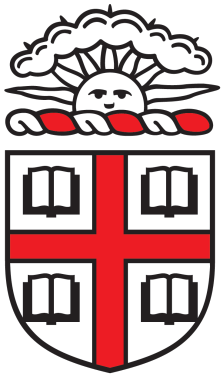
BROWN UNIVERSITY



UNDERGRADUATE HONORS THESIS

Approximate Spike Encoding Neural Networks: ASPENN

Author:
Andrew DUNCOMBE

Advisor and First Reader:
Sherief REDA

Second Reader:
Jacob ROSENSTEIN

A thesis submitted in fulfillment of the requirements for Honors in Computer Engineering

in the

Scalable Energy-Efficient Computing Laboratory (SCALE)
School of Engineering

March 30, 2021

BROWN UNIVERSITY

Abstract

Approximate Spike Encoding Neural Networks: ASPENN

by Andrew DUNCOMBE

Spiking Neural Networks (SNNs) are a neural network architecture that encodes information as a time series of binary pulses, or spikes, in order to perform low-power neural tasks. This work presents the Approximate Spike Encoding Neural Network (ASPENN), a SNN circuit design that uses a novel weight accumulator unit with modular approximate arithmetic components. This circuit is designed using Verilog and synthesized using the Genus Synthesis Tool. This system is tested using the MNIST handwriting digit dataset and analyzed for classification accuracy and design metrics such as chip area and power consumption. ASPENN demonstrates improved computation performance over alternative neuron circuit designs. The circuit design and code has been made open-source, allowing future researchers access to a complete SNN circuit design for use in later works.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

There is a growing desire in a number of fields to implement neural network algorithms on embedded hardware. The explosion of digital data available in the world has greatly increased the need for high-performance tools to process and analyze that data. Modern NNs have achieved remarkable performance on various analysis tasks, and embedded systems are already integrating NNs into a number of developing technologies, from IoT and wearable devices, to medical and industrial scanners.

SNNs are a neuromorphic-inspired architecture that aims to emulate how the human brain processes information. In a SNN, information is passed between neurons using series of binary pulses, or spikes, instead of standard numerical information. This allows SNN computation to avoid the use of expensive multiplication units altogether and instead use simple addition and thresholding operations. SNNs are an active area of research, and have demonstrated cutting edge performance on the MNIST dataset [2], and a variety of other benchmarks [1]. SNNs are not without problems, however. Spiking data is inherently event driven with irregular memory access patterns, which makes implementation on traditional computation systems, such as CPUs and GPUs, difficult. SNNs commonly need to be larger than traditional networks to achieve a similar level of performance, which in turn often results in a greater number of computations, even if each individual computation is simpler.

In this work, we propose the Approximate Spike Encoding Neural Network (ASPENN), a SNN design that leverages the irregular memory access patterns to improve energy-efficiency. This design focuses on improving the core neuron update computation with a modular, multi-stage weight summation unit that is well suited for spiking computation. This unit combines low-power counter compression and block-save adder structures to perform efficient multi-operand addition. We also present an approximation method to further the improve the area and energy requirements of this circuit. This unit is integrated into a complete network circuit that routes spiking information in order to take full advantage of the efficient adder design. This design is verified on the MNIST handwriting digit dataset and achieves 98.42% accuracy, near the cutting edge for this dataset. This performance is achieved by consuming only 2.5μJ of energy per image on average.

Chapter 2

Background

2.1 Spiking Neural Networks (SNN)

Spiking neural networks are a variant of traditional neural networks that take inspiration from how the brain processes information to perform energy-efficient computation. The primary difference is that instead of a neuron always passing along information, it only passes along information when that neuron is activated. This behavior requires a concept of time within the network model. Each neuron, therefore, produces a spike train of neuron activations which are used to perform the network operation.

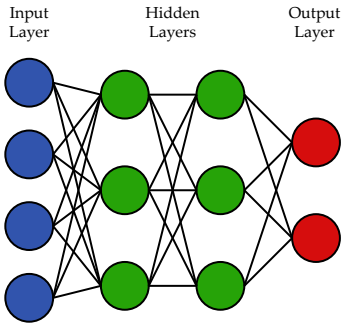


FIGURE 2.1: Feedforward Neural Network Structure

While there are neuron-level differences between traditional artificial neural networks and spiking networks, many structural concepts are shared between them. The feedforward network is the simplest neural network model, and this structure can be easily implemented using a SNN. In a feedforward network with fully connected layers, each pairing of neurons between adjacent layers is associated with some trained weight, and each neuron uses the incoming weights to compute its activation. With standard networks, every neuron from the previous layer contributes the corresponding weight for each neuron update, while with SNNs, only those weights coming from spiking neurons are used. More complex architectures are possible to implement, such as convolutional neural networks, but this work will focus on the basic feedforward structure.

2.2 SNN Neuron Models

The core elements of any neural network are the individual neurons that process data and the synapses that connect neurons. There are a variety of neuron architectures that are commonly used in SNN circuit designs. One such neuron model, the Integrate and Fire (IF) model, is particularly interesting due to its simple design and low-cost implementation. Much of the language for describing SNN behavior is borrowed from neuroscience. Each IF neuron contains a tracking value, called the membrane potential voltage. In each simulated time-step, all synaptic weights leading from spiking neurons in the previous layer to a single neuron in the receiving layer are summed to form

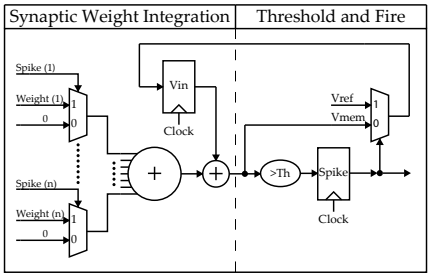


FIGURE 2.2: Integrate-and-Fire Neuron Model

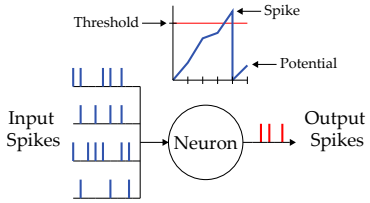


FIGURE 2.3: Spiking Neuron Behavior

the synaptic impulse voltage. This impulse is then added to the neuron's potential, and this new voltage is compared against some threshold value. If the threshold is exceeded, the neuron fires an output spike and resets the membrane potential. If the threshold is not exceeded, the membrane potential is set to the sum of the old potential and the current impulse. This operation is described mathematically in Equation 2.1, where $v_o[t]$ is membrane potential, $v_{imp}[t]$ is the calculated impulse voltage, and v_{ref} is the reference threshold.

$$v_o[t] = \begin{cases} 0 & v_o[t-1] + v_{imp}[t] \geq v_{ref} \\ v_o[t-1] + v_{imp}[t] & \text{otherwise} \end{cases} \tag{2.1}$$

More complex neuron models have been proposed, mostly taking inspiration from various characteristics of a brain's biological neuron behavior. One such model is the Leaky Integrate and Fire neuron (LIF), which subtracts some value from the neuron's potential after each timestep, simulating the gradual degradation of neuron's potential after long periods of inactivity. Another method introduces a refractory period in which a neuron cannot be updated from some small number of timesteps after it has fired a spike. More complex models that better emulate biological neuron behavior have been proposed, although most SNN implementations favor the simpler models due to their more efficient structure and their relative ease of training. This work focuses its attention on the non-leaky IF neuron, although the resulting design could easily be modified to implement these more complex models with relatively simple changes.

2.3 Multi-Input Addition

Multi-input addition circuits are a specialized category of arithmetic that aims to improve performance when adding three or more operands. Thabah *et al.* demonstrate experimentally that Wallace Trees are generally the most efficient multi-input design [15]. Wallace Trees use a network of carry-save adders (CSA) to perform multi-input addition. An example for computing the sum of six inputs is shown in Figure 2.4. Carry-save adders use a chain of full adders, but instead of sending each adder unit's carry signal to the next adder in the chain, the CSA outputs both the sum and carry signals for every full adder unit. The sum and carry signals can then be used by other CSA units. At the end of a CSA network, a single carry-propagate addition is necessary to derive the final result. If a network of CSAs is constructed carefully, then many

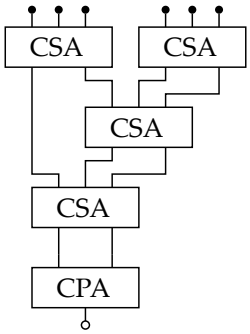


FIGURE 2.4: Wallace Tree with 6 Inputs

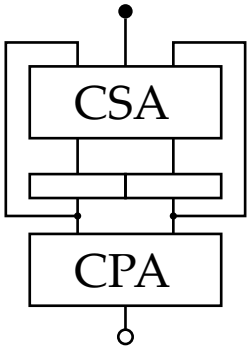


FIGURE 2.5: Carry Save Accumulator

additions can be performed very quickly and efficiently. Wallace Trees do have issues, however. Wallace trees are designed with a fixed number of inputs that are all present at the same time, and the size of this system can become prohibitively expensive when used on a large number of inputs. The SNN accumulate operation has an unknown number of addition operands, which is difficult to implement with Wallace Trees. Furthermore, since multiple neuron computation units will exist side-by-side, making the neuron unit compact is an important concern. These reasons make Wallace trees an unattractive option for performing the SNN accumulate operation.

Instead, a multi-input design that receives inputs sequentially would be better suited for the spiking behavior of SNNs. This would allow an arbitrary number of inputs to be summed together with a compact and efficient circuit. The most common accumulator design is a Carry-Save Accumulator [5]. This design feeds an input into a carry-save adder and latches the outputs to be fed back into the system. Once all inputs have been processed, a CPA unit is applied to derive the final summation result. A diagram showing this operation is shown in Figure 2.5. This system is compact but consumes a lot of energy when applied over many inputs. Input compression is an alternative method to accumulate sequential information [18]. This method requires more area but is more energy efficient. Compression circuits count the number of ones in an input set [11]. These counts are then used by an arithmetic circuit to finish the computation. Counter compression is a low-cost operation, much lower typically than the CSA operation described above. While the second stage arithmetic circuit is typically more expensive, this cost is amortized over many inputs.

2.4 Approximate Neural Network Circuits

Neural networks have been shown to be resilient to computational approximations [17]. This has led to a good deal of research focused on building approximate neural network circuits. These projects can be divided into two categories based on the method used: 1) network-level approximations, and 2) neuron-level approximations. Network-level approximations focus on reducing the neural network to a simpler form [17], which hopefully reduces the number of computations the network performs. Neuron level methods focus on using approximate computing circuits in the actual

neuron computation [6], [9], [7]. These methods aim to introduce errors at the expense of accuracy to improve performance of each computation.

Various works in recent years have attempted to apply network level approximation techniques to spiking neural networks. Some works have directly applied network reduction and pruning strategies to SNNs [14], [12]. Other works have attempted to use the unique computational behavior of SNNs, particularly their time-dependent nature, to dynamically adjust the network during inference [13]. To the best of our knowledge, no previous works have attempted to apply neuron level approximate methods to SNNs beyond simple weight quantization.

2.5 Spiking Neural Networks Circuits

Up until recently, there have been significant hurdles facing wide-scale implementation of spiking neural networks. The most significant hurdle is the difficulties with training SNNs, with often large performance divide between SNN implementations and their more conventional counterparts. In recent years, great strides have been made towards reducing this divide, and SNNs have met, and sometimes even exceeded, performance metrics on several common benchmarks. Several specialized hardware systems have been developed to assist researchers with training and testing of spiking algorithms [1], [3]. These systems have demonstrated impressive performance gains on a variety of common benchmarks, both in terms of accuracy and power consumption.

Other projects have made progress towards developing small-scale and highly-optimized SNN processor chips [19, 13]. These proposed designs focus on providing spiking networks for real-world embedded systems and edge devices. While less applicable to a wider range of problems than the industrial scale implementations popular for research of spiking algorithms and training methods, these small-scale accelerators instead promise to perform a subset of important operations very efficiently. These new systems will play a vital role in pushing spiking neural network implementations into the mainstream.

Chapter 3

Spiking Neuron Design

A great deal of research effort in developing spiking neural network circuits has been devoted to training and optimizing these systems on the network level. Our ASPENN design takes a novel approach by focusing on the lowest level of a SNN, the spiking neuron itself. This section will describe the neuron circuit implementation and in particular the weight accumulation operation itself. It will also present a method for approximating the neuron's accumulate operation.

3.1 Integrate and Fire Neuron

The simplest and most common spiking neuron model is the integrate-and-fire neuron (IF). There are two phases to performing an IF neuron update. The first phase starts by accessing the synaptic weights of all neurons in the previous layer that have fired in the current time step. It then sums these weights together to form the synaptic impulse. In the second phase, this impulse is added to the neuron's current membrane potential and compared against some threshold value. If the threshold is exceeded, the neuron fires and resets. If not, the summed value is latched as the neuron's new membrane potential.

3.2 Weight Accumulation Unit Design

The core operation of an IF neuron is weight accumulation. This operation is required to sequentially add together a variable number of signed, fixed point weights. Fixed point binary representations are commonly used in low-power computations because they can be used to represent fractional numbers without resorting to expensive floating-point computation. Fixed point numbers also make it easy to use 2-s complement conversions to represent signed numbers. ASPENN represents the value, v_b, of a binary number, b, of size k using Equation 3.1. In this equation, b[1] is the least-significant bit.

$$v_b = -2^{k-1} * b[k-1] + \sum_{n=0}^{k-2} 2^n * b[n] \tag{3.1}$$

To perform weight accumulation, ASPENN proposes an energy-efficient design that combines parallel counter compression with adder tree networks [18]. The logic behind choosing this design over a more conventional CSA accumulator deals with the

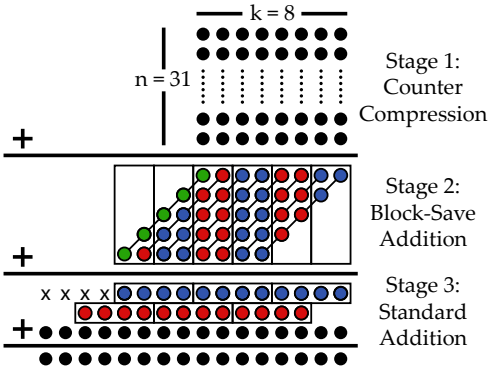


FIGURE 3.1: Weight Summation Unit ($c = 5$): Inputs/Outputs in BLACK. Internal signals in RED, GREEN, and BLUE.

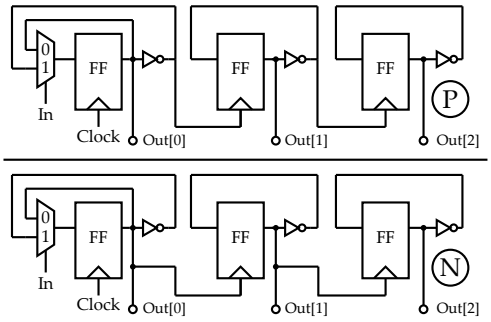


FIGURE 3.2: Counter Chain for c = 3: P is a positive (+1) counter while N is a negative counter (-1).

energy cost per input. While a CSA accumulator uses a moderate amount of energy for every new input, ASPENN's design performs a low-cost counting operation for every new input, and then performs only a few, high-cost operations throughout the computation. These costs are amortized over the many inputs used during a computation and reduce the overall energy consumption significantly. This structure decomposes into three modular stages: 1) Counter Compression, 2) Block-Save Addition, and 3) CPA addition. Figure 3.1 presents a dot-diagram of the proposed addition unit, where each dot represents a single bit of binary data. A more detailed description of each stage is given in the following sections.

3.2.1 Stage 1: Counter Compression

The first stage, Counter Compression, simply counts the number of ones in each bit position for a sequence of inputs. Counter compression is an extremely low-power operation, which benefits greatly from the sequential arrival of synaptic weights. For an input with a width of k bits, this requires k counter compressors. ASPENN implements these parallel counters using short chains of resettable flip-flops [16]. Each of these chains use a fixed number of flip-flops, c. There are two flavors of counter chain used, a positive and a negative counter, which are shown in Figure 3.2. Each of the k bit positions requires a separate counter, so this stage uses $k * c$ flip-flops. In Figure 3.1, the results of each counter is represented as a diagonal line of dots. The $k - 1$ least significant bit positions are counted using a positive counter, but the most significant bit position requires a negative counter. This is due to the signed binary representation expressed in Equation 3.1, where the MSB has a negative contribution to the number's total value. This distinction is represented by the GREEN dots in 3.1.

3.2.2 Stage 2: Block-Save Addition

The second stage, Block-Save Addition, performs a set of small-scale additions on the results of the Counter Compression stages. Each counter result is skewed and arranged by significance, so as to accurately perform the necessary addition operation. This arrangement can be seen in Figure 3.1. These skewed counter results are partitioned into

blocks, and then the sum of each block is computed in parallel. The exact dimensions of these blocks are carefully chosen to ensure that the resulting pieces form a disjoint partition of the block-sum results [18]. Each block is colored with alternating RED and BLUE colors in Figure 3.1, which correspond to the similarly colored inputs to the CPA Addition. Each block in this stage runs in parallel and has a modular design that is well suited for the approximation methods outlined in a later section.

3.2.3 Stage 3: CPA Addition

These block sum results, as well as the neuron's current membrane potential, are added together in this stage to produce an updated membrane potential. Additionally, the sign bit from the MSB counter is extended and incorporated with this addition. These extended bits are marked with an $\times$ in Figure 3. This stage first applies a carry-save layer to reduce the three inputs down to two, and then applies a standard CPA operation. This stage contributes the most to the delay of this system, as it is the only stage that cannot be parallelized effectively. More advanced adder designs could address the latency issue, but often at the expense of additional hardware and energy costs, so ASPENN uses a simple ripple-carry design.

3.2.4 Weight Accumulation Operation

There is a significant issue with this design that must be addressed. In the first stage, the counter compressors have a fixed size, which limits the amount of data they can count before the counter overflows and an error occurs. The capacity for a counter of size c is given by:

$$capacity = 2^c - 1 \tag{3.2}$$

If $c = 5$, for example, then the counters can be filled with at most 31 inputs while still guaranteeing an exact result. The actual number of inputs fed into this system is determined by the number of neurons that have fired in the previous layer. The number of inputs vary wildly from operation to operation. Having a capacity large enough to process the maximum possible number of inputs directly is prohibitively expensive and will mostly be wasted as most operations will not require the larger capacity. This system can be adapted to accumulate an arbitrary number of inputs by updating itself at regular intervals. This update activates Stages 2 and 3 and stores the result to be fed back into the system as the additional input to Stage 3 discussed previously. The Stage 1 counters are then reset and can continue counting without fear of exceeding capacity. This operation continues until the neuron update has summed all necessary weights. Once all spikes for a given timestep have been processed to form the synaptic impulse, this value is compared against some threshold value, and a spike fires if that threshold is exceeded. If a neuron fires a spike, the neuron's potential is reset, if not, then the neuron's potential is updated for the next timestep.

3.3 Neuron Approximation Methods

Approximate computing is a category of optimization techniques that seek to improve some design metric, such as area or energy consumption, at the expense of accuracy.

This technique has proven effective for improving traditional neural networks [6, 9]. Neural networks have been shown to be resilient to computational approximation [17], and SNNs show a similar robustness to computational approximations [14].

SNNs are a promising target for approximate computing optimizations. Since spikes are generated based on thresholding the result of weight accumulation, an approximate computation can produce an equivalent spiking pattern to that of an exact computation if the errors are not too severe. Neuron outputs are time dependent as well, so minor time-shifts in internal spike trains may still produce correct final results.

The weight accumulator design described above performs exact, multi-input addition. Since this circuit uses a highly modular design, it is easy to incorporate approximate components when appropriate. ASPENN incorporates approximation techniques in the Block-Save Addition and the Standard Addition Stages of the multi-input adder described previously. These techniques are described in detail in the following sections. ASPENN's weight accumulator unit has a further advantage in that an approximation applied to the computation amortized any error that occurs over all inputs fed into the system. This reduces the expected error per input significantly.

The first method applied to ASPENN is aimed at developing a technique to convert exact adder trees into approximate adder trees and apply this technique to the block-save operation of Stage 2. The following sections will discuss how to generate optimal, exact adder trees of arbitrary size, various properties of these trees, and finally how to convert exact trees into approximate trees with lower power consumption. The second method applies a standard library of 2-input approximate adders to Stage 3 of the weight accumulator unit.

3.3.1 Exact Adder Trees Generation

Adder trees are circuits comprised of full and half adders that can be used to sum together inputs of arbitrary number and dimension. These trees can be thought of as generalizations of standard addition circuits, which are conventionally limited to two inputs of fixed size. An example of an adder tree for adding 5 inputs of size two is given in Figure 3.3. In this and all other adder tree diagrams in this report, inputs are marked as black dots, outputs are marked as white dots, and the number of dashes through a signal marks the significance of that signal. There are a variety of methods to generate adder trees, but the method generally considered optimal is the Three Greedy-Approach [11]. This method greedily connects signals of the same significance level in groups of three or two, depending on availability. The choice of selection greedily picks the signals with the minimum delay from the input set. Pseudocode for generating optimal, exact adder trees using the Three-Greedy Approach is given in Algorithm 1. Trees generated using this method have the smallest possible size and depth. Size is defined as the total number of adders. Depth is defined as the maximum number of adders on a path between an input and an output. For purposes of determining size and depth, half and full adders both count as 1 adder

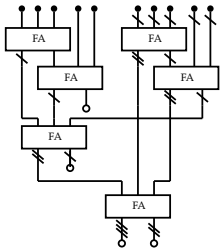


FIGURE 3.3: Exact 5x2 Block Adders. Size: 6, Depth: 4

unit. More fine-grained approaches can simulate the real-world differences between these various circuit components for increased performance, but this project we will use the simpler model for its ease of design.

Algorithm 1 Optimal Adder Tree Generation with the Three Greedy-Approach

DEFINE: $halfAdder \leftarrow \{in_X, in_Y, out_{SUM}, out_{CARRY}\}$
DEFINE: $fullAdder \leftarrow \{in_X, in_Y, in_Z, out_{SUM}, out_{CARRY}\}$
INPUT: signals ← initial list of signal queues
adders ← empty list
$index \leftarrow 0$
while index < length of signals **do**
  if length of signals[index] = 2 **then**
    $Adder_H \leftarrow$ new halfAdder
    $Adder_H.in_X \leftarrow$ dequeue signals[index][0]
    $Adder_H.in_Y \leftarrow$ dequeue signals[index][1]
    $Adder_H.out_{SUM} \rightarrow$ signals[index]
    $Adder_H.out_{CARRY} \rightarrow$ signals[index+1]
    $Adder_H \rightarrow$ adders
  else
    $Adder_F \leftarrow$ new fullAdder
    $Adder_F.in_X \leftarrow$ dequeue signals[index][0]
    $Adder_F.in_Y \leftarrow$ dequeue signals[index][1]
    $Adder_F.in_Z \leftarrow$ dequeue signals[index][2]
    $Adder_F.out_{SUM} \rightarrow$ signals[index]
    $Adder_F.out_{CARRY} \rightarrow$ signals[index+1]
    $Adder_F \rightarrow$ adders
  end if
  if length of signals[index] = 1 **then**
    index += 1
  end if
end while

Before moving on to describing a method to convert exact adder trees into approximate trees, it is helpful to understand some properties of the signals found within these circuits. We reasonably assume random inputs where each input bit has an equal chance of being either a zero or one. For a single full or half adder unit given random inputs, the output bits each have an equal probability of begin zero or one, although these two probabilities are not independent. Since an adder tree is entirely comprised of full and half adder units, and both the inputs and outputs of these units have the same distribution, we observe that all signals in an adder tree have a 50% chance of being either zero or one. Therefore, fixing any of these signals to a specific value has a 50% chance of introducing an error with magnitude equal to the significance of that signal. The sign of that error is determined by what it is set to: setting the signal's value to 1 will cause a positive error, while a value of 0 will introduce a negative error. This signal fixing can be thought of as making a guess as to the result of a signal before it is computed.

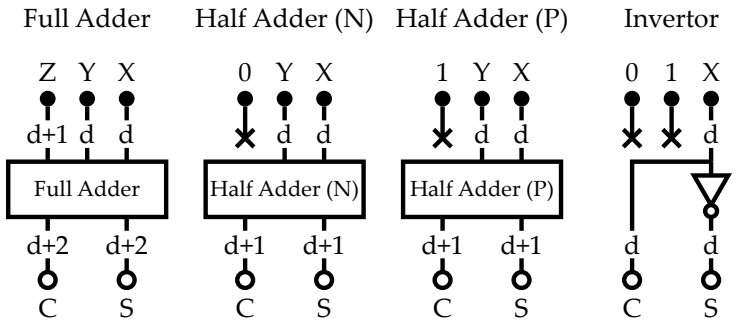


FIGURE 3.4: Full Adder Reduction Diagrams: Full adder with 0, 1, and 2 input signals fixed. Depths of input and output signals based off some depth, "d".

3.3.2 Approximate Adder Tree Conversion

Exact adder trees can be converted into approximate adder trees by performing a sequence of signal fixations. Converting to an approximate adder tree takes two steps, choosing signals to fix, and choosing what values to fix those signals to. Each signal chosen for fixation produces an error with a magnitude determined by the significance of that signal. The choice of the value, either 0 or 1, determines the sign of the error. Any fixed signal will create an approximation, but only by being strategic can this approximation be used to improve performance. Performance can be improved by reducing either the size or the depth of the tree, both of which correspond directly to improvements in energy consumption. For picking signals to fix, we have derived two useful heuristics: The Depth-Mismatch Heuristic, and the ± 1 Heuristic.

The Depth-Mismatch Heuristic can be applied whenever the input signals to a specific adder unit have mismatched depths. First, it is important to note that the output signal of an adder unit is simply the maximum input signal depth plus 1. If the input signal with maximum depth is fixed to some value, then the output signal's depth can be reduced by the difference between the depths of the fixed signal and the next deepest input signal. For example, assume an adder unit has three input signals with depths {d, d, d+1}, then the output signals have depth $(d+1)+1 = d+2$. Now, we fix the third signal, thereby removing it from the depth calculation. The new output signal depth is therefore only $(d)+1 = d+1$. This operation is illustrated in Figure 3.4. Fixing one of a full adder's inputs has the additional benefit of reducing that unit to a simpler half adder, although the choice of what value the signal is fixed to will change the type of half adder it is reduced to. If the input signal is fixed to a 0, corresponding to a negative error, the full adder reduces to a negative-type half adder. Fixing the input signal to a 1 causes a reduction to a positive-type half adder. Truth tables for these two types are shown in Table 3.1. This method can also be applied to half adders with input depth mismatches, in which case the circuit reduces to a single inverter. This rule is helpful when depth mismatches exist and can directly reduce the depth of an adder tree by fixing only one signal.

When there are no adder units with depth mismatches, then the ± 1 Heuristic can be applied instead. This heuristic can only be applied to full adders, and fixes two input signals to opposite values. This operation reduces a full adder down to a single

inverter, as shown if Figure 3.4. As inverters are typically considered negligible circuit components, especially when compared against full and half adders, this reduction essentially removes the full adder's contribution to both the size and complexity of the adder tree. While this method requires twice as many fixations as the Depth-Mismatch Heuristic, it reduces the complexity of the original circuit to a greater degree. To understand why the input signals must be fixed to opposite values, it is best to examine the logical functions of these reductions. Table 3.1) shows the truth tables for a variety of full adder reductions. For convenience, assume that all input combinations are equally likely. From this table, it is clear that the probability that an error occurs for the inverter reduction is %50, the same as for the half adder reductions. Interestingly, however, is that the expected error of this reduction is zero, while the expected error of the half adder reductions is ± 0.5, depending on the type of half adder used. Notably, the variance is higher for the inverter than the half adders.

TABLE 3.1: Full Adder Reductions: Each reduction column contains the carry and sum output values (C and S), as well as the error (E) compared against the full adder result.

Inputs			$FA(X,Y,Z)$		$HA_N(X,Y)$			$HA_P(X,Y)$			$INV(X)$		
Z	Y	X	C	S	C	S	E	C	S	E	C	S	E
0	0	0	0	0	0	0	0	0	1	+1	0	1	+1
0	0	1	0	1	0	1	0	1	0	+1	1	0	+1
0	1	0	0	1	0	1	0	1	0	+1	0	1	0
0	1	1	1	0	1	0	0	1	1	+1	1	0	0
1	0	0	0	1	0	0	-1	0	1	0	0	1	0
1	0	1	1	0	0	1	-1	1	0	0	1	0	0
1	1	0	1	0	0	1	-1	1	0	0	0	1	-1
1	1	1	1	1	1	0	-1	1	1	0	1	0	-1

Once a set of signals have been selected, the errors introduced by these fixations can be represented as a sequence of magnitudes. Selecting values for these signals is an equivalent problem to assigning a sign to every value in this magnitude sequence. The goal of this task is to reduce the absolute value of the expected error generated by this set of signal fixations. By assuming that every error occurs with an equal 50% probability, then the expected error of a set of n fixations, $E[s_f]$, is given by equation **??**. We have developed two heuristics for value selection: Alternating Signs, and Most Significant Sign.

$$E[s_f] = 0.5 * \sum_{k=1}^{n} s_f[k] \tag{3.3}$$

The Alternating Signs Heuristic is best applied when there exist two fixed signals of the same magnitude. This method fixes these two signals to alternating signs, one positive and one negative. By choosing opposite signs, the induced errors have a chance of cancelling out, thereby producing a correct final answer even with approximate components. This method can also reduce the maximum error, as it prevents the

magnitudes from compounding if both signals have an error. It is this principle which is applied to good effect by the ± 1 Rule.

The Most Significant Sign (MSS) Heuristic is best applied when there exists a strictly increasing sequence of fixation magnitudes. An example magnitude set that exhibits this property is $\{2, 4, 8, 16\}$. Applying this rule will fix all signals to the same sign, except the most significant signal which will have an opposite sign. This method minimizes the magnitude of the expected error induced by that set of fixations. For example, applying the MSS method to the sequence shown above will yield a signed magnitude set of $\{+2, +4, +8, -16\}$. This assignment produces an expected error of only -1, which is minimal for this set. The expected error can be reduced even further if the MSS method is expanded slightly by adding a repeated fixation at the least significant signal. So if the fixation set instead has magnitudes $\{2, 2, 4, 8, 16\}$, then the extended MSS method will produce the signed errors $\{+2, +2, +4, +8, -16\}$, which has an expected error of 0.

3.3.3 Approximate Weight Accumulator unit

Each block in Stage 2 of the weight accumulator unit can be converted into an exact adder tree. The tree structure for a completely full 5x2 block is shown in Figure 3.5. All possible Depth Mismatches are marked with colored circles. Also shown are two potential approximate conversions. The first approximate tree, Design #1, has had the Depth-Mismatch Rule applied twice, producing a fixation magnitude set of $\{1, 2\}$. The MSS method is applied to form a signed error set of $\{+1, -2\}$. In the second tree, Design #2, the Depth-Mismatch rule is applied a third time, to produce a fixation magnitude set of $\{1, 2, 4\}$ and a signed error set of $\{+1, +2, -4\}$. These designs reduce the depth of the exact 5x2 adder tree to 3 and 2 respectively. Both have an expected error of -0.5.

Since the results of these block-save adders are used to form pieces of the inputs to the Stage 3 addition, it is helpful to examine the induced errors of adjacent blocks in relation to one another. Suppose there are two adjacent blocks, A and B, arranged $\{A, B\}$. Let Block A be approximated with Design #1, and Block B be approximated

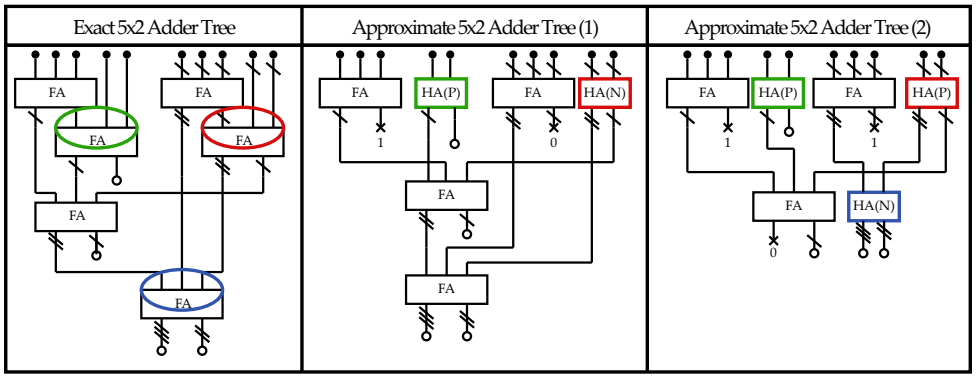


FIGURE 3.5: Approximate 5x2 Adder Conversion: (Left) Exact 5x2 adder. Depth Mismatches marked with RED, GREEN, and BLUE. Size=6, Depth=4. (Center) Approximate 5x2 Adder Design #1. Size=6, Depth=3. (Right) Aapproximate 5x2 Adder Design #2. Size=6, Depth=2.

with Design #2. Block A's significance values are shifted by the width of Block B, so Block A's fixation magnitude set will be multiplied by some power of 2. For instance, in the case where $c = 5$, the block width is 2, so Block A's magnitude set will be multiplied by $2^2 = 4$. Combining the fixation magnitude set from each block forms a new fixation magnitude set: $\{1, 2, 4, 4, 8\}$. This set can be divided into two sub-sequences, $\{1, 2\}$ and $\{4, 4, 8\}$. This first sequence matches the criterion for applying the MSS method, while the second matches the criterion for applying the Extended MSS method. After applying these rules to the sequences, this arrangement has a signed error set of $\{+1, -2, +4, +4, -8\}$, with an expected error of -0.5.

A second approximation method is applied during Stage 3. This method uses an approximate, 2-input adder to perform the final CPA addition. While there are many potential options for approximate adders, [6, 7], the EvoApprox8b Library [10] has the greatest number and range of options. This library has been generated by applying a genetic algorithm to an exact addition circuit, and the results are guaranteed to be near the Pareto frontier for approximate addition. While there were many options balancing accuracy, cost, and power consumption, we chose design 2JY, a 16-bit signed adder. This choice was made after a series of experiments showed that this design was better on a wider range of criterion when synthesized.

Chapter 4

Spiking Neural Network Design

The previous sections detailed the design of a single neuron, and methods to approximate that neuron's computation. This section will now go over how to construct a network of these neurons to perform a neural task. ASPENN is designed and trained to perform handwritten digit classification on the MNIST dataset, a common benchmark for neural network algorithms. This chapter will first describe the network design and discuss the training method used. It will then describe the network-level architecture design and memory management system.

4.1 SNN Design and Training

The Modified National Institute of Standards and Technology (MNIST) dataset is a large collection of images of handwritten digits [8]. Each image is encoded as a 28x28 pixel grid of greyscale values. Each image is classified as a single digit between $0-9$. ASPENN takes as input one of these images and classifies that image with a corresponding digit. Training spiking neural networks has traditionally been a challenging task facing SNN researchers, but great strides have been made in recent years. ASPENN is trained using a technique to convert trained ANNs into high-accuracy SNNs [2]. The first step in this training process is to train an ANN using standard backpropagation techniques on a network of rectified linear unit (ReLU) activations. ReLU activations can be thought of as approximations of a neuron's spiking rate, and so can be converted directly into a network of Integrate-and-Fire neurons.

The final step in this method is weight normalization, which constrains all the weights in a layer to limit the maximum spiking activity during a timestep. Normalization both improves accuracy and reduces convergence time. The normalization training stage guarantees two things about the network weights. The first guarantee is that each weight has a magnitude less than 1. The second is that every neuron in the entire network can use a spiking threshold equal to 1. Furthermore, SNNs have been shown to be robust to small precision weight quantization [14]. This allows ASPENN to use q0.8 signed fixed-point numbers for synaptic weights, and q8.8 numbers for the impulse and membrane potentials, with minimal impact on accuracy as opposed to floating point numbers. A fixed point structure described using Q notation is written as qA.B, where A is the number of integer bits, while B is the number of decimal bits. ASPENN's weights, therefore, require only 8 bits of data per weight. Importantly, ASPENN's training process does not account for the quantization, and instead trains

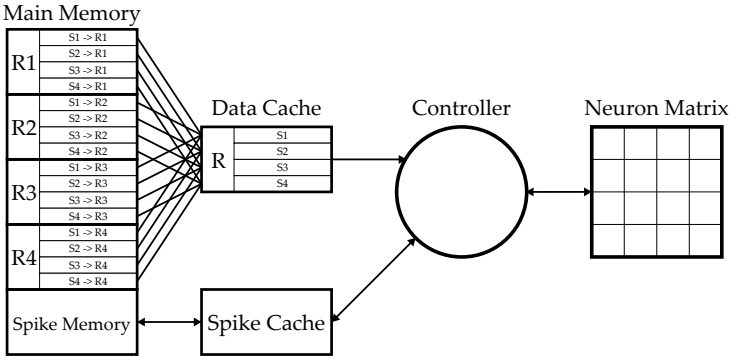


FIGURE 4.1: SNN Architecture (5 Components): (1) Main Memory stores weights and spikes. (2) Cache pre-loads sets of weights for easy access. (3) Controller directs memory flow. (4) Neuron Matrix contains the physical computation units. (5) Spike Cache passes spikes between the memory systems and the Control unit.

the weights purely using floating point numbers. The conversion to fixed point occurs after training.

ASPENN uses a fully connected feedforward architecture to perform MNIST classification. The network parameters used in this project has four layers. The input layer has 784 neurons, one for each pixel in an MNIST image. There are also two hidden layers with 1024 neurons each, and an output layer with 10 neurons, one for each possible digit class. This network is trained in Matlab using a tool developed by Diehl [2]. A network trained with these parameters can achieve classification of accuracy of 98.48%, which is near cutting edge for this dataset. The output class is determined by whichever output neuron has had the highest spiking activity during a classification. Each classification runs over 10 timesteps, which is experimentally determined to be approximately the point where the answer converges to its final value. The input layer's behavior is not based on any neuron update model, but rather is determined directly from the input image. First, a random number is generated for each pixel at each timestep and is compared against that pixel's true value. The corresponding input neuron fires a spike at that timestep if the randomly generated number exceeds the pixel value. This operation is applied to all input pixels for the duration of a classification and generates a set of spike trains for an input image. Generating input spike trains from images will not be handled directly by the ASPENN circuit presented here but will instead be generated during a preprocessing stage and made available for ASPENN at runtime.

4.2 SNN Circuit Architecture

ASPENN's network design is comprised of 5 components. (1) Main Memory, (2) Data Cache, (3) Controller, (4) Neuron Matrix, and (5) Spike Router. A diagram showing the network architecture is shown in Figure 4.1. The first component is the Main Memory. This unit stores all neuron weights, as well as a set of Spikes. The Data Cache periodically has sets of weights loaded from Main Memory and assists with high-speed data transfer by avoiding the need for frequent Main Memory reads and writes. The

Controller unit coordinates data flow from the memory system to the neuron matrix, as well accessing the Spike Cache during neuron computation. The Neuron Matrix contains all physical neuron computation units and is where the core neuron update operation takes place. The last component is the Spike Router, containing both the Spike Memory and the Spike Cache. Spikes are passed back and forth between the Cache and the Memory as needed. The following sections will go over each of these components in more detail.

4.2.1 Main Memory

The Main Memory system for ASPENN contains all weight memory data, as well as Spike Memory. The size requirements to store all the data is far too large for on-chip memory, but an off-chip DDRAM component can implement this functionality easily. For the purposes of this report, we will assume the use of the Micron Technologies 46V DDRAM chip (MT46V). This component is cheap, has significant memory storage, and is easy to use. ASPENN has network dimension {784, 1024, 1024, 10}, which requires nearly 1.86 million weights to be stored. As each weight has a size of 1 byte, this corresponds to roughly 1.8 MB of main memory storage. The MT46V, by comparison, has 256 MB of memory available.

The arrangement of weights inside the memory system is an important consideration in ASPENN's design. Figure 4.2 shows a small-scale feedforward neural network, and a memory system containing all corresponding synaptic weights. This network has a spiking layer (S), and a receiving layer (R). Notably, the weights in memory are grouped by receiving layer, not by spiking layer. As the physical neuron unit performs the operation of the receiving neuron, it makes the neuron computation more efficient to group data in this way. Since multiple physical neurons are being computed in parallel, it is convenient to group neurons in the receiving layer together. Each of these groups will have a size equal to the total number of neurons in the neuron matrix. This grouping divides the main memory into a set of weight blocks, where each block contains the weights leading from every spiking neuron in a layer to a set of receiving neurons. During data loading to the cache, it is a block which is loaded directly into the Data Cache. Sets of these blocks are pre-formed for each layer in the network and placed into the Main Memory for later use. Block sizes are determined by the size of the Neuron Matrix and the number of neurons in a given layer, and is expressed in Equation 4.1.

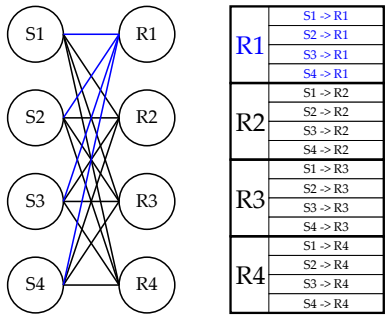


FIGURE 4.2: Main Memory: Weights grouped according to receiving neuron (R).

$$\text{block size\_i} = (\text{layer size\_i}) * (\text{neurons in matrix}) * (\text{weight size}) \tag{4.1}$$

4.2.2 Data Cache

ASPENN's Data Cache contains a block of weights loaded from Main Memory. This cache is implemented on-chip as scratchpad memory. This means that the Controller

unit controls all aspects of this cache directly. The Data Cache implements two primary operations: weight block loading, and weight reading. During each load operation, the entire cache is filled up by a block from memory. Once a block has been loaded, it remains in the cache until the corresponding neurons have completed all their computation and updates. The two important parameters for any cache design are the cache size, and the cache line width. The cache size is the same as the maximum block size, which is simply the block size, as determined in Equation 4.1, for the largest layer in the network.

For example, ASPENN's largest layer has 1024 neurons, and weights are 1 byte each. Assume that there are 64 physical neurons in the Neuron Matrix. These parameters yield a cache size of $1024 * 64 * 1 = 64kB$. During a weight reading operation, a single cache line will be read. A cache line size of one byte is needlessly inefficient, so instead several weights will be present on each cache line. ASPENN uses a cache line of four bytes, so four weights will be received during each read operation. Each of these weights will correspond to different receiving neurons, but the same spiking neuron. In the Neuron Matrix, this group of four neurons is referred to as a Neuron Tile.

4.2.3 ASPENN Controller

The ASPENN Controller coordinates the memory flow during classification. This system performs several operations relating to loading and reading from the cache, accessing the Spike Cache, and sending weight data to the Neuron Matrix. Five parameters are required to coordinate this memory flow. (1) The layer index determines which layer is being computed. ASPENN has three layers of simulated neurons, as the input layer is pre-computed. (2) The page index determines which group of receiving neurons is being computed. (3) The time index tells what time step the system is currently on. An MNIST classification runs for 10 timesteps on ASPENN. (4) The spike index indicates which spike is being read from the Spike Cache. (5) The tile index indicates which tile in the Neuron Matrix weights will be sent to. An algorithm that describes how these parameters are used is given in Algorithm 2.

At a high level, the controller goes through every neuron in every layer by loading and feeding the Neuron Matrix the proper weight data, as determined by the previous layer's spiking activity. Each page loaded into the Data Cache contains all weights required for computing the spike activity for every neuron in that block. As the neuron update requires many internal updates and refreshes, it is easiest to initialize a set of neurons in hardware and compute those neuron's activity for every timestep before moving onto the next set. This avoids the need for loading and reloading the neurons' membrane potential, saving both time and energy. It also allows for each page of weights to be loaded exactly once from memory, at the start of that set of neuron's computations. The Spike Cache must contain all spikes from the previous layer for every timestep. The Spike Cache only needs to be loaded when a new layer begins, as the same spiking activity is shared by every neuron in each layer. A neuron's output spikes are processed by the Spike router, whose operation is described in a later section.

Algorithm 2 ASPENN Controller

```
INITIALIZE: MainMemory ← TrainedWeights
INITIALIZE: SpikeMemory ← InputImage
for (layer_i = 0; layer_i < numLayers; layer_i += 1) do
  SpikeCache ← SpikeMemory
  for (page_i = 0; page_i < numPage; page_i += 1) do
    DataCache ← MainMemory[layer_i][page_i]
    RESET: Neuron Matrix
    for (time_i = 0; time_i < numTimeSteps; time_i += 1) do
      for (spike_i = 0; spike_i < numSpikes; spike_i += 1) do
        Spike ← SpikeCache[time_i][spike_i]
        for (tile_i = 0; tile_i <; tile_i += 1) do
          Weight ← DataCache[spike_i][tile_i]
          Weight → NeuronMatrix[tile_i]
        end for
      end for
      NeuronMatrix.SpikeOut → SpikeMemory
    end for
  end for
end for
```

4.2.4 Neuron Matrix

The Neuron Matrix contains all physical computation units in ASPENN. The matrix is an addressed set of neuron tiles, which in turn are groupings of individual neurons. The Neuron Matrix receives as input a weight from the Controller, as well as an address indicating which tile that weight is being sent to. There are also some control signals, such as a reset signal. Direct tile addressing, where data wires connect the controller directly with each tile and activated using large address decoders, consumes an unnecessarily large amount of routing resources. Instead, ASPENN uses a tree of binary-decision data splitters, where each splitter uses a single bit of the address to decide where to send data. A representation of this routing system is given in Figure 4.3, where each bit in the address is used to direct the data either left or right. ASPENN has a Neuron Matrix of 16 tiles, which requires a 4-bit address. Routing data in this system requires 15 binary splitter junctions. This method is beneficial as the total number of on-chip routing resources is significantly reduced in exchange for only a minor increase in logic resources.

A Neuron Tile is a grouping of several neuron units. The input to a tile is a set of weights from the Controller. ASPENN packs 4 neurons into each tile. Simply copying 4 instances of the neuron unit described in the previous chapter is an inefficient solution. It is worth keeping in mind that since the Controller loops through sending data to every tile in the Neuron Matrix, there will be significant downtime between valid data inputs. This relaxed time constraint allows for some neuron resources to be shared within the tile. The first stage of the neuron update, Counter Compression, cannot be shared, as each counter cannot be overwritten between input arrivals. Therefore, four copies of the Counter Compressor exist in a single Neuron Tile. Stages 2 and 3 of the

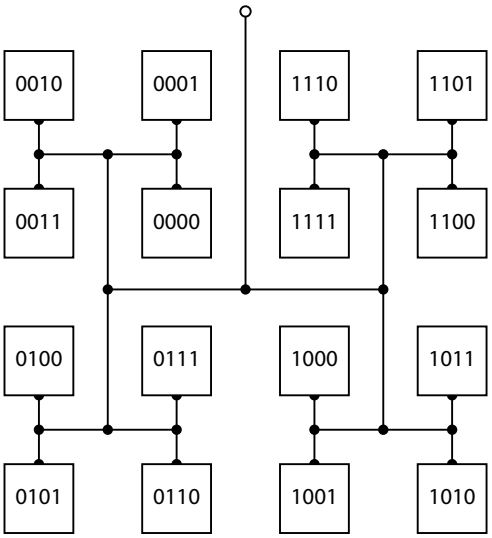


FIGURE 4.3: 16 Tile Neuron Matrix: Each tile is marked with the corresponding 4 bit address. Each bit in the address can be thought of as binary choice the data will make at each intersection.

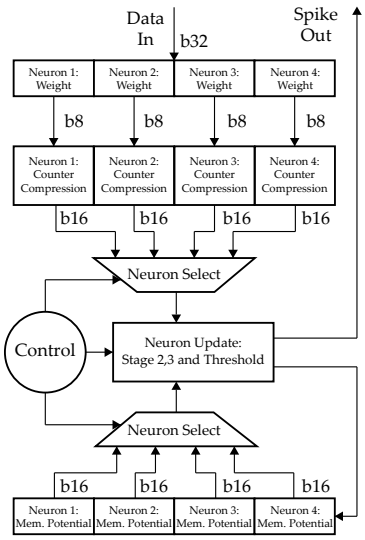


FIGURE 4.4: Neuron Tile: Each neuron tile contains 4 individual neurons. Each neuron has a distinct Stage 1 component. Every neuron shares one Stage 2, Stage 3, and thresholding component.

weight accumulator, as well as the membrane potential thresholding unit, are shared among all neurons in the tile. Each tile contains only a single instance of these units. Inputs to the Neuron Update component are multiplexed between the four Counter Compressors. Membrane potentials for each neuron are stored separately and are similarly multiplexed as inputs to the Neuron Update component. Control logic is also shared, although it does need to be expanded a bit to account for the additional complexity brought on by the shared resources.

4.2.5 Spike Router

The Spike Router is made up of two sub-components, the Spike Memory and the Spike Cache. The Spike Memory exists on the same DDRAM chip as the Main Memory. The Spike Cache exists as additional on-chip scratchpad memory, alongside the Data Cache. Each of these components are divided into time blocks that group spikes that occur at the same simulated time. For ASPENN's design, there are 10 blocks, one for each timestep. Each block needs to be large enough to comfortably allow for the maximum expected spiking activity for a layer in a single timestep. Experimentally, this has been determined to be less than 512, so the Spike Cache has 10 blocks with 512 possible spikes, or 51,200 spikes total. Each spike corresponds to an address indicating which neuron has fired that spike. The size of each spike must be enough to cover the address space of the largest neuron layer. As the maximum layer size in ASPENN is 1024, each spike needs to be 10 bits long. Altogether, this memory system requires approximately 64kB of storage.

At the start of classification, a spiking pattern for a pre-processed input image is loaded into the Spike Memory. Once a new layer begins computation, the entirety of

the Spike Memory contents is loaded into the Spike Cache. During operation, spikes are consecutively read from the Spike Cache and are used as addresses to read from the Data Cache. Once a neuron tile has finished computation for a single timestep, it will fire an output spike pattern for the neurons in that tile. This pattern is interpreted by the Spike Router and converted into a set of valid spike addresses. These spikes are then written to the proper time block in the Spike Memory. This operation continues for every neuron in a layer until the spiking activity for that layer is complete. Once a layer has finished computation and all spikes have been processed and written to memory, the Spike Cache is reloaded with the new Spike Memory contents.

Chapter 5

Evaluation

ASPENN has been evaluated to examine two primary factors, the network's classification accuracy, and the circuit's performance in terms of area and power consumption. A hardware implementation for ASPENN was developed at the register-transfer level using Verilog HDL and synthesized using the Cadence Genus synthesis tool on the gscl-45 nm library. The memory system is evaluated using OpenRAM, an open source tool that models SRAM and DRAM technologies [4]. In the following chapter, ASPENN is analyzed at several levels. First, an ASPENN neuron with an exact weight summation unit is analyzed and compared against alternatives. Next, similar metrics are analyzed for approximate neurons with a variety of configurations. Finally, the ASPENN network is analyzed for classification accuracy and overall area and power consumption.

5.1 Exact ASPENN Neuron

We first evaluate the design of a single exact ASPENN neuron. For this and all other evaluations, neurons have a weight data size of 8 bits, and a Stage 1 Counter size of 5. This analysis is broken down into the several stages of a neuron, as well as for the neuron as a whole. The design analyzed here is that of a single, stand-alone neuron without any of the extra overhead required to form a neuron tile. A carry-save accumulator with the same input dimensions is also analyzed. The CSA is further broken into its two stages, the accumulator, and the final addition. In Table 5.1, the area, power, and timing characteristics of these components are shown.

Circuit	*Area [μm^2]*	*Power [mW]*	*Delay [ps]*	*Energy [pJ]*
1) CSA: Accumulate	426.1	95.1	223	0.103
2) CSA: Update	348	79.3	1349	0.525
3) ASPENN: S1	623.7	20	185	0.037
4) ASPENN: S2/3	599.7	62.3	1405	0.875
5) ASPENN: Complete Neuron	1811.5	116	1488	1.726

TABLE 5.1: Exact Neuron Circuit Metrics: 1) Carry-save adder accumulate circuit, 2) Carry-save adder update circuit, 3) ASPENN Stage 1, counter compressors, 4) ASPENN Stages 2 and 3, block-save addition and CPA addition, 5) ASPENN Complete Neuron, Stages 1, 2, 3, and thresholding

Stage 2 Config.	*Stage 3 Exact/Approx*	*Area [μm^2]*	*Power [mW]*	*Delay [ps]*	*Energy [pJ]*	*Energy Red. [%]*	*Acc [%]*
Exact	Exact	597	62.27	1405	0.875	0	98.33
(0,1)	Exact	580.1	59.83	1314	0.786	-10.17	98.31
(1,2)	Exact	564.1	57.09	**1314**	0.75	-14.29	96.47
Exact	Approx	578.6	59.95	1435	0.86	-1.71	**98.42**
(0,1)	Approx	567.8	57.99	1362	0.79	-9.71	97.9
(1,2)	Approx	**546.7**	**55.17**	1326	**0.732**	**-16.34**	96.49

TABLE 5.2: Approximate Weight Accumulator Metrics: Stage 2 Configurations are either Exact, with all exact adder tree blocks, or approximate with either one or two blocks replaced with an approximate adder tree circuit. Stage 3 Configurations are either an exact CPA adder, or an approximate CPA adder

These results demonstrate two important qualities of the proposed ASPENN design. The first is that the Stage 1 counter compressors consume an order of magnitude less energy than the other stages, which is beneficial as that is the component most frequently active. It also points towards Stages 2 and 3 as the most beneficial to optimize, due to their relatively expensive operation.

The second quality deals with the relative advantage of ASPENN over a CSA accumulator. While at first glance it seems like the ASPENN neuron computation consumes more energy than the CSA, ASPENN gains a significant advantage when the relative frequency of use for each operation is considered. Over a full set of inputs, 31 in the $c = 5$ case, the CSA adder performs an accumulation operation 31 times, and a final addition operation once. This yields an energy per input of: $(31 * E_{acc} + E_{add})/31 = 0.12pJ$. With the same input set, the ASPENN neuron activates the Stage 1 counters 31 times, and the neuron update once, resulting in an energy per input of $(31 * E_{S1} + E_{update})/31 = 0.065pJ$. This is an energy per input reduction of 45.6%, demonstrating a clear advantage for the ASPENN weight accumulator design in terms of energy efficiency. Notably, however, this comes at the expense of nearly twice as much area, although these factors are mitigated due to ASPENN's neuron tiling implementation.

5.2 Approximate ASPENN neuron

An important element of the ASPENN project is its various approximation strategies and their impact on performance. This section details these impacts on the neuron level by modifying the single exact neuron analyzed above. Please refer to section 3.3.3 for a detailed overview of the approximate weight accumulator unit. In brief, Stages 2 and 3 of the weight accumulator unit will be approximated using a variety of configurations. For the Stage 2 approximations, there are two reasonable approximate adder tree designs, marked as Design #1 and Design #2. See Figure 3.5 for diagrams of these two designs. These approximate adder designs can be used to replace the functionality of the exact adder trees used in Stage 2. ASPENN experiments with two configurations of these designs. The first is to use just the less extreme approximation, Design #1, while the second is to use both approximate adder tree designs in adjacent blocks. It

is worth noting that the least significant block does not contain a full set of inputs, and is therefore trivially computed, and will be skipped by these approximation configurations. The Stage 3 approximation uses an open source in-exact adder to perform the final addition. This method is tried with all the Stage 2 configurations. The design metrics of these various configurations are summarized in Table 5.2.

The physical design metrics are derived using the Genus tool with the same synthesis parameters as in the previous section. These metrics are given only for the Stage 2/3 sub-component of the neuron, as the other pieces of the neuron are unaffected by these approximation methods. The accuracy metric is found using a SNN network simulator that uses an emulation of the various approximate neuron configurations. This simulator is applied to the MNIST dataset [8]. For comparison, a purely software SNN with the same structure and floating-point weights achieves 98.46% accuracy on the same dataset. Energy reduction is compared against the exact case.

The first row in Table 5.2 corresponds to an exact ASPENN neuron, and therefore can be used to show the effect of weight quantization on network accuracy. This effect is very small, as MNIST accuracy drops only 0.13% from the software baseline. From the exact circuit, we see progressively lower energy consumption with slightly worsening accuracy. The configuration with the greatest efficiency gain is shown in the last row in the table, with two Stage 2 approximate adder trees as well as the Stage 3 approximate addition. This configuration reduces energy consumption by -16.34% with only a 1.84% reduction in accuracy compared to the exact configuration. Notably, the approximate adder tree replacements in Stage 2 are the primary contributor to the efficiency gains made by these configurations. The Stage 3 approximation only contributes around 2% extra efficiency gain, although this gain is made with either a negligible, or in some cases positive, impact on accuracy.

5.3 ASPENN Network

5.3.1 Network Component Analysis

This project also synthesizes and evaluates a complete ASPENN circuit. Only the on-chip components of ASPENN are physically synthesized. The Main Memory system in particular will not be analyzed, as most of the system's characteristics are determined by the choice of DDRAM chip. The other memory subsystems, the Data Cache and the Spike Cache, are implemented using OpenRAM, an open source SRAM memory compiler [4]. The particular silicon technology used was FreePDK45, an open source process design kit and standard cell library designed for educational and research use. The design characteristics for these two cache systems is shown in Table 5.3.

The other on-chip components, the Controller, and the Neuron Matrix, are designed with Verilog HDL and synthesized using the Genus Synthesis Tool. These components are analyzed at three levels: first a single Neuron Tile, then the complete Neuron Matrix, and finally the matrix plus the Controller. This final system contains all of ASPENN's non-memory components. The results of these components synthesis are outlined in Table 5.4. ASPENN's total area is given by the sum of the cache area, as well as the Neuron Matrix and Controller components. The largest component by

Cache Type	*Area [μm^2]*	*Leakage Power [mW]*	*Read/Write Power [mw]*	*Delay [ps]*	*Energy pJ*
1) Data	633,284	0.529	18.2	648	11.79
2) Spike	70,442	0.052	3.68	519	1.91

TABLE 5.3: ASPENN On-Chip Memory Subsystem Analysis: 1) Data Cache and 2) Spike Cache

System Design	*Area [μm^2]*	*Power [mW]*	*Delay [ps]*	*Energy [pJ]*
1) Neuron Tile	5213	1.79	1942	3.48
2) Neuron Matrix	84,093	21.1	1944	41.02
3) Matrix+Controller	91,250	22.3	2104	46.92

TABLE 5.4: ASPENN On-Chip Computation Subsystem Analysis: 1) Neuron Tile (4 neurons per tile), 2) Neuron Matrix (16 tiles per matrix, 64 neurons total), 3) Neuron Matrix + Controller

far is the Data Cache, although that system uses less energy when compared to the computation units of the Neuron Matrix.

5.3.2 Network Energy Consumption Analysis

ASPENN's net energy consumption can be estimated using these component level metrics. The activity of a network is driven by the spiking activity of the neuron, as this activity directly impacts the rate at which various components are used during a classification. Spiking activity can be used to calculate the effect of the memory subsystem, as well the computation units in the Neuron Matrix and the Controller unit. The total energy consumption of the memory system can be estimated based on the network's expected activity and the individual cost for a read/write operation. During operation, each spike is loaded to the Spike Memory exactly once and is loaded from the memory to the Spike Cache exactly once. A spike is re-used for every neuron active in a Matrix, so it only needs to be read once per active neuron set. This analysis will ignore the energy consumption required to access the Main Memory, as that is specific to the choice of DDRAM chip. The Spike Cache energy contribution is given by Equation 5.1.

$$\begin{aligned} E_{\text{spike cache}} = {} & E_{scwrite} * (numspikes_{layer1} + numspikes_{layer2} + numspikes_{layer3})/size_{tile} \\ & + E_{scread} * (numspikes_{layer1} * size_{layer2} + numspikes_{layer2} * size_{layer3} \\ & + numspikes_{layer3} * size_{layer4})/(size_{matrix} * size_{tile}) \end{aligned} \tag{5.1}$$

The size of each layer is determined by the network parameters, while the number of spikes for each layer varies depending on the image being classified. A similar equation is derived for the Data Cache's energy contribution. Each weight in the SNN network is loaded into the cache once, again grouped by neuron tile. Each spike causes

TABLE 5.5: ASPENN Energy Consumption Analysis

System Component	***Net Energy [nJ]***	***Proportion [%]***
Spike Cache	44	1.76
Data Cache	2,367	94.68
Computation	89	3.56
Total	2,500	100

all corresponding weights to be read from the cache. This behavior results in the total Data Cache energy contribution to be expressed by Equation 5.2.

$$\begin{aligned} E_{\text{data cache}} = E_{dcwrite} * (size_{layer1} * size_{layer2} + size_{layer2} * size_{layer3} + size_{layer3} \\ * size_{layer4})/size_{tile} + E_{dcread} * (numspikes_{layer1} * size_{layer2} \\ + numspikes_{layer2} * size_{layer3} + numspikes_{layer3} * size_{layer4})/size_{tile} \end{aligned} \tag{5.2}$$

The computational unit's energy consumption is also driven by the spiking activity. Each spike initiates a computation in every neuron in the receiving layer, although most of these computations are low-cost Counter Compression operations. For each block of inputs, determined by the size of counters, a single complete update of each neuron tile is performed. The computation energy consumption of ASPENN can therefore be determined by Equation 5.3.

$$\begin{aligned} E_{\text{computation}} = E_{\text{counters}} * (numspikes_{layer1} * size_{layer2} + numspikes_{layer2} * size_{layer3} \\ + numspikes_{layer3} * size_{layer4})/size_{tile} + E_{neuron\_tile} * \left(size_{layer2} * \left\lceil \frac{numspikes_{layer1}}{size_{block}} \right\rceil \right. \\ \left. + size_{layer3} * \left\lceil \frac{numspikes_{layer2}}{size_{block}} \right\rceil + size_{layer4} * \left\lceil \frac{numspikes_{layer3}}{size_{block}} \right\rceil \right) /size_{tile} \end{aligned} \tag{5.3}$$

While the exact spiking activity of each layer is highly dependent on the input image, an average activity can be determined experimentally through simulation. For an exact ASPENN system with the layer parameters {784, 1024, 1024, 10}, the three non-output layers are determined to have a spiking activity of 1029, 279, 610 spikes per classification. These parameters and activity can be used in Equations 5.1, 5.2, and 5.3. The results of the various energy contributions are given in Table 5.5. These results estimate that ASPENN's average energy consumption per classification is 2.5 μJ. Much of this energy consumption is dominated by the memory system.

Chapter 6

Conclusion

Spiking neural networks are a promising alternative to standard network architectures. ASPENN is a novel SNN accelerator design that is intended to offer an energy efficient implementation of the spiking network computation. ASPENN is designed to be modular and parameterizable, so that a variety of network sizes and dimensions can be implemented easily. ASPENN's novel weight accumulator unit improves computational energy consumption by 45.6% over the standard CSA accumulator units at the expense of a modest increase in circuit area. Several approximation methods and configurations are applied to this weight accumulator unit, and a novel method to approximate arbitrary adder trees is presented. The modular design of the accumulator unit makes it easy to incorporate approximate modules into the wider design. These methods contribute to a further 16.34% energy reduction compared against the exact accumulator, while causing a decline in classification accuracy of less than 2%. ASPENN's average energy consumption per classification is estimated to be 2.5 μJ.

Currently, ASPENN only supports feed forwards network architectures with non-leaky Integrate-and-Fire neurons. Future improvements to the ASPENN system could include other neuron variants, such as leaky IF neurons, and refractory periods, as well as more advanced network architectures such as convolutional spiking networks. The approximation technique developed for ASPENN could also be expanded upon, and more techniques could be applied to improve the neuron computation performance.

All code for this project can be found on Github using the following link:

https://github.com/scale-lab/ASPENN.git.

Bibliography

[1] Mike Davies et al. “Loihi: A Neuromorphic Manycore Processor with On-Chip Learning”. In: *IEEE Micro* 38.1 (2018), pp. 82–99.

[2] Peter U. Diehl et al. “Fast-Classifying, High-Accuracy Spiking Deep Networks Through Weight and Threshold Balancing”. In: *International Joint Conference on Neural Networks (IJCNN)* (2015), pp. 1–8.

[3] Steve B. Furber et al. “The SpiNNaker Project”. In: *Proceedings of the IEEE* 102.5 (2014), pp. 652–665.

[4] M. Guthaus et al. “OpenRAM: An open-source memory compiler”. In: *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (2016), pp. 1–6.

[5] S. Kannappan and S. Mastani. “A Survey on Multi-Operand Addition”. In: *Acta Technica Corveniensis - Bulletin of Engineering* 13 (2020), pp. 65–68.

[6] Yongtae Kim, Yong Zhang, and Peng Li. “Energy Efficient Approximate Arithmetic for Error Resilient Neuromorphic Computing”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 23.11 (2015), pp. 2733–2737.

[7] Yongtae Kim, Yong Zhang, and Peng Li. “In energy efficient approximate adder with carry skip for error resilient neuromorphic VLSI systems”. In: *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (2013), pp. 130–137.

[8] Y. LeCun et al. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[9] Vojtech Mrazek et al. “Design of power-efficient approximate multipliers for approximate artificial neural networks”. In: *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (2016), pp. 1–7.

[10] Vojtech Mrazek et al. “EvoApprox8b: Library of Approximate Adders and Multipliers for Circuit Design and Benchmarking of Approximation Methods”. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE)* (2017), pp. 258–261.

[11] J. G. Oklobdzija, D. Villeger, and S. Liu. “A method for speed optimized partial product reduction and generation of fast parallel multipliers using an algorithmic approach”. In: *IEEE Transactions on Computers* 45.3 (1996), pp. 294–306.

[12] Nitin Rathi, Priyadarshini Panda, and Kaushik Roy. “STDP-Based Pruning of Connections and Weight Quantization in Spiking Neural Networks for Energy-Efficient Recognition”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38.4 (2019), pp. 668–677.

[13] Sanchari Sen, Swagath Venkataramani, and Anand Raghunathan. “Approximate Computing for Spiking Neural Networks”. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE)* (2017), pp. 193–198.

[14] Evangelos Stromatias et al. “Robustness of Spiking Deep Belief Networks to noise and reduced bit precision of neuro-inspired hardware platforms”. In: *Frontiers in Neuroscience* 9 (2015), p. 222.

[15] Sheba Thabah, M. Sonowal, and Prabir Saha. "Experimental Studies on Multi-Operand Adders". In: *International Journal on Smart Sensing and Intelligent Systems* 10 (2017), pp. 327–340.

[16] P. Umarani. "A High Performance Asynchronous Counter using Area and Power Efficient GDI T-Flip Flop". In: *Indian Journal of Science and Technology* 8.7 (2015), pp. 622–628.

[17] Swagath Venkataramani et al. "AxNN: Energy-efficient neuromorphic systems using approximate computing". In: *IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)* (2014), pp. 27–32.

[18] Chi-Hsiang Yeh and B. Parhami. "Efficient pipelined multi-operand adders with high throughput and low latency: designs and applications". In: *Conference Record of The Thirtieth Asilomar Conference on Signals, Systems and Computers* 2 (1996), pp. 894–898.

[19] Shihui Yin et al. "Algorithm and Hardware Design of Discrete-Time Spiking Neural Networks Based on Back Propagation with Binary Activations". In: *IEEE Biomedical Circuits and Systems Conference (BioCAS* (2017), pp. 1–5.