Summary

We will be investigating an implementation of Neural Networks into a low-energy FPGA. Neural Networks are a common machine learning algorithm with a high potential for parallelization, which can be exploited by hardware. If Neural Networks become a commonplace mobile application, the battery drain of the software implementation would be a strong deterrent. This project investigated the properties of a low-energy software implementation.

Background:

An artificial neural network is a statistical learning algorithm involving layers of nodes, called perceptrons, which process information in a way that approximates an unknown function. Starting from an input layer, information is filtered, modified, and passed down through a series of hidden layers until reaching the final output layer. One of the major uses of neural networks involves their ability to discern the underlying function connecting a set of input values to a set of output values.

The network structure chosen contains a single hidden layer, and a single output node. Each input node will send its weighted input to all nodes in the hidden layer. The hidden layer nodes then apply a sigmoid function 1/(1+e^(-x)), where x is the sum of their corresponding inputs. This function has the important property that it slopes sharply upward at 0.5, which is called the activation point. The activated outputs form the hidden layer are sent to the single output node in the same fashion and then summed to provide the final output

The key to this training process lies in the set of weights given to the input values. Neural Networks train these weights based on training data with known outputs for certain inputs. The weight calculations will be done using the backpropagation algorithm. This algorithm essentially iterates through the training dataset and alters the weights until we reach a certain error threshold or predefined iteration count.

As seen from the pseudocode below, there are a few clear avenues for parallelism in the algorithm. Specifically, the weighted sum and error update portions can be easily parallelized. They follow a pattern that can be modeled using matrix multiplication which is ideal for an FPGA's numerous parallel multipliers. However, the nature of the backpropagation algorithm introduces data dependencies between layers of the neural network and across iterations of the training dataset.

Sequential implementation of Backpropagation algorithm:

```
while(previous error > error) {
  for(each test datapoint) {
    for(each hidden node) {
      sum = 0;
      for(each input node)
        sum += weight * input;
      hidden_value = 1/(1+Math.exp(-sum));
    }
    sum = 0;
    for(each hidden node)
      sum += weight * hidden_value
    output_value = 1/(1+Math.exp(-sum));

    error += (output_value - true_value)^2;
    output_weight_err = error caused by output weight;

    for(each hidden node) {
      hidden_weight_err = error caused by hidden weight;
      for(each input node)
        new_hidden_weight = old weight + weight change;
      new_output_weight = old weight + weight change;
    }
  }
}
```

Approach:
Technology: Altera de2-115 FPGA, 2.5 GHz Core i7 (3770K) CPU
PC side implementation in Java and C++. FPGA implementation in SystemVerilog.
Used FANN (http://leenissen.dk/fann/wp/) library as a benchmark.

Three different implementations of the same single 4 node hidden layer neural network were used as test comparisons: the original Java implementation, the FANN library implementation in C++, and the FPGA implementation. The input dataset consists of 4 inputs and one output resulting from a linear combination of the inputs. There are a total of 200 input data points each represented using floating point numbers. These go through 200 iterations of the backpropagation training algorithm. Once it is trained, the networks can be tested very quickly by doing the multilayer weighted sum which was trained.

The small neural network results in a somewhat reduced potential speedup, since the parallelism potential of calculation is much less than in large networks. Most CPUs could achieve the level of hardware parallelism we achieved using just SIMD and hyperthreading. The primary benefit, however, of using a hardware implementation is that the power consumption is an order of magnitude lower. The hardware speedup is achieved using a large number of built in multipliers, which perform all of the output calculations simultaneously, leading to single-clock result generation. An alternative implementation would have used much less multipliers, but would have ended up being significantly slower. Reusing the multipliers would also lower the footprint, and potentially lower the power consumption, but would sacrifice speed.

Our neural network is represented using 4 matrices: input, hidden_output, hidden_weights, and output_weights. Running the backpropagation algorithm for a single training example takes a total of 7 clock cycles:

1. Multiply hidden_weights and input storing the value into sum (4x4x3 = 48)
2. Apply sigmoid on sum (3)
3. Multiply sum by output_weights and store into output (4x3 = 12)
4. Apply sigmoid on output (3)
5. Calculate output_weight_err (3)
6. Calculate hidden_weight_err (4)
7. Update both hidden_weights and output_weights (4x4x4 + 4x4 = 76)

The values in parentheses represent the cost of a completely serial implementation assuming that only one arithmetic operation can be done during each clock. In total, the training should take 7x200x200 = 280000 clocks to complete on the FPGA. A clock speed of 1 GHz puts the theoretical running time at around .28 ms.

The overall control flow of the hardware implementation was a 7 stage FSM that could potentially be pipelined in a future version.

Results:

We were able to successfully design and implement the neural network in SystemVerilog. The primary development was done using ModelSim to simulate the hardware and ensure correctness. The synthesis portion of the project was less successful, with some issues in the user interface leading to unusable results. Performance metrics were determined through synthesis. We benchmarked our performance against a FANN library implementation running on a Core i7 3770K. Power usage for the FPGA was measured using Xilinx's official power estimation tool and benchmark values were used for the CPU.

Experimental Setup:

We used a fixed neural network setup with 4 input nodes, 4 hidden nodes, and 1 output node. Each of the 200 training data points consisted of 4 input decimal numbers and a binary output. The inputs are randomly generated between 0-1. The output comes from a sum of randomly generated weights between .15-.35 multiplied by their respective inputs. These data points are preloaded onto the FPGA through a ROM block. The neural network is configured to run 200 iterations of the training dataset. The learning rate is initialized to .2 and the initial hidden weights are ~0.05.

Test data points are also loaded onto the ROM in the beginning. These consist of 4 randomly generated numbers representing the input. After training is completed, the FPGA should alow the user to select which test point they want to use, and the hardware will use the now trained neural network to compute the output. This is displayed and checked against the output of the software side implementation running the same steps.

Limitations:

The neural network implementation using backpropagation that we started with has inherent data dependencies between layers of neurons, data points, and iterations. Neurons receive their inputs from the previous layer. The gradient descent model that forms the basis of backpropagation requires iteratively changing the weights after each run.These factors limited the amount of speedup that can be gained through parallelism.

However even with these limitations, the nature of dedicated hardware gave us the performance boost that we wanted. This is a great example of the tradeoff between performance and flexibility.

Deeper Analysis:

Much of the success of this project revolved around the strict limitations we placed upon the neural network. In terms of performance, if the network had more nodes, we may not have had enough multipliers to do the matrix multiplications in one clock. We will not be able to represent extremely large or extremely precise floats with the fixed point representation we chose.

As for complexity, the preferred way to execute training is to keep iterating through the training data set until the error rate of running the network on the training data is below a

certain threshold, which was not necessary for the scope of this project. Similarly, applying an exact sigmoid function is much more challenging on an FPGA than in software. The training data set and testing data set are both static, and on chip. A real world implementation would utilize a communication protocol like ethernet, which was also outside the scope of this project.
Essentially, this implementation trades the ability to support multiple types of datasets in order to deal with a specific dataset very well. This type of tradeoff is beneficial for dedicated hardware such as wearables.

Detailed Results:
One of the main advantages of an FPGA is its power usage. For example, the reported thermal design power of an Xilinx xc5vlx330 is 30W while standard intel i7 chips hover around 100W. However, since our neural network will not cause either to run at full load, we need a better way to measure energy consumption. Below are some power stats generated using Xilinx's power estimation tool based on running our neural network the newest Altera devices.

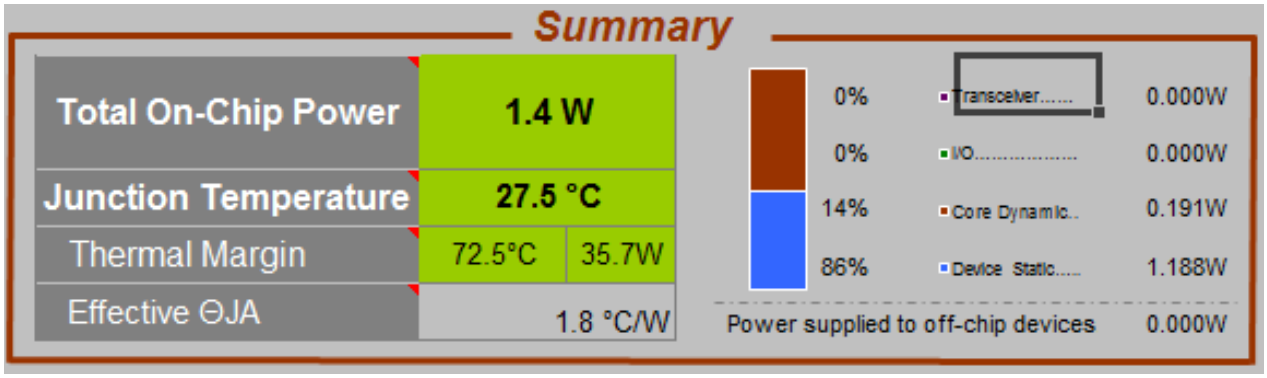
Device	
Family	Virtex UltraScale
Device	XCKU115
Package	FBVA900
Speed Grade	-1L (0.9V)
Temp Grade	Industrial
Process	Typical
Voltage ID Used	
Characterization	Advance, v1.2, 2015-01-20

On-Chip Power

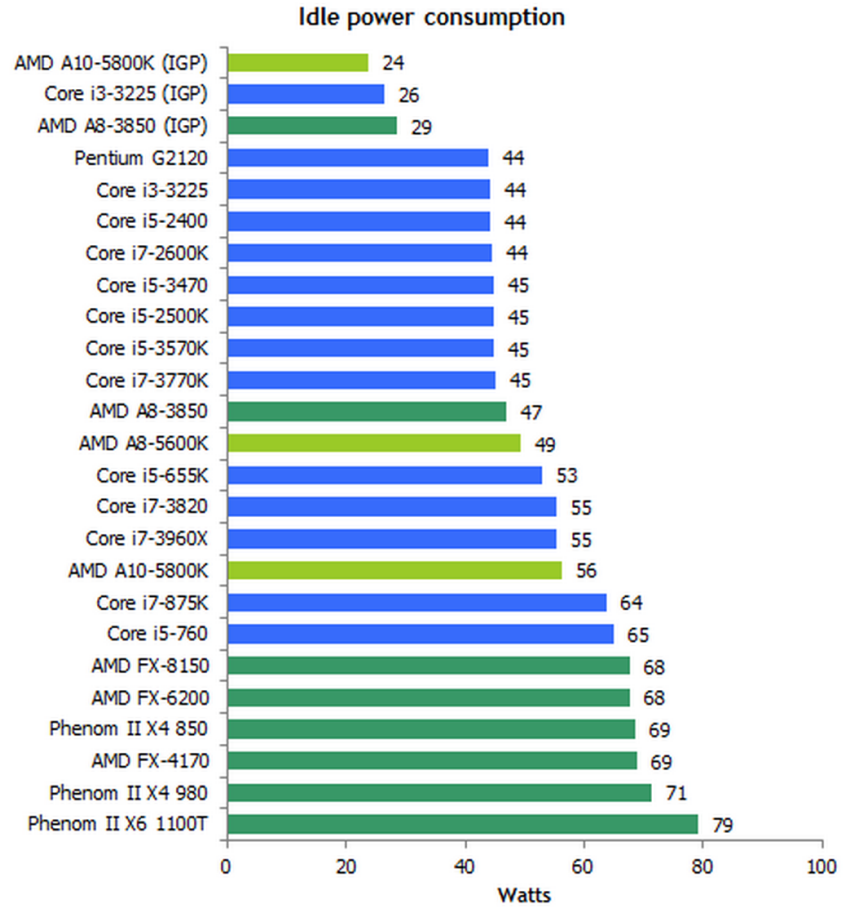
Resource		Power (W)	Power (%)
Core Dynamic	CLOCK	0.050	4
	LOGIC	0.118	9
	BRAM	0.024	2
	DSP	0.000	0
	PLL	0.000	0
	MMCM	0.000	0
	Other	0.000	0
	Hard IP	0.000	0
I/O	IO	0.000	0
Transceiver	GTH	0.000	0
Device Static		1.188	86

Power Supply

Source	Voltage	Total (A)
V_{CCINT}	0.900	0.630
V_{CCINT_IO}	0.900	0.035
V_{CCBRAM}	0.950	0.040
V_{CCAUX}	1.800	0.237
V_{CCAUX_IO}	1.800	0.162
V_{CCO} 3.3V	3.300	
V_{CCO} 2.5V	2.500	
V_{CCO} 1.8V	1.800	
V_{CCO} 1.5V	1.500	
V_{CCO} 1.35V	1.350	
V_{CCO} 1.2V	1.200	
V_{CCO} 1.0V	1.000	
$MGTV_{CCAUX}$	1.800	
$MGTAV_{CC}$	1.000	
$MGTAV_{TT}$	1.200	



Comparing this to idle cpu power consumptions:



Taking our performance calculations from before, the training step on the FPGA takes around 280000 cycles leading to a theoretical performance of .28 ms. The CPU FANN library implementation on the Core i7 took around 22 ms. So we have an energy consumption difference of $\frac{(22\ ms) * 45\ W - (.28\ ms) * 1.4\ W}{(.28\ ms) * 1.4\ W} = 2524\%$. We can see that the FPGA is a much more energy efficient implementation.

Notice that these calculations were calculated assuming a fixed 200 iterations on datasets of size 200. However, the FPGA energy usage scales linearly with iterations and dataset size while FANN energy usage only scales linearly with dataset size:

Fixed Iterations:

Iterations	Dataset Size	FPGA	FANN
200	200	.28 ms	22.36 ms
200	300	.42 ms	33.77 ms
200	400	.56 ms	44.42 ms
200	500	.70 ms	54.80 ms

Fixed Dataset Size:

Iterations	Dataset Size	FPGA	FANN
200	200	.28 ms	22.36 ms
300	200	.42 ms	83.9 ms
400	200	.56 ms	111.75 ms
500	200	.70 ms	139.47 ms

Conclusions:
Based on the above results, we can safely say that we have met our performance and energy goals. However, it came at the cost of usability. Future improvements would involve support for multiple types of neural networks that can be trained using a variety of methods on any data set.

List of work:
dswillis: FPGA design and implementation of neural network, proposal
bohanl: Helper scripts, FANN and java implementations, report