional graph algorithms. In [41], Hamilton et al. demonstrate how the nonlinear dynamics of Spiking Recurrent Neural Networks (SRNNs) can be used to implement low-level graph operations. These low-level graph operations can then be used to solve complex traditional graph problems.

With DANNA2 hardware designed as described in [Section 3.3](#), it is possible to create DANNA2 networks which can be used to find the length of the shortest path within a graph. The way to do this is quite straight forward and presented in [41]. The vertices of the graph are converted into neurons and the edges of the graph are created as synapses. If the graph edges are undirected, then two synapses are created, one for each direction. This is required since each synapse is unidirectional. The synapse weights and neuron thresholds in the graph are set up such that every synapse has enough weight to cause the postsynaptic neuron to fire. Different network edge weights are represented in the graph by changing the delay of the corresponding synapse. A fire on the input neuron will propagate through the network, and the time that the output neuron takes to fire is the length of the shortest path. If the graph is unweighted and each synapse has a delay of one, then the number of cycles the spike takes to propagate through the network is equal to the number of node visits required to traverse the graph. With weighted graphs or synapses with a greater delay, the length of the shortest path can be computed as a function of the time of the first output fire.

There are some limitations to using DANNA2 grid for graph algorithms; the graph network must be able to be mapped directly onto the grid network. This limits the connectivity patterns and the edge weights which can appear in the graph. In the future, a graph network compiler could be used in conjunction with sparse DANNA2 networks to allow for the mapping of a greater variety of graph networks; however, two easy-to-map graph problems were chosen for initial design and testing.

Both of the chosen graph problems are grid-based and map easily to the grid coordinates of DANNA2. The first graph problem is an undirected graph which represents a maze;


```

Create a DANNA2 network with the same dimensions as the graph.

For each vertex in the graph:
    Add a neuron with vertex coordinates, threshold of 0, and refractory of 1.

For each vertex in the graph:
    For each edge connected to the vertex:
        Add a synapse to represent the forward connection
            with a weight of 1 and a delay of 0.

        Add a synapse to represent the backward connection
            with a weight of 1 and a delay of 0.

Set the source of the graph as the input neuron.

Set the sink of the graph as the output neuron.

```

Figure 8.28: Pseudocode for maze graph to DANNA2 network.

by solving the shortest path graph algorithm, the maze is solved [73]. The second graph represents information about roads and intersections in a city [74]. In this problem the graph is a directed graph with edge weights representing the time that would be taken by traveling down the street and waiting at the stoplights. The shortest path represents the fastest path through the city.

Programs were written to generate graphs of any size for each problem, and also to convert the graphs into DANNA2 networks. Generating a DANNA2 network from the undirected maze graph was straightforward. Figure 8.28 shows the pseudocode for the conversion.

Generating the DANNA2 network for the directed city graph is very similar. The pseudocode for this is shown in Figure 8.29. The key difference is that only one synapse is connected per edge, and the delay value is set equal to the edge's weight. However, in practice, the domain of the edge weight is much larger than the domain of the delay values. For DANNA2 hardware, the delay values are constrained to the integers between zero and 15. Because of the delay value constraint, a conversion function is needed to round the edge's weight to the nearest delay value. This edge weight rounding means that the length of the shortest path will be found with some rounding error. If more delay values were available, this error would be reduced. If no rounding is needed, then the exact length of the shortest path can be found.

```

Create a DANNA2 network with the same dimensions as the graph.

For each vertex in the graph:
    Add a neuron with vertex coordinates, threshold of 0, and refractory of 1.

For each vertex in the graph:
    For each edge connected to the vertex:
        Add a synapse to represent the connection with a weight 1
        and delay equal to the to the edge weight.

Set the source of the graph as the input neuron.

Set the sink of the graph as the output neuron.

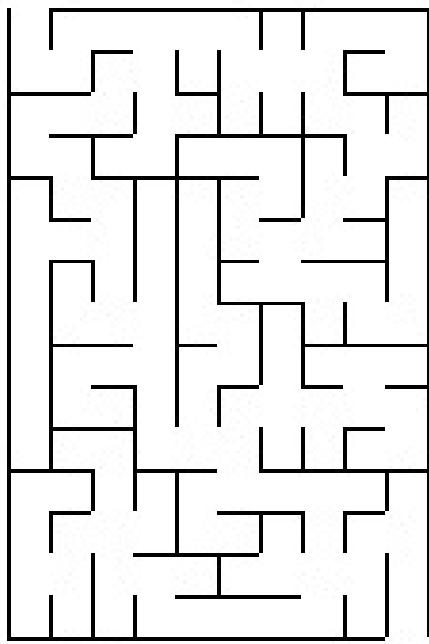
```

Figure 8.29: Pseudocode for city graph to DANNA2 network.

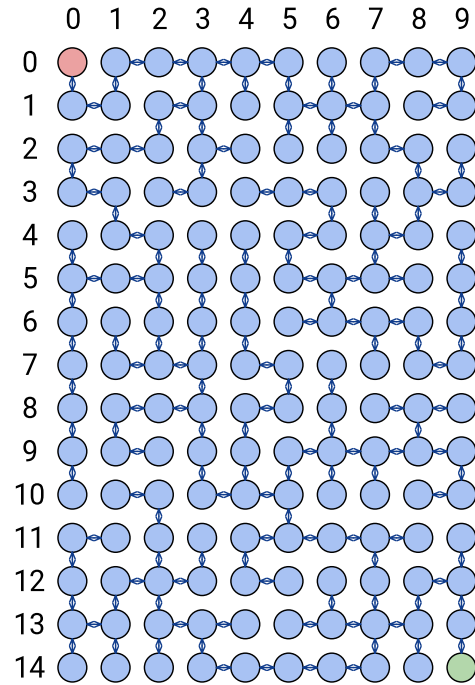
After converting the networks, DANNA2 was used to find the length of the shortest path. The DANNA2 networks are able to find the length of the shortest path, and their value matches that of traditional algorithms used to find the length of the shortest path. The city graph networks are able to match the shortest path found by traditional algorithms when the edge weights can be directly mapped to delay. When the weights must be rounded, the length of the shortest path is accurate within the rounding error. [Figure 8.30](#) shows a maze and the resulting DANNA2 network, and [Figure 8.31](#) shows a city and the resulting DANNA2 network.

8.4.1 Requirements to Find the Shortest Path

With the discussed setup, the length of the shortest path can be found. There are a few challenges in determining what the shortest path actually is. The basic idea is to use the SRNN's Spike-Timing-Dependent-Plasticity (STDP) to learn the shortest path [\[41\]](#). STDP is a mechanism to strengthen synaptic connections that cause the post-synaptic neuron to fire and to weaken the synaptic connections which do not cause the neuron to fire; however, here the mechanism is used to record the shortest path through the network. (For more information about DANNA2's STDP mechanism see [Section 3.3.1](#).) As the spikes propagate through the network, the synaptic weights of the connections that cause the neurons to fire have their weights increased, while the ones which do not cause the fire have their weights

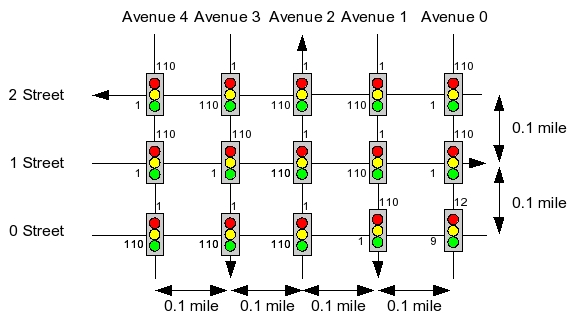


(a) Maze [73]

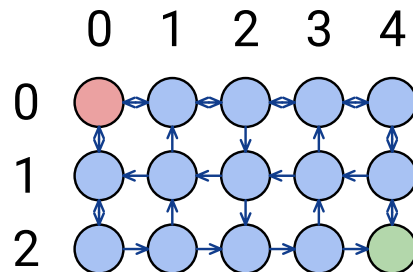


(b) DANNA2 Network

Figure 8.30: A maze and the resulting DANNA2 network used to find the length of the shortest path through the maze.



(a) City [74]



(b) DANNA2 Network

Figure 8.31: A city graph and the resulting DANNA2 network used to find the length of the shortest path through the city.

decreased. Only the increase of weight is required to remember the path. Using this method requires that DANNA2 hardware additionally supports 1) STDP to record the backlink for the shortest path, 2) infinite refractory period to prevent multiple fires from modifying the weights, and 3) a method to read the modified weight values from the elements.

DANNA2 was currently lacking each of these requirements, and STDP was untested on hardware. There was also no native support for infinite or approximately infinite refractory period. (Although an infinite refractory period could be modeled with additional supporting network structure.) And lastly, there was no logic in place to read values from the array in hardware. The programming bus was designed such that it could theoretically be used for bi-directional communication, but without STDP in use, there was no need to read from the elements, since they would not change configuration. Each of these features were tested and added to DANNA2 to support the shortest path finding algorithm. The details of this are found in the following sections.

8.4.2 Adding Element Reading

Adding element reading was the first challenge tackled. Element reading makes the remaining features easier to implement, since the synaptic weight changes and fire state can be read from the element. Element reading queries the state of the element, which includes the loaded configuration of the element, as well as additional debugging and monitoring fields.

Element reading was designed to be efficient, flexible, and to make full use of the programming bus described in [Section 3.3.3](#). The reading is additionally designed to support querying only the desired elements. This allows for many possibilities, including querying singular elements, the entire array, or specific regions. The DANNA2 communication chain design contrasts with the DANNA capture-shift design, which would capture the state of all of the elements and read out the information in parallel. With DANNA, the entire array had to be read at once, and the information was obtained by row and had to be reordered before it could be used.

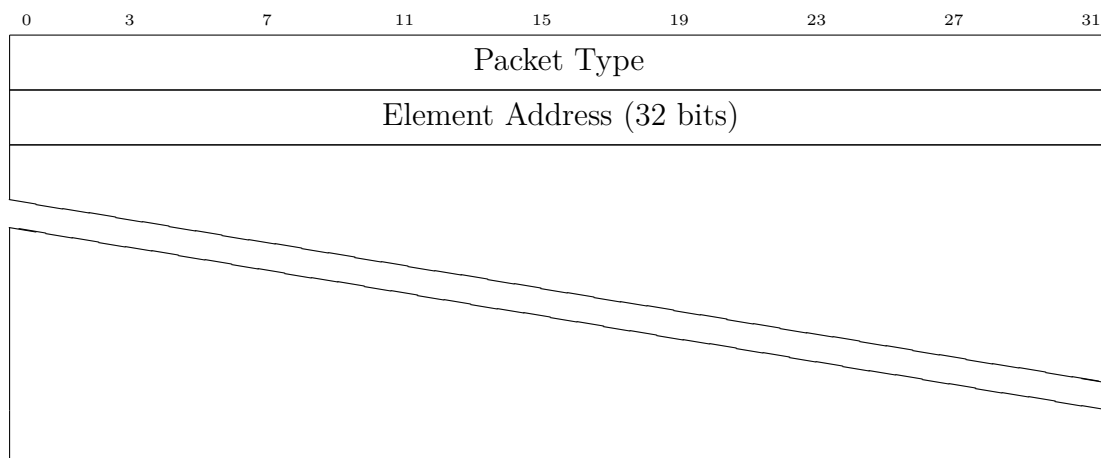


Figure 8.32: New DANNA2 element read input packet description.

Host Packets

Whereas DANNA used a single packet to read the state of the entire array over multiple response packets, DANNA2 uses a single read packet which will query the state of a single element. The neuromorphic processor will respond with one packet containing the state from the requested element. To read from multiple elements, one read packet is sent for each element and each element returns one state packet. Because the host to neuromorphic processor communication links have high-throughput, support large burst transfers, and have separate upstream and downstream channels, there is no performance limitation that results from sending one read request packet per response packet. The total amount of data that must be received to read from the entire array is comparable to the total amount of data received from a capture-shift command. Since one packet is received per response, the data does not have to be reordered in order to interpret the information.

[Figure 8.32](#) shows the structure of a DANNA2 element read packet. This packet has a newly added read op-code and contains the address of the element to read. [Figure 8.33](#) shows the corresponding element state packet which is sent by the neuromorphic process in response. This packet is also known as a read output packet. The element state packet has an identical structure to the packets used to configure the elements. The packet type code is even the same for both the configuration and element state packets. This means that

0	3	7	11	15	19	23	27	31
Packet Type								
Element Address (32 bits)								
Threshold						N	O	F
						R		L
								S
Weight 0			Delay		Weight 1			Delay
Weight 2			Delay		Weight 3			Delay
Weight 4			Delay		Weight 5			Delay
Weight 6			Delay		Weight 7			Delay
Weight 8			Delay		Weight 9			Delay
Weight 10			Delay		Weight 11			Delay
Weight 12			Delay		Weight 13			Delay
Weight 14			Delay		Weight 15			Delay
Weight 16			Delay		Weight 17			Delay
Weight 18			Delay		Weight 19			Delay
Weight 20			Delay		Weight 21			Delay
Weight 22			Delay		Weight 23			Delay

Figure 8.33: New DANNA2 element state output packet description. The ‘N’ and ‘O’ bits are newly added. They represent whether the neuron has fired (N) and whether the element is configured as one-shot (O). This packet description is the same as the new configuration packet description with the exception that ‘N’ is not used.

Table 8.2: Input Packet Opcodes

Input Opcode	Binary	Value
No-Op	00000000	0
Input Fire	00000001	1
Program	00000010	2
Reset	00000100	4
Clear	00001000	8
Read	00010000	16

Table 8.3: Output Packet Opcodes

Output Opcode	Binary	Value
Fire Output	00000001	1
Read Output	00000010	2

the element state packets could be redirected as configuration packets to create a duplicate neural network on an additional DANNA2 neuromorphic processor. [Tables 8.2](#) and [Table 8.3](#) shows the input and output opcodes respectively. The host software has been updated to support sending and receiving read packets. The TENNLab framework was likewise updated to support reading configuration state from DANNA2 hardware.

Programming Bus Changes

The element array controller was updated and the programming bus operation was extended to support element reading. Since the programming bus now supports reading element state, has multiple packet types, and is a daisy chain bus, it will now be referred to as the element communication chain. To guarantee that the element's values do not change during element reading, the evaluation of the network must be paused while the read takes place. A read command is sent along the element communication chain, and the element state is sent by the element back to the array controller via the communication chain. In order to make efficient use of the communication chain and to increase the performance of multiple read operations in a row, multiple in-flight read operations are allowed at a time.

Performance Considerations

With the forwarding behavior of the element communication chain, multiple packets can be passed along the chain concurrently. This is also used in configuring the network, where configuration packets are sent one after the other along the communication chain. Multiple packets are in-flight at different points along the chain. When the packet reaches the destination element, that element is configured. Each element receives and forwards a word of the packet each clock cycle. Previously, each element would always forward the packets along the chain; now elements are designed to forward any packet where the packet destination identification number (ID) does not match the element's ID. If the ID matches, the packet is destined for the current element. If the packet is a configuration packet, it will not be forwarded. If the packet is a read packet, the packet will be forwarded with its contents replaced with the current element's state. Since in all cases the packet is always forwarded the next cycle, the ability to have multiple in-flight packets along the communication chain is preserved without the problem of a data backup along the chain.

A counter was added to the array controller to count the number of in-flight read packets currently in the communication chain. When a new read packet is sent on the chain, the counter is incremented. When a state packet is received from the chain, processed, and the information is sent to the host, the counter is decremented. The incoming host packet parser only processes element read packets when the in-flight counter is greater than one. This allows for multiple read operations to be pipelined without allowing the neuromorphic array to continue executing while the read is taking place. When the parser encounters a different packet type, it waits for the in-flight counter to decrement to zero before operation is continued.

Host to Communication Chain

The incoming host packet parser now understands element read packets. When one is received, an internal read packet is sent out along the element communication chain. This read packet is ten words long, which is the same length as the configuration packet and the required length for packets sent along the communication chain. [Figure 8.34](#) shows a picture of the

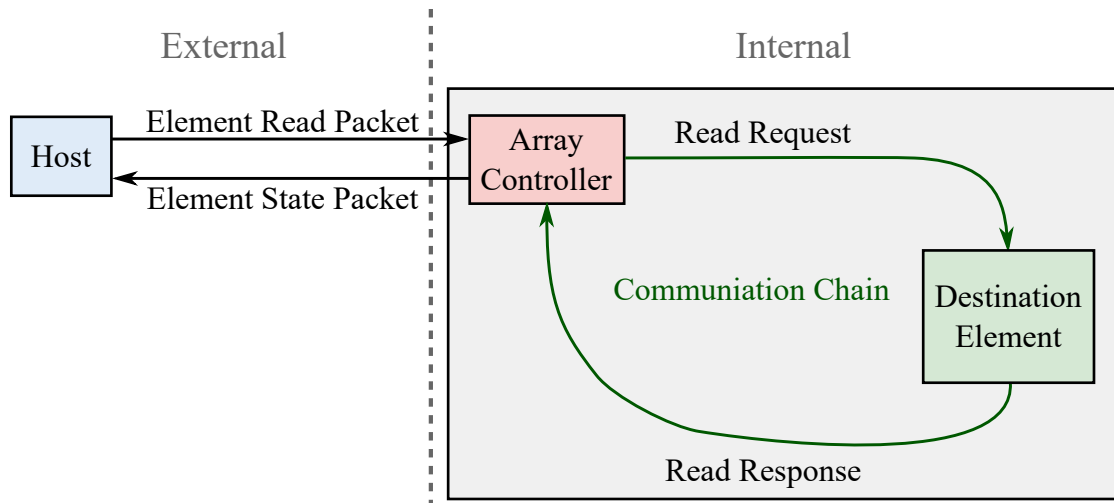


Figure 8.34: Diagram of the element communication chain. Only the destination element is shown, although every element is connected along the communication chain.

communication chain. The read packet only has bits set in the first word, which is shown in [Figure 8.35](#). This first word includes the address of the element to read, and a set bit indicating that this packet is a read packet. The remaining words are all zeros and will be filled in by the target element. The element will fill in the remaining words with the same content that it expects from a configuration packet, as well as any additional debugging and state fields. The previously discussed [Figure 3.11](#) shows the current layout for the eight synapse configuration words which follow after the first word. [Figure 8.38](#), discussed later, shows the current layout for the last word, which contains the neuron configuration and additional state fields. If the target element does not exist, then the packet will return back to the array controller with all zeros.

Communication Chain to Host

The array controller was extended to receive packets back from the communication chain. The controller is looking for packets received with the read bit set. It will ignore configuration packets which do not have a destination. The controller will then take the information provided from the element and send the information back to the host as a element state output packet. An additional counter is added to keep track of the current packet word while parsing the communication chain packet.

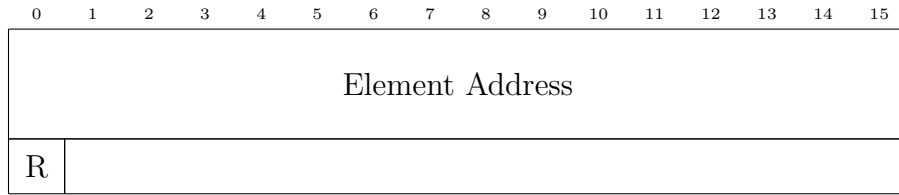


Figure 8.35: Layout of the initial addressing packet over the element communication chain with added element read bit.

Element Changes to Support Reading

The element design also had to be updated to support reading. The element will now consume packets which are used to configure it. In other words, it will no longer forward packets which are used to program it. Although this step is not necessary since the controller will ignore the configuration packets it receives, it is useful for debugging since any received programming packet means that the corresponding element does not exist and was not programmed successfully. There are three options for each packet received on the communication chain:

1. Forward the packet if the packet ID does not match the element ID.
2. Do not forward the packet and use it to configure the element if the packet ID matches and the packet is a programming packet.
3. Replace the content of the packet with the element's current state if the packet ID matches and the packet is a read packet.

The logic of the element was changed to correctly handle these three options. Additionally, the logic for reading and writing the synapse table was updated to support the status reporting operation.

Implementation Plan

To make adding element reading easier and to allow for iterative testing, the process of adding element reading was divided into five steps. These steps were to 1) make sure the programming chain would work with the element consuming packets destined to it, 2) add a bit to the programming chain packets what would signify an element read instead of a write,

3) make the elements send their internal state out along the programming chain when a read packet is destined to it, 4) make the DANNA2 array controller forward read packets to the host when they are received from the programming chain, and 5) add read packet parsing to the device driver to correctly receive the read packets. This breakdown of steps worked well and adding element reading was added without any major difficulties. The incremental development allowed each step to be verified before starting the next step.

Overhead for Element Reading

The element reading implementation is not without a logic utilization penalty. Since each element has to be updated to support reading, the amount of logic required on the FPGA also increases. With all the element features enabled, the maximum size for a DANNA2 array on the KCU1500 is 50×25 . When STDP, fan-in, leak, and reading are disabled, the maximum DANNA2 array size is increased to 70×35 . Similarly, with the 690T and all the features implemented, the maximum array size is 35×20 ; however, with the features disabled, the maximum array size increases to 50. Because reading does increase utilization, and thus decreases the number of implementable elements, a generic was added to the VHDL project to allow the reading feature to be included or excluded in the design. The build scripts were likewise updated to support this generic.

8.4.3 Testing and Debugging Hardware STDP

Spike Timing Dependent Plasticity (STDP) is a common method of implementing a form of online learning with spiking neuromorphic systems. DANNA2 has a simplified, generic pairwise STDP, implemented with a lookup table keyed on the time difference between a neuron fire and the fire of its incoming synapse. This time difference is calculated as shown in (8.3).

$$\Delta t = t_{\text{synapse_fire}} - t_{\text{neuron_fire}} \quad (8.3)$$

When the time delta falls within the STDP window, then the weight of the synapse is adjusted according to (8.4).

$$W(i, j)_t = W(i, j)_{t-1} + S(\Delta t) \quad (8.4)$$

W is the weight of the synaptic connection and S is the STDP weight adjustment. The value of $S(\Delta t)$ is stored in the STDP lookup table at key Δt .

Previously, STDP capabilities had been added to the digital design, but STDP had not been verified on DANNA2 hardware or used with any applications running on DANNA2 hardware. Now that element reading is implemented, STDP can be verified by reading out the weight changes as a result of network activity. Additionally, applications which rely on reading the weight change, like the graph algorithms, can be run on hardware implementations of DANNA2.

A few issues were found when testing STDP which caused the hardware STDP not to work correctly. Some of these issues were related to bugs from refactoring the code, such as `neuron_time` being unset in the synapse unit. Various other hardware bugs related to STDP were also fixed, included making the initial synapse time '111' instead of '000' and adjusting the neuron time bus, since it is 3-bit not 4-bit. Other changes were related to design differences between the hardware and the simulator.

The hardware would unintentionally allow synapse connections to be created by STDP. If a neighboring synapse fires near a neighboring neuron which also fires, a new connection would be made. This is not the desired behavior (although it could be further investigated as a new method of on-chip learning). The issue arises because the hardware does not have a mechanism to differentiate between a disconnected synapse and a connected synapse with a weight of zero. The general question is how to handle synapse connections when the weight changes to zero. There are multiple possible ways to handle this situation without adding an additional mechanism to signify connected synapses, including the following:

1. If the synapse weight is zero, then the synapse is not connected and the synapse weight will not change from zero.
2. The synapses weight will be clipped to a minimum weight (or maximum weight if negative) and the sign of the weight will not change.
3. The synapse weight cannot be changed to zero but is allowed to change signs.

Table 8.4: STDP Lookup Table Values

Δt	Value
+0	-5
+1	-4
+2	-2
+3	-1
-4	1
-3	2
-2	4
-1	5

Option one was chosen since it is the easiest to implement and is sufficient for evaluating graph algorithms. If a different behavior is desired, then the changes to the logic should be minor and limited to the synapse unit.

The simulator is able to distinguish between connected synapses with a weight of zero and disconnected synapses. The simulator's logic was changed to also match option one. The synapse connection is not removed, but the weight of the synapse is no longer allowed to potentiate or depress from a weight of zero.

Another difference between the simulator and the hardware was that they used different STDP curves, both implemented as a lookup table. The values in the lookup tables were changed to match each other, thereby making the STDP curves the same. Both designs allow easy reconfiguration of the lookup table values. The STDP curve is largely arbitrary and can be changed easily; the theory is that the curve should be roughly exponential. If one wants to make changes to the STDP curves, then after making the change in software, the program will need to be recompiled, and after making the change in hardware, the FPGA bitfile will need to be regenerated. [Table 8.4](#) shows the current values in the lookup table and [Figure 8.36](#) shows a plot of these values as a step function.

With all these changes, STDP on the simulator matches the behavior of STDP on the hardware. Multiple test networks were used in addition to the graph networks to verify the correct operation of hardware STDP.

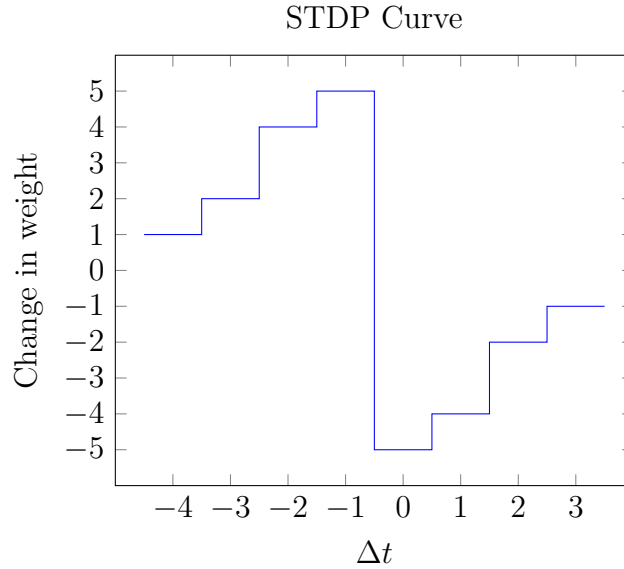


Figure 8.36: Plot of STDP lookup table values.

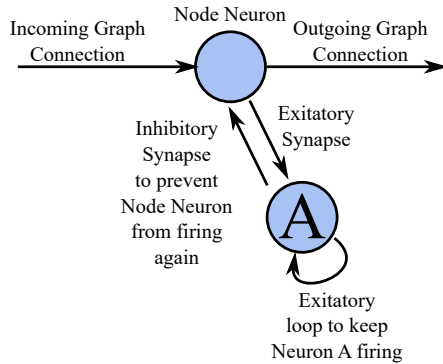
8.4.4 Adding Infinite Refractory Period

The next component to implement is the infinite refractory period. Hardware support for an infinite refractory period is not needed since an infinite refractory period can be implemented with additional elements to prevent the graph element from firing again after it fires for the first time. This additional structure, which can be used in place of hardware support, is shown in [Figure 8.37](#).

Although not necessary to evaluate graph networks, adding single fire or “one-shot” support to the element is easy and will allow for larger graph networks to be evaluated since support elements will not be needed. A new configuration bit was added to the elements to indicate that the element is set to only fire once, and an additional state bit was added to indicate that the element has fired. The fired state bit was necessary to implement one-shot, and it is available for reading even when the element is not in one-shot mode. This allows elements to be queried for activity after a network run.

A one-shot enable bit (‘O’) and a fired state variable (‘N’) were both added to the neuron programming word shown in [Figure 8.38](#). The compare and fire logic was modified to record the fire state variable, which is set on a fire and cleared on a reset or clear activity, and it was modified to allow the element to fire only once when the neuron is configured as one-shot.

With Internal Loops



Without Internal Loops

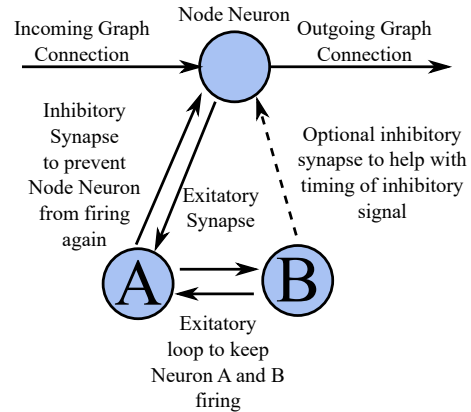


Figure 8.37: One-shot loop, used to implement infinite refractory period through the use of additional elements.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15				
Threshold										Refractory			Leak						
S	F	O	N																

Figure 8.38: Layout of the neuron programming information sent over the element communication chain with additional neuron fields added. ‘O’ is a configuration bit which specifies if the neuron is set to be one-shot. ‘N’ is a status bit which indicates that the neuron has fired since the last clear activity was performed.

The logic uses the fire state variable to check if the neuron has already fired and prevent it from firing again.

The configuration packet was updated to include the one-shot enable bit, and the element state output packet was updated to include the one-shot enable bit and fired state variable as shown previously in [Figure 8.33](#). The host driver was updated to create and parse these new packets. The simulator was likewise changed to support one-shot operation of the neurons and to record the fire state. The DANNA2 network files were updated to version 0.2 and now include the one-shot enable or disable for the network. The software is still backwards compatible with version 0.1 and the file version will be updated by reading and saving an old DANNA2 network file. The TENNLab framework and DANNA2 simulator currently only support enabling/disabling one-shot on a network-wide level through the use of a new DANNA2 parameter. If for some reason one-shot needs to be set on a per-element level, then the framework and simulator can be changed. The hardware already enables and disables one-shot on a per element level.

8.5 Evaluating Graph Algorithms using DANNA2 Networks

Now that STDP is tested, element reading is added, and one-shot mode is added, the shortest path can be found using DANNA2 neuromorphic processors. In order to support shortest path finding, the code which converts the graph into a DANNA2 network now also configures the DANNA2 network to enable STDP and one-shot. The DANNA2 networks are run the same way, with a single fire on the input. With one-shot enabled, only one fire will be seen on the output neuron. This fire time still represents the length of the shortest path, as discussed prior.

The modified DANNA2 network is now read back from the DANNA2 processor; the synapse weights changed by STDP can be used to read out the shortest path through the graph. The pseudocode to read out the shortest path is shown in [Figure 8.39](#). This pseudocode is guaranteed to find a shortest path through the network based on the values of the synapse


```

Neuron start = network.get_input_neuron(0); // Source of graph
Neuron neuron = network.get_output_neuron(0); // Sink of graph
vector <Coords> path; // Array to hold the shortest path

// Find the path from the end to the beginning
while (neuron->coords != start->coords):
    // Add current neuron to the packet of the array
    path.push_front(neuron->coords);

    // Follow the shortest path backwards by following a synapse which caused
    // the neuron to fire
    for synapse in neuron->synapses:
        if (synapse.weight > synapse.original_weight)
            neuron = synapse.from;
            break;

// Add the Source of the graph to the path
path.push_front(start->coords);

```

Figure 8.39: Pseudocode to read out the shortest path from a DANNA2 network modified by STDP.

delay. If you want to detect the presence of multiple shortest paths, or to find all shortest paths, then they can be found by following each backwards path along which the synapse weight increased from its original value.

The two programs written to convert the graph problems to DANNA2 problems were extended to also create a DANNA2 instance, load the DANNA2 network, evaluate the network, read back the modified network, and find the shortest path from the modified network. Since both programs are written within the TENNLab framework, they support running the graph algorithms on the DANNA2 simulator, on a single board FPGA, and on SNACC. The city map solver has a parameter which allows a bit shift amount to be entered to shift the weight values by a certain amount before adding it to the network. Doing so is sometimes necessary to allow the graph to map within the maximum synaptic delay of 15. Just like before, the shortest path for weighted networks is found within the rounding error of converting the edge weight to a four-bit delay value.

8.5.1 Results

The two graph algorithm applications for maze solving and city map path planning were also extended to measure the time used to find the shortest path. In each case the time needed to find the shortest path is recorded for the method selected. For traditional maze solving, the program measures the time necessary to perform a depth-first search to find the end of the maze. For DANNA2 maze solving, the program measures the time required to configure, reset, apply input, simulate, receive output, and read out the network on DANNA2. These times do not include the time spent following the shortest path in the modified neural network or printing the shortest path to the screen.

The city map application similarly supports measuring the time to solution. For the traditional algorithm, the program measures the time needed to perform Dijkstra’s algorithm to find the shortest path. For the neuromorphic algorithm, the program measures the time necessary to configure, reset, apply input, simulate, receive output, and read out the network on DANNA2. Also included in the measurement is the time taken to traverse the modified DANNA2 network to calculate the exact floating point path length instead of just using the reduced precision estimated shortest path length.

These programs were used to collect the timings from many different runs with randomly generated graphs of varying sizes. To collect data for this section, multiple test runs were performed for different network sizes and solution methods. For each graph size, ten random mazes and ten random city graphs were created. Then the solution method was timed for 100 runs for each of the ten different graphs. The solution methods measured are the traditional algorithms, the DANNA2 software simulator, DANNA2 running on the KCU1500, and DANNA2 running on SNACC.

For DANNA2 running on the KCU1500, various network sizes were tested based on the general sizing patterns for DANNA2 networks. Single board networks were tested from a height of five, increasing by five rows and maintaining a width of the nearest multiple of five that is half of the height. Although this sizing pattern is not enforced, networks with a width half of the height tend to perform well for more traditional neural network tasks, although for graph algorithms, the best size is the size which matches the graph problem being solved.

In this case, since graph problems can be generated for any size, a standard DANNA2 sizing pattern was chosen for this testing as well. The smallest size tested on the KCU1500 was 5×5 and the largest size tested was 50×25 . A network size of 50×25 is the largest network that can be built on the KCU1500 when following the sizing pattern with leak, STDP, and element reading features enabled.

The sizes of the SNACC networks tested were generated similarly, except this time the networks of the sub-arrays started at 5×5 and increased by a height of five with the width the nearest multiple of five that is half of the height. This means that the smallest two-by-two SNACC network size tested is 10×10 and the largest size tested was 80×40 . A total network size of 80×40 is the largest network which can fit on the two-by-two SNACC development system using 690Ts for the sub-arrays and with leak, STDP, and element reading enabled.

The software-based approaches evaluate all the sizes tested by either the single-board FPGA or SNACC methods. In addition to the measured solution times, an ideal neuromorphic hardware time was calculated and included with the other solutions. This ideal time is based on the length of the shortest path and the DANNA2 hardware clock frequency used by the hardware solutions. This ideal time is the time required by the DANNA2 hardware to evaluate the neural network and to update the weights with STDP for the shortest path problem. The network only needs to be run until the first fire is seen on the output. When the first fire is seen on the output, the solution to the shortest path problem is found, and the solution is stored within the modified synapse weights found within the network. The ideal neuromorphic time does not include the time required to load the network or read back out the network, but only the time necessary for the hardware evaluation of the neural network. The ideal time serves as a comparison point to help demonstrate the interest in solving graph algorithms using neuromorphic hardware from a performance standpoint.

Figure 8.40 shows the time taken to solve ten randomly generated 40×20 mazes with the various solution methods. Table 8.5 shows the same information in a table. From these charts, the ideal neuromorphic hardware solution is the fastest, followed by the traditional algorithm. The DANNA2 simulator is the next fastest, followed by the KCU1500 and lastly SNACC. The low solution time of the ideal neuromorphic time illustrates why the neuromorphic solution to graph problems is exciting. Since the solution time is based on

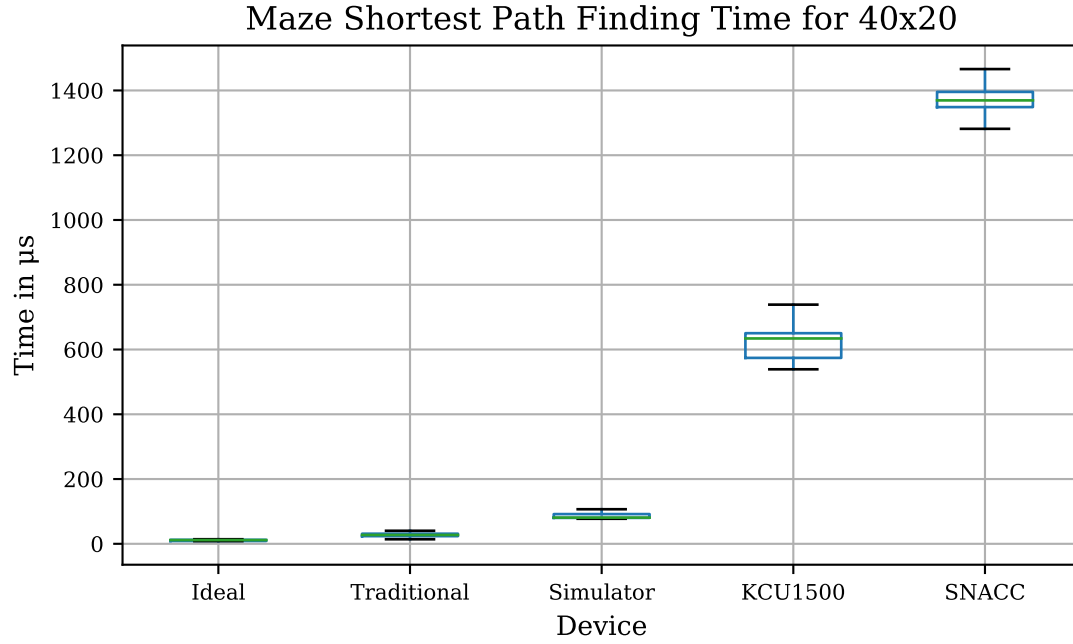


Figure 8.40: Maze shortest path solving time comparison for different methods with a 40×20 maze.

Table 8.5: Maze 40x20 Shortest Path Finding Time

Device	Mean	Std
Ideal	10.88	1.96
Traditional	27.41	5.31
Simulator	83.16	9.59
KCU1500	681.11	149.63
SNACC	1374.53	47.67

Table 8.6: City Map 40x20 Shortest Path Finding Time

Device	Mean	Std
Ideal Shifted	23.47	0.59
Ideal	94.11	2.36
Simulator	111.64	21.09
Traditional	163.43	24.11
KCU1500	695.08	144.37
SNACC	1353.51	74.34

the path length, there is great performance potential. The simulator time falls behind both the ideal and the traditional implementations because loading and reading the network have greater runtime complexities than the depth-first search solution. On top of configuring and reading, the event-based software simulator also has to evaluate the network’s activity. The hardware-based solutions further trail behind due to the added packet creation and parsing time, and the added communication time. SNACC is in last place, since it requires more packets than the KCU1500.

The shortest path solution times for the weighted city map graphs, shown in [Figure 8.41](#) and [Table 8.6](#) for a 40×20 city map, are similar to the unweighted maze graph solution times. Like before, the ideal neuromorphic hardware solution is the fastest, followed by the DANNA2 simulator, DANNA2 on the KCU1500, and lastly DANNA2 on SNACC. However, unlike before, the simulator was faster than the traditional algorithm. This result is caused by the simulator only measuring the time required to evaluate the neuromorphic simulation. If instead the time measured the creation of the DANNA2 network, the traditional algorithm would be faster than the DANNA2 simulator. The ideal time is computed based on both the actual length of the shortest path and also on the length of the shortest path with the reduced precision required to map the weight into four bits. Both ideal times are included in the figures, with ideal being the time based on the shortest path in the graph and ideal shifted being the reduced precision shortest path time. What is interesting with the neuromorphic weighted graph problem is that since the length of the shortest path correlates to how long the evaluation takes, the solution time can be sped up by reducing the precision of the weights.

The next series of figures looks at how the evaluation time needed to find the shortest path changes as the size of the graphs change. [Figure 8.42](#) shows the evaluation time for finding the shortest path in the maze for different numbers of vertices in the maze graph. The number of vertices is the same as the number of elements in the DANNA2 network, which is equal to the height times the width of the network. [Figure 8.43](#) shows the same information but includes only the faster, software-based implementations. The KCU1500 line is shorter than the others since it is unable to implement as many neurons as SNACC. From these graphs the trends appear roughly linear with different slopes. The ideal neuromorphic solution has the shallowest slope and performs the best. The slopes for the KCU1500 and SNACC are roughly the same, but with different offsets. The slope for the KCU1500 and SNACC is driven by the time required to create, send, receive, and parse packets with the hardware devices.

Since the ideal neuromorphic hardware line is calculated based on the length of the shortest path and the DANNA2 network clock frequency, its slope is a line when graphed with the length of the shortest path on the x-axis. This graph is shown in [Figure 8.44](#). When the other implementations are graphed with the length of the shortest path on the x-axis, the lines are jagged, as shown in [Figures 8.45](#) and [8.46](#). This is because the length of the path is only loosely correlated to the number of elements in the graph, since the graphs were randomly generated.

[Figures 8.47](#) and [8.48](#) show how the evaluation time needed to solve the city map graph problem changes as a function of the number of vertices in the graph (or elements in the neural network). The city map evaluation times trend similarly to the maze evaluation times. It is interesting that the ideal and simulator lines have the same shape even though the simulator is performing the ideal shifted operation. This means that although the simulator's performance is much worse than that of ideal shifted, the trend of simulator and ideal is the same as the graphs get larger.

8.5.2 Limitations and Conclusion

This work successfully added the capability to use a DANNA2 neuromorphic processor to perform shortest path finding for a graph problem. This ability was implemented with the

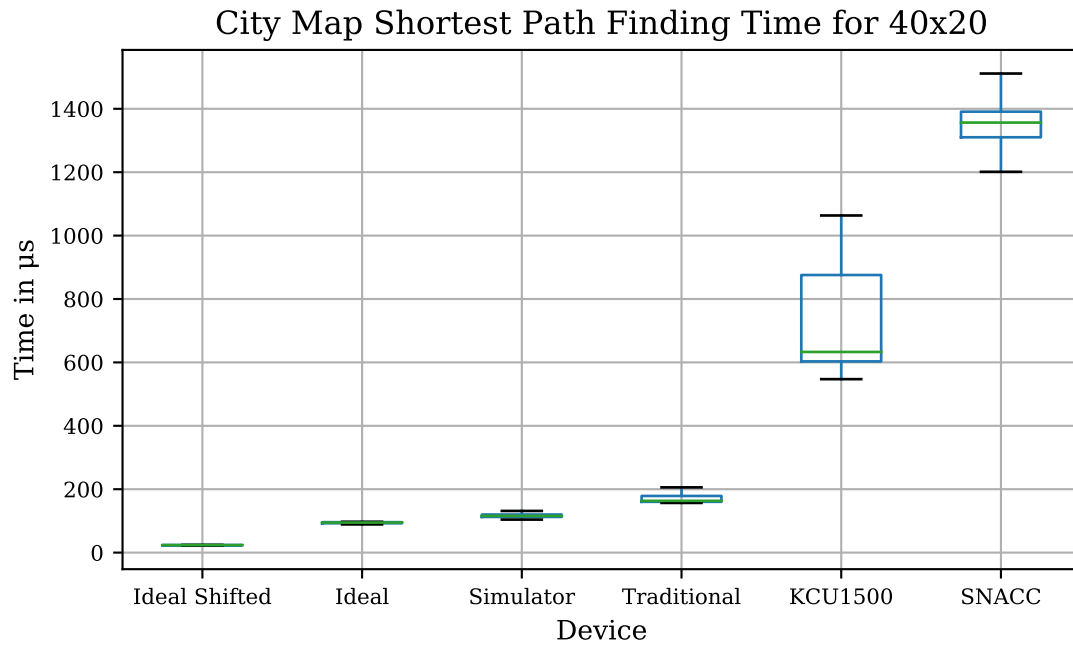


Figure 8.41: City map shortest path solving time comparison for different methods with a 40×20 city.

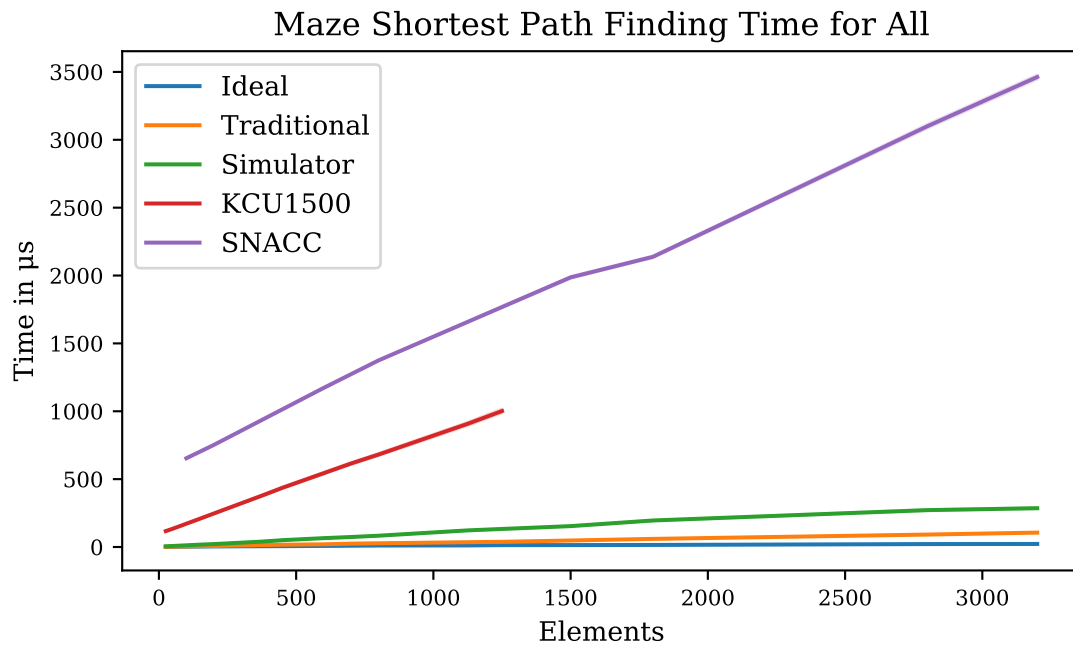


Figure 8.42: Maze shortest path solving time for graphs with different numbers of vertices (i.e. DANNA2 elements).

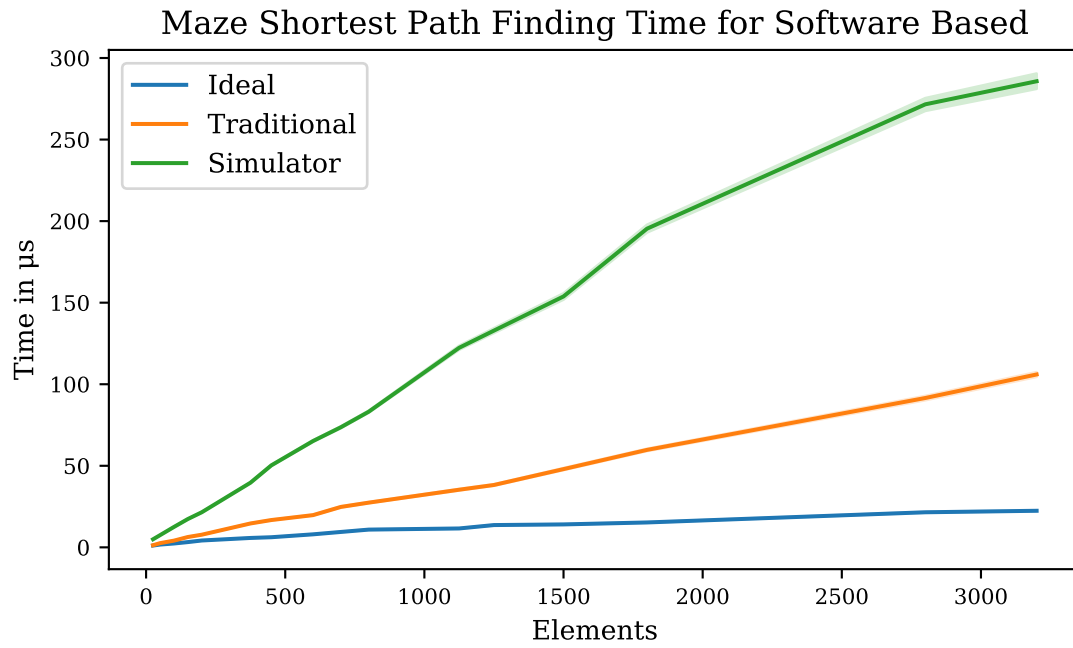


Figure 8.43: Maze shortest path solving time for graphs with different numbers of vertices (software implementations only).

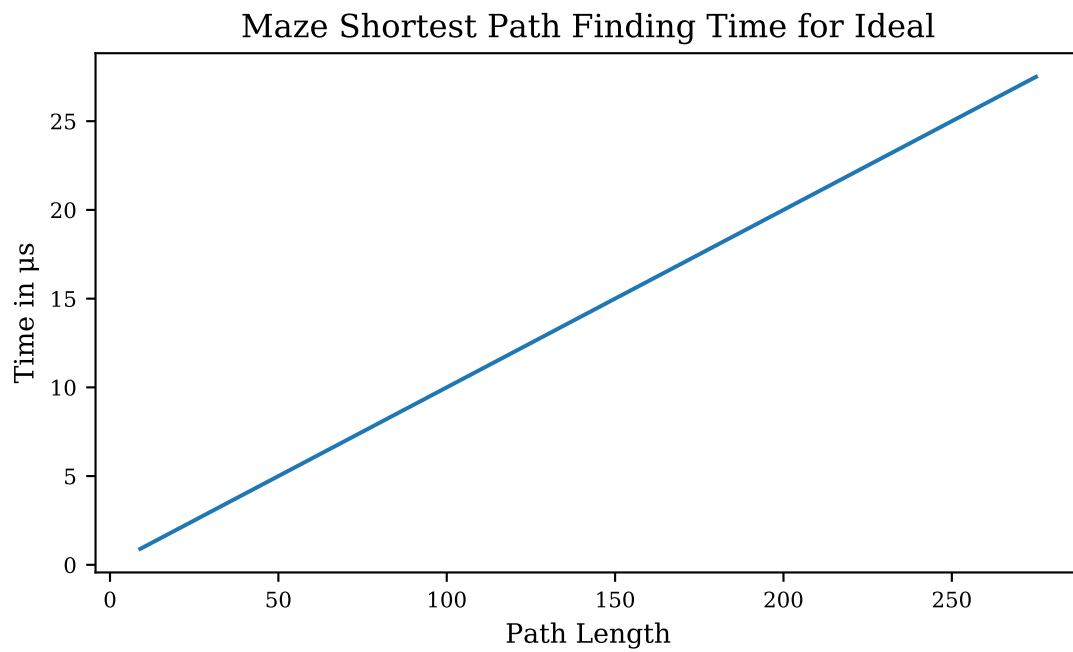


Figure 8.44: Maze shortest path solving time for the ideal neuromorphic solution versus length of the shortest path.

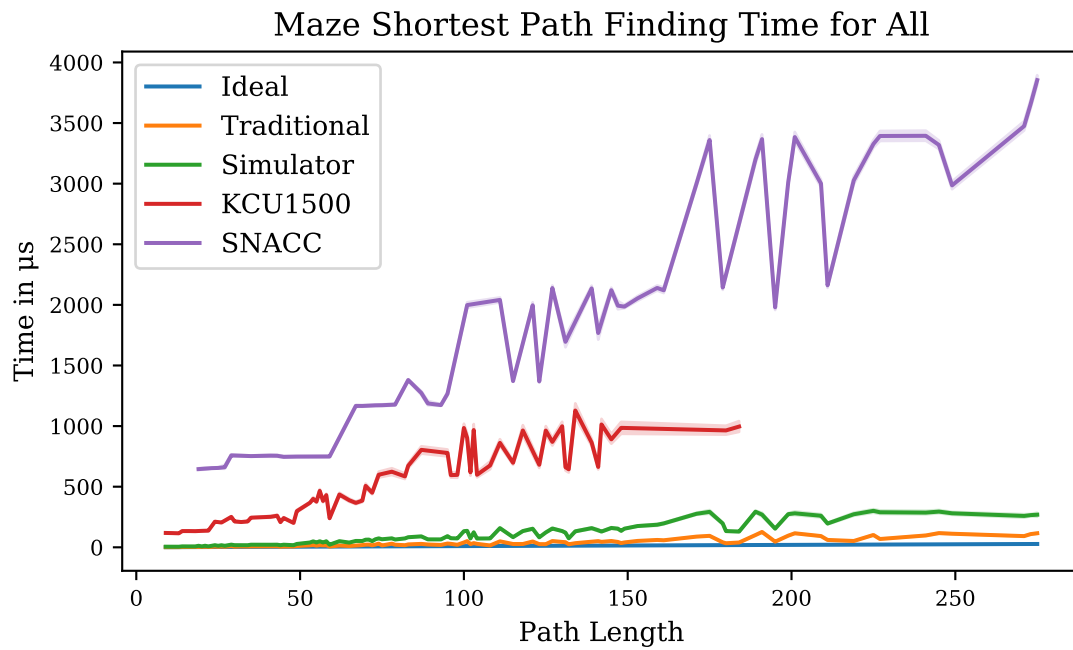


Figure 8.45: Maze shortest path solving time for graphs of different shortest path lengths.

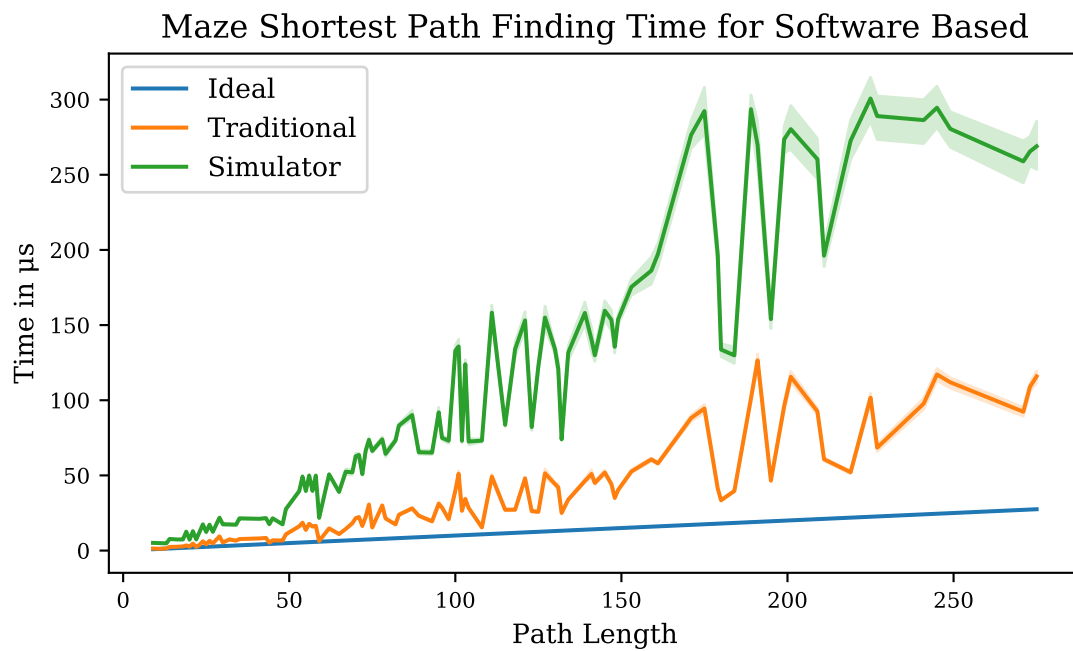


Figure 8.46: Maze shortest path solving time for graphs of different shortest path lengths (software implementations only).

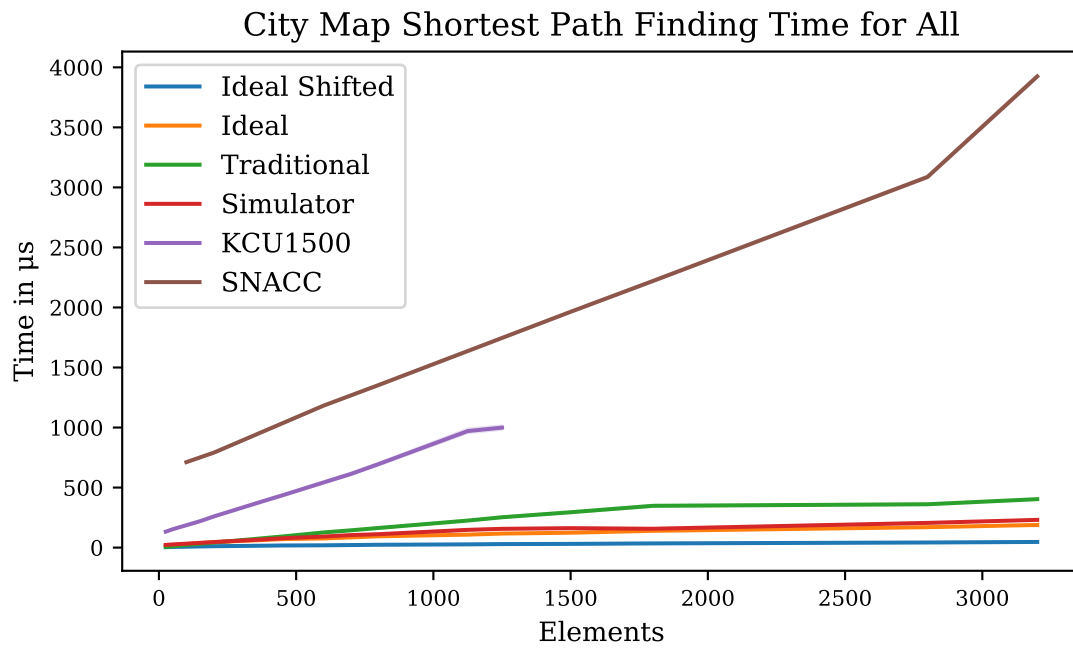


Figure 8.47: City map shortest path solving time for graphs of different numbers of vertices.

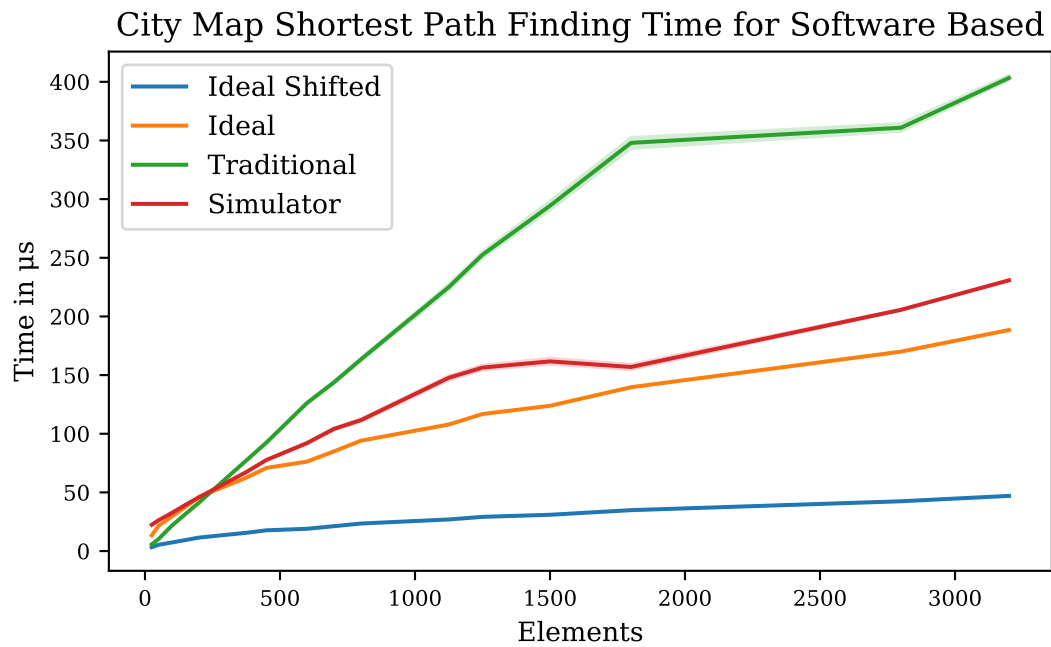


Figure 8.48: City map shortest path solving time for graphs of different numbers of vertices (software implementations only).

addition of element reading and one-shot mode features. STDP was also required, and this work verified STDP on DANNA2 hardware for the first time. Although graph algorithms can now be run on DANNA2 hardware, there are still some limitations and performance considerations. DANNA2 grid has a limited number of neurons that can be used for input and output. Only the first column can be used for input, and only the last column can be used for output. Additionally, the graph problem must be able to be placed on a grid with only local connections. This limits the types of graph algorithms which can be solved with DANNA2 grid arrays. Maze and city map graph problems were used to test and evaluate the shortest path finding since both graphs map well to a DANNA2 grid network. Furthermore, they can be randomly generated to any number of vertices, which lets them test the scaling of solving larger graphs on DANNA2.

Switching from DANNA2 grid arrays to sparse arrays relaxes the mapping constraints on the graph algorithms. Now input and output neurons can be anywhere in the coordinate system, and connections are no longer limited to nearest neighbor. Some limitations remain; the total number of connections for one element is 24 (without fan-in), and elements still have to be placed on integer coordinates. Additionally, the elements only have 15 delay slots that can be used to model edge weight within the graph.

Other graph algorithms can be implemented using the low-level graph operations supported by DANNA2 hardware. One additional example is to find connected sub-graphs. This can be done by firing a neuron in one region of the graph, and then reading the elements to see if it has fired or not. The elements which fired are part of the connected sub-graph. The addition of element reading, infinite refractory period, and fire flags make DANNA2 much more suited for running low-level graph operations, which will allow DANNA2 to support a variety of graph problems.

Although path finding on hardware is slower than running on the simulator and both are slower than using traditional algorithms, the neuromorphic hardware is able to find the shortest path or the length of the shortest path in the same number of network cycles as the shortest path. This has the potential to be much faster than traditional algorithms on large graph problems as predicted by the results. However, the main limitation for using DANNA2 to solve graph algorithms is the complexity of loading a new network and reading back out

the modified network. As mentioned earlier, the ideal neuromorphic hardware solution time is shown in (8.5).

$$\text{Ideal Solution Time} = \text{Length of Shortest Path} \cdot \text{Network Cycle Time} \quad (8.5)$$

This scales quite nicely with the size of the graph problem since the number of vertices along the shortest path is usually much less than the number of vertices in the graph. The runtime complexity for the ideal neuromorphic hardware solution is then $O(|P|)$ where P is the shortest path and $|P|$ is its length. For comparison, the runtime complexity of depth-first search is $O(|V| + |E|)$ and the runtime complexity of Dijkstra's algorithm is $O(|E| + |V| \log |V|)$.

As mentioned before, the limitation of the DANNA2 approach to solving the shortest path algorithm is the need to create, load, and read back the neuromorphic networks. Currently, in order to use DANNA2 to solve the graph problem, the graph has to first be converted into the corresponding DANNA2 network. This process is $O(|V| + |E|)$ as each edge and vertex must be converted into a synapse or neuron, respectively. Then the network must be loaded onto the neuromorphic processor, which is $O(|V|)$ where V corresponds to the number of DANNA2 elements. The E term goes away since it is replaced with the constant 24. Twenty-four is the maximum number of synapses per element and the load operation consists of one packet per element. After loading the network, evaluation is performed for $|P|$ cycles with $O(|P|)$ complexity. Next the DANNA2 network must be read back from the neuromorphic processor, so again, the complexity is $O(|V|)$, since each DANNA2 element must be read back out. Finally, the shortest path must be read out from the modified DANNA2 network. This process is also $O(|P|)$. Therefore, with the exception of ideal, the combined runtime complexity of both the traditional and the DANNA2 shortest path algorithms reduce down to $O(|V|)$, or $O(|V| \log |V|)$ for Dijkstra's algorithms, with the constraint that each vertex can only have 24 edges maximum.

This result matches what is seen in the results, since the time needed to find the shortest path is roughly linear to the number of vertices in the graph. The slope of the line is driven by the number of $O(|V|)$ operations which must be performed. The traditional algorithms

only go through the vertices once, whereas the DANNA2 algorithm must go through the vertices about five times. This causes the slope of the line for DANNA2 implementations to be much greater than the traditional implementation, as seen in the figures. The hardware DANNA2 implementations are much slower than simulated DANNA2, because the majority of the time is spent loading and reading the elements, which the simulator is able to do faster than the hardware. The running of the neural network only occurs for a short number of cycles, with very few total fire events. This combination makes the event simulator especially fast. The hardware is better suited for jobs which run for long periods of time, with many fires.

Even with these present limitations, there are some instances where running path-finding graph algorithms on neuromorphic processors makes the most sense. One instance is for path finding with neuromorphic hardware, which could be useful for planning the path of neuromorphic robots like NEON and GRANT. Neuromorphic path solving will be most useful for fast path finding in fixed or slowly changing environments such that multiple queries can be run on the same network without having to reload the network. With using neuromorphic hardware for path finding, the source of the graph can be the destination and the sink can be the robot's current position. Then the robot can follow the shortest path backward, only reading data from the elements which fall on the shortest path. If the environment changes slowly, only the affected elements need to be changed and the rest can be left unchanged.

Regardless of the practicalities of running graph algorithms on DANNA2 near-term, this application was successfully used to test STDP and element reading via the communication chain. Furthermore, future applications might be found which would benefit from a neuromorphic implementation. The applications which would perform the best are large, able to be mapped to neuromorphic hardware, and would be queried multiple times with a mostly unchanging graph.

8.6 Real-time Network Evaluation

DANNA2 neuromorphic processors can also be used in time-critical, real-time applications, such as robotic control applications or near-sensor, edge-computing, streaming, real-time

classification applications. Although there are no hard guarantees on how long a DANNA2 network will take to evaluate, soft real-time guarantees can be made based on the expected time for network evaluation. The inability to make hard timing guarantees results from 1) using a Linux based host without a real-time operating system to encode and decode spiking data, 2) using a PCIe bus to communicate between the host and the hardware system, and 3) running the DANNA2 neuromorphic processors at different clock frequencies than the host machine and the communication channels.

This problem is further exacerbated with SNACC, where the NACC and each neuromorphic sub-array operate at different clock frequencies. Although the exact execution time is unknown, each of these components takes roughly the same amount of elapsed real time to complete each operation. The SNACC simulator has identified various bottleneck types, and if no bottleneck is present, the entire asynchronous system performs with roughly the same network cycle time as a single-board FPGA DANNA2 processor. For the hardware systems, the major source of execution time is the latency in communication with the host system. This latency, although not the same every run, is comparable for similar operations and will probabilistically complete within a given time window. Therefore, if the total time needed to fully evaluate a neural network for the particular task is sufficiently less than the real-time deadline, then the neuromorphic system can be used for the real-time application. In the case of real-time control applications, this deadline is the decision update frequency, which is driven in part by the update frequency of the sensors and actuators. For most real-time applications the updated deadline is around 20 ms. Therefore, the goal is to demonstrate that SNACC is able to run a neural network for a control application significantly under 20 ms in order to demonstrate that SNACC can be used for real-time applications.

8.6.1 GRANT

The Ground-Roaming Autonomous Neuromorphic Targeter (GRANT), shown in [Figure 8.49](#), from TENNLab is used as the real-time application to demonstrate that SNACC can be used for real-time network evaluation [2]. GRANT is the first DANNA2 neuromorphic processor powered robot and is also the first neuromorphic powered robot with network reconfiguration for multi-function objectives to be able to find and navigate towards a target while avoiding

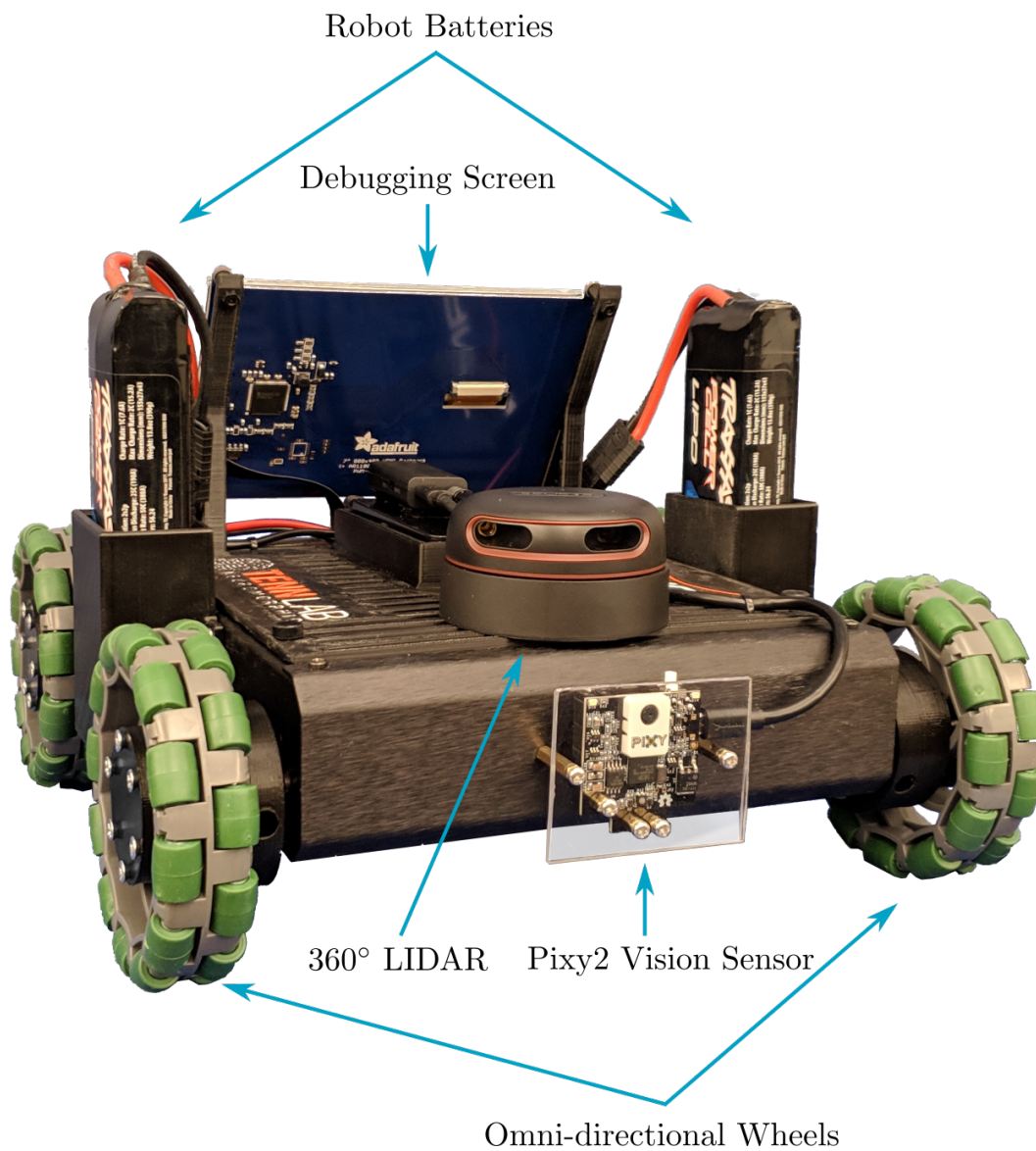


Figure 8.49: The Ground-Roaming Autonomous Neuromorphic Targeter [2].

obstacles. GRANT is designed as a general purpose neuromorphic robotic development platform capable of being easily programmed with different neural networks to perform a variety of tasks. GRANT is controlled with a Xilinx Zynq-7000 System on Chip (SoC) placed on a PYNQ-Z1 board [30]. The DANNA2 neuromorphic processor is implemented in the FPGA portion of the Zynq SoC, and the ARM Cortex-A9 processor in the Zynq SoC is used as the host system. Altogether, the PYNQ-Z1 board implements an embedded SoC version of the host and DANNA2 processor combination. Figure 8.50 shows the block diagram of the SoC design. An AXI4 interconnect and DMA engine are used to transmit the DANNA2 packets between the ARM processor and the DANNA2 processor.

The two tasks on GRANT which are used to test real-time operation of SNACC are an obstacle avoidance exploration task (roam) and a target navigating and obstacle avoidance task (target). Both of these tasks are implemented with the **robonav** application. Roam uses a single neural network to roam around a room and avoid obstacles. It was trained to maximize the coverage of the room traversed. Target is unique since it has two phases which are each implemented with a single network. One phase is the roaming phase which uses the roam network. The second phase occurs when a target is sighted and the DANNA2 processor is reconfigured to use a network that was trained to drive towards a target while avoiding obstacles. If the robot loses sight of the target for too long, the DANNA2 processor is reconfigured back to the roam network. This means that the target application needs to be able to also swap out active networks while the application is running. Both roam and target represent real-time control applications which work with the embedded SoC DANNA2 processor. The real-time update deadline for both applications is 10 ms. By controlling the operation of GRANT for these applications, SNACC’s real-time capabilities were tested.

8.6.2 Design

The basic approach to using SNACC as the neuromorphic processor for GRANT is to network SNACC’s host computer and GRANT together. Then, a network socket is used to transmit the DANNA2 packets data from GRANT to SNACC’s host computer, where SNACC will then evaluate the packets and send the result back to GRANT. Both Ethernet and Wi-Fi connections were tested as ways to network GRANT and SNACC. The testing and design was

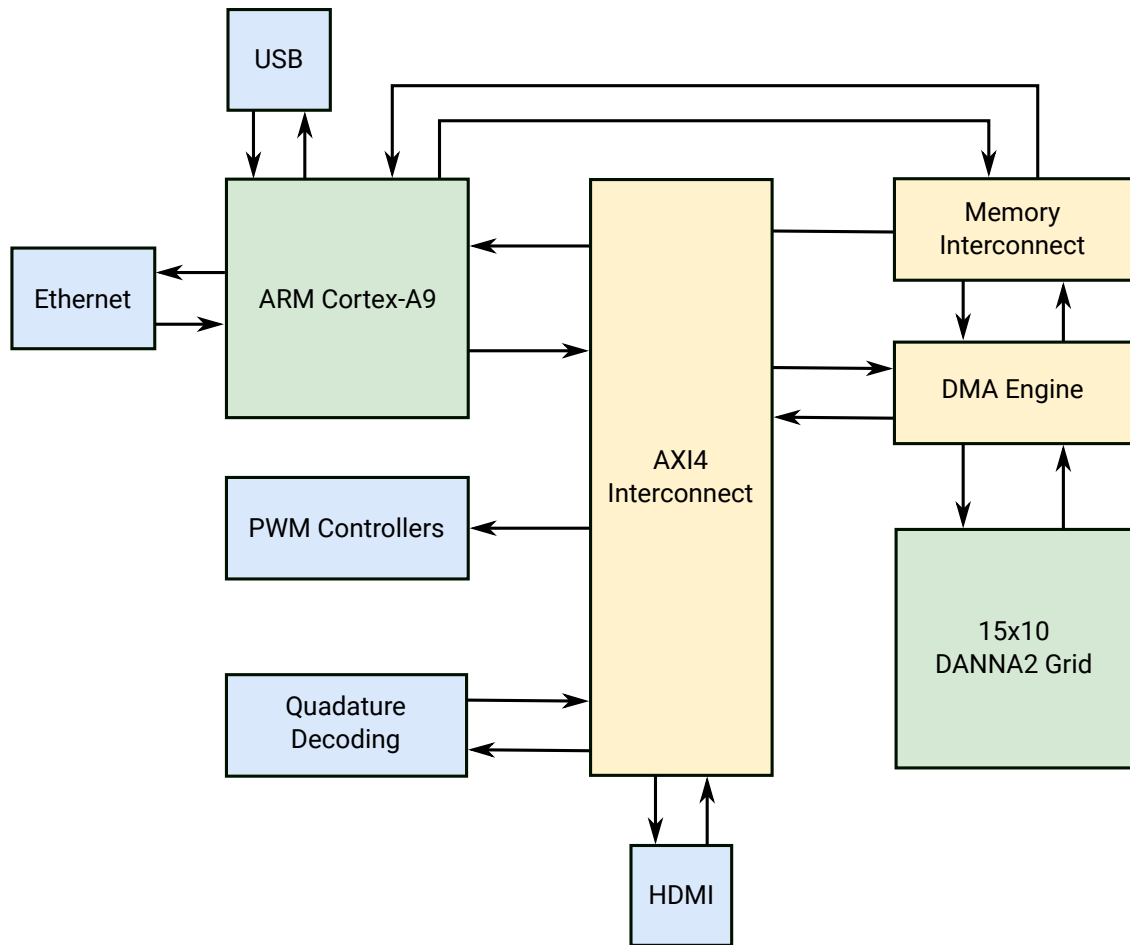


Figure 8.50: GRANT SoC design block diagram [2].

conducted using GRANT and a PC connected to both a single-board DANNA2 (KCU1500) and to SNACC. To make the explanations of the setups easier, GRANT refers to the ARM processor on the robot, PC refers to the host machine for single-board DANNA2 and SNACC, KCU1500 refers to the single-board DANNA2 implementation, and SNACC will refer to the multi-board DANNA2 implementation.

This section will go over the iterations of the communication design and discuss the performance increases along the way. The metric of performance used during the design is the average time in microseconds needed to evaluate the decision for the next cycle. This includes updates from the sensors, packet encoding, evaluating the neural network, decoding the packet, and updating the motor speeds. For the target application, the target position relative to the robot is also updated. The time this decision takes is measured from GRANT.

In the first test, GRANT was connected to the PC and GRANT sent binary DANNA2 packets over the network. The PC then used the `ncat` Linux utility to redirect packets between the network socket and the device file to communicate with the KCU1500. Although this approach worked, the time needed for the decision update was approximately 50 ms to 60 ms. For comparison, the update time when run locally on GRANT is roughly ~ 1 ms.

The primary reason that this implementation took so long is that `ncat` does not send an empty write to tell Xillybus to flush the sending buffers. Therefore, for the second test, a custom network-socket-to-device-file program was written, which would try to improve the performance by sending an empty write system call to flush the Xillybus buffers after completing a transfer. This test proved successful. Without including the flushing, the performance increased to roughly 24 ms, and with flushing the performance increased to roughly 6 ms. Looking closer at the resulting times revealed contention on the port when only one network socket was used. The average performance was around 1 ms, but would periodically jump to around 40 ms. This contention issue was resolved by changing the program to use two sockets, one for sending and one for receiving. With two sockets the performance is now close to local performance, averaging around 1 ms.

Although redirection of packets works fine for running on the KCU1500, it will not work for SNACC. With SNACC, the packets must be routed correctly among the multiple sub-array boards. Therefore a new utility program was created, called `streaming_run`, which allows

for higher-level communication with the DANNA2 processors and supports running on the DANNA2 simulator, single-board hardware, and SNACC. This utility defines a high-level ASCII command interface that can be directly mapped to function calls within the DANNA2 driver similarly to `single_run`. However, `streaming_run` is able to support multiple simulate calls, unlike `single_run`. As the name suggests, `streaming_run` takes a stream of commands and runs the commands on the specified DANNA2 processor implementation. The output from the processor is then parsed and converted to ASCII output to be sent back. `streaming_run` can be used with pipes or network sockets.

The third test used `streaming_run` to run GRANT with a simulator on the PC, the KCU1500, and SNACC. Jonathan Ambrose, a TENNLab graduate research assistant, updated the code on GRANT to be able to send and receive packets using the high-level command interface used by `streaming_run`. This test was also successful, with approximately 3 ms as the worst-case cycle update time.

In the fourth test, GRANT's code was updated to run the neural network evaluation at a target speed of 10 ms with sensor updates every 150 ms. Prior to this change, the decision cycle only occurred every 150 ms with the network evaluation thread waiting for the sensor update threads to finish collecting sensor data. This test showed reduced update times when sensors were not read and caused the mean of all the decision cycle times to improve by about half a millisecond. This change also improved the behavior of the robot and allowed it to more closely match the simulated robot's behavior.

In the fifth test, GRANT's code was updated to send larger transfers of data. Now GRANT sends all the DANNA2 packets over at once instead of sending them one at a time. The effect on the PC is that Xillybus is only flushed when GRANT is expecting a response, and larger transfer blocks can be used. This test had the best performance by far with around 0.5 ms evaluation time for all DANNA2 processor implementations. This final setup is the one used to collect detailed performance data for the different DANNA2 implementations and connection methods.

This testing shows that `streaming_run` has great performance and is able to work across any of the DANNA2 processor implementations since it utilizes the TENNLab DANNA2 libraries. It further shows that two network sockets should be used to get the best

Table 8.7: Average Decision Update Times for Different Test Runs

Test Run	Update Time (ms)
Test 1	56.21
Test 2 No Flush	23.61
Test 2 Flush	6.07
Test 2 Flush Two Sockets	1.36
Test 3 KCU1500	1.66
Test 3 Sim	3.67
Test 3 SNACC	1.68
Test 4 Sim	0.86
Test 4 KCU1500	0.99
Test 4 SNACC	1.73
Test 5 Sim	0.47
Test 5 KCU1500	0.55
Test 5 SNACC	0.58
GRANT on Local	0.68

communication performance and that the size of transfers and locations of flushes are crucial to achieving peak performance. [Figure 8.51](#) shows a graph of the performance improvement for these test cases, and [Table 8.7](#) shows the performance improvement as a table of the average updated time values.

With the network socket approach taken, switching from Ethernet to Wi-Fi is easy. Since GRANT does not have built-in Wi-Fi and using a USB adapter proved to be difficult, a network router was added onto GRANT. The router connected via Ethernet to the PYNQ board and connected wirelessly to the PC. This router could now be both powered by and mounted to GRANT to allow wireless operation with SNACC. Further benchmarking results, including Wi-Fi results, are shown and discussed in the next section.

8.6.3 Results

Performance data was collected using the same design setup described in the previous section. Samples of the time spent updating the decision during the update cycles were collected by GRANT for running the DANNA2 network locally and for running DANNA2 externally on a separate PC through the software simulator, single board DANNA2 system, and SNACC

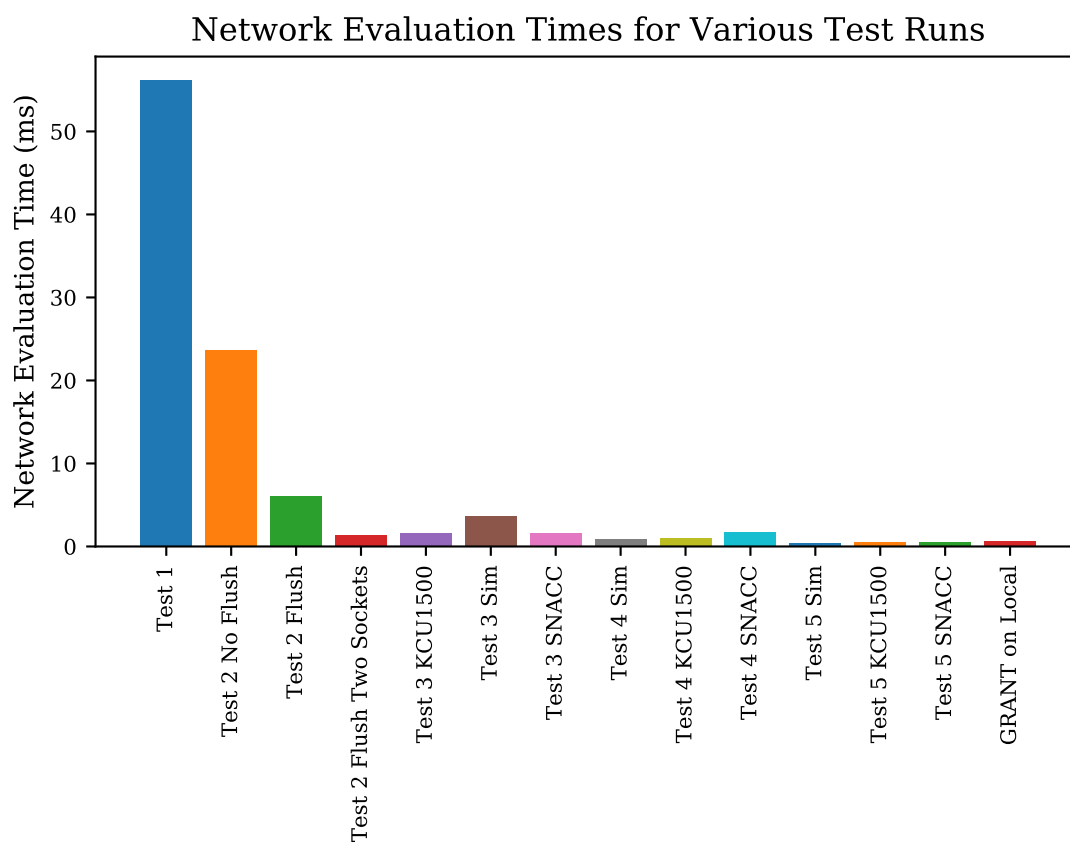
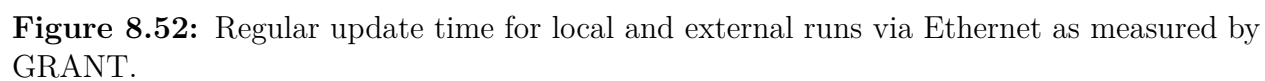


Figure 8.51: Network evaluation performance improvement as different approaches are tried.

system. Both Wi-Fi and Ethernet were tested as means of connecting GRANT with the PC. For the tests which used the PC, the PC measured how long the network evaluation took from its perspective. Both the roam and target applications were used to collect data. Each test was run for around one minute, which resulted in over 5000 samples. The samples were separated into 3 groups. The samples in which a sensor update occurred (every 150 ms) were separated from the regular update cycles, which occurred every 10 ms. Additionally, for the target application, the cycles in which a new network was loaded were also pulled out. To make the graphs in this section, data from 5000 regular update cycles, 387 sensor update cycles, and five network changing cycles was used. The graphs of network evaluation time as measured from the PC also used 5000 samples.

Figure 8.52 shows a box plot of the time taken for a regular update decision to evaluate the neural network running both locally on GRANT and on the external system via an Ethernet connection. This plot shows that both the local and Ethernet cycle evaluations are able to easily meet the 10 ms real-time decision making deadline. Of the external neuromorphic processors, the simulator was able to respond the quickest and with the least variability. The simulator's quick response is likely from being evaluated on the PC without further communication to a hardware DANNA2 implementation. The KCU1500 also seemed to outperform SNACC. This behavior likely comes from the fact that more packets have to be generated and routed on SNACC than for the KCU1500. For a regular update cycle, the SNACC system has roughly twice as many packets to create and parse than the single board setup. Even with the added complexity of DANNA2 hardware, the evaluation times are close together. This is likely because a major part of the evaluation time comes from the Ethernet communication.

To explore this further, Figure 8.53 shows the DANNA2 network evaluation time as measured by the host PC. As suggested by the previous figure, this figure shows that the majority of the evaluation time comes from the Ethernet communication overhead. Additionally, the simulator greatly outperforms the hardware implementations for the small 15×10 neural networks used by GRANT. The KCU1500 has slightly better performance than SNACC. This figure also shows that the slower outliers seen for SNACC are caused by a slowdown in the network evaluation, not from the Ethernet channel.



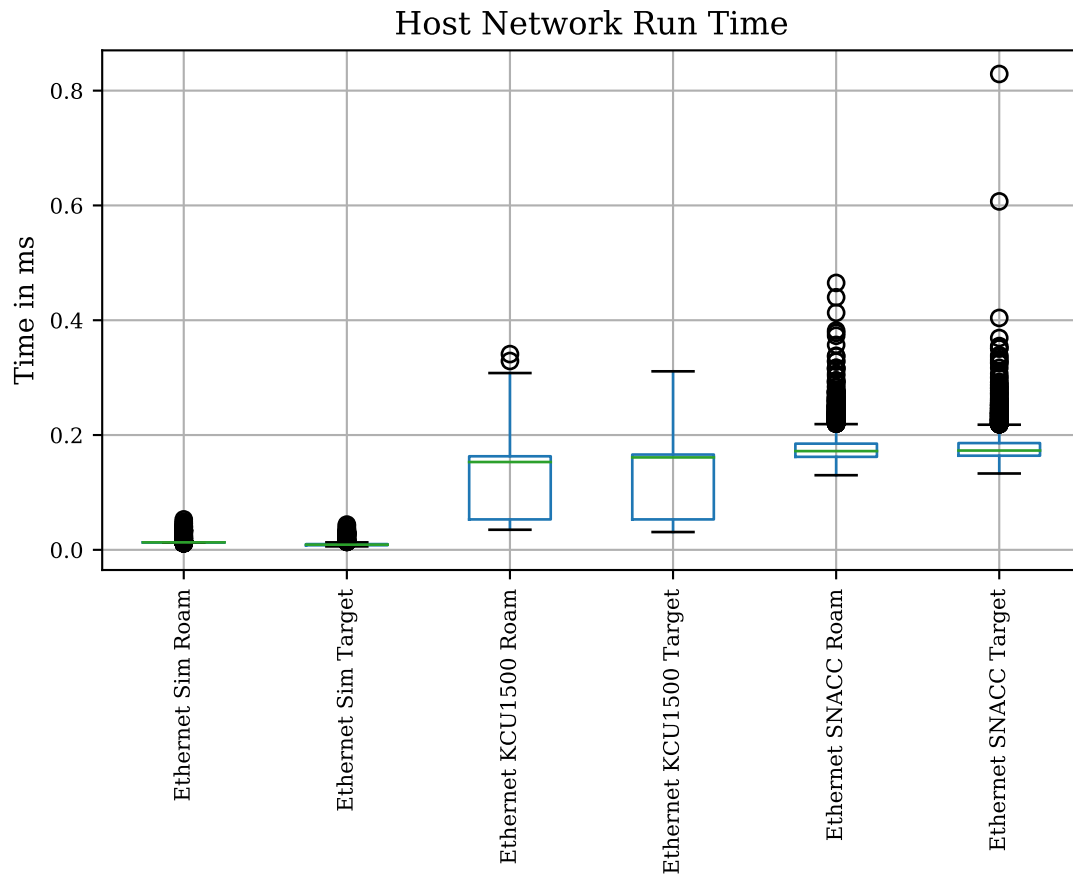


Figure 8.53: Network evaluation time for external runs via Ethernet as measured by the external host PC.

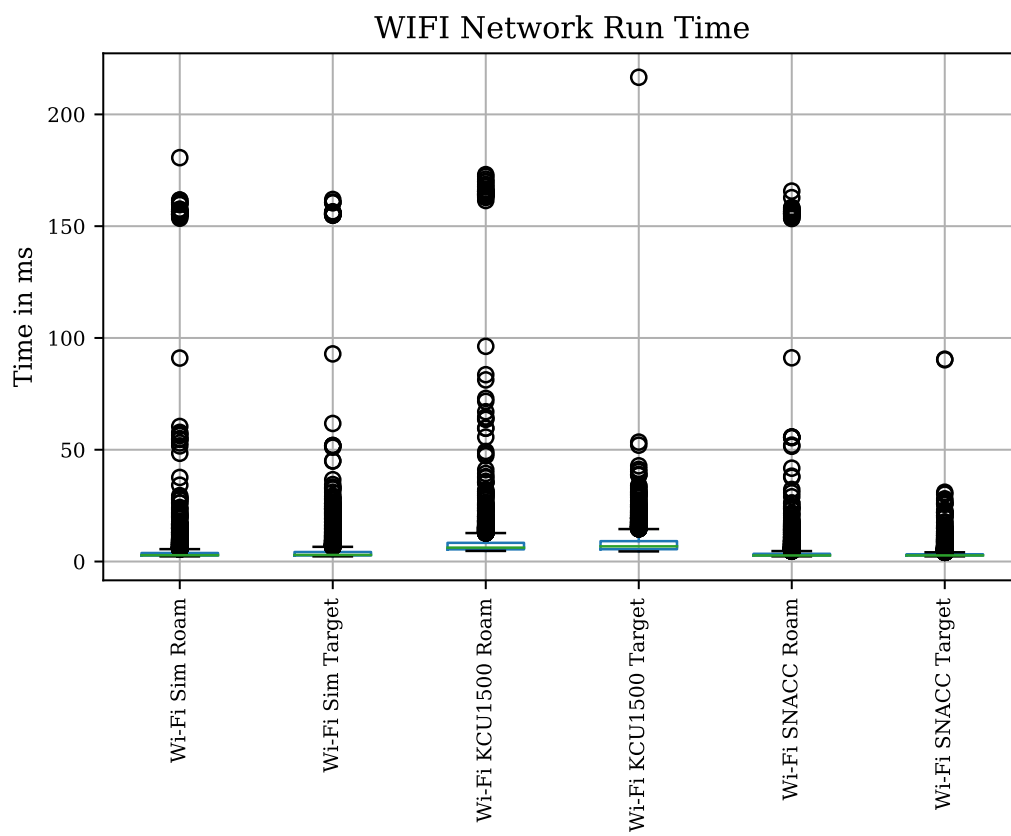


Figure 8.54: Regular update time for external runs via Wi-Fi as measured by GRANT.

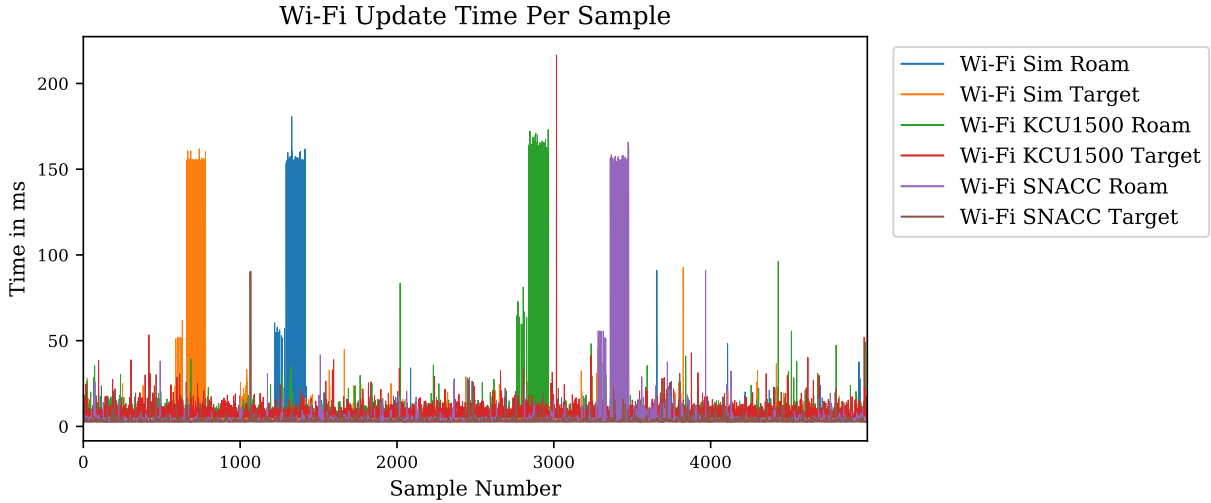


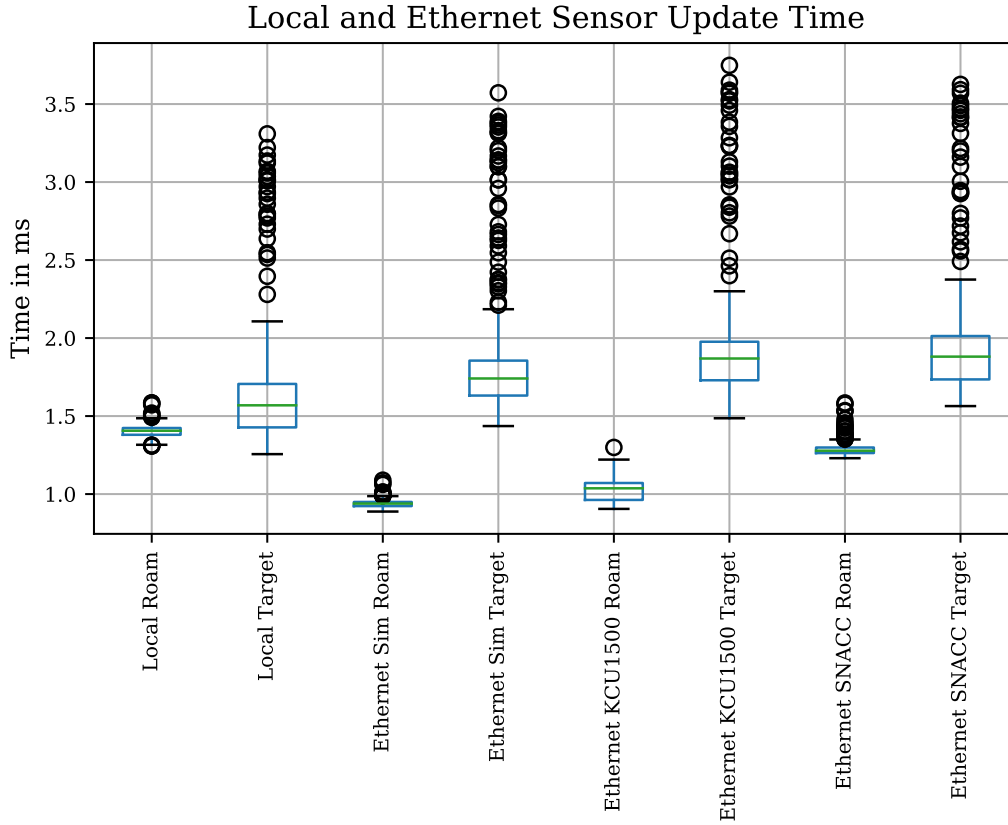
Figure 8.55: Line plot of the regular update time samples via Wi-Fi as measured by GRANT.

Figure 8.54 shows a similar box plot of the regular update cycle evaluation time, now with the external system connected via Wi-Fi. As shown by the graph, the mean evaluation time for Wi-Fi is still low; however, there are many slow outliers hurting performance. To further explore the lower performance, Figure 8.55 shows a line plot of the individual samples. As seen from this graph, the Wi-Fi connection generally performs well, but will fall into pockets of poor performance with evaluation times greater than 150 ms. If these pockets of poor Wi-Fi performance could be mitigated or removed, then real-time performance over Wi-Fi seems possible.

Table 8.8 shows a summary of the numerical results from the various setups used to run the GRANT applications. As shown by the table, all of the test setups meet the real-time deadline of 10 ms on average; however, only the local and Ethernet setups meet the deadline for every sample. Even the worst case update time for the Ethernet setups of 1.29 ms is almost an order of magnitude less than the deadline. The table also shows the percentage of the samples which meet the real-time deadline. The local and Ethernet setups meet the deadline 100% of the time and the Wi-Fi setups met the deadline an average of 92.77% of the time, with a best case of 98.8% and a worst case of 80.88%.

Table 8.8: GRANT Application Results

Setup	Min	Mean	Max	Percent < 10 ms
Local Roam	0.50	0.68	0.82	100.00
Local Target	0.26	0.32	0.66	100.00
Ethernet Sim Roam	0.39	0.47	0.68	100.00
Ethernet Sim Target	0.42	0.48	0.74	100.00
Ethernet KCU1500 Roam	0.43	0.56	0.83	100.00
Ethernet KCU1500 Target	0.42	0.60	0.85	100.00
Ethernet SNACC Roam	0.52	0.58	0.98	100.00
Ethernet SNACC Target	0.53	0.62	1.29	100.00
Wi-Fi Sim Roam	2.30	4.53	180.65	97.74
Wi-Fi Sim Target	2.31	4.75	161.94	97.02
Wi-Fi KCU1500 Roam	4.74	8.41	173.06	84.90
Wi-Fi KCU1500 Target	4.53	7.94	216.60	80.88
Wi-Fi SNACC Roam	2.31	4.40	165.69	97.28
Wi-Fi SNACC Target	2.30	3.35	90.49	98.80

**Figure 8.56:** Sensor update evaluation time for local and external runs via Ethernet as measured by GRANT.

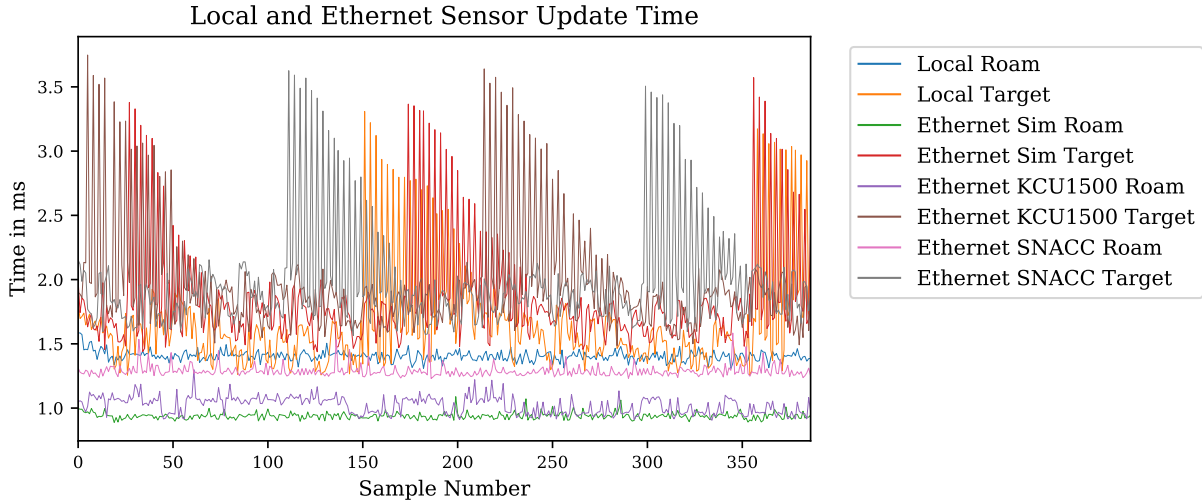


Figure 8.57: Line plot of the sensor update evaluation time samples for local and external runs via Ethernet as measured by GRANT.

Figure 8.56 shows a box plot of the decision update time on cycles which additionally update network input based on information from the sensors. This graph shows that the real-time deadline of 10 ms is still met on cycles which update the sensor data. The graph further shows that the sensor reading for the target application is greater than for the roam application. This difference in update time is likely caused by the addition of the Pixy2 camera used for target sensing. To further explore this update time difference, Figure 8.57 shows a line plot of the individual samples. This graph shows that the update times for the target application follow a decreasing sawtooth pattern. This update pattern is not seen in the roam application.

Figure 8.58 shows that performing a network switch results is the longest time taken for an update cycle. Even with switching network configurations, the real-time deadline is still met. Interestingly, the local network switch takes the longest of the tests, with the external evaluations via Ethernet taking almost 4 ms less to complete. Although this seems counter-intuitive at first glance, the explanation is likely from the performance discrepancy between the PC and ARM processors while generating configuration packets. Both systems must generate the load packets each time a new network is configured, and the PC is able to generate them much faster. Additionally, the high-level communication protocol

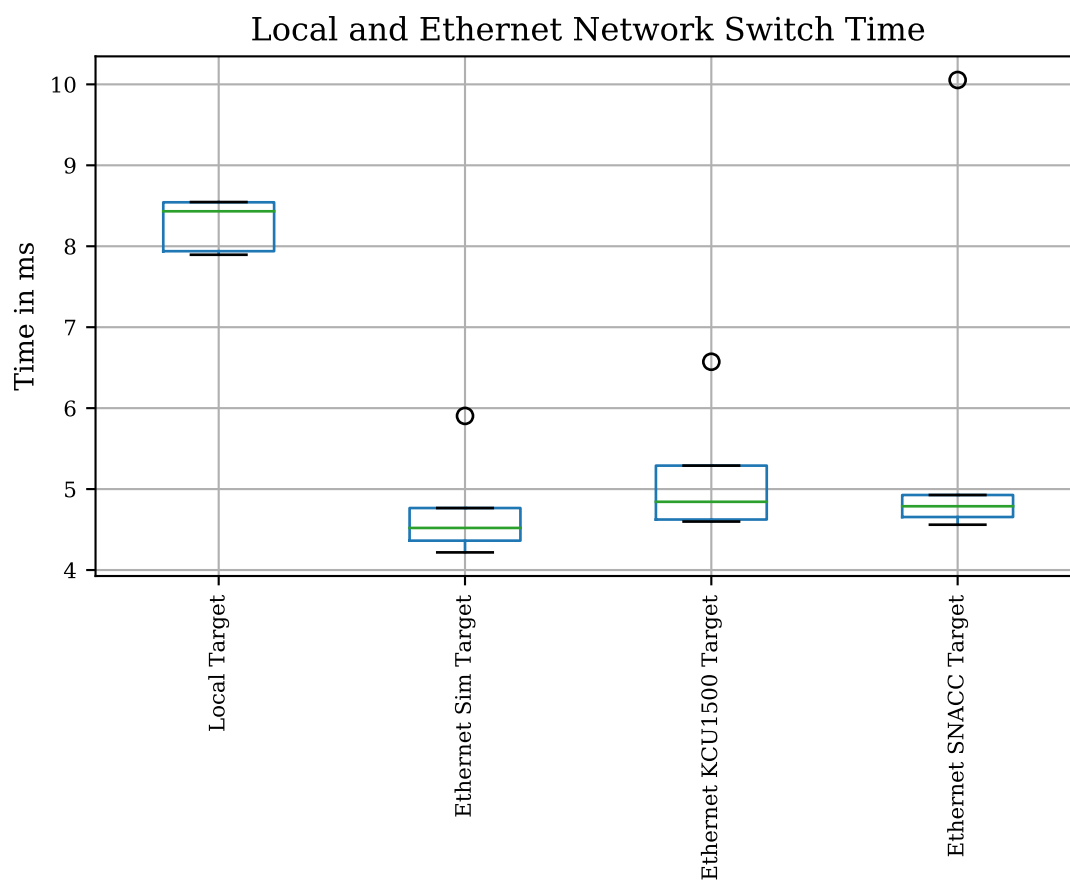


Figure 8.58: Neural network switch update evaluation time for local and external runs via Ethernet as measured by GRANT.

for `streaming_run` minimizes the processing which must be done to instruct the external DANNA2 processor to load a new network. This cuts down on the processing the ARM processor has to do to load a new network configuration, as well as reduces the amount of information which must be sent over the network.

8.6.4 Conclusion

The `streaming_run` utility allows easy communication with a variety of DANNA2 neuromorphic processor implementations through a high-level interface which can be used locally or across network sockets. With the utility, SNACC has been shown to be able to evaluate DANNA2 networks remotely for use in real-time systems. Although strict evaluation timing for DANNA2 cannot be guaranteed, the system behaves reliably and will complete the evaluation in a window of time with a high probability.

To demonstrate this capability, GRANT was connected to SNACC, and SNACC was used to evaluate the DANNA2 networks quickly enough for GRANT to operate in real-time. With an Ethernet connection, SNACC was able to easily meet the timing deadline for all the update cycles. However, if Wi-Fi is used instead of Ethernet, the update timing deadline is only met around 90% of the time. The Wi-Fi connection would periodically slow to an update time around 150 ms. If SNACC is used in a soft real-time application, the Wi-Fi connection might be good enough; however, in other cases the Ethernet connection works much better. This setup also enables the testing of larger DANNA2 networks for the GRANT applications. Previously, the system was limited to the sizes which could fit on the Zynq-7000; now larger sizes, with more inputs, can be tested.

Chapter 9

Accomplishments

The human brain starts working the moment you are born and never stops until you stand up to speak in public.

– George Jessel

My dissertation work has resulted in many accomplishments and this section serves to record them in one place. Details on these accomplishments are found throughout this document.

9.1 Literature Review

For my dissertation work, I completed a thorough literature review of the communication systems used by other neuromorphic hardware systems, and I discussed considerations and challenges associated with spiking communication. This literature review has been published in an *IEEE Access* journal article titled “A Review of Spiking Neuromorphic Hardware Communication Systems” [120]. A majority of this article is republished as [Chapter 2](#).

9.2 Aurora ACK

I also created a custom Aurora acknowledgment ARQ design which makes high-performance, robust, board-to-board communication possible via an Aurora channel with custom logic for retransmission on errors. See [Section 3.4.2](#) for additional information on Aurora and see

[Section 7.3.1](#) for the design of the custom Aurora acknowledgment ARQ. This new component greatly benefited the spiking communication between FPGAs via Aurora since the channel is now guaranteed to be error-free without a significant impact on performance.

9.3 DANNA2

I also improved upon the DANNA2 neuromorphic system, both by adding new utilities to interact with the processors, as well as adding new features to the hardware design. I added element reading to the DANNA2 hardware design and also tested the hardware STDP design (see [Section 8.4](#)). In addition to adding hardware support within the software framework, I also created useful utilities to generate networks with test patterns to easily use the DANNA2 neuromorphic processors (see [Section 8.1](#)). Two of the primary utilities are `single_run` and `streaming_run`, which allow direct access to the underlying DANNA2 library using a higher-level command-based interface (see [Sections 6.3](#) and [8.6.2](#) respectively.). The `streaming_run` utility makes it possible to use this command-based interface remotely over a network socket.

9.4 DANNA2 Visualizer

I wrote a new DANNA2 network visualization program to visualize DANNA2 network files. See [Section 5.7](#) for details on this visualization program.

9.5 Single-board FPGA Implementation

For this dissertation work, I designed and tested the single-board DANNA2 system built using a KCU1500. See [Chapter 6](#) for details. Along with the hardware design, I also updated the TENNLab framework and created software drivers to allow the DANNA2 hardware systems to be used as easily as the DANNA2 software simulator (see [Section 6.2.3](#)).

9.6 SNACC

The SNACC multi-board DANNA2 neuromorphic system was designed, built, and tested. The SNACC system is the primary accomplishment and contribution of this dissertation. A conference paper on SNACC has been submitted to IJCNN 2020 and is pending publication. See [Chapter 7](#) for details.

In order to verify the SNACC communication system, a simulator of this system was written. It can simulate the various communication patterns which are present during the operation of SNACC. The simulator also includes a GUI able to visualize packets moving around the board. Built with the help of Adam Foshie, the SNACC simulator is described in detail in [Section 7.3.2](#). Fake DANNA2 was also designed and built to verify the hardware communication system without a neuromorphic core.

9.7 Build System

For both the single-board DANNA2 hardware processor and SNACC, I created easy-to-use build scripts to generate DANNA2 bitfiles for the FPGAs using the VHDL source files. The scripts allow the parameters of the DANNA2 hardware processor to be specified when building the bitfiles. See [Section 6.2.2](#) for details on the KCU1500 build system and see [Section 7.4.1](#) for details on the SNACC build system.

9.8 Applications

Another major contribution from this work is the testing and performance evaluation of multiple neuromorphic applications on the software simulator, the single-board neuromorphic processor, and the SNACC system. Each of these three methods of evaluating DANNA2 networks are deterministic and result in the same output, albeit with different runtimes. The performance of the DANNA2 evaluation methods was analyzed with various test networks, and factors which affect the runtime were identified and explored. Additionally, the applications of reservoir training, EONS, GRANT, and shortest path finding were tested, and their performance was assessed using each method of evaluation. The resulting output of these

applications matches each other regardless of the method of evaluation, since each evaluation method has matching deterministic behavior. See [Chapter 8](#) for details.

Chapter 10

Future Work

You have brains in your head. You have feet in your shoes. You can steer yourself any direction you choose. You're on your own. And you know what you know. And YOU are the one who'll decide where to go...

– Dr. Seuss, Oh, the Places You'll Go!

There are multiple directions in which future work on SNACC can be conducted. This chapter looks at many of these directions and discusses potential design ideas and considerations for each future research direction.

10.1 Additional Neuromorphic Applications

Now that a two-by-two sub-array SNACC prototype system has been built, there are a couple of different directions for future work. One direction is to explore training additional neuromorphic applications and to use the hardware implementations to train and evaluate larger neural networks than were previously possible. Real-time operation of SNACC could additionally be utilized to perform real-time classification while the new data is being streamed in.

10.2 Move SNACC Routing from Host to NACC

Another area for future work is to move the routing of SNACC sub-array packets from the host system to the NACCs. Currently, the host decides how to route the packets to the sub-array. The host is able to perform this task efficiently; however, as SNACC scales to larger numbers of sub-arrays, this routing will become more complex and the difficulty of the routing could vary depending on the number of sub-arrays in use. To hide this additional complexity from the host, the routing of sub-array packets could be moved to the NACC (or NACCs in the case of very-large systems). Another advantage of moving the routing to the NACCs is the added potential of being able to send fewer packets to a NACC by allowing it to broadcast a single packet to multiple sub-arrays.

In order to implement routing on the NACCs, the NACCs must be made aware of how many elements each sub-array implements. Since a NACC is currently only used to create a host to sub-array communication channel, it presently does not know anything about the information being transmitted to the sub-array. However, for a NACC to be able to perform routing, it will need to know how many elements are implemented on each board so that it knows how to route the packets to each sub-array. The NACCs could either be designed such that the routing information for the packets can be loaded into the hardware, or the NACC build scripts could be modified to set these routing parameters when the bitfiles are generated.

Another consideration with moving the routing to a NACC is how to handle sending the potentially large number of input fires. One option is to send one packet for every 104 input neurons present in the combined array. Another option is to implement support for variable length packets between the host and a NACC. This would allow the host to only send the required amount of packets, depending on the number of fires at the timestep, which potentially reduces the number of packets sent between the host and the NACCs even more. Similarly, variable length packets could also be used for the output fires, additionally reducing the number of packets sent back and forth.

10.3 DANNA2 Array Sub-partitioning

Now that very-large DANNA2 arrays can be built, the ability to sub-divide the array into multiple smaller partitions is desirable. These sub-partitions could be used to evaluate multiple networks generated by EONS in parallel, thus further accelerating EONS performance in hardware. It could also be used to evaluate multiple composited networks concurrently, as well as allow the hardware to be used for multiple unrelated tasks.

DANNA2 hardware was designed with the idea that the DANNA2 arrays would be made out of multiple five-by-five element tiles. (This is the reason the standard hardware sizes are multiples of five.) So to support sub-partitioning, the large DANNA2 array would need to be divided into five-by-five tiles with multiplexers and demultiplexers routing the input and output of the tiles. The outputs from a tile should be able to be routed to the next tile or routed to the control logic. Likewise, the inputs should be able to come from a neighboring tile or the control logic. However, if fan-in is limited between tiles, then the first column's fan-in port 0 could always come from the control logic. This limitation might be acceptable since fan-in has not been used in practice by any application. Additionally, if the software prevents illegal synapse connections, then the outputs could just be routed to the next tile and to the control logic.

With software imposed rules and restricted fan-in, the multiplexers and demultiplexers would not be needed to run multiple standard networks. The changes required to support sub-partitions would be to add additional connections into the control logic and to design a new packet scheme which can handle the added complexity. Although sub-partitioning has not yet been implemented with DANNA2, much of the groundwork is in place to make it possible.

10.4 SNACC Sparse Array Support

Currently, SNACC does not support DANNA2 sparse arrays. There are many challenges in supporting sparse arrays with SNACC. One primary challenge is the limited number of fires which can cross sub-array boundaries. This limitation is more restricting if a fire must cross

multiple sub-array boundaries. Additionally, increasing the number of communication hops further increases the inherent synaptic delay caused from communication.

To get around these hardware fire-routing restrictions, a hardware compiler would need to be created to minimize the communication distance for the synapses within the sparse array. Although the sparse array supports connections to any of the other elements, this freedom is restricted in SNACC, and a compiler is needed both to reduce the distance of these connections and also to minimize the number of connections across sub-array boundaries. Many neuromorphic systems already use network compilers to map neural networks onto hardware, and these compilers typically use similar algorithms to the place-and-route algorithms used by VLSI and FPGA tools.

Just as sparse arrays on single-board DANNA2 neuromorphic processors require regenerating the bitfile to load a new network, the loading of sparse arrays on SNACC would require regenerating the bitfiles. The difficulty of loading a sparse network on SNACC is made even more time consuming since the network will first have to be transformed by the network compiler, and then bitfiles will need to be regenerated for each sub-array in SNACC. Not every sparse network will be able to be mapped to SNACC. Depending on the network, the network compiler might not be able to find a way to transform the network to meet the restrictions placed on it by SNACC. Even with these limitations, sparse arrays on SNACC might be useful for large, already-trained networks. Once sparse networks are supported by SNACC, additional research can be performed to analyze the effects of array placement on sub-array communication.

10.5 Increase Density and Performance

Further work can also be done to increase the density of the sub-arrays or the performance of DANNA2 Hardware. If Time-Division Multiplexing (TDM) is used, multiple elements could be bundled together into a single element with even more logic sharing. This would reduce the frequency of the effective global network clock but would increase the neuron count per chip. Since SNACC has a fixed minimum latency between sub-arrays, this reduction

of network frequency would reduce the number of delay cycles needed to hide this latency. However, supporting TDM would require significant modifications to the element.

The host to NACC communication throughput can be increased by switching to Xillybus revision XXL. Revision XXL has 8 times the bandwidth of revision A and uses an internal data width of 256 bits. This revision allows a maximum bandwidth of 6600 MB/s. This bandwidth is four times the bandwidth of Xillybus revision B, which is used in a NACC, and two times the bandwidth of Xillybus revision Xl, which is used in DANNA2 single-board FPGA implementation.

DANNA2 could also be run at higher clock speeds for a little extra performance. The 690T can handle an element clock of 150 MHz (15 MHz network speed). Since the KCU1500 uses a newer FPGA, it can support an element clock of 200 MHz (20 MHz network speed). The exact maximum frequency the FPGAs are able to support depends on the size of the DANNA2 network being built. The operating frequency of sub-arrays will be further increased with a VLSI implementation of a DANNA2 neuromorphic processor.

Another area of potential performance improvement is the creation of hardware packets. One way packet creation could be improved is by creating packets in parallel and immediately, instead of waiting for a simulate call to sequentially create packets.

10.6 Streaming EO

As discussed in [Section 8.2.3](#), another area of future work is to implement streaming training with EONS on hardware DANNA2 neuromorphic processors. This feature would allow the hardware to be better used for training networks with evolutionary optimization.

Chapter 11

Conclusion

The highest level of mastery is simplicity. Most information is irrelevant and most effort wasted, but only the expert knows what to ignore.

– James Clear

A new large-scale neuromorphic communications system was designed and demonstrated with the Scaled-up Neuromorphic Array Communications Controller (SNACC) system. This system makes use of local point-to-point connections and a Neuromorphic Array Communications Controller (NACC) to enable the scaling of DANNA2 neuromorphic processors to very large scales with the potential for millions of neurons and billions of synapses. The point-to-point connections are implemented with the Aurora 64B66B protocol with a newly-created, custom, automatic repeat request protocol. Xilinx GTX and GTH transceivers are used to send the signal over SMA cables between the components. The performance characteristics of SNACC and DANNA2 on a single-board system were explored with the creation of DANNA2 networks with various test patterns. The performance of multiple neuromorphic applications and training methods were also collected and discussed. The new SNACC system allows for the scaling of DANNA2 networks to massive sizes without compromising the performance of DANNA2. SNACC can be used to prototype large neuromorphic systems to enable development of such systems before they are implemented into an on-chip VLSI design. SNACC is able to scale to meet the future needs of the TENNLab research group at the University of Tennessee and is flexible enough to be modified to support additional neuromorphic processor designs.

Bibliography

- [1] F. Akopyan et al. “TrueNorth: Design and Tool Flow of a 65 mW 1 Million Neuron Programmable Neurosynaptic Chip”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34.10 (Oct. 2015), pp. 1537–1557. ISSN: 0278-0070. DOI: [10.1109/TCAD.2015.2474396](https://doi.org/10.1109/TCAD.2015.2474396) (cit. on pp. 1, 2, 14, 19, 22–24, 27, 31).
- [2] Jonathan Daniel Ambrose. “GRANT: Ground Roaming Autonomous Neuromorphic Targeter”. MA thesis. University of Tennessee, 2020 (cit. on pp. 44, 238, 239, 241).
- [3] Arnon Amir et al. “Cognitive computing programming paradigm: a corelet language for composing networks of neurosynaptic cores”. In: *The 2013 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2013, pp. 1–10 (cit. on p. 14).
- [4] Rajagopal Ananthanarayanan et al. “The cat is out of the bag: cortical simulations with 10 9 neurons, 10 13 synapses”. In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM. 2009, p. 63 (cit. on p. 2).
- [5] ARM. *AMBA 4 AXI4-Stream Protocol. Specification*. Version 1.0. ARM. 2010 (cit. on pp. 73–75).
- [6] Lars Asplund et al. *VUnit*. 2019. URL: <https://vunit.github.io/> (cit. on p. 89).
- [7] Frederico A. C. Azevedo et al. “Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain”. In: *Journal of Comparative Neurology* 513.5 (2009), pp. 532–541 (cit. on p. 2).
- [8] John Backus. “Can Programming Be Liberated from the Von Neumann Style?: A Functional Style and Its Algebra of Programs”. In: *Communications of the ACM* 21.8 (Aug. 1978), pp. 613–641. ISSN: 0001-0782. DOI: [10.1145/359576.359579](https://doi.org/10.1145/359576.359579). URL: <http://doi.acm.org/10.1145/359576.359579> (cit. on p. 7).
- [9] F. Barchi et al. “Flexible on-line reconfiguration of multi-core neuromorphic platforms”. In: *IEEE Transactions on Emerging Topics in Computing* (2019), pp. 1–1. ISSN: 2168-6750. DOI: [10.1109/TETC.2019.2908079](https://doi.org/10.1109/TETC.2019.2908079) (cit. on pp. 13, 30).
- [10] 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 pp. 9, 23, 25–27, 34).

- [11] 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](https://doi.org/10.1109/ISCAS.2007.378041) (cit. on p. 18).
- [12] K. Boahen. “A neuromorph’s prospectus”. In: *Computing in Science Engineering* 19.2 (Mar. 2017), pp. 14–28. ISSN: 1521-9615. DOI: [10.1109/MCSE.2017.33](https://doi.org/10.1109/MCSE.2017.33) (cit. on pp. 18, 22, 25, 26, 36).
- [13] Maxence Bouvier et al. “Spiking Neural Networks Hardware Implementations and Challenges: A Survey”. In: *J. Emerg. Technol. Comput. Syst.* 15.2 (Apr. 2019), 22:1–22:35. ISSN: 1550-4832. DOI: [10.1145/3304103](https://doi.org/10.1145/3304103). URL: <http://doi.acm.org/10.1145/3304103> (cit. on p. 8).
- [14] S. Buckley et al. “Design of superconducting optoelectronic networks for neuromorphic computing”. In: *IEEE International Conference on Rebooting Computing*. Tysons, VA, Nov. 2018, pp. 36–42 (cit. on p. 50).
- [15] N. Cady et al. “Adapting Spiking Networks to Neuromorphic Hardware Subject to Process Variability”. In: *44th Annual GOMACTech Conference*. Albuquerque, Mar. 2019. URL: <http://neuromorphic.eecs.utk.edu/raw/files/publications/2019-Gomac-Cady.pdf> (cit. on pp. 3, 49).
- [16] Nathaniel C. Cady et al. “Full CMOS-memristor Implementation of a Dynamic Neuromorphic Architecture”. In: *2018 GOMACTech Conference*. Miami, Mar. 2018 (cit. on pp. 3, 49).
- [17] G. Chakma et al. “A Mixed-Signal Approach to Memristive Neuromorphic System Design”. In: *60th IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*. Boston, MA, Aug. 2017 (cit. on pp. 3, 49).
- [18] Jason Chan. “Implementation of a Neuromorphic Development Platform with DANNA”. MA thesis. University of Tennessee, 2015 (cit. on pp. 48, 69).

- [19] V. Chan, S. Liu, and A. van Schaik. “AER EAR: A Matched Silicon Cochlea Pair With Address Event Representation Interface”. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 54.1 (Jan. 2007), pp. 48–59. ISSN: 1549-8328. DOI: [10.1109/TCSI.2006.887979](https://doi.org/10.1109/TCSI.2006.887979) (cit. on p. 24).
- [20] Eric H. Chudler. *The Hows, Whats and Whos of Neuroscience*. Apr. 11, 2019. URL: <https://faculty.washington.edu/chudler/what.html> (cit. on p. 17).
- [21] Gregory Cohen et al. “Event-based Sensing for Space Situational Awareness”. In: *The Journal of the Astronautical Sciences* 66.2 (June 2019), pp. 125–141. ISSN: 2195-0571. DOI: [10.1007/s40295-018-00140-5](https://doi.org/10.1007/s40295-018-00140-5). URL: <https://doi.org/10.1007/s40295-018-00140-5> (cit. on p. 24).
- [22] Wikipedia contributors. *Sliding window protocol — Wikipedia, The Free Encyclopedia*. 2019. URL: https://en.wikipedia.org/wiki/Sliding_window_protocol (cit. on pp. 113, 283).
- [23] M. Davies et al. “Loihi: A Neuromorphic Manycore Processor with On-Chip Learning”. In: *IEEE Micro* 38.1 (Jan. 2018), pp. 82–99. ISSN: 0272-1732. DOI: [10.1109/MM.2018.112130359](https://doi.org/10.1109/MM.2018.112130359) (cit. on pp. 14, 19, 30, 35, 37).
- [24] Andrew P. Davison et al. *HBP Neuromorphic Computing Platform Guidebook. About the BrainScaleS hardware*. June 26, 2019. URL: https://electronicvisions.github.io/hbp-sp9-guidebook/pm/pm_hardware_configuration.html (cit. on p. 36).
- [25] M. E. Dean and C. Daffron. “A VLSI Design for Neuromorphic Computing”. In: *IEEE Annual Symposium on VLSI (ISVLSI)*. IEEE, July 2016. DOI: [10.1109/ISVLSI.2016.81](https://doi.org/10.1109/ISVLSI.2016.81) (cit. on p. 47).
- [26] 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. 3, 16, 47, 66).

- [27] 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. 69).
- [28] M. V. DeBole et al. “TrueNorth: Accelerating From Zero to 64 Million Neurons in 10 Years”. In: *Computer* 52.5 (May 2019), pp. 20–29. ISSN: 0018-9162. DOI: [10.1109/MC.2019.2903009](https://doi.org/10.1109/MC.2019.2903009) (cit. on pp. 37, 38).
- [29] Manaskant Desai and Karthik Sivaramakrishnan. *Optimizing Usb 3.0 Throughput With Ez-Usb Fx3*. AN86947. Version C. May 9, 2017 (cit. on pp. 125, 127).
- [30] Digilent. *PYNQ-Z1*. 2020. URL: <https://reference.digilentinc.com/reference/programmable-logic/pynq-z1/start> (visited on 02/15/2020) (cit. on p. 240).
- [31] Adam W. Disney. “Software Support for Dynamic Adaptive Neural Network Arrays”. PhD thesis. University of Tennessee, Aug. 2018. URL: https://trace.tennessee.edu/utk_graddiss/5049 (cit. on p. 47).
- [32] Peter Dittrich. “Chemical Computing”. In: *Unconventional Programming Paradigms*. Ed. by Jean-Pierre Banâtre et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 19–32. ISBN: 978-3-540-31482-0 (cit. on p. 3).
- [33] P. J. Eckhart. “Tiled DANNA: Dynamic Adaptive Neural Network Array Scaled Across Multiple Chips”. MA thesis. University of Tennessee, 2017 (cit. on pp. 75, 77–82).
- [34] 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](https://doi.org/10.1109/ISCAS.2008.4541501) (cit. on p. 17).
- [35] S. Fok and K. Boahen. “A Serial H-Tree Router for Two-Dimensional Arrays”. In: *2018 24th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*. May 2018, pp. 78–85. DOI: [10.1109/ASYNC.2018.00026](https://doi.org/10.1109/ASYNC.2018.00026) (cit. on p. 27).
- [36] S. B. Furber et al. “Overview of the SpiNNaker System Architecture”. In: *IEEE Transactions on Computers* 62.12 (Dec. 2013), pp. 2454–2467. ISSN: 0018-9340. DOI: [10.1109/TC.2012.142](https://doi.org/10.1109/TC.2012.142) (cit. on p. 36).

- [37] S. Furber, S. Temple, and A. Brown. “On-chip and inter-chip networks for modeling large-scale neural systems”. In: *2006 IEEE International Symposium on Circuits and Systems*. May 2006, pp. 1945–1948. DOI: [10.1109/ISCAS.2006.1692992](https://doi.org/10.1109/ISCAS.2006.1692992) (cit. on p. 23).
- [38] Stephen Furber and Andrew Brown. “Biologically-inspired massively-parallel architectures-computing beyond a million processors”. In: *Ninth International Conference On Application of Concurrency to System Design, 2009. ACSD’09*. IEEE. July 2009, pp. 3–12. DOI: [10.1109/ACSD.2009.17](https://doi.org/10.1109/ACSD.2009.17). URL: <https://eprints.soton.ac.uk/270985/1/PID871138.pdf> (cit. on pp. 11, 26, 27, 29, 30).
- [39] Tristan Gingold. *What is GHDL?* 2019. URL: <https://ghdl.readthedocs.io/en/latest/about.html> (cit. on p. 86).
- [40] Tristan Gingold et al. *GHDL*. 2019. URL: <https://github.com/ghdl/ghdl> (cit. on p. 86).
- [41] K. E. Hamilton, T. M. Mintz, and C. D. Schuman. “Spike-based primitives for graph algorithms”. In: *CoRR* abs/1903.10574 (2019). eprint: [1903.10574](https://arxiv.org/abs/1903.10574). URL: <http://arxiv.org/abs/1903.10574> (cit. on pp. 208, 210).
- [42] M. S. Hasan et al. “Biomimetic, Soft-Material Synapse for Neuromorphic Computing: From Device to Network”. In: *IEEE 13th Dallas Circuits and Systems Conference (DCAS)*. Nov. 2018. DOI: [10.1109/DCAS.2018.8620187](https://doi.org/10.1109/DCAS.2018.8620187). URL: <https://ieeexplore.ieee.org/abstract/document/8620187> (cit. on p. 50).
- [43] Jonathan Heathcote. “Building and operating large-scale SpiNNaker machines”. PhD thesis. The University of Manchester (United Kingdom), 2016 (cit. on p. 36).
- [44] HiTech Global. *HTG-777: Virtex 7 FPGA FMC Module*. 2019. URL: http://www.hitechglobal.com/Boards/Virtex-7_FMC.htm (cit. on p. 86).
- [45] Sebastian Hoppner. “SpiNNaker2. Towards extremely efficient digital neuromorphics and multi-scale brain emulation”. In: *Nice Workshop*. 2018 (cit. on pp. 11, 37).
- [46] Human Brain Project. *Hardware*. Apr. 20, 2019. URL: <https://www.humanbrainproject.eu/en/silicon-brains/how-we-work/hardware/> (cit. on p. 36).

- [47] Human Brain Project. *Short Overview of the Human Brain Project*. Apr. 10, 2019. URL: <https://www.humanbrainproject.eu/en/about/overview/> (cit. on p. 9).
- [48] Intel. *Intel’s Pohoiki Beach, a 64-Chip Neuromorphic System, Delivers Breakthrough Results in Research Tests*. July 15, 2019. URL: <https://newsroom.intel.com/news/intels-pohoiki-beach-64-chip-neuromorphic-system-delivers-breakthrough-results-research-tests/> (cit. on p. 37).
- [49] Eugene M. Izhikevich and Jeff Moehlis. “Dynamical Systems in Neuroscience: The geometry of excitability and bursting”. In: *SIAM review* 50.2 (2008), p. 397 (cit. on p. 1).
- [50] A. Jimenez-Fernandez et al. “AER-based robotic closed-loop control system”. In: *2008 IEEE International Symposium on Circuits and Systems*. May 2008, pp. 1044–1047. DOI: [10.1109/ISCAS.2008.4541600](https://doi.org/10.1109/ISCAS.2008.4541600) (cit. on p. 25).
- [51] Ole Kröger. *Neuromorphic computing: An Overview*. June 18, 2019. URL: <https://opensourc.es/blog/neuromorphic> (cit. on p. 92).
- [52] N. M. Linke et al. “Comparing the architectures of the first programmable quantum computers”. In: *2017 Conference on Lasers and Electro-Optics Europe European Quantum Electronics Conference (CLEO/Europe-EQEC)*. June 2017, pp. 1–1. DOI: [10.1109/CLEOE-EQEC.2017.8087391](https://doi.org/10.1109/CLEOE-EQEC.2017.8087391) (cit. on p. 3).
- [53] De Ma et al. “Darwin: A neuromorphic hardware co-processor based on spiking neural networks”. In: *Journal of Systems Architecture* 77 (2017), pp. 43–51. ISSN: 1383-7621. DOI: <https://doi.org/10.1016/j.sysarc.2017.01.003>. URL: <http://www.sciencedirect.com/science/article/pii/S1383762117300231> (cit. on pp. 2, 14, 18, 24, 35, 39).
- [54] Henry Markram. “A countdown to a digital simulation of every last neuron in the human brain”. In: *Sci. Am* 306.6 (2012) (cit. on p. 83).
- [55] Henry Markram. “The blue brain project”. In: *Nature Reviews Neuroscience* 7.2 (2006), p. 153 (cit. on p. 1).

- [56] Henry Markram et al. “Introducing the Human Brain Project”. In: *Procedia Computer Science* 7 (2011). Proceedings of the 2nd European Future Technologies Conference and Exhibition 2011 (FET 11), pp. 39–42. ISSN: 1877-0509. DOI: <https://doi.org/10.1016/j.procs.2011.12.015>. URL: <http://www.sciencedirect.com/science/article/pii/S1877050911006806> (cit. on p. 9).
- [57] A. J. Martin and M. Nystrom. “Asynchronous Techniques for System-on-Chip Design”. In: *Proceedings of the IEEE* 94.6 (June 2006), pp. 1089–1120. ISSN: 0018-9219. DOI: [10.1109/JPROC.2006.875789](https://doi.org/10.1109/JPROC.2006.875789) (cit. on pp. 23, 130).
- [58] Alain J. Martin. *Programming in VLSI: From communicating processes to delay-insensitive circuits*. Tech. rep. CALIFORNIA INST OF TECH PASADENA DEPT OF COMPUTER SCIENCE, 1989 (cit. on p. 23).
- [59] Alain J. Martin et al. *The design of an asynchronous microprocessor*. Tech. rep. CALIFORNIA INST OF TECH PASADENA DEPT OF COMPUTER SCIENCE, 1989 (cit. on p. 23).
- [60] P. Merolla et al. “A digital neurosynaptic core using embedded crossbar memory with 45pJ per spike in 45nm”. In: *2011 IEEE Custom Integrated Circuits Conference (CICC)*. Sept. 2011, pp. 1–4. DOI: [10.1109/CICC.2011.6055294](https://doi.org/10.1109/CICC.2011.6055294) (cit. on pp. 14, 25, 34).
- [61] 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 pp. 10, 28).
- [62] J. P. Mitchell. “DANNA2: Dynamic Adaptive Neural Network Arrays”. MA thesis. University of Tennessee, 2018. URL: http://trace.tennessee.edu/utk_gradthes/5167 (cit. on pp. 49, 53, 68, 92).
- [63] J. P. Mitchell et al. “DANNA 2: Dynamic Adaptive Neural Network Arrays”. In: *International Conference on Neuromorphic Computing Systems*. Knoxville, TN: ACM, July 2018. DOI: [10.1145/3229884.3229894](https://doi.org/10.1145/3229884.3229894) (cit. on pp. 3, 49, 53, 54).

- [64] J. P. Mitchell et al. “NeoN: Neuromorphic Control for Autonomous Robotic Navigation”. In: *IEEE 5th International Symposium on Robotics and Intelligent Sensors*. Ottawa, Canada, Oct. 2017, pp. 136–142. DOI: [test](#) (cit. on p. 44).
- [65] Don Monroe. “Neuromorphic computing gets ready for the (really) big time”. In: *Communications of the ACM* 57.6 (2014), pp. 13–15 (cit. on pp. 3, 42, 104).
- [66] S. Moradi et al. “A Scalable Multicore Architecture With Heterogeneous Memory Structures for Dynamic Neuromorphic Asynchronous Processors (DYNAPs)”. In: *IEEE Transactions on Biomedical Circuits and Systems* 12.1 (Feb. 2018), pp. 106–122. ISSN: 1932-4545. DOI: [10.1109/TBCAS.2017.2759700](#) (cit. on pp. 15, 20, 21, 24, 26, 30, 32, 34, 39).
- [67] A. Neckar et al. “Braindrop: A Mixed-Signal Neuromorphic Architecture With a Dynamical Systems-Based Programming Model”. In: *Proceedings of the IEEE* 107.1 (Jan. 2019), pp. 144–164. ISSN: 0018-9219. DOI: [10.1109/JPROC.2018.2881432](#) (cit. on pp. 9, 36).
- [68] 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. 49).
- [69] PixyCam. *Pixy2*. 2020. URL: <https://pixycam.com/pixy2/> (cit. on p. 25).
- [70] J. S. Plank et al. “A Unified Hardware/Software Co-Design Framework for Neuromorphic Computing Devices and Applications”. In: *IEEE International Conference on Rebooting Computing (ICRC 2017)*. Washington, DC, Nov. 2017 (cit. on p. 52).
- [71] J. S. Plank et al. “The TENNLab Exploratory Neuromorphic Computing Framework”. In: *IEEE Letters of the Computer Society* 1.2 (July 2018), pp. 17–20. DOI: [10.1109/LOCS.2018.2885976](#). URL: <https://doi.ieeecomputersociety.org/10.1109/LOCS.2018.2885976> (cit. on pp. 42, 45, 46, 52).

- [72] J. S. Plank et al. “The TENNLab Suite of LIDAR-Based Control Applications for Recurrent, Spiking, Neuromorphic Systems”. In: *44th Annual GOMACTech Conference*. Albuquerque, Mar. 2019. URL: <http://neuromorphic.eecs.utk.edu/raw/files/publications/2019-Plank-Gomac.pdf> (cit. on p. 44).
- [73] James S. Plank. *COSC 302: Data Structures/Algorithms II, Lab 7: Solving Mazes with DFS*. 2013 (cit. on pp. 209, 211).
- [74] James S. Plank. *COSC 302: Data Structures/Algorithms II, Lab 8: Dijkstra’s Algorithm – City Streets*. 2013 (cit. on pp. 209, 211).
- [75] Queensland Brain Institute. *Action potentials and synapses*. Nov. 9, 2017. URL: <https://qbi.uq.edu.au/brain-basics/brain/brain-physiology/action-potentials-and-synapses> (cit. on p. 17).
- [76] Bipin Rajendran et al. “Low-Power Neuromorphic Hardware for Signal Processing Applications”. In: *arXiv* (Jan. 11, 2019). arXiv: <http://arxiv.org/abs/1901.03690v2> [cs.ET] (cit. on p. 8).
- [77] J. J. M. Reynolds, J. S. Plank, and C. D. Schuman. “Intelligent Reservoir Generation for Liquid State Machines using Evolutionary Optimization”. In: *IJCNN: The International Joint Conference on Neural Networks*. Budapest, 2019, pp. 1–8. DOI: [10.1109/IJCNN.2019.8852472](https://doi.org/10.1109/IJCNN.2019.8852472) (cit. on pp. 45, 204, 205).
- [78] J. Sawada et al. “TrueNorth Ecosystem for Brain-Inspired Computing: Scalable Systems, Software, and Applications”. In: *SC ’16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Nov. 2016, pp. 130–141. DOI: [10.1109/SC.2016.11](https://doi.org/10.1109/SC.2016.11) (cit. on pp. 19, 22, 30, 33, 34, 39).
- [79] J. Schemmel, J. Fieres, and K. Meier. “Wafer-scale integration of analog neural networks”. In: *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*. June 2008, pp. 431–438. DOI: [10.1109/IJCNN.2008.4633828](https://doi.org/10.1109/IJCNN.2008.4633828) (cit. on pp. 9, 26, 28).

- [80] 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](https://doi.org/10.1109/ISCAS.2010.5536970) (cit. on pp. 9, 26).
- [81] Johannes Schemmel. “Towards the Second Generation BrainScaleS System”. In: *NICE Workshop*. 2018 (cit. on p. 11).
- [82] 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. 12, 18, 26, 29, 30).
- [83] 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](https://doi.org/10.3389/fnins.2011.00117). URL: <http://journal.frontiersin.org/article/10.3389/fnins.2011.00117> (cit. on pp. 9, 24).
- [84] 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. 47).
- [85] 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](https://doi.org/10.1371/journal.pone.0080455) (cit. on p. 46).
- [86] 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](https://doi.org/10.1016/j.bica.2013.05.001) (cit. on p. 46).
- [87] C. D. Schuman, J. D. Birdwell, and M. E. Dean. “Neuroscience-Inspired Dynamic Architectures”. In: *Biomedical Science and Engineering Center Conference (BSEC), Oak Ridge National Laboratory*. Apr. 2014 (cit. on p. 3).

- [88] C. D. Schuman et al. *A Survey of Neuromorphic Computing and Neural Networks in Hardware*. arXiv:1705.06963. May 2017. URL: <https://arxiv.org/abs/1705.06963> (cit. on p. 8).
- [89] C. D. Schuman et al. “An Evolutionary Optimization Framework for Neural Networks and Neuromorphic Architectures”. In: *International Joint Conference on Neural Networks*. Vancouver, July 2016 (cit. on p. 44).
- [90] C. D. Schuman et al. “Neuromorphic Computing: A Post-Moore’s Law Complementary Architecture”. In: *International Workshop on Post-Moore’s Era Supercomputing (PMES)*. Salt Lake City, UT, Nov. 2016 (cit. on p. 3).
- [91] Catherine D. Schuman. “Neuroscience-Inspired Dynamic Architectures”. PhD thesis. University of Tennessee, May 2015 (cit. on p. 46).
- [92] William Severa et al. “Training deep neural networks for binary communication with the Whetstone method”. In: *Nature Machine Intelligence* 1.2 (Jan. 2019), pp. 86–94. ISSN: 2522-5839. DOI: [10.1038/s42256-018-0015-y](https://doi.org/10.1038/s42256-018-0015-y). URL: <http://dx.doi.org/10.1038/s42256-018-0015-y> (cit. on p. 45).
- [93] 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 p. 2).
- [94] Juncheng Shen et al. “Darwin: a neuromorphic hardware co-processor based on Spiking Neural Networks”. In: *Science China Information Sciences* 59.2 (Feb. 2016), pp. 1–5. ISSN: 1869-1919. DOI: [10.1007/s11432-015-5511-7](https://doi.org/10.1007/s11432-015-5511-7). URL: <https://doi.org/10.1007/s11432-015-5511-7> (cit. on pp. 14, 15, 34).
- [95] Slamtec. *RPLIDAR-A2 Laser Range Scanner*. 2020. URL: <https://www.slamtec.com/en/Lidar/A2> (cit. on p. 25).
- [96] TENNLab. *TENNLab - Neuromorphic Architectures, Learning, Applications*. 2019. URL: <http://neuromorphic.eecs.utk.edu/> (cit. on p. 42).

- [97] Texas Instruments. *LMK03328EVM Ultra-Low-Jitter Clock Generator EVM With 2 PLLs, 8 Differential Outputs, and 2 Inputs*. 2019. URL: <http://www.ti.com/tool/LMK03328EVM> (cit. on p. 86).
- [98] Chetan Singh Thakur et al. “Large-Scale Neuromorphic Spiking Array Processors: A Quest to Mimic the Brain”. In: *Frontiers in Neuroscience* 12 (2018), p. 891. ISSN: 1662-453X. DOI: [10.3389/fnins.2018.00891](https://doi.org/10.3389/fnins.2018.00891). URL: <https://www.frontiersin.org/article/10.3389/fnins.2018.00891> (cit. on pp. 8, 15, 39).
- [99] The University of Manchester. *SpiNNaker Project - Boards and Machines*. 2019. URL: <http://apt.cs.manchester.ac.uk/projects/SpiNNaker/hardware/> (cit. on pp. 34, 36).
- [100] The University of Manchester. *SpiNNaker Project - System Software*. 2019. URL: <http://apt.cs.manchester.ac.uk/projects/SpiNNaker/software/> (cit. on p. 34).
- [101] R. Tsui. “Molecular-scale engineering for future electronics”. In: *2002 IEEE International Symposium on Circuits and Systems. Proceedings (Cat. No.02CH37353)*. Vol. 2. May 2002, pp. II–. DOI: [10.1109/ISCAS.2002.1010918](https://doi.org/10.1109/ISCAS.2002.1010918) (cit. on p. 3).
- [102] G. Urgese et al. “Optimizing Network Traffic for Spiking Neural Network Simulations on Densely Interconnected Many-Core Neuromorphic Platforms”. In: *IEEE Transactions on Emerging Topics in Computing* 6.3 (July 2018), pp. 317–329. ISSN: 2168-6750. DOI: [10.1109/TETC.2016.2579605](https://doi.org/10.1109/TETC.2016.2579605) (cit. on p. 22).
- [103] M. Mitchell Waldrop. “The chips are down for Moore’s law”. In: *Nature* 530.7589 (Feb. 2016), pp. 144–147. DOI: [10.1038/530144a](https://doi.org/10.1038/530144a) (cit. on p. 3).
- [104] R. Weiss et al. “A Soft-Matter Biomolecular Memristor Synapse for Neuromorphic Systems”. In: *IEEE Biomedical Circuits and Systems Conference (BioCAS)*. Oct. 2018. DOI: [10.1109/BIOCAS.2018.8584668](https://doi.org/10.1109/BIOCAS.2018.8584668). URL: <https://ieeexplore.ieee.org/abstract/document/8584668> (cit. on p. 50).
- [105] Wikichip. *Loihi - Intel*. Aug. 8, 2019. URL: <https://en.wikichip.org/wiki/intel/loihi> (cit. on pp. 14, 30, 37).

- [106] J. C. Willis. “Middleware and Services for Dynamic Adaptive Neural Network Arrays”. MA thesis. University of Tennessee, 2015 (cit. on pp. 66, 69).
- [107] J. Wu, S. Furber, and J. Garside. “A Programmable Adaptive Router for a GALS Parallel System”. In: *2009 15th IEEE Symposium on Asynchronous Circuits and Systems*. May 2009, pp. 23–31. DOI: [10.1109/ASYNC.2009.17](https://doi.org/10.1109/ASYNC.2009.17) (cit. on pp. 2, 16, 18, 24, 29).
- [108] Xilinx. *7 Series FPGAs Data Sheet: Overview*. DS180. Feb. 27, 2018. URL: https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf (cit. on p. 56).
- [109] Xilinx. *Aurora 64B/66B Protocol Specification*. SP011. Version 1.3. Xilinx. Oct. 1, 2014. URL: https://www.xilinx.com/support/documentation/ip_documentation/aurora_64b66b_protocol_spec_sp011.pdf (visited on 06/14/2017) (cit. on p. 72).
- [110] Xilinx. *Aurora 64B/66B v12.0. LogiCORE IP Product Guide*. PG046. Version 11.0. Xilinx. May 22, 2019. 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. 72, 73).
- [111] 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 p. 72).
- [112] 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. 72).
- [113] Xilinx. *Vivado Design Suite*. 2019. URL: <https://www.xilinx.com/products/design-tools/vivado.html> (cit. on p. 85).
- [114] Xilinx. *Xilinx Kintex UltraScale FPGA KCU1500 Acceleration Development Kit*. 2019. URL: <https://www.xilinx.com/products/boards-and-kits/dk-u1-kcu1500-g.html> (cit. on p. 86).

- [115] Xilinx. *Xilinx Power Estimator (XPE)*. 2020. URL: <https://www.xilinx.com/products/technology/power/xpe.html> (cit. on p. 166).
- [116] Xilinx. *Xilinx Virtex-7 FPGA VC707 Evaluation Kit*. 2019. URL: <https://www.xilinx.com/products/boards-and-kits/ek-v7-vc707-g.html> (cit. on p. 86).
- [117] 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. 70, 71).
- [118] Xillybus Ltd. *Revision B/XL/XXL user notes*. 2019. URL: <http://xillybus.com/doc/revision-b-xl> (visited on 06/12/2017) (cit. on pp. 72, 129).
- [119] A. Yen, Y. Mensah, and M. E. Dean. “Poster: SABR: Development of a Neuromorphic Balancing Robot”. In: *EURECA: Exhibition of Undergraduate Research and Creative Achievement, University of Tennessee*. Knoxville, TN, Apr. 2019 (cit. on p. 44).
- [120] A. R. Young et al. “A Review of Spiking Neuromorphic Hardware Communication Systems”. In: *IEEE Access* 7 (2019), pp. 135606–135620. DOI: [10.1109/ACCESS.2019.2941772](https://doi.org/10.1109/ACCESS.2019.2941772) (cit. on pp. 1, 7, 255).
- [121] A. R. Young et al. “Neuromorphic Array Communications Controller to Support Large-Scale Neural Networks”. In: *IJCNN: The International Joint Conference on Neural Networks*. Rio de Janeiro, Brazil, July 2018 (cit. on pp. 69, 122).
- [122] Aaron Reed Young. “Scalable High-Speed Communications for Neuromorphic Systems”. MA thesis. University of Tennessee, 2017 (cit. on pp. 69, 71, 83, 153, 170).

Appendix

A Abbreviations and Symbols

AER	Address-Event Representation
AMBA	Advanced Microcontroller Bus Architecture
ARM	Advanced RISC Machine
ARQ	Automatic Repeat Request
ASIC	Application-Specific Integrated Circuit
AXI	Advanced eXtensible Interface
CI	Continuous Integration
CRC	Cyclic Redundancy Check
CSV	Comma-Separated Values
DANNA	Dynamic Adaptive Neural Network Arrays
DIP Switch	Dual In-line Package Switch
DMA	Direct Memory Access
DP	Data Pair
ECC	Error Correction Code
EFEC	Enhanced Forward Error Correction
EO	Evolutionary Optimization
EONS	Evolutionary Optimization of Neuromorphic Systems
EZ-USB FX3	Cypress USB 3.0 Peripheral Controller
FEC	Forward Error Correction
FIFO	First In, First Out
FMC	FPGA Mezzanine Card
FPGA	Field-Programmable Gate Array
FSM	Finite-State Machine
GALS	Globally Asynchronous Locally Synchronous
GPIF	General Programmable Interface
HPC	High Pin Count
IP	Intellectual Property
LIDAR	Light Detection and Ranging

LIF	Leaky Integrate-and-Fire
MGT	Multi-gigabit Transceivers
NACC	Neuromorphic Communications Controller
NIDA	Neuroscience-Inspired Dynamic Architecture
No-Op	No Operation
OS	Operating System
PCB	Printed Circuit Board
PCIe	Peripheral Component Interconnect Express
PC	Personal Computer
RISC	Reduced Instruction Set Computer
SATA	Serial ATA
SDK	Software Development Kit
SECDED	Single-bit Error Correction and Double-bit Error Detection
SERDES	Serializer/Deserializer
SMA Connector	SubMiniature Version A Connector
SNACC	Scaled-up Neuromorphic Array Communications Controller
SRNN	Spiking Recurrent Neural Network
Tcl	Tool Command Language
TCP	Transmission Control Protocol
TDM	Time-Division Multiplexing
TENNLab	Laboratory of Tennesseans Exploring Neural Networks
USB	Universal Serial Bus
VLSI	Very-Large-Scale Integration

B Pseudocode

B.1 Go-Back-N

Go-Back-N pseudocode [22]:

N = window size

Rn = request number

Sn = sequence number

Sb = sequence base

Sm = sequence max

Receiver :

 Rn = 0

 Do the following forever :

 If the packet received = Rn and the packet is error free

 Accept the packet and send it to a higher layer

 Rn = Rn + 1

 Else

 Refuse packet

 Send a Request for Rn

Sender :

 Sb = 0

 Sm = N + 1

 Repeat the following steps forever :

 1. If you receive a request number where $Rn > Sb$

$Sm = (Sm - Sb) + Rn$

$Sb = Rn$

 2. If no packet is in transmission ,

 Transmit a packet where $Sb \leq Sn \leq Sm$.

 Packets are transmitted in order .

B.2 Window Buffer

Add Packet :

```
if (wr_tvalid == 1 and full == 0) then
    window_memory[p_end] = wr_tdata;
    keep_memory[p_end] = wr_tkeep;
    p_end++;
    count++;
end if;
```

Drop Packet :

```
if (drop == 1 and empty == 0) then
    p_start++;
    count--;
end if;
```

B.3 Aurora ACK Send

ack_set	indicates that a new acknowledgment is ready to be sent.
ack_send	is the last packet number received , ready to be acknowledged.
ack_num	is the last packet number sent which has been acknowledged.
Sm	is the max sequence number. It is used to add packets to the window buffer .
Sn	is the Sequence number.
Si	is the window index. This number is used by the send control to indicate the next packet to be sent.
send_state	is the state of the send and drop state machine.
MAX_FLOOD	is how many resends to issue before waiting.
FLOOD_WAIT	is how long to wait once MAX_FLOOD is reached.
resend_count	is a counter which counts how many resends have been sent.
wait_count	is a counter which counts how many cycles have been waited when flood control is active.

From the Receive Process :

```
ack_set
ack_send
```

ack_num

Read Control(clk, reset):

if (reset) then

Sm <= 1

end if;

Repeat on clock:

If a new valid packet is received and there is room in the window buffer then

Add the packet to the window buffer. The packet's Sn = Sm.

Sm++;

Reset resend_count.

end if;

Send and Drop Control(clk, reset):

If (reset) then

Si <= 0;

send_state <= START;

resend_count <= 0;

wait_count <= 0;

end if;

Repeat on clock:

if (resend_count == MAX_FLOOD) then

wait_count++;

if (wait_count == FLOOD_WAIT) then

resend_count <= 0;

wait_count <= 0;

end if;

end if;

switch send_state:

when START:

If the AXIS output is ready, the window buffer has data, and

resend_count < MAX_FLOOD then

```

        send__state <= SEND_PACKET;
    end if;

    If the window buffer is not empty and the ack number is greater than
    or equal to the first packet's Sn or the ack number has wrapped around
    then
        send__state <= DROP;
    end if;

    If there is a new ack number and the sender is ready then
        send__state <= SEND_ACK;
    end if;

when SEND_PACKET:
    Si++;
    if (Si >= window_size) then
        Si <= 0;
        resend_count++;
    end if;
    Send packet window_buffer[Si] with D=1, An=ack_send, and Sn=Sn.
    send__state <= START;

when DROP:
    Drop the first packet from the window buffer.
    Si--;
    send__state <= START;

when SEND_ACK:
    Send ACK packet with D=0, An=ack_send.
    send__state <= START;

end switch;

```

B.4 Aurora ACK Receive

ack_set indicates that a new acknowledgment is ready to be sent.
ack_send is the last packet number received, ready to be acknowledged.
ack_num is the last packet number sent which has been acknowledged.
Sn is the Sequence number.
Rn is the last received sequence number.

Receive Process(clk, reset):

 if (reset) then

 Rn <= 0;

 ack_set <= 0;

 ack_send <= 0;

 ack_num <= 0;

 end if;

Repeat on clock:

 When data is successfully sent to the received axis, increment the ACK count.

 Rn++;

 ack_send <= Rn;

 ack_set <= 1;

 If packet is received and CRC is valid then

 If the packet is a data packet with full data or the packet is a header only with only one valid word then

 Pull out the ack_number

 ack_num <= packet[An];

 end if;

 If the packet is a valid data packet, the packet is the next in the sequence (packet[Sn] = Rn), and the receiver is ready then

 Send the packet to the receiver.

 end if;

 If the packet is the current packet then

```
        resend ack
        ack_set <= 1;
    end if;
end if;
```


Vita

Aaron Reed Young is from Knoxville, Tennessee. He graduated from Hardin Valley Academy in 2012 and then began his computer engineering studies at the University of Tennessee, Knoxville. During summers as an undergraduate he completed internships at Siemens Medical Systems, Oak Ridge National Laboratory's Manufacturing Demonstration Facility, and Garmin International. Throughout graduate school he worked with TENNLab neuromorphic computing research group. 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 graduated with his Doctorate of Philosophy degree in Computer Engineering in May of 2020.