8-2018

Software Support for Dynamic Adaptive Neural Network Arrays

Adam W. Disney
University of Tennessee, adisney1@vols.utk.edu

To the Graduate Council:

I am submitting herewith a dissertation written by Adam W. Disney entitled "Software Support for Dynamic Adaptive Neural Network Arrays." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

James S. Plank, Major Professor

We have read this dissertation and recommend its acceptance:

Douglas S. Aaron, Mark E. Dean, Garrett S. Rose

Accepted for the Council:
Dixie L. Thompson

Vice Provost and Dean of the Graduate School

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

Software Support for Dynamic Adaptive Neural Network Arrays

A Dissertation Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Adam W. Disney

August 2018

Acknowledgments

I would like to thank my doctoral committee members, Dr. Mark Dean, Dr. Garrett Rose, and Dr. Doug Aaron. I would like to thank Dr. James Plank especially for his advice and guidance for 6 years! I couldn't have done it without you. Thank you to all my family and friends, new and old, that have been with me through this journey. A huge thank you to Harry Wagner. Without your help I would have never started this journey. Finally, thank you to my wife, Danielle. You have stuck by me for nearly half my life from high school dropout to PhD graduate.

Abstract

Moore's Law fairly accurately modelled advancements in traditional computing architectures for multiple decades, but it has come to an end. This has led researchers to put more focus on alternative computing architectures such as neuromorphic computing. DANNA (Dynamic Adaptive Neural Network Array) is a computing architecture that was designed in 2014 to meld features of recurrent, spiking, plastic neuromorphic computing systems with very efficient hardware implementations. Its hardware design and FPGA implementation preceded any software support or simulation. This work describes the software support for DANNA, including four different simulators, that has enabled TennLAB to explore the capabilities of the architectures. Additionally, we generalized a well-known neuromorphic experiment from 2008 to fit within the TennLAB software structure. We use this experiment to compare the capabilities of DANNA and TennLAB's other neuromorphic models.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Overview

Moore's Law has fairly accurately modelled advancements in traditional computing architectures for quite a while, but it is coming to an end [47] [30]. This has led researchers to put more focus on alternative computing architectures [44] such as quantum [22], molecular [45] and even chemical [46] [20]. One of the more popular alternatives is called neuromorphic computing. Neuromorphic computing takes inspiration from the brain. It promises to be able to do tasks that are difficult for a traditional computer while using much less power [25] [42].

There are two ends to the neuromorphic computing spectrum. On one end, neuromorphic computing models seek to mimic the brain as precisely as possible. The other end realizes that there may be no need to mimic the brain precisely to have a useful computing device. Over the past three years, the Neuromorphic Computing group at the University of Tennessee and Oak Ridge National Laboratory, called "TENNLab," has developed three neuromorphic computing models that lean towards the latter, called NIDA, DANNA and mrDANNA. All three are spike-based [42], and feature programmable neurons and synapses that are configured in 2D and 3D space. Of particular interest to this dissertation is the model DANNA, and the battery of software research and support that has been performed for this dissertation.

DANNA was developed specifically with digital hardware in mind. The characteristics of the neuromorphic model, such as programmable elements that may be configured as neurons or synapses, digital synaptic delays and nearest neighbor connectivity, were selected to facilitate implementation on FPGA's and eventually ASIC's. However, like any hardware project, DANNA requires a large amount of software for support, communication, simulation, validation, and programming. The research in this dissertation addresses every facet of software for the DANNA neuromorphic model. Each research project is described in subsequent chapters. The remainder of this introduction describes the TENNLab approach to neuromorphic computing, followed by a detailed dive into the DANNA neuromorphic computing model.

1.2 The TENNLab approach to neuromorphic computing

TENNLab approaches their neuromorphic computing design with three major components: Applications, Learning, and Models [34]. Applications are written towards a generic neuromorphic device interface that allows various neuromorphic models to be interchanged in the application without any modification. The learning aspect mostly involves an Evolutionary Optimization (EO), that is also abstracted such that any model can be used with the same learning algorithm when supplied with the parameters and constraints for the particular model. Figure 1.1 shows the stack of these components.

For example, the "Flappy Bird" cell phone game is an application from TENNLab. In this game, a player is constantly forced forward while having to navigate through gaps in

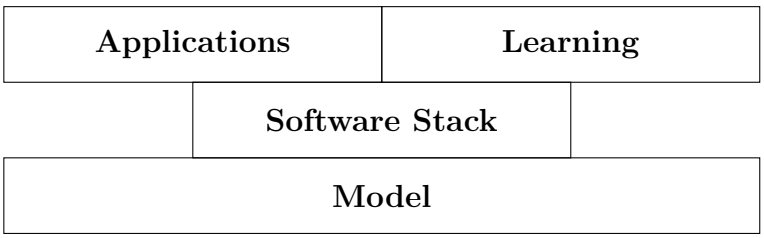


Figure 1.1: Components to TENNLab approach

approaching walls. The application can be played by a human player or a neuromorphic model. The neuromorphic model receives information about the game state encoded as spikes as input, and its output firing events translate to an action in game. The specific configuration of the neuromorphic device is learned with EO. EO must use fitness functions specified by the application. In this case, fitness is based on the duration of the game, with longer times being better. The fitness function plays the game with the model as the player and notes the length of the game. Based on the fitness of each configuration, the EO will modify the population of configurations in an attempt to make better configurations.

In this example it is easy for the application to use any of the backend models such as NIDA, DANNA, mrDANNA, or any other future model that is integrated into this framework. A different learning method can also be used with little to no additional support from the application.

1.3 Dynamic Adaptive Neural Network Arrays

Dynamic Adaptive Neural Network Arrays (DANNA) were designed specifically for hardware implementation before any software was written to support it. That software is explained in the subsequent chapters. In this section, the DANNA model is described in detail to understand the software research. The details are enough for understanding the model behavior, but hardware implementation details are left out. DANNA has been described in [12] and extensions in [8] though not in enough detail to reproduce behavior.

1.3.1 DANNA Overview

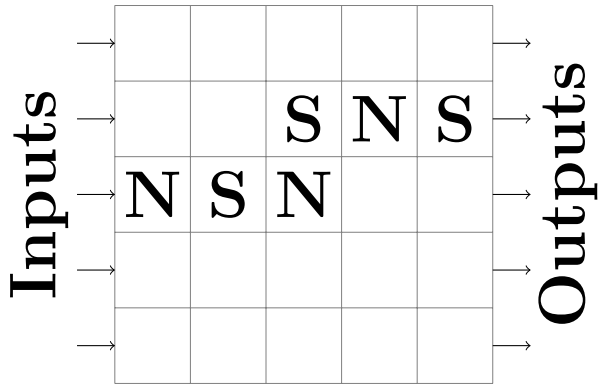
DANNA is a neuromorphic device that simulates a spiking neural network. It is a 2D array of elements where each element is user configurable to be a synapse or neuron. The array has pre-defined locations at the edge of the array for external inputs and outputs. A basic picture of an array is shown in figure 1.2a. Each element can connect to its neighbors marked as blue shown in figure 1.2b. External inputs can only be read by the direct element to which the input is connected. For example, in figure 1.2a only elements in the first column receive inputs that are external to the array. The configured type of the element does not

matter as long as the element is programmed to listen to the input. Basically, the input acts as a synapse element that is one space to the left of the array. Similarly the output acts as a synapse that is one space to the right of the array. At a high level, the model is straightforward.

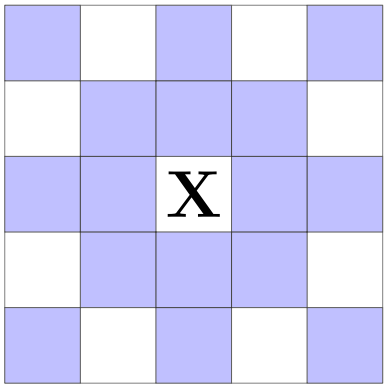
1.3.2 The Neuron

Neurons are based on simple accumulate-and-fire neurons, where the neuron accumulates charge from neighboring elements and fires if the total accumulated charge is over its programmed threshold. Once the neuron fires, the amount of charge in the neuron is reset. For example, suppose a neuron's threshold is ten and the current charge of the neuron is zero. If it accumulates five units of charge, then it will store five units of charge indefinitely. If at some other time it accumulates ten units of charge for a total of 15 (which is greater than or equal to the threshold), it will fire, and then reset back to zero.

The DANNA hardware in reality handles this in a slightly different way. The accumulator for a neuron is an n-bit register treated as an unsigned integer that uses saturation arithmetic. The neuron threshold is fixed at 2^{n-1}, because this is very simple to check in hardware, by checking the state of the highest bit, thereby eliminating the need for a full comparison. To implement the concept of a configurable threshold, the hardware allows configuration of the default charge (**dcharge**). The **dcharge** is the charge level to which the neuron resets



(a) Sample DANNA array with six elements configured



(b) Element X can connect to the elements in the blue locations

Figure 1.2: Basic Overview of DANNA elements and connectivity

after firing. For example, if the user asks for a threshold of one, this translates to setting the **dcharge** to $2^{n-1}-1$, so that the neuron needs one unit of charge to reach the fixed threshold.

This implementation decision can have a heavy impact on reasoning about the DANNA behavior. Suppose a neuron has a threshold of 127, and the number of bits for the accumulator is eight. Subsequently, a charge of -100 comes into the neuron. In the conceptual view with signed integers, the neuron would start at 0 and end up at -100 charge for a total delta of -100. For the hardware, the neuron would start with a **dcharge** of one ($2^7 - 127$). When reading the -100 the neuron would be saturated at the bottom of its range to zero, for a total delta of -1. That is quite different from the conceptual view, so this must be taken into consideration when reasoning about neuron charge in DANNA.

Otherwise, the neuron is extremely simple with only the threshold and enabled inputs as parameters. The enabled inputs determines which of the neuron's neighbors from which the neuron will accumulate charge. This is due to the broadcasting nature of the array discussed in section 1.3.4.

1.3.3 The Synapse

Synapses conceptually function like directed graph edges, connecting nodes which are neurons. Signals from the source neuron are transmitted to the sink neuron. Thus a synapse can only listen to one neighbor, and when it detects the source firing, it will then fire some time later based on its delay parameter. The intent of the delay parameter is to simulate the physical length of a synapse. The amount of charge that the end neuron receives is based on the synapse's configured weight. The weight is stored in the same n-bit accumulator as the neuron, but is treated as a signed integer. Additionally the synapse has depression and potentiation mechanics (plasticity) that change its weight during operation. If the synapse detects that it was responsible for the destination element firing, then its weight will increase. Alternatively, if the element was already firing thus meaning the synapse was not responsible, then it will decrease. Once the weight changes, it cannot change again for a number of cycles, based on the synapse's refractory parameter. It is also possible to toggle plasticity at load time per synapse. In total the synapse has the following parameters:

- **input**
- **weight**
- **dp_port** (more in section 1.3.4)
- **dp_toggle** (plasticity is often called 'dp' in the library short for depression/potentiation)
- **delay**
- **refractory**

1.3.4 Subtleties of Configuration

Overall the conceptual view is that neurons are connected to neurons by synapses. One might think that part of the synapse definition would be something like "connect neuron A to neuron B." In the hardware reality, when elements fire, the fire broadcasts to all neighbors. The neighbors must be configured to listen to the broadcast individually. Figure 1.3 shows neuron A connected to neuron B through synapse S.

This scheme creates a few subtleties. For instance, this makes a synapse a one-to-many element instead of one-to-one, since any neighbor to the synapse can be configured to listen to it. For purposes of plasticity, the synapse must configure which single neighbor it will monitor (called the **dp_port**). This means that it is possible to configure it to a neighbor that is not even listening to the synapse. While allowed, obviously the behavior of the plasticity will be

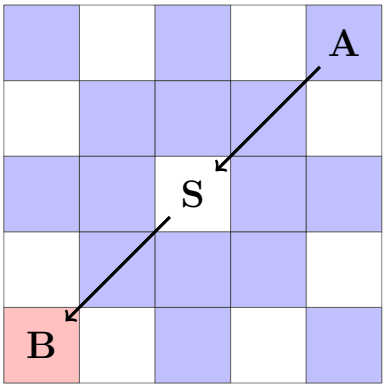


Figure 1.3: Neuron A connected to neuron B through synapse S. When S fires, all of the colored squares receive the event, but only B is configured to read it (colored in red).

illogical, since the behavior of that element is not related to the synapse. Figure 1.4 adds two new neurons to figure 1.3. Neuron C is listening to S; thus C will accumulate charge when S fires. Neuron D is not listening to S thus D will not accumulate charge when S fires, even though the firing signal will arrive at D. Furthermore, S's **dp_port** can be set to A, B, C, D, or any of the other colored squares, though it would only be logical to set it to B or C.

Additionally, due to this broadcasting scheme, it is possible to connect neurons to neurons. Again the behavior here will be illogical but allowed. The amount of charge the sink neuron receives is the source neuron's **dcharge** parameter. Also, due to the timing in the hardware, it is possible that the source fire will be read twice or not at all. Timing details are discussed in section 1.3.5.

Synapse to synapse connections are fine and work logically. Often this is required to connect two neurons that are far apart in the array, or if the user desires a delay higher than the maximum for the delay parameter. Synapses ignore the incoming value of a fire event, such that the charge that the sink neuron will receive will be the weight of the final synapse in the chain. It is possible that the synapses before the final synapse change weight due to the configured **dp_port**. This does not matter to a synapse listening to it, but would affect a neuron so this mechanic should be kept in mind. Figure 1.5 shows such a configuration.

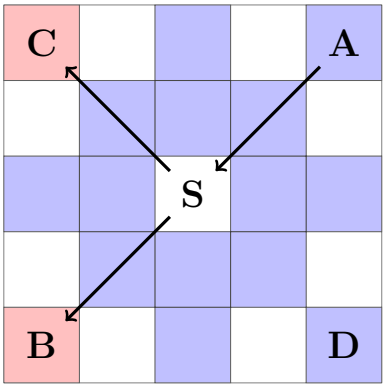


Figure 1.4: Neuron A fires causing synapse S to fire. Neurons B, C, D and all the other blue squares will receive the fire event, but only B and C are configured to pay attention to the data (colored in red).

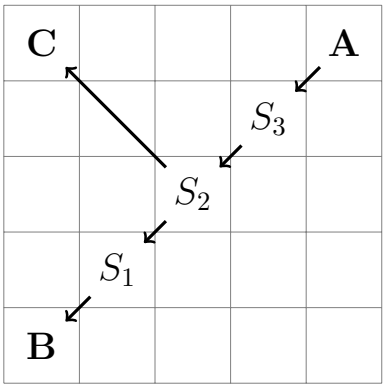


Figure 1.5: If S_2 has a dp_port configured to listen to S_1, then its plasticity mechanics will happen illogically. In this case, C can possibly get unexpected values from S_2.

1.3.5 Timing

The previous sections have described the parameters of the elements, but without any time components. This section will explain the timing of events with timing diagrams.

Clocks

There are two major clocks for the timing of all events in DANNA: the global clock and the port clock. The port clock runs at 16 times the speed of the global clock, as seen in figure 1.6. Timestamps on outputs are in relation to the global clock. The port clock is generally used for neurons to scan through each of their input ports and accumulate available input from the neighbor on that port. Since there are 16 neighbors per element that all must be read per global cycle, its speed must be 16 times faster than the global clock.

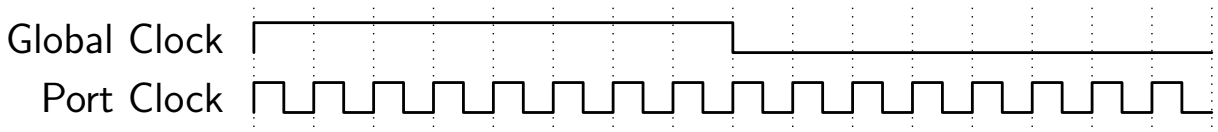


Figure 1.6: Global Clock and Port Clock

```
uint8_t port_select()
{
  // Sample the number we want
  uint8_t rv = (seed >> 15) & 1;
  rv |= ((seed >> 31) & 1) << 1;
  rv |= ((seed >> 47) & 1) << 2;
  rv |= ((seed >> 61) & 1) << 3;

  // Update seed - bit[61] xnor bit[62] is the new bit to be shifted in
  uint64_t bit = 1 ^ (((seed >> 61) & 1) ^ ((seed >> 62) & 1));
  seed = (seed << 1) | bit;

  return rv;
}
```

Figure 1.7: LFSR used to select initial port at the beginning of each global cycle. A 4-bit number is built from bits 15, 31, 47, and 61 in the register, then the register is updated by shifting left. The bit shifted in from the right is the `xnor` of bits 61 and 62.

Port Select

At the beginning of the global cycle, a port number is selected randomly with a 64-bit linear feedback shift register (LFSR). Figure 1.7 contains C++ code for the update and sampling used at each global cycle.

After the first port clock cycle, the port select simply increments by one (with wrap around), such that all 16 ports will eventually be the port select. This process repeats for each global clock cycle. Ports are sampled on the rising edge of the port clock when that port's enable signal is also high. In figure 1.8, the signal labeled **Port 4 Enable** shows when port 4 enable is high. If the neighbor on port 4 is firing during the moment when the port 4 enable is high, then the neuron will accumulate the incoming charge. Figure 1.8 illustrates with a timing diagram. This shows that ports are sampled at the end of their duration as the port select. Understanding this will help with reasoning about future timing diagrams.

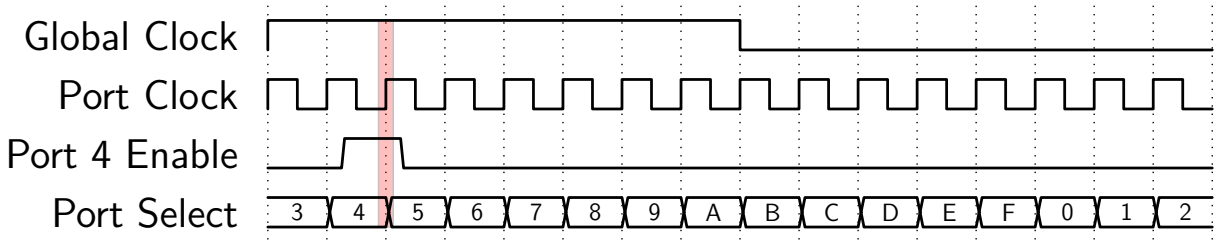


Figure 1.8: Port select over a whole global cycle. Port 4 is sampled when the port 4 enable is high. This is highlighted in the figure. The neuron accumulates charge on the rising edge of the Port clock.

1.3.6 Port Orientations

To support the depression and potentiation mechanics on synapses, there is a concept of port orientation. The ports are oriented differently around each element, depending on its location in the array. This means that one cannot simply associate a port with an offset in the array. The offset depends on the location in the array. Figure 1.9 shows the pattern of the ports throughout the array. This repeats for each four by four tile. The inner ring of ports represents moving to the immediate neighbor that is one square away in that direction. The outer ring of ports represents moving two squares away. Thus, for example, the elements in the first and third columns of the first row of the array are connected to each other by the port labeled 'A'.

Why make the ports complicated like this? It makes port numbers between two neighbors the same. If one were to move to the neighbor at a particular port number, then it is possible to follow the same port number back. For example, consider the colored element pairs in figure 1.10. Notice that the offset to get to the neighbor is reversed between the two elements of the same color. Take the red pair for example. One would offset to the left one square for port number 6 but the other would offset right one square.

Due to the way elements scan their ports, this allows a synapse to know with certainty if it is responsible for a neighboring neuron firing. The timing of all this will be covered in the following sections.

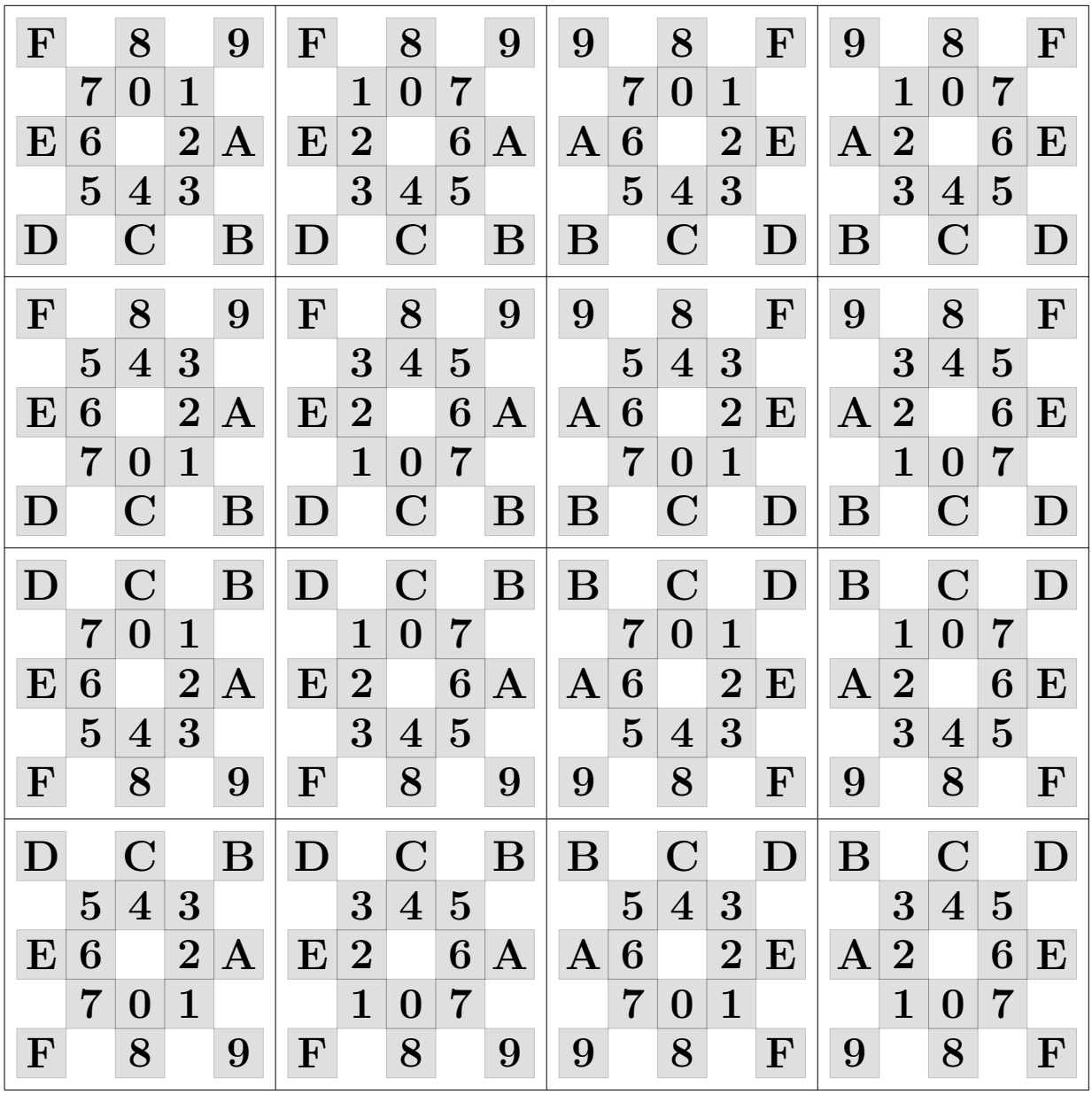


Figure 1.9: Orientation pattern that repeats every 4 by 4 tile

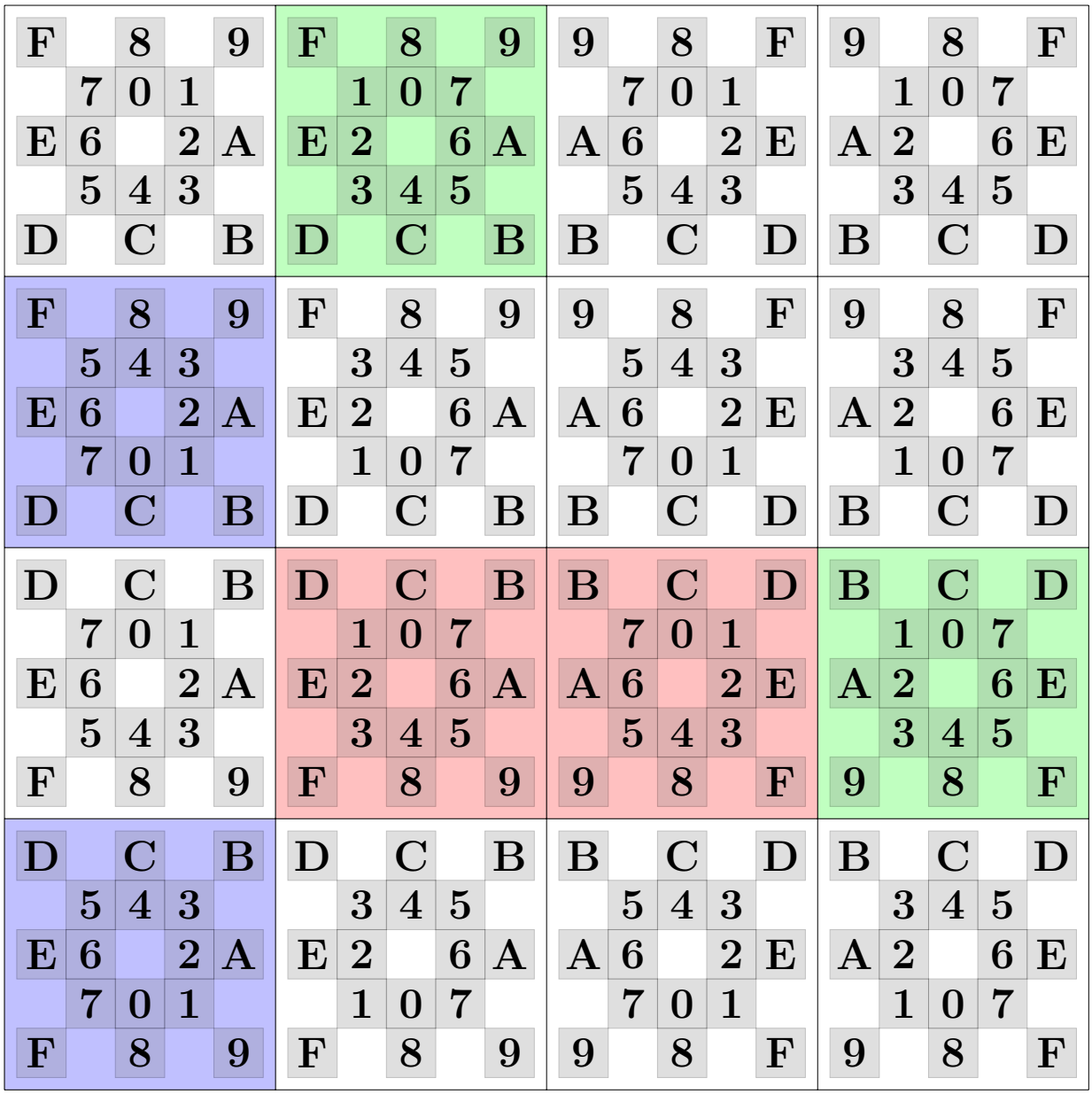


Figure 1.10: The two red elements are connected by port number 6. The two blue elements are connected by port number C. The two green elements are connected by port number B.

1.3.7 Neuron Timings

Neurons accumulate charge from neighbors on each cycle of the port clock, if the current port select is enabled. If that charge makes the total charge greater than or equal to the threshold, then the neuron will fire on the next port clock cycle and reset to its default charge (**dcharge**). Neurons fire for the length of a global cycle such that neighboring synapses will always read the fire (this will be more apparent in section 1.3.8).

Figure 1.11 shows a neuron accumulating charge from a neighbor on port eight, and then firing on the next cycle. During the fire, the neuron will still accumulate charge from neighbors but will not check against the threshold. This means the amount of charge can be above the threshold without firing. This is shown in figure 1.12, which is identical to figure 1.11 except the neuron accumulates some charge from port F while firing. If later the neuron were to accumulate a firing event, and afterwards the accumulator still has a value with the highest bit set, then it would fire even if the incoming charge is negative.

1.3.8 Synapse Timings

Unlike neurons, synapses always read their configured input on the last port cycle (in other words, right when the global clock goes high). Since a neuron fires for 16 port cycles (i.e. the duration of a global cycle), a firing neuron will always be scanned exactly once by a listening synapse. Figure 1.13 illustrates this with the neuron firing from figure 1.11. The timing of this neuron to synapse interaction can be delayed by one global cycle if the neuron fires due to input on either of the last two port cycles. Figure 1.14 illustrates this situation.

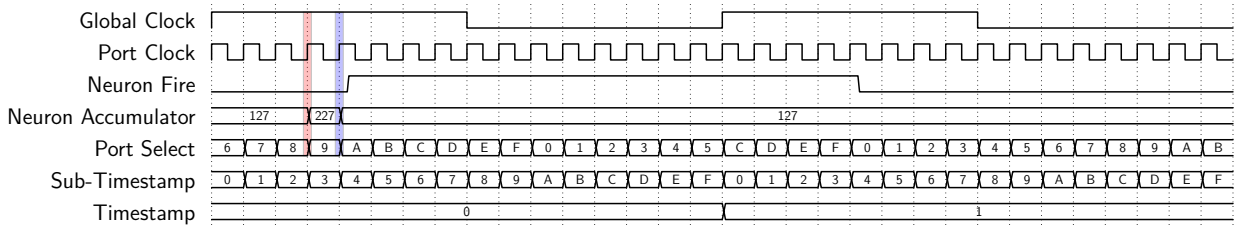


Figure 1.11: A neuron accumulating 100 units of charge on port eight, and then firing on the next cycle. The fire stays active for 16 port cycles.

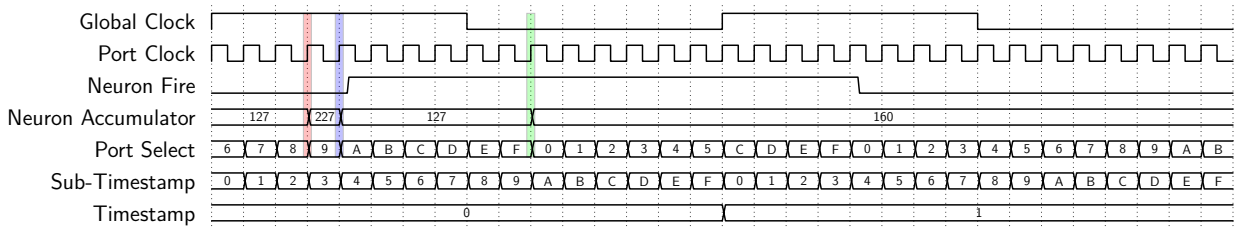


Figure 1.12: Same as figure 1.11, except the neuron accumulates charge from port F while firing.

If a synapse is to fire, then it always begin firing on the first port select, and fires for a duration equal to the global cycle. In this way, a reading neuron (or synapse) is guaranteed to have that port come up as the select, because every port number shows up during this time. This works out great but can be a bit confusing while reasoning about the model.

Synapses ignore incoming values and only pay attention to the fact there is a fire event. The synapse then fires based on its configured delay. Figure 1.15 is a continuation of figure 1.13 showing when the synapse would fire with a configured delay of 0. A delay greater than 0 simply shifts the synapse fire by a number of global cycles equal to the delay.

The synapse's **dp_port** controls the element to which the synapse will listen for depression and potentiation mechanics. This is where the orientation patterns from section 1.3.6 become important. If the synapse is currently firing when its **dp_port** comes up as the port select, it will initiate a depression/potentiation check (**dp_check**). On the next port cycle, the synapse will check the firing state of the neighboring neuron on the **dp_port**. If it is already firing, then the synapse will depress because it can know with certainty that it did not cause that neighboring neuron to fire. If it depresses, then this sequence ends. If it did not depress, then on the next cycle it will again check the firing state

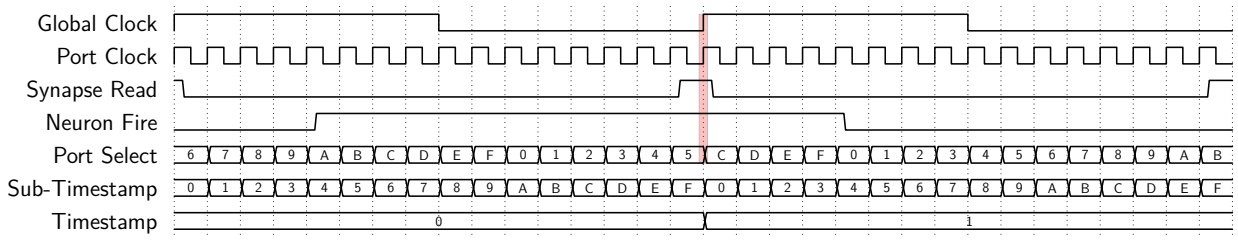


Figure 1.13: Synapse reads from neuron on the global cycle highlighted in red.

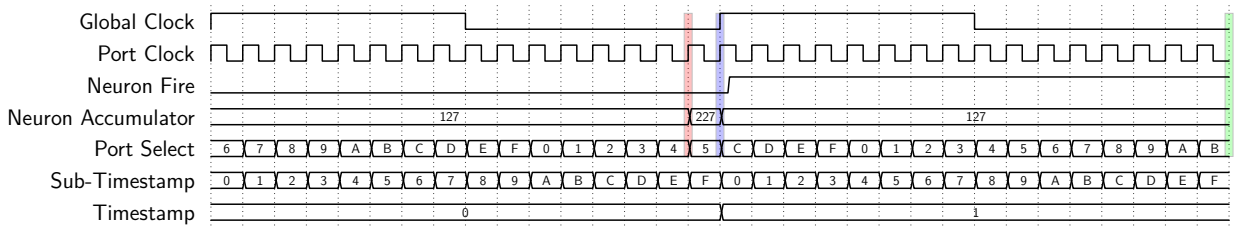


Figure 1.14: Similar to figure 1.11 except the neuron reads input on the 15th port cycle (highlighted in red). The neuron will begin firing on the port cycle highlighted in blue but this is also when a listening synapse would read from the neuron. Thus, the synapse will not read the fire here but on the next global cycle (highlighted in green).

of the neighboring neuron. If the neighboring neuron is firing, it can know with certainty that it caused that neighboring neuron to fire, and it will potentiate.

Note that all of the depression/potentiation mechanics assume that the **dp_port** points to a neuron. All of the mechanics still activate, regardless of the neighboring element's type. So, if the neighbor is a synapse, then the weights can still change though in an illogical manner. This is often not a problem, because synapses ignore incoming values of fires. This should be kept in mind though, because it is possible in some configurations of elements that it would be important. For example, a neuron listening to a synapse in the middle of a chain of synapses like in figure 1.5.

Figure 1.16 illustrates the timing of a synapse potentiating after a neuron fires, due to the charge the synapse provides. For this example, the neuron has a threshold of one, so the **dcharge** is 127, and obviously the neuron is listening to the port of the synapse. The synapse is configured to have a weight of 100, and the **dp_port** is set to listen to the port of the neuron (which, due to the orientation pattern is the same port number). The port number between these two elements is port eight. When the port select is eight, the neuron

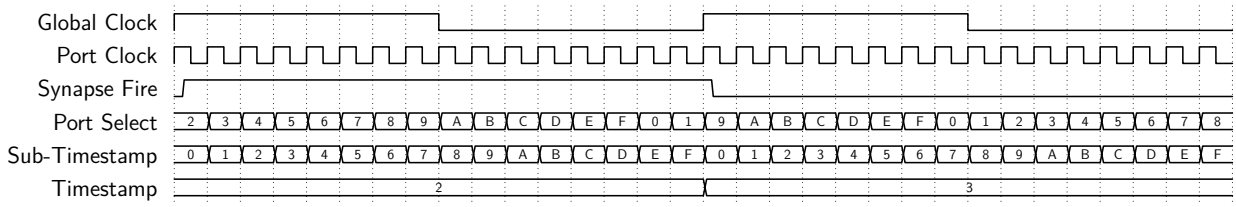


Figure 1.15: Synapse firing after reading the neuron fire from figure 1.13.

reads the synapse which is firing, so it accumulates charge equal to the synapse's weight. This puts the neuron's charge above the fixed threshold of 128, which causes it to fire on the next port cycle and reset its charge to its **dcharge** value.

Meanwhile, the synapse will initiate its **dp_check** when the port select is eight on port cycle six. On port cycle seven, the synapse checks the neuron's firing state and can detect that it is not firing. Thus, it does not depress and continues the **dp_check**. On port cycle eight, the synapse checks again and the neuron is firing, which causes the synapse to potentiate on port cycle nine. Notice that because the neuron was not firing on port cycle seven, but was firing on port cycle eight, the only possible conclusion is that the neuron fired because of charge gained from this particular synapse.

Figure 1.17 shows a similar situation to figure 1.16, except the synapse depresses due to the neuron reading input from port six and firing before port eight comes up.

1.3.9 Capture and Shift

To this point, the only output emitted by DANNA has been the output element firing events. Sometimes it is of interest to gather information about the internal state of the DANNA array. To accomplish this on hardware, DANNA has a set of commands called Capture and Shift.

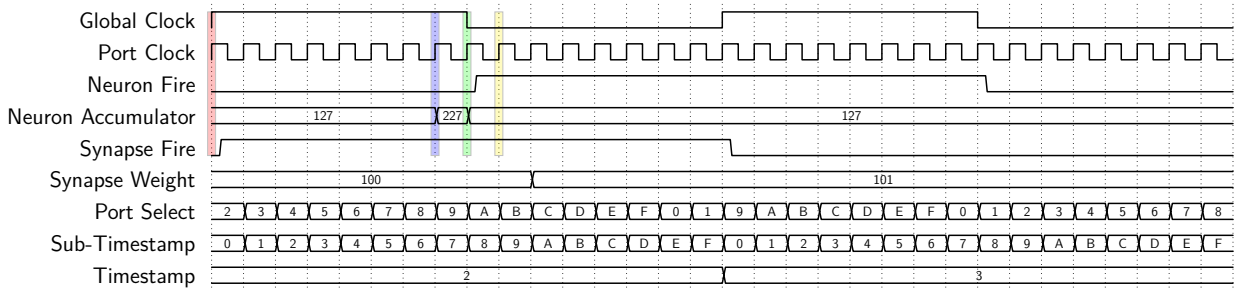


Figure 1.16: Synapse dp_check with potentiation. The red section highlights when the synapse begins firing. The blue section highlights when the neuron reads that fire. The blue section is also when the synapse begins the dp_check. The green section is the depression check. The yellow section is the potentiation check. The synapse then potentiates on the next port cycle.

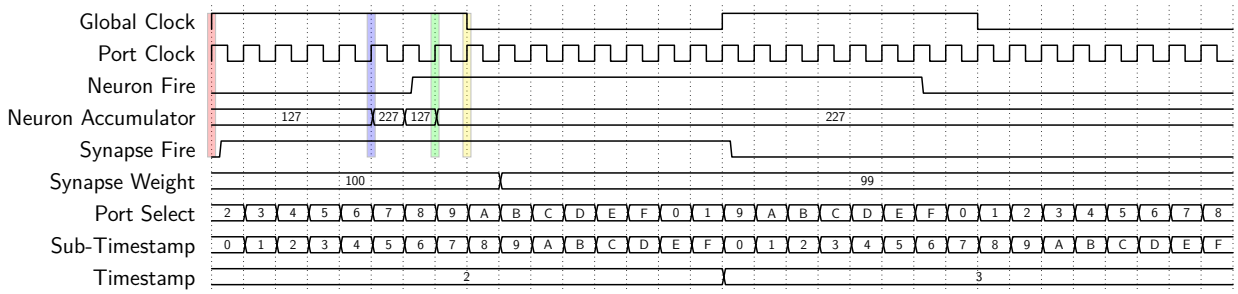


Figure 1.17: Synapse dp_check with depression. The red section highlights when the synapse begins firing. The blue section highlights when the neuron reads a fire from a different neighbor. The green section highlights when the dp_check begins. The yellow section is the depression check. The synapse then depresses on the next port cycle.

The Capture command snapshots three values for each element in the array: the number of fires since last capture, the value in the accumulator, and the number of queued fires for the element. The first value is very straight forward. The second value corresponds to a neuron's stored charge and a synapse's weight. The third value only applies to synapses and is the number of detected fires that have yet to fire. Essentially this is the number of fires that are still traveling across the synapse.

Once these values are captured, the information is moved off the chip with the Shift command. Each element holds a total of 32-bits to represent the data captured. Each column is treated as a single $32 \times R$-bit shift register where R is the number of rows in the array. Each time the Shift command is issued, a single bit from each column is shifted out from the top of the array. Thus, a total of $32 \times R$ Shift commands must be issued to the array to get all of the Capture information. Figure 1.18 illustrates how the Shift command functions.

1.4 New Features for DANNA

After verification of hardware and the development of many applications using the model, we developed several new features for DANNA. Each feature is describe in the following subsections.

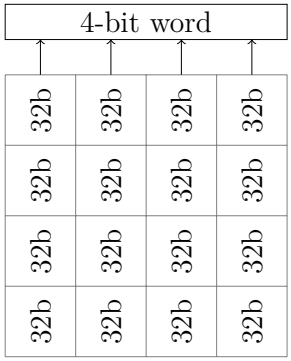


Figure 1.18: Capture information is stored within each element as 32-bits. Each shift command shifts all the bits in the column up once with the top bit from each column appearing in the output data from the array.

1.4.1 Neuron Leak

Neuron leak is a common feature in neuron models and is essential for certain techniques like using DANNA as a reservoir in reservoir computing [24]. The DANNA implementation of leak has two parameters that are configurable per neuron: leak amount and leak delay. After the leak delay number of global cycles, the neuron will normalize towards the default charge value by the leak amount. For example, if the neuron has a default charge of 100, current charge of 125 and a leak of 10 then the current charge will become 115 when it leaks. If the current charge had been 75 then it would become 85.

The leak happens when the neuron reads the first port of the cycle. If an input is read at the same time, then the leak and charge are accumulated simultaneously before a threshold check occurs. Leak alone does not trigger a threshold check. Figure 1.19 shows an example of the leak. If an input of 10 charge was read at the same time of the leak, then the charge would remain at 125.

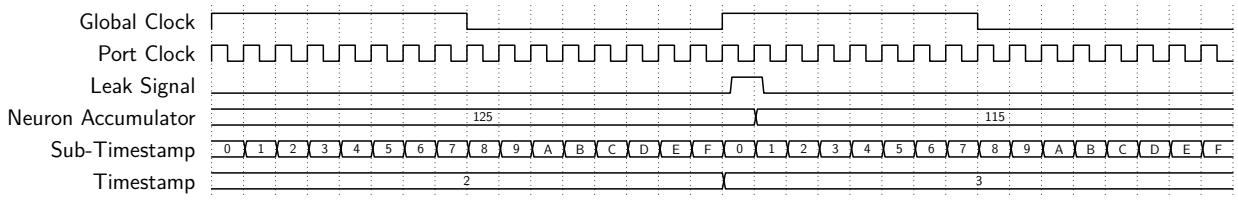


Figure 1.19: Example of neuron leak of 10 when the neuron's default charge is 100.

For the clock-based simulations, leak is simply done at the end of each global cycle. The addition of leak to the event-based simulator was fairly simple. The element now has space to save its leak amount and leak delay, but a new event type was not needed. Since the leak cannot trigger the neuron to fire it is not important to update the neuron's charge exactly when it would happen. The neuron stores when it last updated leak, such that the real current charge is calculable when it would be important like on an incoming fire or when the state of the network is being captured.

An alternative implementation of leak may be implemented to approximate exponential leak function, $C_t = C_0 e^{(-t/\tau)}$, where C is the neuron charge, t is the current time, and τ is the time constant on the leak. To avoid the need for division units, the equation may be phrased as $C_t = C_0 2^{(-t/\tau)}$. In this way, a simple division by 2 may be made every τ cycles.

1.4.2 New Port Selector

A new random port selector addresses an issue with the original port selector that inherently gave a bias towards certain connections. In the original design, a port is randomly selected by the linear feedback shift register (LFSR) at the beginning of the global cycle. For each port cycle, the port number is simply incremented by one with wrap around such that all 16 ports are read by the end of the global cycle. The issue with this approach can be illustrated with an example.

Suppose a neuron has two inputs, one on port 5 and another on port 7. If both inputs are firing during the same global cycle, the chance that port 5 gets scanned before port 7 is 14 out of 16. If the initial select is anything other than 6 or 7, then 5 will come before 7 during the global cycle. This gives 5 a better chance to cause the neuron to fire and potentiate.

The new random port selector uses a slightly modified version of the Fisher-Yates shuffle described in Knuth's *The Art of Computer Programming* [21]. This version uses a vector containing all the port numbers and permutes them in place to create a selector sequence for the global cycle. Initially the vector is simply set to the sequence 0 through 15. Each global cycle the vector is shuffled with the algorithm shown in figure 1.20. Normally after each swap, the **random_number()** range would be reduced by one. This would require hardware that can calculate division which would take quite a bit of real estate on the FPGA.

```
for(int i = 15; i >= 0; i -= 1)
  swap(select_array[i], select_array[random_number()]);
```

Figure 1.20: Select array shuffle algorithm. random_number() generates a 4-bit number with the LFSR from the original port select implementation shown in figure 1.7.

Through empirical testing, it takes millions of global cycles before a select sequence repeats. This is much improved over the 16 possible sequences with the original select sequence.

1.4.3 Spike Timing Dependent Plasticity

A Spike Timing Dependent Plasticity (STDP) feature for DANNA has been implemented that is essentially a more complicated version of the existing depression and potentiation mechanics. STDP mechanics may be of use for some applications. Details on a DANNA modification to support STDP are detailed here.

Generally the STDP model adjusts synapse weights upward when read before the destination neuron fires and adjusts the weight downward if read after the neuron fires. Figure 1.21 shows a basic function of STDP.

To implement this in the DANNA model, there needs to be a way to tell if a synapse fire is read before or after the neuron begins to fire. To enable this functionality, the synapse now

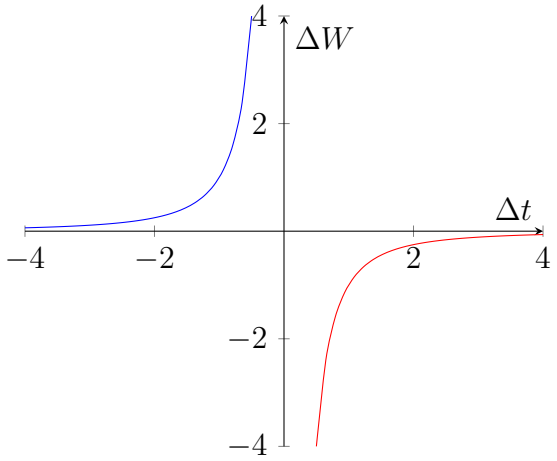


Figure 1.21: Weight changed (ΔW) based on synapse fire time relation (Δt) to neuron fire

has a depression flag (**dflag**), a potentiation counter and a depression counter. The dflag is used to catch corner cases when a synapse stops firing before proper checks occur. The counters can be the same piece of hardware since they are never used at the same time, but it is easier to think about them separately. Additionally, the device will have a configured time window and a table of weight change values. For a weight change to occur, there needs to be a synapse fire and a neuron fire that both occur within the time window.

The counters are used to capture when one of the two elements fires. If the synapse fires with both counters at zero, then the potentiation counter will start. If the neuron fires while this counter is counting down, then there is a pair of events within the window where the synapse fire came before the neuron fire; thus the synapse should potentiate.

If the neuron fires without a counter going, then the depression counter will start. If the synapse fires while this counter is counting down, then there is a pair of events within the window where the synapse fire came after the neuron fire; thus the synapse should depress.

That is the general idea, but there are some details required to make sure all situations are handled correctly. There are five cases to ensure correctness. Two of these cases are during the **dp_check**. With STDP, this check only happens once after the **dp_port** is the select.

Case 1: If the neighboring neuron is not firing, then it means that the synapse fire will contribute to the neuron firing, or possibly be the final charge that makes the neuron fire. If the depression counter is not counting, then the synapse can start the potentiation counter.

Case 2: If the neighboring neuron is firing, then it means that the synapse fire will be read after the neuron has already fired. If the potentiation counter is not counting, then the synapse will set the dflag and start the depression counter. Now this case guarantees that there is a synapse/neuron firing pair in the window, but all weight changes are deferred to the global cycle. This is why the depression counter is still started.

The last three cases happen on the global cycle.

Case 3: This case is when the depression counter is counting. If the synapse is firing OR the dflag is set, then the synapse will depress based on the count currently in the depression counter. The count currently in the counter works as a table lookup for the weight change value.

Case 4: If the neuron is firing and the potentiation counter is counting, then the synapse will potentiate. Again the value in the counter works as a table lookup.

Case 5: If the neuron is firing and the potentiation counter is not counting, then the depression counter starts. This means that there is only this neuron fire within the window so far. Additionally, if the synapse is firing while this is true, then the dflag is set.

This scheme only considers the first synapse firing event that pairs with the neuron firing event as opposed to all the synapse firing events within the window. This is a concession due to hardware space limitations.

1.5 Summary

Having thus described the intricacies of the DANNA model and its hardware implementation, the reader may appreciate the complexities involved in writing a simulator for DANNA, and for the other software elements that surround DANNA. The rest of this dissertation is organized as follows.

Chapter 2 discusses work related to neuromorphic simulation and software support. Chapter 3 discusses the software framework created for this dissertation which includes the software simulation, hardware communication, and how it facilitates application development. Chapter 4 discusses the GPU implementation of the software simulator. Chapter 5 discusses the performance of the simulators. Chapter 6 discusses an experiment involving the detection of patterns in noise with DANNA and TennLAB's other neuromorphic models.

Chapter 2

Related Work

There are no previous works on DANNA specific simulation and little software support [48], but there exist many spiking neural network simulators in software such as BRAIN [18], CARLsim [5], NEST [17], and NeMo [33]. They all simulate on many common neuron and synapse mathematical models such as Integrate-and-fire, Hodgkin-Huxley, or Izhikevich.

BRAIN was created to be flexible in which models can be simulated, while being fast to simulate and easy to use. It is written in Python and utilizes the NumPy library's Basic Linear Algebra Subprograms (BLAS) with the option to use optimized C code for some of the functions. The simulation is clock driven such that events are scheduled at fixed time intervals. Since these models are expressed typically as differential equations, BRAIN has support to define new differential equations in standard mathematical notation without the need to write any new code.

NEST (Neural Simulation Tool) differs in that it is an event based simulation that uses C++, MPI (Message Passing Interface), and pthreads (POSIX threads) to distribute work across many machines. It uses its own scripting language to define networks. Just like BRAIN, the simulation is clock driven with discrete time steps. The work within the time step is parallelized, and then all machines will synchronize to exchange event information.

CARLsim is a C++ library that uses GPUs to accelerate simulation. It provides a similar interface to PyNN [10], a common interface abstraction for several spiking neural network simulators including NEST and BRAIN. The simulation uses a sparse network representation similar to an adjacency list for graphs to highly reduce memory usage. The parallelization

techniques used are described in [31] as Neuronal parallelism and Synaptic parallelism. The simulation loop uses the Neuronal parallelism to update neuron states, and then the Synaptic parallelism to update the synapse states. CARLsim's GPU implementation achieves much faster speeds over their single threaded CPU simulation.

NeMo is designed to simulate neuromorphic hardware. It uses a Parallel Discrete-Event Simulation (PDES) to distribute the simulation across multiple machines via MPI. This uses an optimistic simulation scheme that has the possibility to execute events out of order. If this happens, it detects the causality errors and applies a reverse computation. This effectively rollbacks back the simulation to before the issue, and then reapplies the events in the proper order. Currently the only model simulated by NeMo is IBM's TrueNorth neuromorphic hardware [1] though it does not simulate it to cycle accuracy.

There are several hardware level projects as well, some with accompanying software simulations. Probably the most notable is IBM's TrueNorth architecture [3, 19, 43, 27]. TrueNorth consists of a tiling of neurosynaptic cores, where each core consists of 256 inputs referred to as axons, 256 outputs referred to as neurons, and a synaptic crossbar mesh that allows fully configurable connectivity between the two. The tiles are interconnected with a bus-like architecture that shuffles neuron firing events to other cores' axons. TrueNorth works on discrete time steps and integer values.

TrueNorth is programmed with an internally developed Corelet environment within MATLAB [2]. Programmers define the configuration of cores and package them as corelets which are treated as black boxes with a defined set of inputs and outputs. The idea is that a corelet will solve some problem much like a function or subroutine in a standard programming language. The programmer can then simply stitch these corelets together to form a solution to a larger problem. The Corelet environment will solve the physical placement of cores on the TrueNorth hardware.

The Compass simulator [35] is a cycle accurate simulator for the TrueNorth architecture. Unlike the previously mentioned simulators, it only simulates TrueNorth. It makes use of MPI and OpenMP to scale to multiple machines. There is an alternative version that uses PGAS (Partitioned global address space) in place of MPI that reduces runtime by half in the best situation. For each time step, the TrueNorth cores are simulated in parallel before

reaching a synchronization barrier, followed by a networking phase that moves firing events between cores.

SpiNNaker [36] takes another approach to hardware. It uses traditional von Neumann architecture for the computational power but interconnects them with a novel communication system designed for high performance with many small communication packets. SpiNNaker runs a specialized operating system with an event based API to which programs are written. In general, any number of programs can be written with this API. A wrapper around this API, called PACMAN, presents a more familiar interface for neural network developers. There is an emulator of the SpiNNaker hardware available to facilitate application development, since the SpiNNaker system does not support the typical computing environment.

Neurogrid [4] and the FACETS project, later continued as the BrainScaleS project [37], both have a mixed analog and digital approach to neuromorphic hardware.

Loihi [9] is the newest hardware model coming from Intel that is similar to the TrueNorth architecture with the addition of online learning mechanics and x86 processor cores.

Chapter 3

Software Framework

The first major project, which composes the bulk of this dissertation, is the basic software framework for DANNA. This is composed of three parts: simulation, hardware communication, and application support. Each is described in the sections below. Parts of this chapter have been previously published in [15] and [14].

3.1 Simulation

It should be evident from the description in section 1.3 that writing a cycle-accurate simulator for DANNA is a challenging task. The first goal in the simulator development was to perform hardware verification. However, as part of the overall research project in neuromorphic computing, the role of the DANNA simulator is quite varied. The goals for DANNA simulation that go beyond hardware verification are detailed below.

Communication: The input and output to DANNA come in the form of packets that are consumed and generated at the speed of the global clock. The hardware realization of DANNA requires a communication component so that DANNA may talk with a host computer. Originally, this component was realized by a Cypress FX3 USB kit; subsequent research has been to replace this functionality with custom hardware [49]. Software simulation of DANNA has been used both to develop the communication modules and to drive the packet design.

Training: To employ DANNA in applications, we train networks using a genetic algorithm called *Evolutionary Optimization for Neuromorphic Systems (EONS)* [41]. Candidate networks are generated by EONS, randomly at first, but then using reproductive operations on previously evaluated networks. Each network must be loaded onto a DANNA device, which is then directed by the application to execute on a training suite of inputs/tasks. Thus, the activities of loading networks, processing input and output packets, and running the device, are performed many, many times. While the DANNA hardware may be employed for this task, it is typically more efficient to use standard computing resources for training. Therefore, the performance of the DANNA simulator is important for the timely completion of EONS.

Utilization of available computing resources: The reason that software simulation is more efficient for training than hardware is the fact that general computing resources are abundant, and much easier to manage and leverage than FPGA's. EONS parallelizes very naturally [40], which again points to using software simulation for training. Many modern computers also contain GPU's, which offer additional opportunities for speeding up simulation.

Exploration of large devices: The DANNA hardware team has been performing active research in scaling DANNA by composing multiple DANNA chips [49, 16]. This places an additional focus on simulation, not only for verification, but on the ability to simulate very large DANNA networks efficiently.

Exploration of future hardware: Similarly, the DANNA hardware team has been exploring new functionalities within DANNA as described in section 1.4. These functionalities are best explored in simulation, to evaluate their potential effectiveness, rather than in hardware.

To summarize, simulation achieves many goals within the research project of DANNA. To achieve these goals, the simulators must have three sometimes conflicting properties: cycle accuracy for hardware development and verification, speed for training, and flexibility for exploration. It is for this reason that four separate simulators for DANNA have been developed.

The first, described in section 3.1.1, is named the "Clock-based" simulator, and its goal is hardware verification and abstract model development. It was briefly described in [15]. The second, described in section 3.1.2, is named the "Event-based" simulator. It abstracts away many of the hardware details in order to be faster, and thereby facilitate application programming. It was briefly described in [40]. The last two are an attempt to leverage GPU processors. They are described in their own chapter (chapter 4).

3.1.1 Clock-based Simulation

The clock-based simulator was the first simulator developed for DANNA. Its primary goal was hardware verification; however, until subsequent simulators were developed, it was also employed for training, producing the first DANNA networks that solved applications such as data classification [39] and pole balancing [11]. This simulator occupies a very small memory footprint, requiring only 40 bytes for the representation of each grid element.

Each element in the hardware array can be configured as a neuron or a synapse. Certain hardware components are shared between the two modes. For instance, the accumulator that holds a neuron's charge is the same accumulator that holds a synapse's weight. In simulation, this maps well to using a union of two structures, where shared hardware values are aligned. The hardware assumes that neurons lead to synapses and synapses lead to neurons. The simulation can make the same assumption and not have any complicated type checking or other conditionals. This makes matching hardware easy and allows the simulation to store only a single vector of these unions.

Since this simulation is polling neighbors for updates, there is an issue with incorrect or out of bounds access. For example, if the top left corner element attempts to read an element one row above, that's obviously out of bounds. A simple solution to this adds padding rows and columns of elements. For example, a 10 by 10 array will become 14 by 14 in memory. This avoids constantly checking bounds during simulation and makes for an elegant solution for external inputs. Since inputs act as synapses outside the bounds of the array, the padding elements are a natural way to represent these external synapses. When an external input should fire, then the firing state and weight of a padding element can simply be updated to

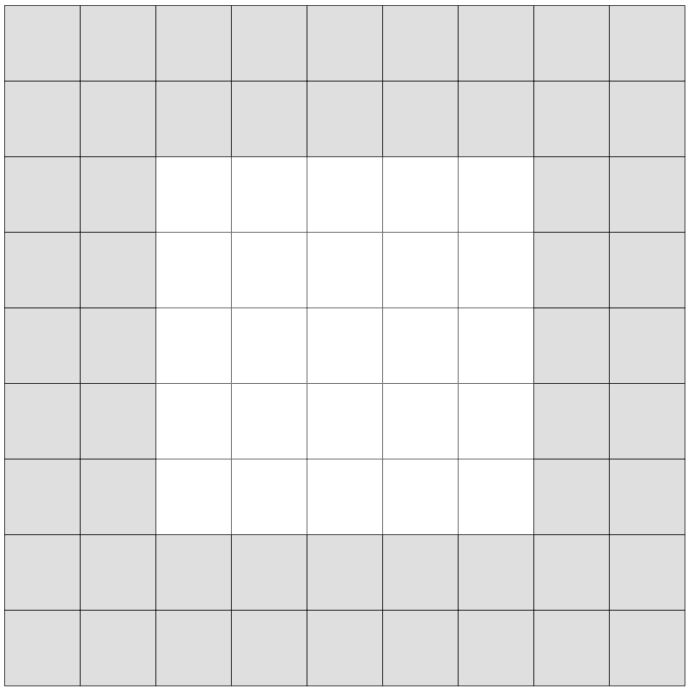


Figure 3.1: Sample five by five DANNA array with padding elements in grey. Elements are essentially empty, but can be activated as an elegant solution for external inputs.

reflect that, and nothing special needs to happen in the simulation loop to handle external inputs.

The access patterns of DANNA are difficult to reason about, but not without reason. As described in section 1.3.6, the ports are numbered such that neighbors have the same port number to each other. During simulation, the port select determines which port each element should be accessing. Since the logical direction of that port is not consistent across all elements, there is a look-up table of offsets indexed on port number and orientation. When an element needs to access a neighbor, the neighbor's location is simply the element's location plus the table's offset. This allows the enabled inputs to be stored as a bit field rather than a list of pointers or indices which would require much more memory.

Since the DANNA model in hardware is based on clocking, one of the most major issues with simulation is timing of events. While the simulation is a simple loop through the elements, in reality, the hardware processes all elements in parallel. At each step the hardware simultaneously updates elements and reads state. The clock simulator deals with this by having two loops per port clock cycle. This could be accomplished with a single loop but

would require an additional copy of the array. Figure 1.8 shows generally what section of timing the two loops are handling. The first loop, in blue, handles updating the state of elements based on what was read from the previous cycle. The second loop, in red, then reads the state of neighbors, but does not update its own state.

3.1.2 Event-based Simulation

After verification of hardware, a major focus was simulation speed, since the runtime of EONS training is heavily dependent on the simulator's performance. Event based simulation avoids any useless work done in the clock based version. The work on the clock based simulation made it clear what events in hardware were needed. The important events boil down to the following 10 events:

- FIRE COUNT UP: Increment fire count that is reported during a capture command.
- FIRE STOP: Toggle firing state to off.
- FIRE START: Toggle firing state to on.
- FIRE ARRIVE: The time when the element would detect a neighbor firing.
- WEIGHT CHANGE: Weight change event due to potentiation or depression.
- REFRAC OFF: Toggle refractory state of synapse to off.
- REFRAC ON: Toggle refractory state of synapse to on.
- DP START: Begin the events that detect potentiation and depression events. This does not initiate if in refractory.
- DEPRESS: Check if synapse should depress.

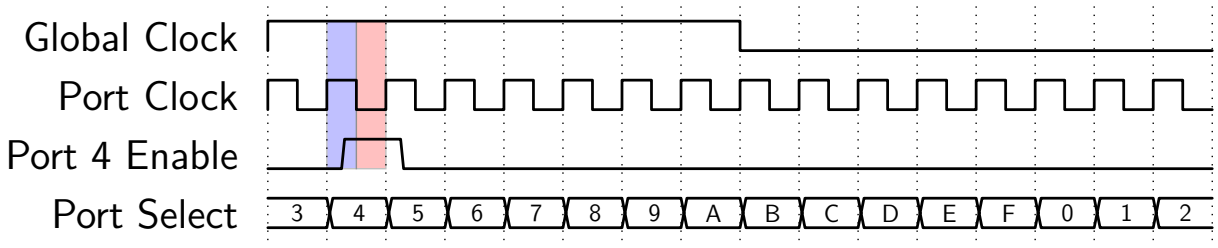


Figure 3.2: Figure 1.8 with coloring to illustrate the sections of time that the simulation loops handle. The first loop, in blue, updates element state based on reads from the previous cycle. The second loop, in red, reads state from neighboring elements.

- POTENT: Check if synapse should potentiate.

Events go into a priority queue and are executed in time order, with events on the same timestamp being prioritized in the order above. This preserves the required ordering that the clock based simulation handles with two loops.

The array no longer needs padding like the clock based simulation, because elements no longer poll neighbors. Instead elements receive an event from the neighbor when it fires. Each element has a broadcast list that is filled in at load time. The type of the broadcast lets the element know when to schedule the FIRE ARRIVE event on its neighbors.

To further improve, some additional memory is used to eliminate three of the event types: FIRE COUNT UP, FIRE STOP, and REFRAC OFF. By storing the timestamp when an element begins firing, the current firing state can be determined with a comparison with the current timestamp. Thus, an event that toggles the firing state to off is not required. This information also allows elements to increment firing count immediately. An adjustment to the value can be made when a capture command is issued, if it is incorrect. The same concept applies to the refractory state of a synapse.

3.2 Hardware Communication

Hardware communication is obviously important in order to use the DANNA hardware device. DANNA was designed with real time applications in mind. Commands are read on every global cycle from a First In First Out queue (FIFO). If this FIFO is empty, then the array will read a **noop** command. When an output event happens on DANNA, an output packet is put into an outgoing FIFO. In a real time application, these FIFOs can be removed and directly connected to sensors or other devices.

The hardware team chose USB for communicating between the DANNA device and traditional host computers [7]. This allows kits to be made that are simple to set up and use. The original clock based simulation was extremely valuable in troubleshooting the USB communication. Since it reads and writes packets in the same format as the hardware, errors such as packet misalignment were extremely easy to notice. USB is simply a middleware means of communication between the host and the device. Other communication layers, like

PCI Express or Ethernet, can be used. The software stack uses the library **libusb** [23] for this USB communication.

The largest challenge with hardware communication is the real time design of the device. Commands are read on each cycle and executed on the next cycle. If the host wants to fire an input on timestamp 10, then the fire command must be the command read at timestamp 9. This presents two major issues. First, this requires that all commands need to be known ahead of execution time. For example, if the command to start the device running is issued while it is paused on timestamp 0, and the host wants a fire to happen on timestamp 10, then the host must guarantee that the hardware reads the start, 9 noop commands and then the fire command. In other words, if the FIFO is not constantly filled with something, then the host cannot know when a command will execute. In real time applications, this is a non-issue since exact timing is not required, but it is an issue when trying to precisely verify with the simulator or run a host specific application. Second, this requires high bandwidth from the USB and host. If the host cannot keep the input FIFO filled and empty the output FIFO quickly enough, then inputs or outputs can be lost or incorrect.

The simulators deal with this issue by requiring an input at every cycle. In this way, the simulator does not have the issue of losing contact with the host. If an input packet is not provided, the simulator stalls. Proper handling of the FIFOs is not a problem with USB 3.0 bandwidth; It was with USB 2.0. The hardware team is currently looking into a new communication system that will behave much like the simulator [49]. It will pause the array execution when an input is not available or the output FIFO is full.

Another issue involves a problem with halting. It is difficult on the software side to know when to stop reading from the hardware. With an arbitrary set of commands, the DANNA may not halt, or the set of commands can make it halt and restart several times. The host must determine how many halts to expect, to know when to stop reading, and possibly inject a halt if the commands do not halt the hardware.

To understand an example of this halting issue, one must first understand the commands available for the DANNA device, which are as follows:

- **NOOP**: A null command that indicates no additional action should be taken.

- **LOAD**: Configures an element as either a neuron or synapse with appropriate parameters.
- **HALT**: Halts the global clock essentially pausing simulation.
- **RUN**: Begins the global clock for an indefinite amount of time.
- **STEP**: Begins the global clock for a fixed amount of cycle then halts automatically.
- **FIRE**: Issues input fires to the array.
- **RESET**: Clears the configuration of all the elements in the array and resets the timestamp to 0.
- **CAPTURE**: Captures state from the array described in section 1.3.9.
- **SHIFT**: Shifts out capture information as described in section 1.3.9.

Now consider the following sequences of commands:

- Suppose the user issues a single **RUN** command. This will never halt, thus never returning to the user.
- Suppose the user issues a **STEP 10** followed by a **FIRE**. This generates a single halt at the end of the 10 cycles the step command allowed.
- Suppose that **FIRE** is changed to a **HALT**. This will still generate a single halt packet after 1 cycle, since the halt is issued right after the array begins to run due to the step command. The halt basically cancels the step command.
- Suppose the issued commands are **STEP 10**, **NOOP 10**, and **HALT**. This will generate two halts because the step command's cycles run out thus halting the array and generating a halt packet, but then the halt command issued will generate another halt packet.

Obviously this can get much more complicated, and the host machine will have to analyze the commands to solve how many halts are generated.

To alleviate these issues, a special halt was added called an admin halt. This is a command that the software does not allow the user to issue. This admin halt is always appended to the set of commands issued by the user. When the hardware reads this halt, an additional flag is set in the response packet to let the host know that the array halted due to the admin halt. This guarantees the hardware will halt and the software does not

have to do any complex analysis of the commands being issued. The host simply waits for a single admin halt response from the array. Additionally, this helped the hardware team dealing with some USB bursting issues.

3.2.1 Tiled DANNA

Patricia Eckhart's master thesis work involved implementing tiled DANNA devices [16]. This required additional communication details and simulator support to verify. Her original test has only two devices tiled left to right. The left device has the inputs to the network and the right device has the outputs. Both are 32 by 32 DANNA arrays, so the whole array is 32 rows by 64 columns. The interconnect between the 32nd and 33rd columns forms a border that has limited connectivity across it, as well as additional time delay. Elements directly on the border can connect to the usual elements on the same device, but only connect to the neighbor directly across the border on the other device. Figure 3.3 shows this limited connectivity.

Since these are separate devices, the host is required to coherently write to the left device and read from the right device, and to know that both devices have halted. This type of communication is difficult to get correct and does not generically scale well. The hardware team will be working on a hardware arbiter that handles the details for the host machine [49]. The host machine will then be able to treat it as any other singular device.

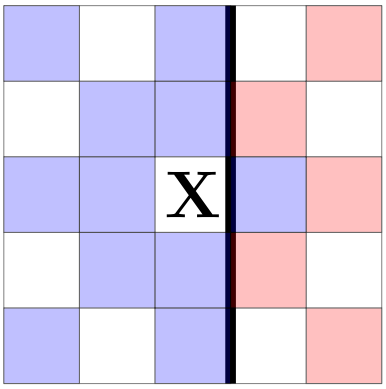


Figure 3.3: X can connect to the blue locations. Due to the border (noted with the thick black line) it cannot connect to the red locations.

The borders in the simulator make the same connectivity limitations as the hardware. Borders always span the entire height or width of the array. For example, the 32 by 64 test hardware was simulated by specifying a vertical border between columns 31 and 32. During simulation, new broadcast types handle this border correctly. The only difference is when the event would arrive at the neighbor, so it is not much different from previous types.

3.3 Application Support

3.3.1 DANNA Library

Obviously none of this is useful if it cannot be utilized. Thus there is a C++ library to facilitate application developers. The DANNA library consists of four major objects: Cmds, Outputs, Config, and Sim.

Config

The Config object holds information about non-user configurable settings for a DANNA device. This includes the size of the array and the location of external inputs and outputs. On a physical device, these parameters are generally not user configurable. An instance of a Config object is required for the other objects in the library. This allows all objects to appropriately error check and instantiate.

Cmds

The Cmds object holds a set of commands to be issued to DANNA devices. The following commands are supported:

- **Neuron**: Configure an element as a neuron.
- **Synapse**: Configure an element as a synapse.
- **Noop**: A null command. Do nothing.
- **Run**: Begin the hardware running until a halt is issued.
- **Step**: Begin the hardware running for specified number of cycles.
- **Fire**: Fire a specified combination of the external inputs.

- **Halt**: Halt the device.
- **Reset**: Reset the timestamp to 0 and remove all configured elements.
- **Capture**: Capture state information about the device.

These map closely to the hardware commands discussed in section 3.2 with little abstraction. Commands are generally kept in the order that they are issued, but to optimize EONS performance it was of interest to allow the manipulation of the Neuron and Synapse commands in place. This presents an issue with simply keeping them in order.

The general loop of EONS tests the fitness of a population of networks, and then modifies the population based on those fitness values. A network in terms of the Cmds object is simply a set of Neuron and Synapse commands. EONS may want to change, delete, or add elements in this network. To allow the user to get at arbitrary Neuron and Synapse commands, there is a C++ map keyed on coordinates that points to the packet in the buffer of hardware packets. Modification is simple to implement, as this simply requires changing the packet in place.

A major issue arises with insertion and deletion. Commands are kept in a vector contiguous in memory. This preserves the order in which commands are issued, but also is required for libusb to send the commands. If a new element is simply inserted into the commands, then it will naturally go to the end of the vector. If the user had previously issued a **Reset** command then this will not work as intended. A network will be loaded, some commands issued, the array then resets and loads this new element by itself. With deletion, the load command is removed but leaves a gap in the vector, so every command after it must be moved to preserve order. This makes deletion a linear operation.

The solution to this is to constrain the ordering of commands. **Neuron** and **Synapse** commands are separated from the other commands. These commands are logically before any other commands issued. Due to the admin halt discussed in section 3.2, the DANNA device will always be in a halted state when issuing a Cmds object. Since these commands will only be issued while the array is halted, there is no issue with the ordering of them. Therefore, when an element needs to be added to the network, the command can simply be appended to the vector of **Neuron** and **Synapse** commands. When an element needs to be

deleted it can be swapped with the last element, and then the last element is deleted, much like a heap pop operation.

Due to this reordering of these commands to the front, it also only makes sense for a single **Reset** to exist in the Cmds object, and it has to happen before the **Neuron** and **Synapse** commands. Otherwise, it would delete all of the elements from the device, and then any subsequent commands would accomplish nothing, since there would no longer be a network configured. If the user really wants to run two networks in sequence, then they must split the commands into two Cmd objects.

The Cmds object does other error checking and constraints on commands to make them more user friendly. For example, all port configurations of **Neuron** and **Synapse** commands are checked to make sure they are valid. This includes that the neighbor is inside the array and obeys connectivity at any borders as described in section 3.2.1. The ports are also specified by direction and a number. For example, "SW2" means to listen to the neighbor that is two squares to the southwest. This frees the user from having to understand the port orientations discussed in section 1.3.6.

Outputs

The Outputs class simply keeps the outputs from DANNA devices, such as fire events along with their timestamps, and captured state information from the Capture command. This is by far the simplest library object.

Sim

The Sim class is an interface with only one method: simulate(). An instance of a Sim is backed by one of the implementations of a DANNA device, such as the clock-based sim, event-based sim, or the hardware interface. In this way an application developer can quickly and easily use any backing device with a simple one line change at instantiation.

3.3.2 NeoN Study Case

TennLAB researches several neuromorphic devices. Thus it is of interest to try applications on all of them. Rather than requiring application developers to write multiple versions of their applications for the various devices, TennLAB's Neuro library is designed to generically interface with them all. On the DANNA side, this is built upon the objects described in section 3.3.1. In this way, an application only need to be written once and will generally work for any backend model. Neuromorphic Control System for Autonomous Robotic Navigation (NeoN) is one such application that was realized with DANNA hardware [29].

NeoN is a tank drivetrain robot with a scanning LIDAR sensor and bottom mounted whisker switches designed to explore while avoiding obstacles and ledges. The controlling agent used in NeoN is an FPGA DANNA device which controls the tank drivetrain's power on each tread. The decisions are based on the input given by the LIDAR and whisker switches.

The configuration for the DANNA device, in terms of neuron and synapse placement, is solved by EONS. As mentioned above, EONS solves applications via evolutionary optimization. EONS first creates a set of random solutions to the problem which with these neuromorphic devices is a random configuration of neurons and synapses. This population of solutions is tested for their fitnesses via a fitness function. This function must be specified by each application since this meaning changes based on the task. For example, a classification task's fitness function is often some function of the percentage of data points classified correctly.

Using these fitness values, EONS makes decisions on which solutions "die" and which solutions "survive". EONS then will generate a new population through "breeding" and mutation operations. Breeding, or crossover, operations take two solutions and mix them together in the hopes that the resulting solution will have a better fitness. Mutations are random changes to the original solution. For example, randomly changing the weight of a synapse or the threshold of a neuron is a typical mutation. With this new population of solutions, EONS repeats the process of checking fitness and creating new populations.

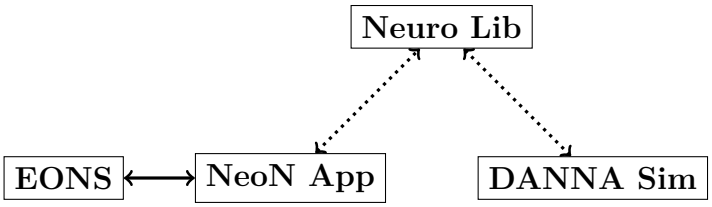


Figure 3.4: Components of a NeoN EONS run

In NeoN's case, fitness is defined by room coverage and avoidance of obstacles. Solutions are configurations of DANNA neurons and synapses. The solution is loaded on the DANNA device and fitness is determined by running the robot with DANNA as the controlling agent. This is done with software simulations of the NeoN robot and DANNA devices to allow faster than real time training. Figure 3.4 illustrates this setup. The Neuro abstraction allows any model to be dropped in as the controlling agent for NeoN, though the solution would have to be re-trained for that model. Once a solution is trained, it can be loaded onto the physical setup of NeoN. This setup is illustrated in figure 3.5.

The EONS training for NeoN was run on the entirety of ORNL's Titan supercomputer for 24 hours. This is why DANNA simulation is extremely important. A large portion of EONS training involves simulating the neuromorphic device. The faster this portion completes, the faster EONS can explore the solution space. The Sim interface on DANNA allows an easy change between software and hardware or to even use them in tandem for training.

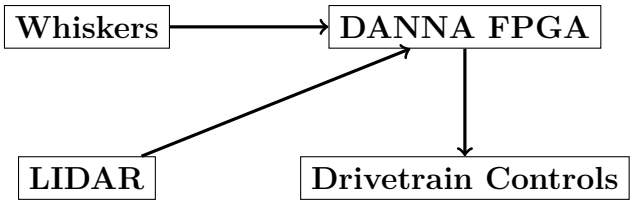


Figure 3.5: Components of the NeoN robot using DANNA FPGA hardware

Chapter 4

GPU Implementation

Parts of this chapter have been previously published in [14]. The prevalence of Graphics Processing Units (GPUs) in modern computing systems led us to explore how we could leverage these resources in DANNA's simulation. Due to causality issues with parallelizing an event simulation, the clock simulator is the basis for the GPU simulation. For example, suppose an event-based simulator tries to process 32 events in parallel. The first event might generate an event that would be placed before any of the next 31 events. That new event could affect the outcome of the other 31 events that were already processed. In addition, GPU programming has many constraints that must be taken into consideration to get performance out of the GPU.

4.1 Basics of GPU Architecture and Programming

Nvidia's *CUDA Toolkit Documentation* [32] gives an excellent description of optimal GPU workloads:

> "More specifically, the GPU is especially well-suited to address problems that can be expressed as data-parallel computations - the same program is executed on many data elements in parallel - with high arithmetic intensity - the ratio of arithmetic operations to memory operations. Because the same program is executed for each data element, there is a lower requirement for sophisticated flow control, and because it is executed on many data elements and has high

arithmetic intensity, the memory access latency can be hidden with calculations instead of big data caches."

The clock based simulation at first glance appears to fit this workload. To understand the design of the simulation, details of GPU architecture and programming are important to understand. This will be described in terms used by Nvidia.

4.1.1 GPU Architecture

The heart of the GPU hardware consists of processors called Streaming Multiprocessors (SMs). The number of SMs on a particular GPU varies between models. SMs execute and manage threads in groups of 32 called warps. The threads of a warp all execute the same instruction on each clock cycle. It is possible for threads to diverge, but the threads not involved in the alternate path will accomplish no work. Generally divergence should be avoided as much as possible to avoid wasting resources. Figure 4.1 shows a simple example.

The SM does its own scheduling of warps that reside on it. The SM have computational units called cores that run the threads of the warps. SMs typically have enough cores to handle 1 to 4 warps executing simultaneously. The SM also has limited capacity for other resources which depend on the GPU model. The SM scheduler can only maintain 16 to 32 blocks at a time (blocks will be explained in section 4.1.2), up to 2048 threads total, and up to 64 warps total. There are up to 64K to 128K registers total, with a maximum of 255

```
if(A)
  funcA();
else
  funcB();
```

Figure 4.1: Threads within the warp that have condition A true will execute while the others run no operations. Afterwards the threads with condition A false will execute while the others run no operations. If funcA() and funcB() have roughly the same runtime, divergence will double the total runtime.

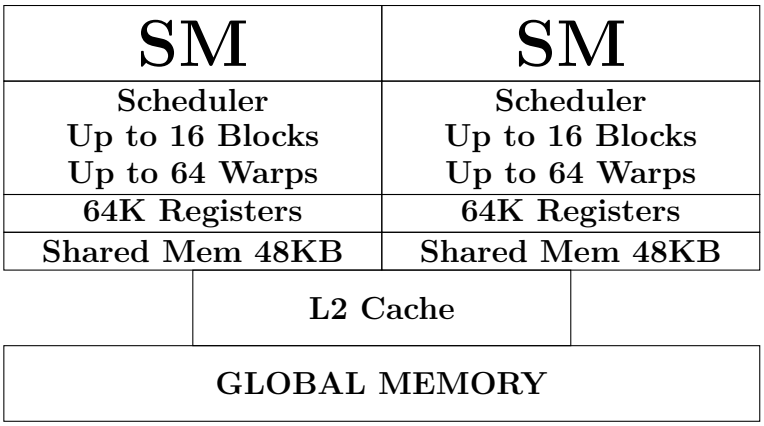


Figure 4.2: A simple visualization of GPU hardware. GPUs have multiple processing units called Streaming Multiprocessors (SM). Each SM has a hardware scheduler that performs best when at full capacity but various limited resources can prevent this such as registers and shared memory. All SMs are connected to the global memory which has a global L2 cache.

registers per thread and a user controlled cache called shared memory that is 48KB to 96KB in size. Figure 4.2 illustrates these resources.

All of this must be considered when looking to exploit a GPU to its fullest potential. Usually the best goal is to have full occupancy in the scheduler with 64 warps. This allows the SM to hide memory latency by swapping in warps that are ready to compute while other warps are waiting on memory accesses. There are many barriers to this, though, with the various limited resources. If a kernel is launched with 64 blocks with 1 warp each, then the limitation of 16 to 32 blocks resident at a time prevents having 64 warps in the scheduler. Additionally there can be too many registers used per thread. Suppose the SM has 64K registers and each thread uses the maximum 255 registers. Then there can only be 257 threads or 9 warps (one of which only has a single thread). There is also a limit on shared memory. If a single block uses the maximum memory allowed per block of 48K, and the SM only has 48K, then only one block can occupy the SM at a time.

The other major issue to consider is the latency of memory access. The global memory of a kernel is accessible by all threads within the kernel, but it has very high latency. There are two major mechanics to work around this. First, threads of a warp can perform coalesced memory accesses. This is accomplished by threads of a warp accessing data that is contiguous

in memory. This essentially lets the data be fetched with a single request, rather than 32 individual requests. Second, data may be loaded into the shared memory that all threads within a block can access. This shared memory is much faster than the global memory, akin to the cache on a CPU, except the programmer controls this manually. This is where the need for high arithmetic intensity comes into play. Data needs to be either part of a relatively expensive calculation and/or used by multiple threads via the shared memory to achieve best results. Any information that needs to be kept after the kernel finishes must be written back to global memory.

4.1.2 GPU Programming

GPU functions are referred to as kernels. A kernel is like a normal C style function except it will execute multiple times via multiple threads. Each thread is assigned a thread id, usually used by the thread to index the data element on which it will do work.

Threads are grouped into blocks, and the collection of all of the blocks form the grid. The number of threads and blocks are determined when an instance of the kernel is launched. Figure 4.3 shows a simple illustration of a kernel's threads. By default, kernel calls are non-blocking so that CPU tasks can run in parallel with the GPU kernels. There are synchronization primitives to wait on kernel calls to finish. There are also primitives for synchronizing all threads of a block from within the kernel. There are no synchronization primitives between blocks in a grid. Blocks can only really communicate between each other through global memory. Blocks are scheduled on SMs and remain there until all threads return from the kernel. Memory allocation is done through special C style memory management functions like `malloc()` and `memcpy()`.

A Simple Example

Figure 4.4 shows a simple example of a GPU kernel written in Nvidia's CUDA. The kernel is simply receiving a pointer to an array of elements and the size of that array. The threads of the kernel simply loop over all the elements and multiply by two.

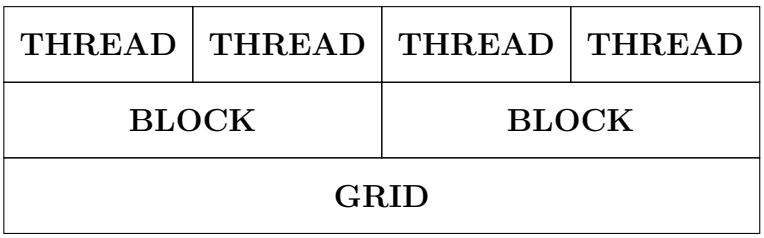


Figure 4.3: GPU kernels are launched with a user specified number of blocks and number of threads per block. The collection of all the blocks is called the grid. Thus, a grid consists of some number of blocks and each block consists of some number of threads. Blocks are assigned to an SM at launch and the threads within are divided into warps for the SM's scheduler.

```
__global__ void double_em(int *array, int size)
{
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  while(i < size)
  {
    array[i] *= 2;
    i += gridDim.x * blockDim.x;
  }
}
```

Figure 4.4: Simple kernel that multiplies all the integers of an array by two.

When the kernel is launched, the user specifies the number of threads per block and the number of blocks. These numbers may be specified in three dimensions, but in this example we have assumed one dimension. Thus, **blockIdx.x** is identifying the block id in which the thread is contained, and **threadIdx.x** is the thread's id within that block. Just like arrays these are always numbered zero to $N - 1$ where N is the number of blocks or threads.

As mentioned in section 4.1.1, groups of 32 threads are partitioned into warps. The threads of a warp are always numbered sequentially so the first warp will contain threads 0 through 31. Now suppose this kernel is launched with two blocks each with 32 threads. This will create two warps. **blockDim.x** will be 32, since that is how large the blocks are in the x dimension.

The first warp will begin by multiplying the array elements 0 through 31, since it is in **blockIdx.x** of 0 and the threads are numbered 0 through 31. Since these elements are contiguous in memory, the GPU may coalesce the memory request of each thread. The same will happen when the answer is written back to memory. The threads then increment their indices by 64. This is often referred to as the stride of the loop. It is 64 because **gridDim.x** is two, which is the number of blocks with which we launched the kernel and **blockDim.x** is 32. One might think we have skipped elements 32 through 63 now, but the second warp is responsible for those elements.

This kernel may be launched with any number of blocks and threads safely to allow it to scale depending on the GPU hardware. For example, if the kernel had been launched with ten blocks, then the stride would be 320 instead of 64. If the GPU has ten SMs, then it is possible for all ten blocks to be running simultaneously. It is also possible that the GPU only has resources to schedule one of the two blocks that were launched. In this case, every other 32 elements in the array would be multiplied by 2 and that block would finish. This frees resources to schedule the second block and complete the other half of the array. Additionally, this example has a low arithmetic intensity so it may be better to launch the kernel with less blocks with more threads. This will locate more threads on the same SM and the scheduler may context switch to help hide memory latency.

There is no divergence in this example except for the last portion of the array. If the array size is not a multiple of 32, then the threads with an index out of bounds of the array

will not execute the multiplication on the last iteration of the loop. The threads will still have to iterate through the same instructions as the other threads but will not actually execute them. For this example it is not a big deal, but divergence can easily disrupt GPU efficiency. At this point, the reader may appreciate that even in this simple example there are many configuration details to consider in GPU programs.

4.2 DANNA simulation on GPUs

This section discusses specifics of mapping the DANNA simulation to GPUs. The clock simulation at first glance seems to fit the paradigm well.

4.2.1 Single Network Simulation

The obvious first approach is to have the GPU run a single simulation. This is a standard GPU loop like described in section 4.1.2. Each element can be processed in parallel during the loop. The data of the array is changed from an array of structures to a structure of arrays, a common practice in GPU programming. This allows individual variables within the original structure to be contiguous in memory, forming coalesced memory requests.

However, because elements in the array may be one of two types, this causes divergence that heavily affects performance. This is the situation shown in figure 4.1, where the condition is testing the element's type. Figure 4.6 shows a small 4 by 4 array. For simplicity,

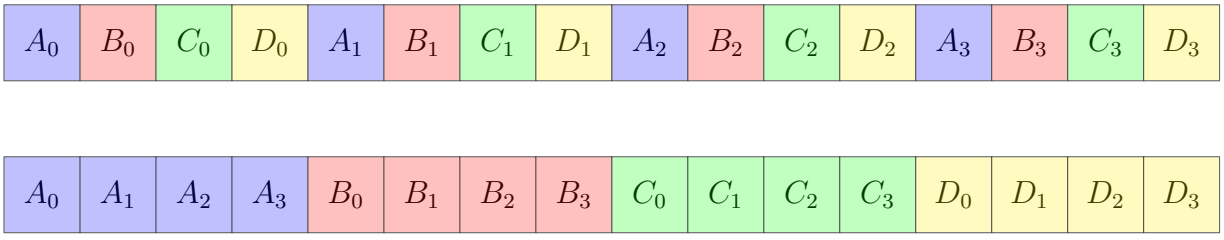


Figure 4.5: Memory layout for an array of structures with 4 members named A, B, C, and D. Typically in programming one would use an Array of Structures (top), but when threads of a warp access the same member of different structures the memory accesses cannot be coalesced. Instead if the memory is laid out as a Structure of Arrays (bottom), coalesced memory accesses can be achieved.

this example will use a warp size of 4. The figure is colored to show how the warps would access elements. If the elements within a single color are not of the same type, then there is divergence which greatly increases the runtime.

To eliminate this divergence, warps are divided into groups of neuron workers and synapse workers. A list of all the neuron indices and a list of all the synapse indices are maintained. Figure 4.7 illustrates this change. Now the threads of the warp are guaranteed to work on the same types.

Unfortunately, this creates another issue. Remember that for best performance, groups of cores should access data that is contiguous in memory, but groups of workers now access specific elements that are most likely not contiguous. For this simulator though, the benefits of eliminating the divergence outweighs the penalty of non-contiguous memory access.

Another barrier to performance involves synchronization. Since this version is split across multiple blocks, it must use the completion of kernel calls for synchronization. This results in each port cycle requiring two different kernel calls for the loops described for the clock based simulation in section 3.1.1. Figure 4.8 illustrates this issue.

Not only is the setup for a kernel call rather expensive, but all state must be written to global memory as well, which is also slow. Furthermore, it became obvious that there is not a high arithmetic intensity for the simulation. Only one port is scanned, and then the state must be updated. This precludes any meaningful use of the GPU's shared memory. A high arithmetic intensity is often achieved with the use of shared memory. DANNA simulation has

N	S	N	S
N	S	N	S
N	S	N	S
N	S	N	S

Figure 4.6: A simple 4 by 4 array where N represents a neuron and S represents a synapse. Assuming a warp size of 4, elements of the same color would be processed by the warp at the same time. Since it has a mixture of element types, this causes divergence.

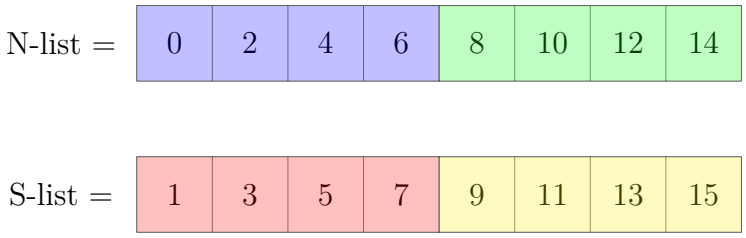


Figure 4.7: The N-list and S-list contain the indices of all the neurons and synapses respectively. Warps assigned to work on a particular type will loop through this list working on this list divergence free.

a low arithmetic intensity, largely due to these unavoidable synchronization points and that all operations are simple integer operations rather than expensive floating point operations.

Since this simulation is more data bound, large networks with high activity are required to outpace the CPU simulators. This allows the GPU to hide the latency by context switching warps as they wait for data. It also mitigates the cost of the synchronizations because the simulation takes longer in between synchronizations while the cost of them is constant. Additionally, at high activity the clocking approach of simulation scales better than the event approach.

Other network activities such as loading elements and array I/O are handled by the CPU.

4.2.2 Multiple Networks Simulation

Unfortunately, large networks are at present not the norm with DANNA applications. For example, the DANNA network for NeoN was on a 15 X 15 array. Currently, the most common

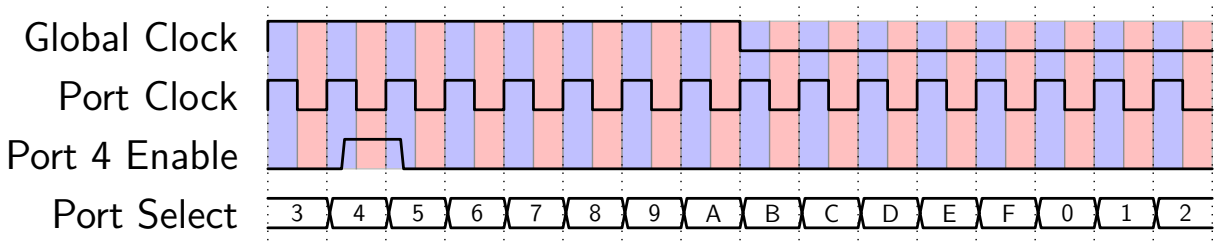


Figure 4.8: This figure shows all locations of the two loops shown in figure 3.2. Every color transition requires a synchronization making for 32 synchronization points for a single global cycle.

use case for the simulator is to perform EONS on large populations of small networks. To leverage the GPU for this use case, this version is designed to simulate multiple DANNA devices simultaneously. Each block in the grid is set up to be a stand-alone simulation of DANNA. Now only a single kernel call is needed per global cycle, because the block level synchronization now may be used.

Each network is completely independent. This allows the configuration of elements and inputs to be unique per network. The only restriction is that all networks must be designed for the same array size. While this version also needs to synchronize 32 times per global cycle, it is a bit less costly because it is at the block level rather than the entire GPU. Otherwise, this version has much in common with the single network version.

One might ask why not simply run multiple versions of the single network simulation? There is a limitation on the number of kernels that can be run simultaneously. For most models of Nvidia GPUs, this limit is either 16 or 32. This would hardly make full use of the GPU.

Chapter 5

Performance

5.1 Stress Test

To compare the performance of the simulators, we employ a dense, recurrent network, created by tiling a 4 by 4 grid composed of four neurons and twelve synapses originally presented in [38]. This network is shown in Figure 5.1. While this network does not solve any particular application, it serves as a stress test for the simulators' performance since most networks generated by EONS are less dense and less recurrent. All synapses are configured with weights of 127 and delay of 1. All neurons are configured with thresholds of 1.

The performance between the clock-based and event-based simulators varies on different conditions. The clock-based simulation's performance is $\mathcal{O}(tn)$ where n is the size of the array and t is the number of cycles simulated. It is also minorly affected by network activity. The

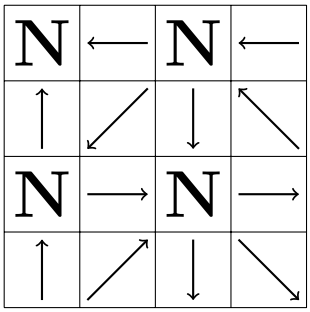


Figure 5.1: 4 by 4 repeating tile for simulator stress testing. N represents neurons and arrows indicate a synapse connection.

event-based simulation's performance is $\mathcal{O}(e \log e)$ where e is the number of firing events. It is also minorly affected by array size.

To compare the simulators, we employed networks that were 15 by 15 and 80 by 80 in size. We chose the 15 by 15 size, because it has been a common size used by EONS to solve many of our applications. The 80 by 80 array size was chosen because it is the largest DANNA array size to fit on a single FPGA to date. We ran 224 tests for each array size for 10,000 cycles. In each test, we varied the number of events generated by randomizing the inputs to the network. Since the clock-based simulator is minorly affected by the number of events processed, this test should reveal a threshold where it becomes better than the event-based simulator for a particular array size. We ran tests on a machine with two Xeon E5-2697 v3 @ 2.60GHz CPUs and a GeForce GTX TITAN GPU. Simulators were compiled with gcc v5.4.0 and CUDA 9.1.

In Figures 5.2 and 5.3, the Event, Clock and GPU Single simulators show the timing of each of the 224 runs as a scatter plot. The GPU Multiple version ran all 224 tests in parallel with a single execution. Thus, its timing is a single line representing the total time of execution divided by 224. This represents the average timing required per test to sequentially run all 224 tests as fast as the GPU Multiple version.

Figure 5.2 shows the 15 by 15 timings. The single network GPU version does horribly on this test, because it barely uses any of the GPU's available resources, due to the small array size. The multiple network GPU version, on the other hand, does use all available resources, because it runs all tests at once, in effect simulating a 15 by 15 by 224 array. While this appears great for multiple networks on GPU, one must remember that this is a comparison of the whole GPU to a single core on the CPU.

As for the CPU versions, the Clock-based simulation starts to surpass the Event-based simulation at approximately 500,000 events over the 10,000 cycles. That means nearly a quarter of all array elements must fire every single cycle for Clock to pull ahead. This is highly uncommon in our experience and why the Event-based simulation greatly speeds up EONS training (see below in Section 5.2).

Figure 5.3 shows the 80 by 80 timings. Now, single networks on the GPU begin to pull ahead because it is closer to fully utilizing the available resources. Similarly, multiple

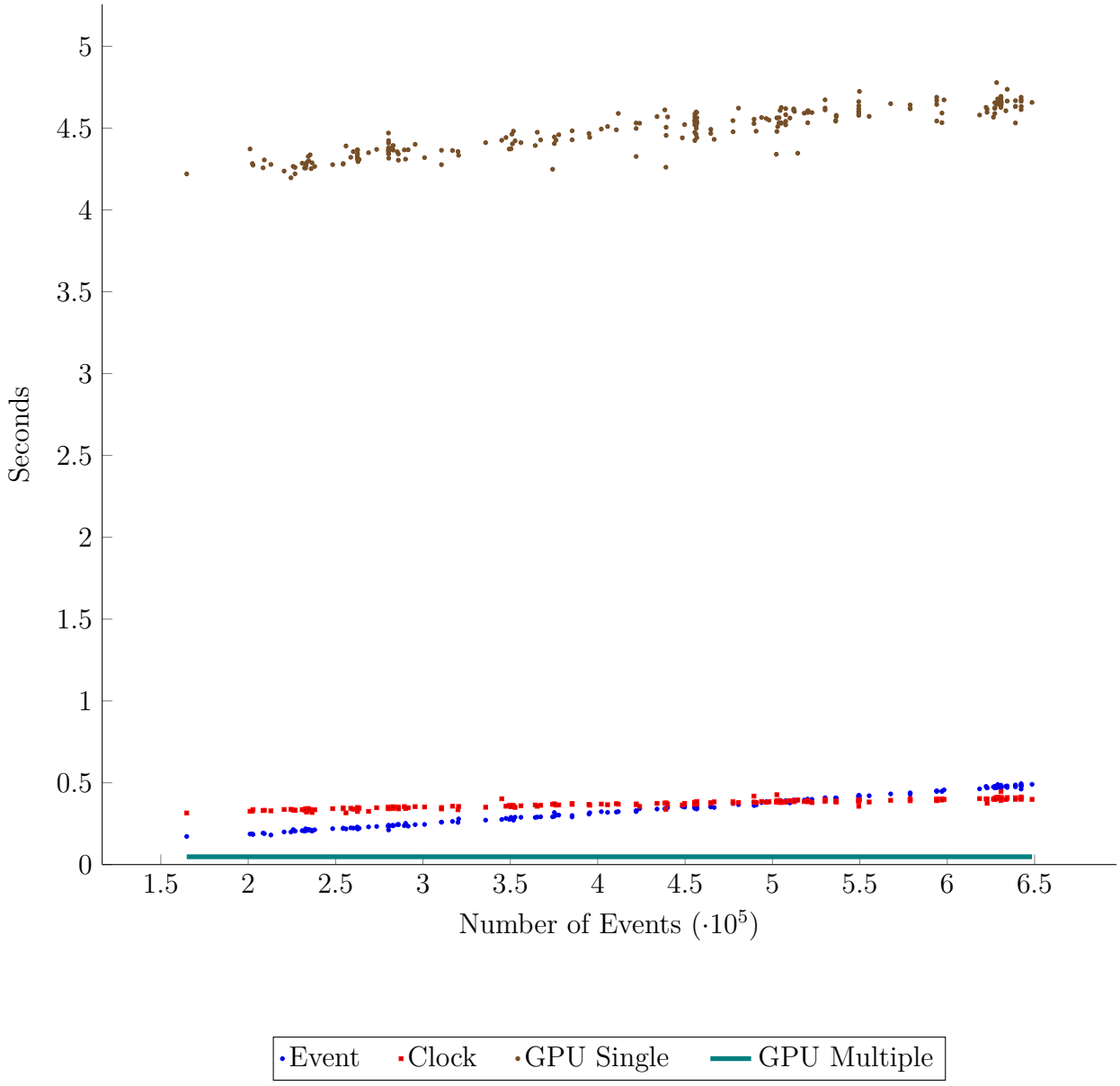


Figure 5.2: 224 runs of a 15 by 15 grid pattern for 10K cycles

networks on the GPU show a larger improvement over the CPU versions. With more available work, the GPU is better at dealing with the memory accesses, mentioned in section 4.1.1. For this test, the intersection of Event and Clock performance is approximately 11,000,000 events. This requires roughly a sixth of all array elements to fire every single cycle for Clock to be superior.

Additionally to get a sense of scaling, we ran a single test on an array size of 1000 by 1000. The results are shown in table 5.1. While we currently do not use networks of this size, it is of interest for future work (e.g. for a reservoir computing model). Here the GPU simulation dominates, and will likely be the simulator of choice for very large dense networks.

5.2 EONS Test

While the stress tests highlight each simulator's strengths, an EONS training run is a realistic use case. We ran EONS training with a population of 1,000 networks for 20 epochs on a classification application with the Breast Cancer Wisconsin dataset from UCI [13]. The array size for this application was 27 by 27, to fit the appropriate input and output requirements. The dataset has 699 entries of which we used 350 for training. Each run used the same randomization seed so that all the same networks would be generated for each of the simulators.

Table 5.2 shows the timings for these EONS runs, using each of the simulators. For each of the epochs, all 350 entries are classified by each of the 1,000 networks to get a fitness score. It takes a total of 105,000 cycles to run all 350 entries through DANNA. With that fitness, EONS makes a new population of 1,000 networks then starts a new epoch. Thus, over the 20 epochs a total of 20,000 networks are created. In this test, the average number of

Table 5.1: A single run of a 1000 by 1000 grid pattern for 10K cycles which generated 1,645,539,386 events.

Simulator Type	Timing in seconds
Event	2182.65
Clock	1608.66
GPU Single	88.4453

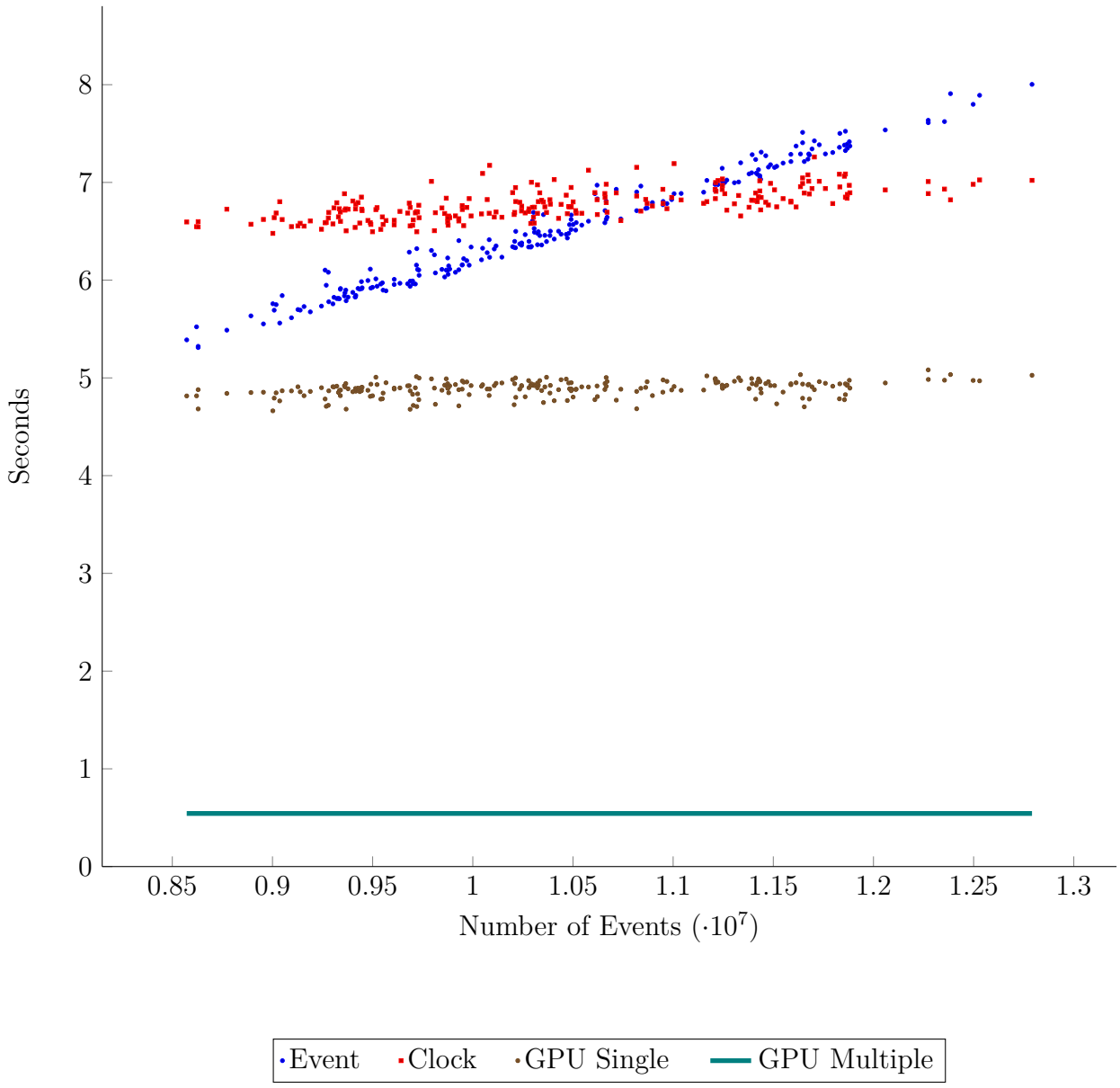


Figure 5.3: 224 runs of a 80 by 80 grid pattern for 10K cycles

neurons and synapses in these networks was 34.9 and 61.8 respectively. The average number of events generated over the 105,000 cycles was 26,413.

The Event simulation is magnitudes better suited for this particular application than the Clock simulation. There are several reasons the Event simulation does much better. First, EONS is designed to utilize all available CPU cores by instantiating multiple simulators and dividing the networks among them for fitness testing. Thus, this compares the usage of all CPU cores rather than a single core like the stress test. The testing machine has two Xeon processors with 14 cores each and hyper-threading, thus allowing 56 threads to run simultaneously. Additionally, networks generated by EONS are not nearly as dense or recurrent as the stress test, and they generate far fewer events. The average element usage for these networks is approximately 13%, and they only generate an average of one event every four cycles.

All the performance tests are summarized in table 5.3 with the best for each test in bold.

Table 5.2: EONS training on Breast Cancer Wisconsin for 20 epochs with a 1,000 network population. Running this test as single networks on GPU takes on the scale of days so it is ignored here.

Simulator Type	Timing in seconds
Event	59.453
Clock	1973.44
GPU Multiple	6738.43

Table 5.3: Summary of performance tests. Bold times are the best of the four simulators for that particular test. Measured in seconds.

	Event	Clock	GPU Single	GPU Multiple
15 by 15, 165K events	0.171972	0.315259	4.22024	**0.047789**
15 by 15, 648K events	0.490571	0.398727	4.65699	**0.047789**
80 by 80, 8.57M events	5.38886	6.59532	4.81528	**0.543299**
80 by 80, 12.8M events	8.00327	7.02117	5.02648	**0.543299**
1000 by 1000, 1.65B events	2182.65	1608.66	**88.4453**	-
EONS, 528M events	**59.453**	1973.44	-	6738.43

Chapter 6

Detection of Patterns in Noise

This chapter describes an experiment where EONS and the neuromorphic models at TennLAB attempt to solve an experiment first presented by Masquelier, Guyonneau and Thorpe in 2008 [26]. In this experiment, the authors employ a simple spiking neuromorphic network composed of a single neuron and 2000 synapses. The neuron is an accumulate-and-fire neuron with leak. Each of the synapses fires into the neuron at its own rate, determined by independent Poisson distributions that change over time. At random times, a fixed pattern involving some fixed subset of the synapses is pulsed. The synapses not involved in the pattern produce firings based off of their probability distributions; however, the synapses involved in the pattern only fire the pattern during that period. The goal is for the neuron to fire while the pattern is being presented. In the original experiment, the synapses are uniformly seeded with some initial weight, but have STDP mechanics implemented so that they potentiate when they fire at times that closely precede the neuron's firing, and they depress when the fire at times that closely follow the neuron's firing. Over time, their system would learn to detect the pattern, using only the STDP mechanics, and no supervised training.

This chapter describes the development of an application, based on Masquelier *et al's* experiment, within the TENNLab software development framework. The goal of the application is to explore how the various TENNLab neuromorphic device models, including DANNA, fare at this experiment.

6.1 Spike Train Generation

The application generates input events in a way to match [26], but also to be configurable in a general way. Herein, all of the parameters that may be modified are explained. Each input starts at some firing rate r (measured in Hz). The time step between generating firing events is x milliseconds. Since this is a discrete Poisson distribution, the chance for a spike at each step is a simple Bernoulli distribution based on r and x. Once it is determined if a spike exists for a particular time step, the firing rate r is updated based on that input's change in firing rate dr (measured in Hz/s). Once r is updated, dr is also updated by a random value in the range $[\Delta dr_-, \Delta dr_+]$ (measured in Hz/s) but dr itself is capped in the range $[dr_{min}, dr_{max}]$ (measured in Hz/s). r is capped in the range $[r_{min}, r_{max}]$. Since hardware TennLAB models have discrete time steps, there is no additional jitter added to the timestamp of fires. The pattern to be detected is simply sampled at some random point during this spike train. The duration of the pattern is also configurable. All of these parameters and their settings used for this experiment are summarized in table 6.1.

Table 6.1: Summary of configurable parameters for spike train generation and their default settings used for this experiment.

Parameter	Default	Description
r	45 Hz	Firing rate
x	1 ms	Granularity of discrete time step
dr	0 Hz/s	Rate of change in firing rate
Δdr_-	-360 Hz/s	Minimum singular update of rate of change in firing rate
Δdr_+	360 Hz/s	Maximum singular update of rate of change in firing rate
dr_{min}	-1800 Hz/s	Minimum rate of change in firing rate
dr_{max}	1800 Hz/s	Maximum rate of change in firing rate
r_{min}	0 Hz	Minimum firing rate
r_{max}	90 Hz	Maximum firing rate

6.2 Models Tested, and Model Specific Adjustments

In this experiment, there are four neuromorphic models tested. These are DANNA, NIDA, mrDANNA and DANNA2. In the subsections that follow, each model is briefly described, along with the adjustments to the experiment that had to be made for each of these models.

6.2.1 DANNA

DANNA arrays typically have inputs on the far left column and outputs on the far right column. Due to element connectivity, this creates a minimum response time for the DANNA array. The input cannot possibly cause an output response for some amount of cycles. Since the original experiment requires the neuron to respond while the pattern is being presented, DANNA could not possibly respond once the array is sufficiently large.

To adjust, an expected delay parameter moves this response window. Suppose the pattern (and thus the response window) starts at cycle 50. With an expected delay of 100, the application looks for a response for this pattern starting at cycle 150.

For DANNA in the EONS tests in section 6.3 below, the expected delay is set to 4 times the number of inputs. This allows enough time for any inputs to propagate through the array to the output and EONS can adjust the delay parameters properly.

Additionally, DANNA is currently the only model with physical placement issues. Typically for DANNA devices, inputs are lined across the left column and outputs are on the right column. Thus, for 2000 inputs there must be 2000 rows. By default, EONS makes a configuration that is square so the entire array would be 2000 by 2000. EONS then creates generic networks that have to be placed on the DANNA array. This placement algorithm is slow for DANNA and greatly extends the runtime of EONS when the array sizes are large. For this reason, the 2000 inputs test uses a 2000 by 30 array size, which greatly reduces the runtime compared to 2000 by 2000.

6.2.2 NIDA

NIDA [38] was the first model developed by TENNLab. It features analog neurons and synapses laid out in 3D space, where synaptic distance is equal to the euclidean distance

between the neurons that a synapse connects. NIDA is implemented in software simulation only. The fact that synapse delays are functions of neuron placement makes this application challenging with NIDA.

The default construction for NIDA networks in EONS places all the inputs in a line on one end of the network and the outputs in a line on the other end. Both lines exist on the same plane, and the entire network space is a cube. Figure 6.1 shows the two dimensional plane on which the inputs and outputs exist. This shows the *minimum* delay to get the input information to the output. As the size of the network grows, it can become impossible for all the input information to propagate to the output within the response window.

To adjust for this, there is a time dilation parameter. This determines how many device cycles are used per time step of the spike train. By default, it is set to one cycle per time step in the spike train but this will not work for NIDA. NIDA instead uses twice the number of inputs per time step of the spike train. This allows all inputs the chance to propagate to the output within a single time step of the spike train with time to spare for indirect routes.

6.2.3 mrDANNA

mrDANNA is a mixed analog and digital architecture that makes use of memristor technology [6]. It is loosely based on DANNA. mrDANNA has little restriction, because the hardware implementation has yet to be finalized. Thus, it is flexible in connectivity like NIDA. Synapse delays are configurable to any value regardless of location. There are no modifications specific to mrDANNA for this experiment.

6.2.4 DANNA2

DANNA2 is the successor to DANNA designed by Parker Mitchell [28]. Its design is heavily influenced by the lessons learned from our research with DANNA. It is also implemented on FPGA hardware with a sparse mode that alleviates the placement constraints from the original DANNA. Each element may connect to 24 neighbors, with more possible through a pass-through mechanic. There are no modifications specific to DANNA2 for this experiment.

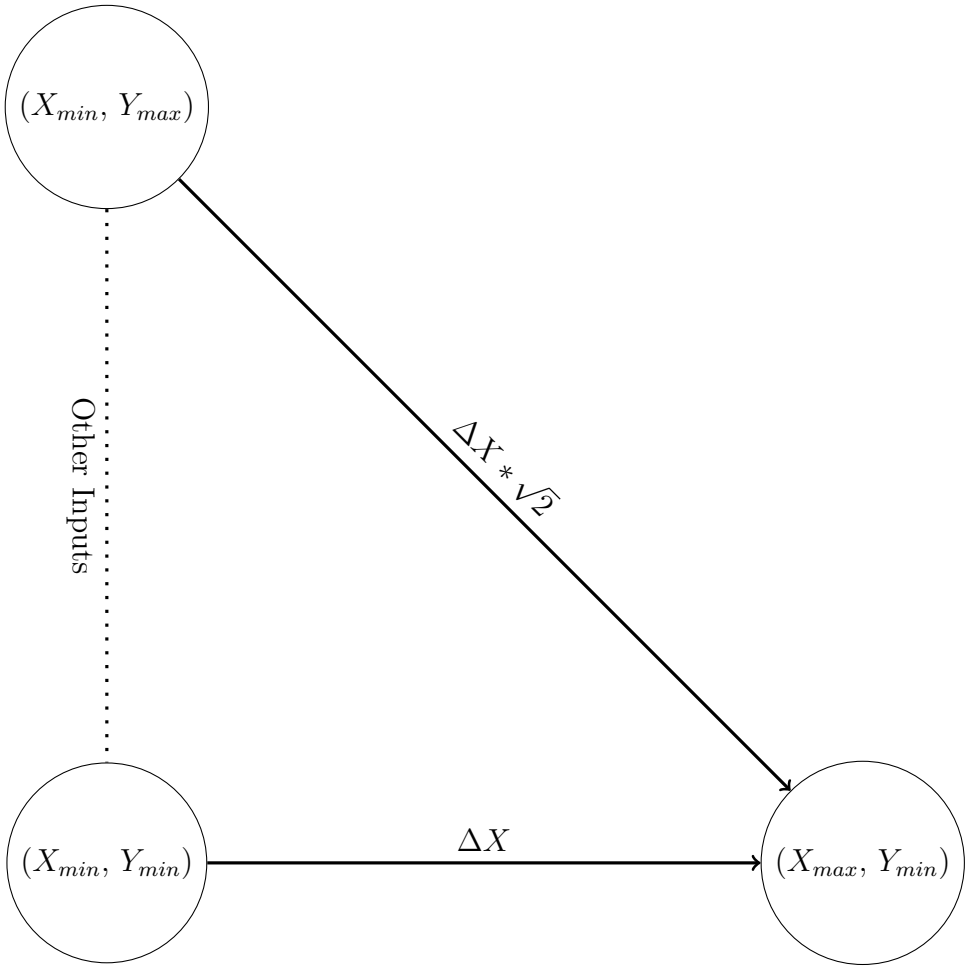


Figure 6.1: NIDA inputs line the left side of the plane and outputs line the right side of the plane. Since the plane is square, the distance between the top input and bottom output is $\sqrt{2}$ times longer than the bottom input to bottom output.

6.3 EONS Configuration

The fitness function used for this experiment is a common measure of accuracy for binary classification called the F_1 score. True positives are counted the first time the network responds within the response window. False positives are fires outside of the window or additional fires within the window. This is to encourage the networks to fire only a single time within the window much like the neurons from the original experiment. False negatives are counted when the network fails to fire at all within a particular window.

For each of the four models, inputs of 10, 150, and the original experiment's 2000 were used. The population size for 10 and 150 inputs is 1000. For 2000 inputs, a population size of 100 is used to mitigate the run time of EONS on such large networks. The spike train generation parameters match those of the original experiment and are listed in the previously mentioned table 6.1.

6.4 Results

Some of the results in this section use boxplots. The box in the boxplots represents the 0.25-quantile to the 0.75-quantile with the dividing line representing the 0.5-quantile. The whiskers represent the smallest (or largest) data point within 1.5IQR (Inner Quantile Range) of the box.

For each network size, there are six sets of 100 EONS runs.

1. DANNA with default parameters (which include depression/potentiation (DP) mechanics).
2. DANNA with the DP mechanics turned off.
3. NIDA with default parameters (which include DP mechanics).
4. NDIA with the DP mechanics turned off.
5. mrDANNA with default parameters (which include DP mechanics).
6. DANNA2 with default parameters (which does not include DP mechanics).

For 10 input runs shown as a box plot in figure 6.2, all models are able to learn the pattern to some extent with DANNA2 performing the best. A larger divide in performance appears with the 150 input test shown in figure 6.3. mrDANNA does extremely poorly with the original DANNA not far ahead. DANNA2 does exceptionally well with the lowest fitness of 93%, and all but two tests achieved a fitness above 96%. NIDA does well, but not quite as well as DANNA2. This is believed to be due to a couple of factors. First, these two models have no spatial limitation on neighbor connectivity. DANNA2 neurons can only have 24 incoming synapses, but the location of the neighboring neuron can be anywhere in the array. Second, DANNA2 is unique from the other models, because it has an integration window like typical neuron models. All inputs on a given timestamp are summed before checking against the threshold. NIDA and DANNA both read inputs one at a time and are checking against the threshold after each.

For the 2000 input runs, DANNA2 achieves similar results, while all of the other models fail to achieve any fitness above 55%, with most close to 0%. Networks of this size are extremely difficult for EONS to find solutions, as it creates an extremely large search space. NIDA and mrDANNA have complete freedom to connect any elements in addition to using floating point numbers for thresholds and weights which exacerbate the issue.

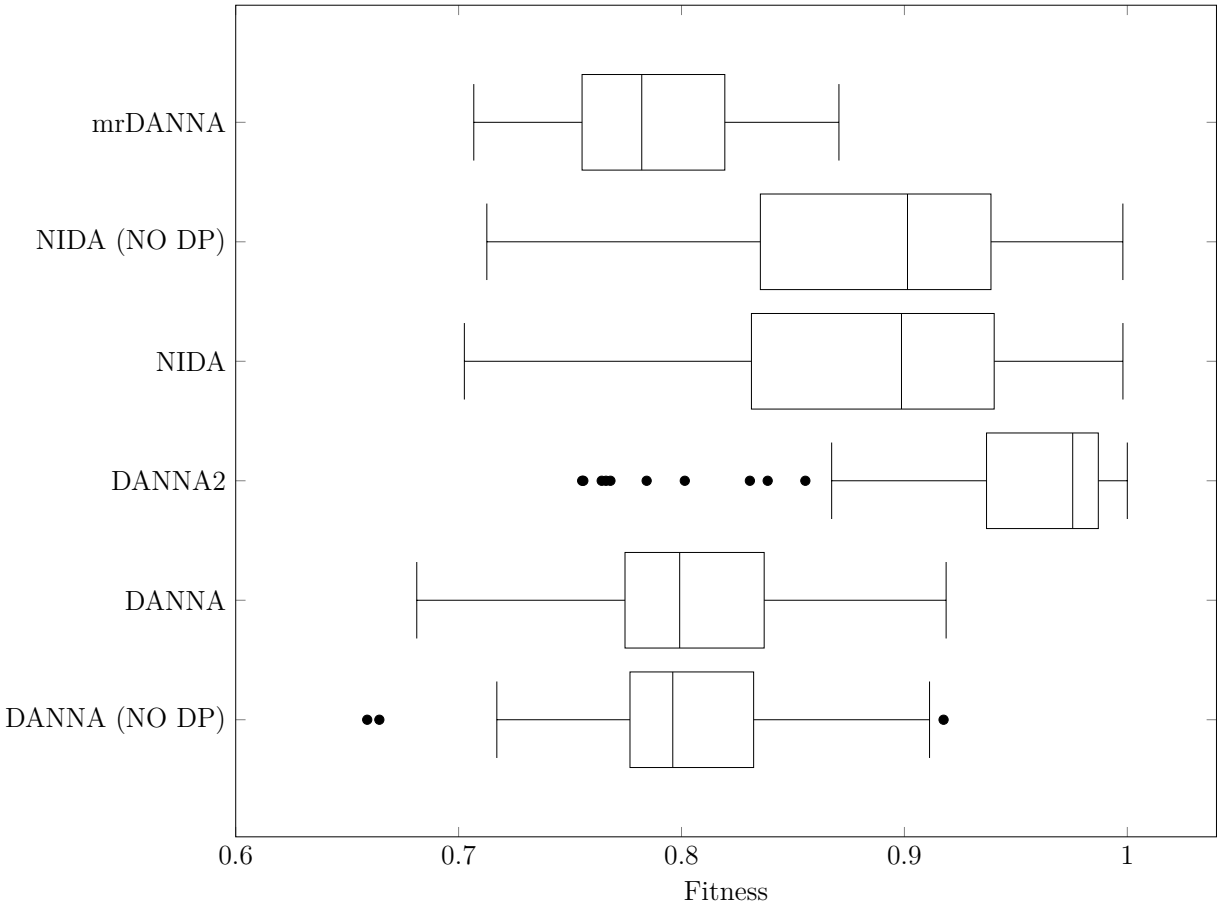


Figure 6.2: Box plot for EONS 10 input test. 5 out of the 10 inputs are involved in the pattern. There are 100 data points, which represent the best fitness achieved for 100 different patterns.

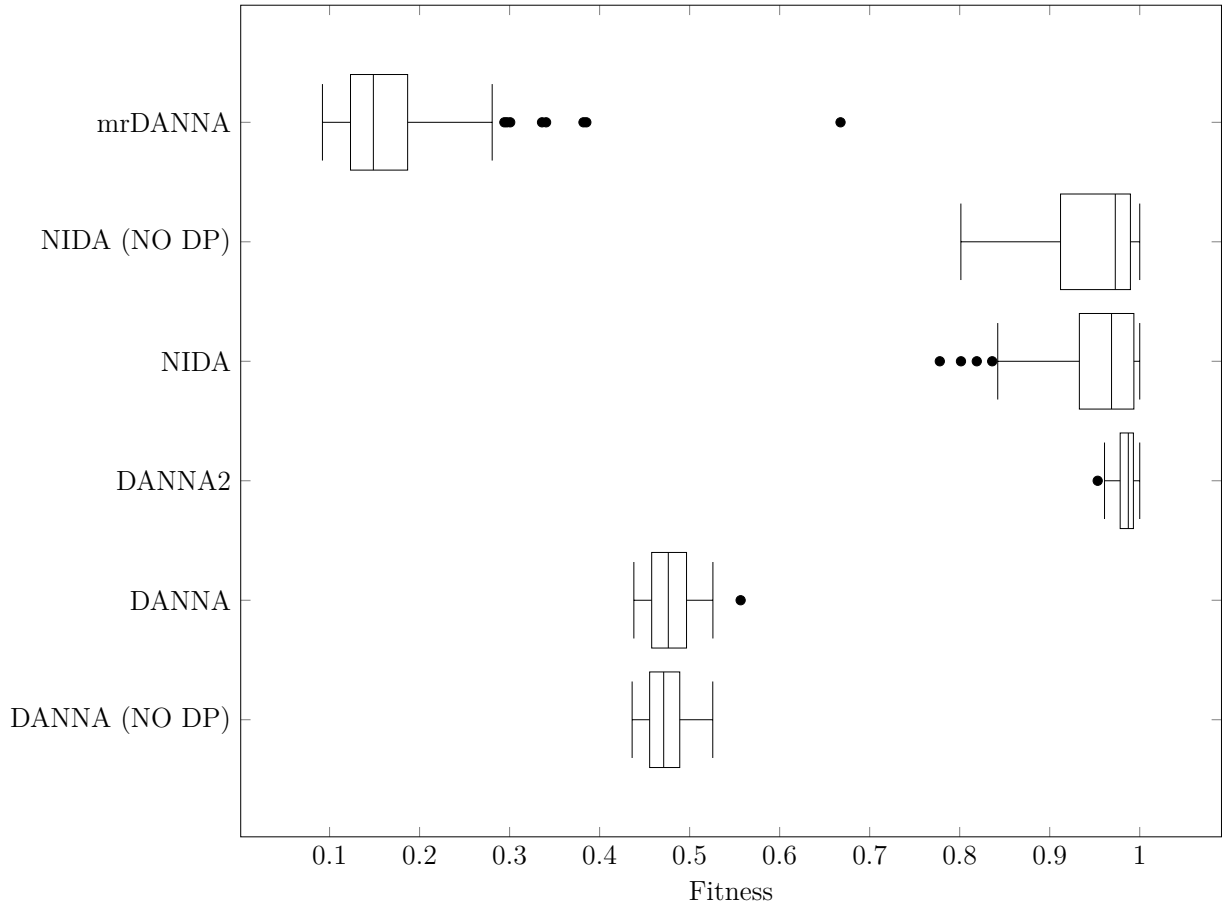


Figure 6.3: Box plot for EONS 150 input test. 75 out of the 150 inputs are involved in the pattern. There are 100 data points, which represent the best fitness achieved for 100 different patterns.

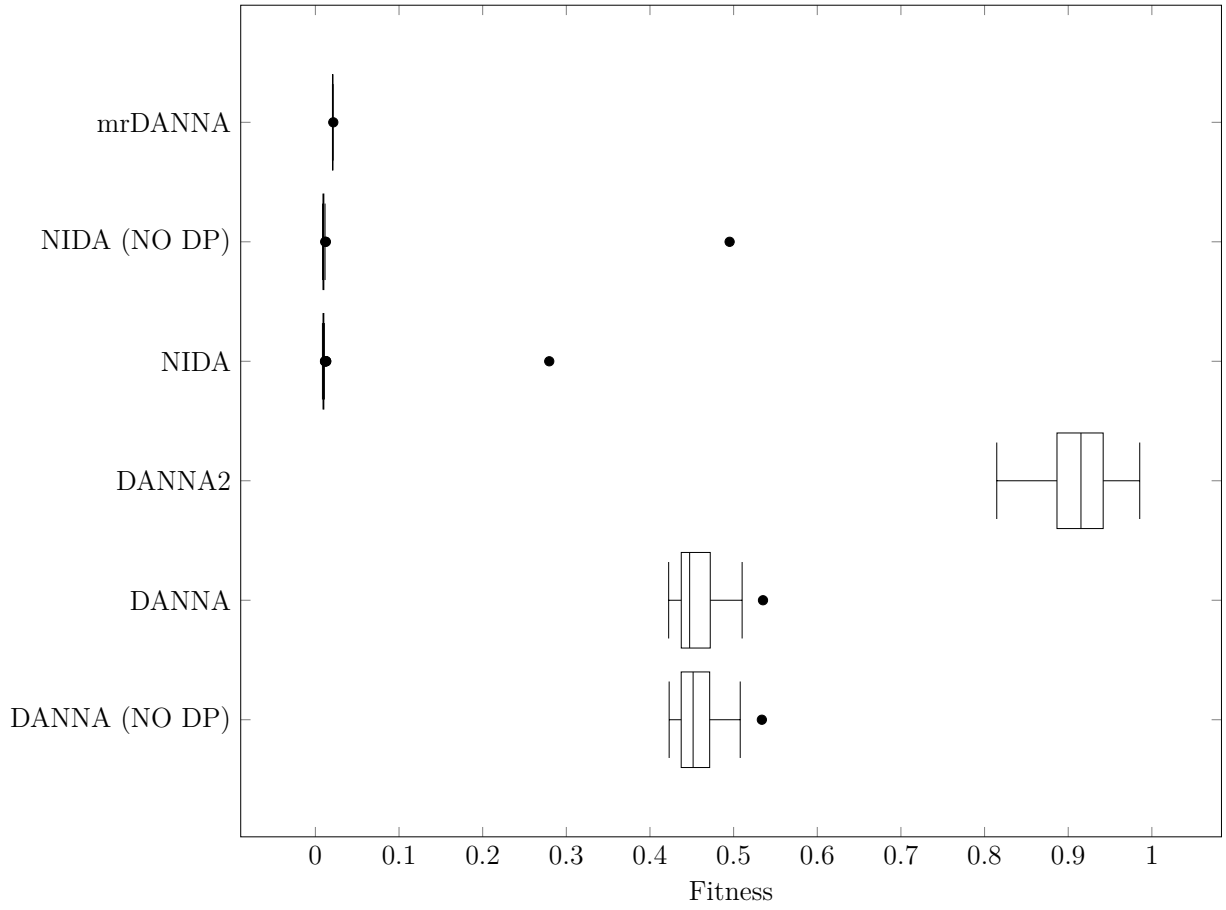


Figure 6.4: Box plot for EONS 2000 input test. 1000 out of the 2000 inputs are involved in the pattern. There are 100 data points, which represent the best fitness achieved for 100 different patterns.

Chapter 7

Conclusions

Neuromorphic architectures are a promising technology with much to explore. DANNA is one such architecture that focuses on simplicity and FPGA implementation. This dissertation has enabled the verification and exploration of the DANNA platform through the four cycle accurate software simulations, hardware communication layer, and application support.

The clock-based simulation was invaluable for the initial phases of DANNA. The mimicry of the hardware counterparts helped with the detection of many bugs and keeps the memory footprint small. However, it is inefficient for training, and thus the event-based simulation was created. It greatly increased the speed of simulation though at the cost of memory. The GPU implementations enable the use of an extremely common computing resource on modern compute nodes but have been challenging to achieve high performance due to the properties of the model.

Ultimately this has led to the development of a successor platform DANNA2 that has considered the lessons learned from the exploration of DANNA. Reasoning about DANNA's behavior is difficult due to many of the model's properties. DANNA2, with the assistance of a new hardware communication layer, looks to change this. While the work here is with the original DANNA, it will accelerate the development and exploration of DANNA2.

Bibliography

[1] Akopyan, F., Sawada, J., Cassidy, A., Alvarez-Icaza, R., Arthur, J., Merolla, P., Imam, N., Nakamura, Y., Datta, P., Nam, G.-J., et al. (2015). Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(10):1537–1557. 24

[2] Amir, A., Datta, P., Risk, W. P., Cassidy, A. S., Kusnitz, J. A., Esser, S. K., Andreopoulos, A., Wong, T. M., Flickner, M., Alvarez-Icaza, R., et al. (2013). Cognitive computing programming paradigm: a corelet language for composing networks of neurosynaptic cores. In *Neural Networks (IJCNN), The 2013 International Joint Conference on*, pages 1–10. IEEE. 24

[3] Arthur, J. V., Merolla, P., Akopyan, F., Alvarez-Icaza, R., Cassidy, A., Chandra, S., Esser, S. K., Imam, N., Risk, W. P., Rubin, D. B. D., et al. (2012). Building block of a programmable neuromorphic substrate: A digital neurosynaptic core. In *IJCNN*, pages 1–8. Citeseer. 24

[4] Benjamin, B. V., Gao, P., McQuinn, E., Choudhary, S., Chandrasekaran, A. R., Bussat, J.-M., Alvarez-Icaza, R., Arthur, J. V., Merolla, P. A., and Boahen, K. (2014). Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations. *Proceedings of the IEEE*, 102(5):699–716. 25

[5] Beyeler, M., Carlson, K. D., Chou, T.-S., Dutt, N., and Krichmar, J. L. (2015). Carlsim 3: A user-friendly and highly optimized library for the creation of neurobiologically detailed spiking neural networks. In *Neural Networks (IJCNN), 2015 International Joint Conference on*, pages 1–8. IEEE. 23

[6] Chakma, G., Dean, M. E., Rose, G. S., Beckmann, K., Manem, H., and Cady, N. (2016). A hafnium-oxide memristive dynamic adaptive neural network array. In *International Workshop on Post-Moore's Era Supercomputing (PMES)*, Salt Lake City, UT. 60

[7] Chan, J. (2015). Implementation of a neuromorphic development platform with DANNA. Masters Thesis, University of Tennessee. 31

[8] Daffron, C., Chan, J., Disney, A., Bechtel, L., Wagner, R., Dean, M. E., Rose, G. S., Plank, J. S., Birdwell, J. D., and Schuman, C. D. (2016). Extensions and enhancements for the DANNA neuromorphic architecture. In *IEEE SoutheastCon 2016*, Norfolk, VA. 3

[9] Davies, M., Srinivasa, N., Lin, T.-H., Chinya, G., Cao, Y., Choday, S. H., Dimou, G., Joshi, P., Imam, N., Jain, S., et al. (2018). Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro*, 38(1):82–99. 25

[10] Davison, A. P., Brüderle, D., Eppler, J. M., Kremkow, J., Muller, E., Pecevski, D., Perrinet, L., and Yger, P. (2009). Pynn: a common interface for neuronal network simulators. *Frontiers in neuroinformatics*, 2:11. 23

[11] Dean, M. E., Chan, J., Daffron, C., Disney, A., Reynolds, J., Rose, G. S., Plank, J. S., Birdwell, J., and Schuman, C. D. (2016). An application development platform for neuromorphic computing. In *International Joint Conference on Neural Networks*, Vancouver. 28

[12] Dean, M. E., Schuman, C. D., and Birdwell, J. D. (2014). Dynamic adaptive neural network array. In *13th International Conference on Unconventional Computation and Natural Computation (UCNC)*, pages 129–141, London, ON. Springer. 3

[13] dheeru, d. and karra taniskidou, e. (2017). uci machine learning repository. 53

[14] Disney, A., Plank, J. S., and Dean, M. (2018). Four simulators of the danna neuromorphic computing architecture. In *Proceedings of the International Conference on Neuromorphic Systems*. ACM. 26, 40

[15] Disney, A., Reynolds, J., Schuman, C. D., Klibisz, A., Young, A., and Plank, J. S. (2016). DANNA: A neuromorphic software ecosystem. *Biologically Inspired Cognitive Architectures*, 9:49–56. 26, 28

[16] Eckhart, P. J. (2017). Tiled DANNA: Dynamic adaptive neural network array scaled across multiple chips. Masters Thesis, University of Tennessee. 27, 34

[17] Gewaltig, M.-O. and Diesmann, M. (2007). Nest (neural simulation tool). *Scholarpedia*, 2(4):1430. 23

[18] Goodman, D. and Brette, R. (2008). Brian: a simulator for spiking neural networks in python. *Frontiers in neuroinformatics*, 2. 23

[19] Imam, N., Akopyan, F., Arthur, J., Merolla, P., Manohar, R., and Modha, D. S. (2012). A digital neurosynaptic core using event-driven qdi circuits. In *Asynchronous Circuits and Systems (ASYNC), 2012 18th IEEE International Symposium on*, pages 25–32. IEEE. 24

[20] Katsikis, G., Cybulski, J. S., and Prakash, M. (2015). Synchronous universal droplet logic and control. *Nature Physics*, 11(7):588–596. 1

[21] Knuth, D. E. (2007). Seminumerical algorithms. 19

[22] Lau, F.-L. A. and Fischer, S. (2017). Embedding space-constrained quantum-dot cellular automata in three-dimensional tile-based self-assembly systems. In *Proceedings of the 4th ACM International Conference on Nanoscale Computing and Communication*, page 22. ACM. 1

[23] libusb (2018). libusb. 32

[24] Maass, W., Natschläger, T., and Markram, H. (2002). Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural computation*, 14(11):2531–2560. 18

[25] Markram, H. (2012). The human brain project. *Scientific American*, 306(6):50–55. 1

[26] Masquelier, T., Guyonneau, R., and Thorpe, S. J. (2008). Spike timing dependent plasticity finds the start of repeating patterns in continuous spike trains. *PloS one*, 3(1):e1377. 57, 58

[27] Merolla, P., Arthur, J., Akopyan, F., Imam, N., Manohar, R., and Modha, D. S. (2011). A digital neurosynaptic core using embedded crossbar memory with 45pj per spike in 45nm. In *Custom Integrated Circuits Conference (CICC), 2011 IEEE*, pages 1–4. IEEE. 24

[28] Mitchell, J. P. (2018). Danna2: Dynamic adaptive neural networks arrays. Master's thesis. 60

[29] Mitchell, J. P., Bruer, G., Dean, M. E., Plank, J. S., Rose, G. S., and Schuman, C. D. (2017). NeoN: Neuromorphic control for autonomous robotic navigation. In *IEEE 5th International Symposium on Robotics and Intelligent Sensors*, pages 136–142, Ottawa, Canada. 38

[30] Monroe, D. (2014). Neuromorphic computing gets ready for the (really) big time. *Communications of the ACM*, 57(6):13–15. 1

[31] Nageswaran, J. M., Dutt, N., Krichmar, J. L., Nicolau, A., and Veidenbaum, A. V. (2009). A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors. *Neural networks*, 22(5-6):791–800. 24

[32] Nvidia (2018). Cuda c programming guide. 40

[33] Plagge, M., Carothers, C. D., and Gonsiorowski, E. (2016). Nemo: A massively parallel discrete-event simulation model for neuromorphic architectures. In *Proceedings of the 2016 annual ACM Conference on SIGSIM Principles of Advanced Discrete Simulation*, pages 233–244. ACM. 23

[34] Plank, J. S., Rose, G. S., Dean, M. E., Schuman, C. D., and Cady, N. C. (2017). A unified hardware/software co-design framework for neuromorphic computing devices and applications. In *IEEE International Conference on Rebooting Computing (ICRC 2017)*, Washington, DC. 2

[35] Preissl, R., Wong, T. M., Datta, P., Flickner, M., Singh, R., Esser, S. K., Risk, W. P., Simon, H. D., and Modha, D. S. (2012). Compass: A scalable simulator for an architecture for cognitive computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 54. IEEE Computer Society Press. 24

[36] Rast, A. D., Jin, X., Galluppi, F., Plana, L. A., Patterson, C., and Furber, S. (2010). Scalable event-driven native parallel processing: the spinnaker neuromimetic system. In

Proceedings of the 7th ACM international conference on Computing frontiers, pages 21–30. ACM. 25

[37] Schemmel, J., Briiderle, D., Griibl, A., Hock, M., Meier, K., and Millner, S. (2010). A wafer-scale neuromorphic hardware system for large-scale neural modeling. In *Circuits and systems (ISCAS), proceedings of 2010 IEEE international symposium on*, pages 1947–1950. IEEE. 25

[38] Schuman, C. D. (2015). *Neuroscience-Inspired Dynamic Architectures*. PhD thesis, University of Tennessee. 50, 59

[39] Schuman, C. D., Disney, A., and Reynolds, J. (2015). Dynamic adaptive neural network arrays: A neuromorphic architecture. In *Workshop on Machine Learning in HPC Environments*, Austin, TX. ACM. 28

[40] Schuman, C. D., Disney, A., Singh, S. P., Bruer, G., Mitchell, J. P., Klibisz, A., and Plank, J. S. (2016a). Parallel evolutionary optimization for neuromorphic network training. In *Machine Learning in HPC Environments, Supercomputing 2016*, Salt Lake City. 27, 28

[41] Schuman, C. D., Plank, J. S., Disney, A., and Reynolds, J. (2016b). An evolutionary optimization framework for neural networks and neuromorphic architectures. In *International Joint Conference on Neural Networks*, Vancouver. 27

[42] Schuman, C. D., Potok, T. E., Patton, R. M., Birdwell, J. D., Dean, M. E., Rose, G. S., and Plank, J. S. (2017). A survey of neuromorphic computing and neural networks in hardware. arXiv:1705.06963. 1

[43] Seo, J.-s., Brezzo, B., Liu, Y., Parker, B. D., Esser, S. K., Montoye, R. K., Rajendran, B., Tierno, J. A., Chang, L., Modha, D. S., et al. (2011). A 45nm cmos neuromorphic chip with a scalable architecture for learning in networks of spiking neurons. In *CICC*, pages 1–4. 24

[44] Shalf, J. M. and Leland, R. (2015). Computing beyond moore's law. *Computer*, 48(12):14–23. 1

[45] Stojanovic, M. N., Stefanovic, D., and Rudchenko, S. (2014). Exercises in molecular computing. *Accounts of chemical research*, 47(6):1845–1852. 1

[46] Suzuno, K., Ueyama, D., Branicki, M., Toth, R., Braun, A., and Lagzi, I. (2014). Maze solving using fatty acid chemistry. *Langmuir*, 30(31):9251–9255. 1

[47] Waldrop, M. M. (2016). The chips are down for moore's law. *Nature*, 530(7589):144–147. 1

[48] Willis, J. C. (2015). Middleware and services for dynamic adaptive neural network arrays. Master's thesis. 23

[49] Young, A. R., Dean, M. E., Plank, J. S., Rose, G. S., and Schuman, C. D. (2018). Neuromorphic array communications controller to support large-scale neural networks. In *IJCNN: The International Joint Conference on Neural Networks*, Rio, Brazil. 26, 27, 32, 34

Vita

Adam Disney was born on August 12th, 1982 in Knoxville, TN. After dropping out of high school and earning a GED in 2000, he returned to school at Pellissippi State Community College in the spring of 2009. He then transferred to The University of Tennessee in Knoxville where he earned his Bachelors in Computer Science summa cum laude in the summer of 2013. He continued at The University of Tennessee to earn his Doctorate in Computer Science in the summer of 2018. The opportunity to return to school greatly enriched his life, and he hopes in the future to return the favor by being a teacher himself.