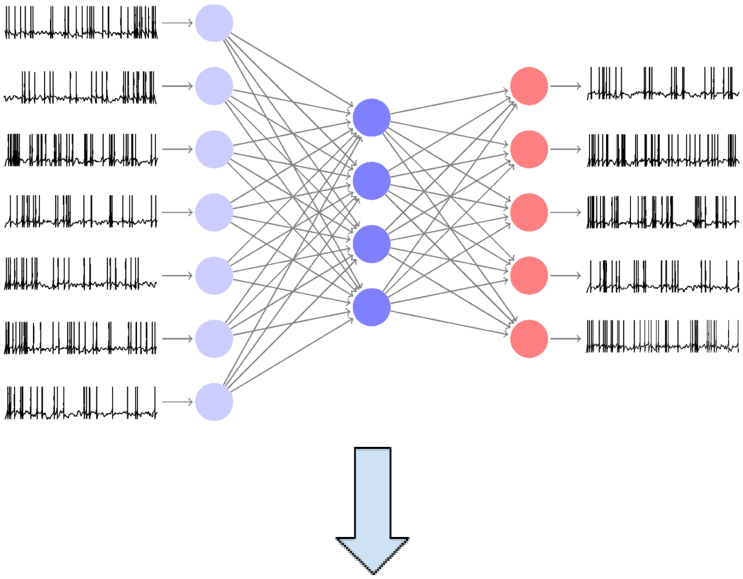
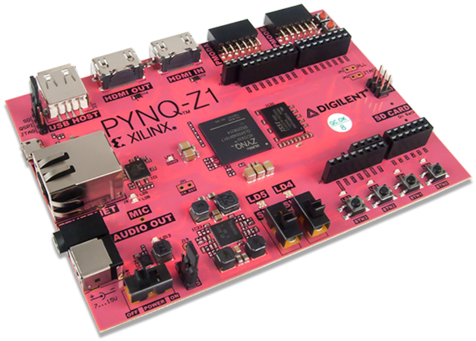
CSE237D Final Report

Pynq Snn Accelerator





Srinithya Nagiri
Tanaya Kolankari
Navateja Alla

Final Report

Abstract:

Spiking Neural Networks (SNNs) are the third generation neural networks gaining importance due to their similarity to biological neural systems. However, the real world engineering applications of SNNs have been limited due to the increased computational costs associated with these networks. It is important to accelerate their hardware implementation to enable widespread integration in smart embedded systems. In our project, we aim to model SNNs using the Leaky Integrate and Fire (LIF) neuron model and implement an SNN hardware accelerator on a Programmable System on Chip (PSoC) which has an FPGA and a general purpose processor on the same chip. The accelerator implementation on the PYNQ Z1 board provides an almost 40x improvement over the software implementation on the ARM processor and a 2x improvement over normal laptops/computers. Our results show that performance improvement and strong scaling can be attained through an efficient division of work between the processor and FPGA in PSoC. Our results will help indicate the suitability of the LIF model for large scale SNN hardware implementation.

Introduction:

Spiking neural networks intend to model the way the brain works. In our brain, each neuron has a membrane potential and when a combination of inputs result in this potential to cross a threshold, neuron spikes. These spikes are sparse in time and hence carry a high amount of information. SNNs use this spike timing information for processing. Spiking networks provide solutions to a broad range of specific problems in applied engineering, such as fast signal-processing, event detection, classification, speech recognition, spatial navigation or motor control. To lower the computational costs and to promote large scale simulations of SNNs, we need low-power, highly parallel, low cost and re-programmable hardware. FPGA has held an outstanding performance in low-power and large-scale parallel computing domain. Compared to CPU or GPU which are based on the Von Neumann or Harvard Architecture, FPGA has a more flexible framework to implement algorithms. The instructions and data in FPGA can be designed in a more efficient way without the constraint of fixed architectures, which is suitable for designers to explore the high performance implement approaches in power or computing sensitive fields. FPGA with its inherent parallel architecture and other above-mentioned characteristics is best suitable for this task. Our project is about implementing inference of a Spiking Neural Network on FPGA.

We used PYNQ-Z1 board for our project. PYNQ or Python for ZYNQ is an open source project from Xilinx is a programmable System on Chip (SoC) which integrates a multi-core processor (Dual-core ARM processor) and a Field Programmable Gate Array (FPGA) into a single integrated circuit. It has been widely used for machine learning research and prototyping and doesn’t require any ASIC-style design tools to design the programmable hardware. A PYNQ enabled Zynq board can be easily programmed in Jupyter Notebook using Python. In PYNQ, programmable logic circuits are presented as hardware libraries called overlays and these overlays abstract the details of FPGA programming. We chose PYNQ board as it is a new platform and because of its relative ease of use as mentioned above.

There are different spiking neuron models like Spike response model, Leaky Integrate and Fire model and etc., available for modeling a spiking neural network. In our project, we use Leaky Integrate and Fire neuron model for implementing SNN as it is historically the most common spiking neuron model. The LIF model makes use of the fact that neuronal action potentials of a given neuron always have roughly the same form. If the shape of an action potential is always the same, then the shape cannot be used to transmit information: rather information is contained in the presence or absence of a spike. Therefore action potentials are reduced to ‘events’ that happen at a precise moment in time. Input stimulus is encoded into input spikes using rate encoding. Spike train generated has frequency proportional to the corresponding membrane potential of the input neuron.

We tried different SNN architectures with the different number of layers, with and without convolutions and also using NengoDL library. The respective python implementations were ported to C++ for performing High-Level Synthesis to generate a bitstream that can be used to program the FPGA. We used Xilinx Vivado suite, a set of High-Level Synthesis (HLS) tools that allow hardware programming using high-level languages like C/C++ without the need to manually create RTL, and design tools to integrate hardware blocks together and generate a “bitstream”. We performed High-Level Synthesis on two of our SNN architectures implemented and generated bitstreams.

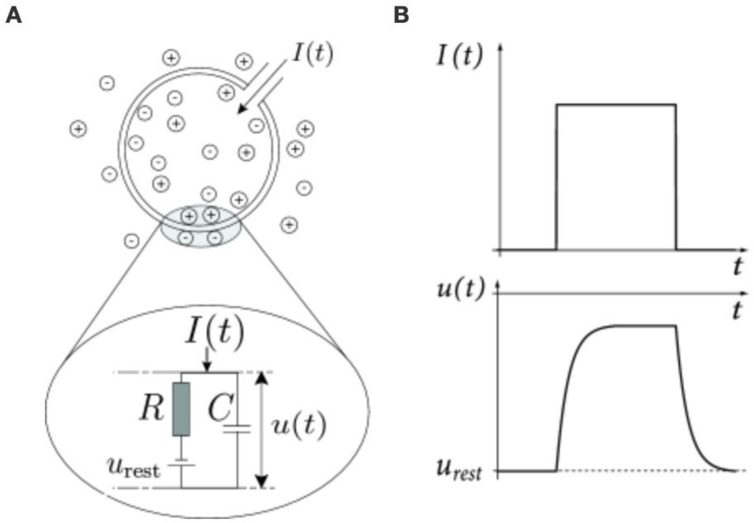
We did a performance comparison between the software implementation and the FPGA implementation. Our final demonstration is about passing an input spike train obtained by encoding an image from MNIST handwritten digits dataset and observing the output spikes of neurons in the output layer.

Project objectives:

1. Train the spiking neural network in software and obtain the weight matrices
2. Implement the neural network using pretrained weights on the programmable logic of PYNQ Board
3. Optimize the code for better performance and do a performance comparison between software and hardware implementations.

Background about LIF neuron model:

Neuronal dynamics can be conceived as a summation process (sometimes also called ‘integration’ process) combined with a mechanism that triggers action potentials above some critical voltage. The basic electrical circuit representing a leaky integrate-and-fire model consists of a capacitor C in parallel with a resistor R driven by a current $I(t)$ is as follows:



u_i is the momentary value of the membrane potential of the neuron *i*, u_{rest} is the resting potential which is the potential of neuron in absence of input and if a neuron receives input current I(t) the potential u_i will deflect from resting potential. If the driving current I(t) vanishes, the voltage across the capacitor is given by the battery voltage u_{rest}. In order to analyze the circuit, we use the law of current conservation and split the driving current into two components as follows

$I(t) = I_R + I_C$

$I(t) = \frac{u(t) - u_{rest}}{R} + C\frac{du}{dt}$

τ_m is the time constant of the leaky integrator

$\tau_m \frac{du}{dt}$ = -[u(t) - u_{rest}] + RI(t)

We assume that at t=0, the membrane potential has a value $u_{rest} + \Delta u$. For t > 0, the input vanishes and I(t) =0. The membrane potential decays by the time constant τ_m. So, if we wait long enough, the membrane potential relaxes to u_{rest}. The solution of the above differential equation with initial condition $u(t_0) = u_{rest} + \Delta u$ is

$u(t) - u_{rest} = \Delta u \, exp(-(t - t_0)/ \tau_m)$

With initial condition *u(0)* = u_{rest}, the solution for the differential equation for 0 < t < Δ is

$u(t) = u_{rest} + R I_0 [1 - exp(-t / \tau_m)]$

This is how the membrane potential builds up as the neurons receive input current. And when this membrane potential crosses a threshold value, the neuron spikes.

Technical Material:

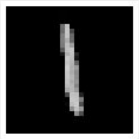
The overall objective of the project is to implement inference of a Spiking Neural Network on an FPGA.

Software Implementation:

We have implemented multiple SNN architectures, one, a pure python implementation in reference to the git repo https://github.com/Shikhargupta/Spiking-Neural-Network and other architectures using NengoDL library which is a graphical and scripting based Python package for simulating large-scale neural networks. Nengo can create sophisticated spiking neural simulations in a few lines of code.

1. A 2-layer SNN for binary and multiclass classification

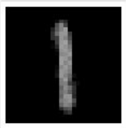
It is a simple 2-layer network with 784 neurons in the input layer and 20% more neurons than the number of neurons actually needed to classify n classes. For example, if we want to perform binary classification, 2 neurons in the output layer are actually sufficient but we place 3 neurons in the output layer. Spike-Time Dependent Plasticity (STDP) learning method is used for training. If we track the output spikes of neurons through the training, we can observe that initially, each output neuron responds to every pattern. But as the network goes through training, neurons learn to respond to specific patterns. Few neurons respond to multiple patterns and thus learn noise. That is the reason why we take a few extra neurons. Also if the weights learned during training by each neuron are reconstructed into an image, it resembles the respective pattern the neuron learned as shown below.

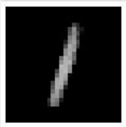

neuron1.png

neuron2.png

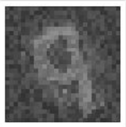
neuron3.png

We trained the network for 10 classes by increasing the number of neurons in the output layer to 12 and by tweaking some parameters like learning rate, initialization of weights, etc., but even after many attempts, the neurons couldn’t learn all the 10 patterns. It requires fine parameter tuning and seemed to be out of the scope of this project. The patterns learned by neurons are as follows


neuron1.png


neuron2.png

neuron3.png


neuron4.png

neuron5.png

neuron6.png

neuron7.png

neuron8.png

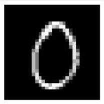
neuron9.png

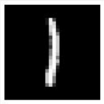
neuron10.png

neuron11.png

neuron12.png

We tried training the network for six classes as mentioned in the repo and the neurons could learn all the six patterns successfully. We ported this python implementation to C++ to go forward with High Level Synthesis.


neuron1.png


neuron2.png

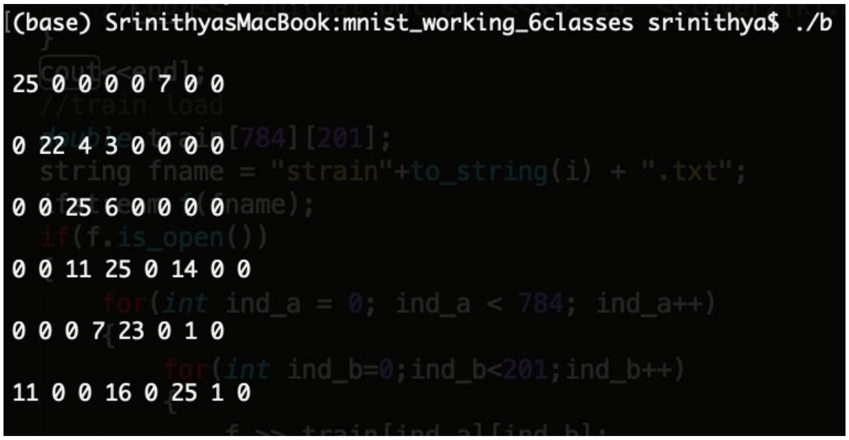
neuron3.png

neuron4.png

neuron5.png

neuron6.png

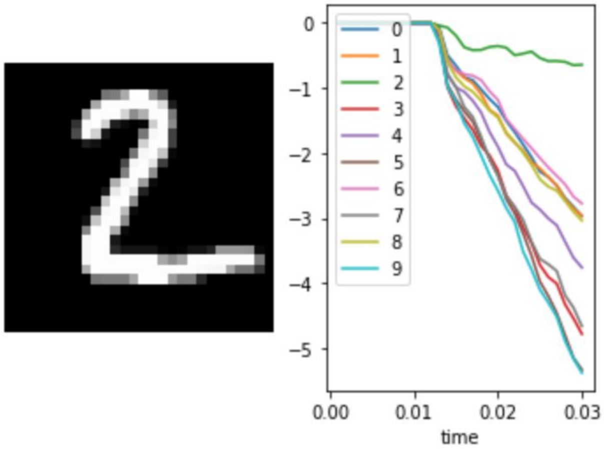
The output of the C++ code is shown in the below screenshot. It can be observed one neuron is firing more spikes than all other neurons for one specific pattern.



```
(base) SrinithyasMacBook:mnist_working_6classes srinithya$ ./b

25 0 0 0 0 7 0 0

0 22 4 3 0 0 0 0

0 0 25 6 0 0 0 0

0 0 11 25 0 14 0 0

0 0 0 7 23 0 1 0

11 0 0 16 0 25 1 0
```

2. 5-layer SNN architecture using NengoDL library

   The network example mentioned in the site had a 5 layer architecture comprising of one input layer, followed by 3 convolution layers and a linear readout. The last two convolution layers are followed by average pooling layers and each convolution layer is followed by a neural nonlinearity. The accuracy obtained after training is 98.75%. The below figure shows the variation of the potential of each of the output neurons. As can be seen, over the timesteps the potential of the relevant class is greater than the remaining neurons by a large margin.



We modified the 5 layer architecture to have only fully connected layers by removing all the convolution layers for simplicity. This architecture had 784, 256, 64, 16 and 10 neurons respectively in the five layers. The initial implementation uses Tensorflow function in the last layer and as a result, the weights of the last layer were not trainable and also accessible as the Tensorflow internally manages the layer object. We had to modify the last layer to use only Nengo objects to obtain the weights. The accuracy achieved after training for this architecture is 93.5%. We ported this implementation to C/C++ including all dependent classes and functions which involved understanding the internal functionality of NengoDL library and performed High Level Synthesis. The HLS performed on the C/C++ code failed as it was using too many resources (around 300% more). We modified the code to make it efficient and HLS still failed but this time the resource utilization was around 27% more. The 5-layer network has too many parameters and hence is difficult to fit.

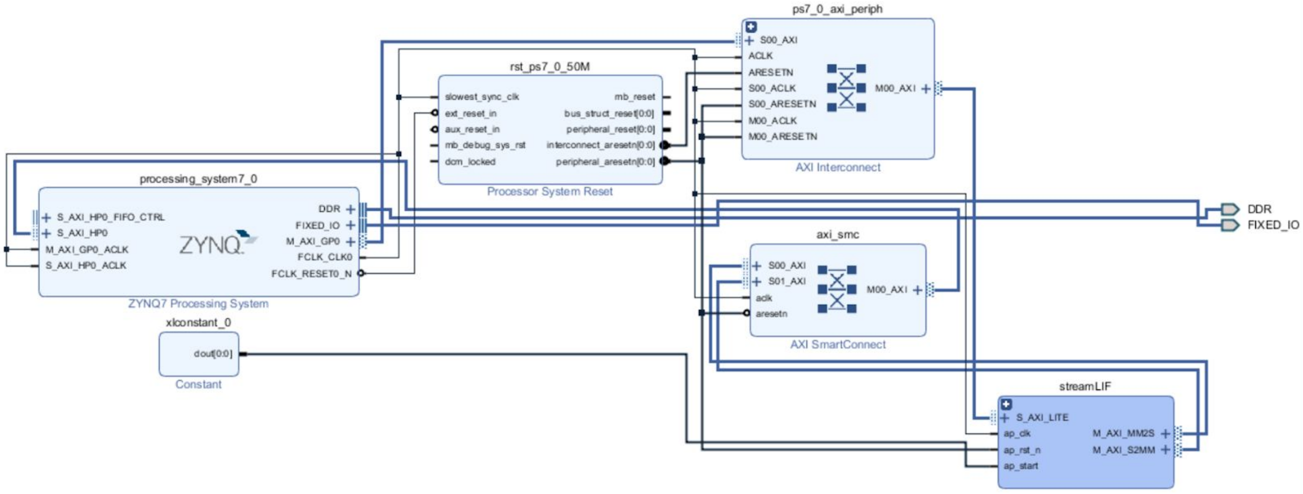
3. SNN with one hidden layer using NengoDL library
   We reduced the number of layers from 5 to 3 to make it fit onto the FPGA. Accuracy was observed with the different number of neurons in the hidden layer varying from 16 to 128. Also, we tried to tweak the number of epochs. The architecture with 64 neurons in the hidden layer and training for 10 epochs gave the best accuracy. We implemented the same architecture in C/C++ and successfully synthesized the design in Vivado HLS. The initial design had a resource utilization of less than 50%. The PYNQ board was programmed with the obtained bitstream and output spikes were generated.

Synthesis on Vivado HLS:

We ported the simple 2 layer spiking neural network for 6 class (0-5 digit) classification and the more complex nengo based 3 layer network for complete 10 class classification into C++. This code was then passed through the entire Vivado HLS Synthesis flow that has been described below:

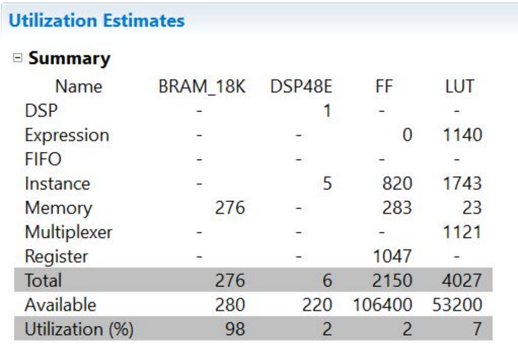
- Conversion of all dynamic arrays into proper fixed arrays or pointers
- Division of the code into submodules for efficient porting to hardware
- Design of testbench for verification of the design
- Synthesize and Export RTL to generate an IP of the design
- Import the custom IP into the Vivado project
- Create the block diagram of the implementation using the custom IP, DMA IP and the ZYNQ Processing system
- Customize the individual IPs based on the requirement of the block design
- Validate the final design to ensure correctness
- Create an HDL wrapper around the block design and generate bitstream

The generated bitstream was then imported in a notebook running on the linux platform of PYNQ Arm processor. The input spike train is sent by the ARM processor to the network on the programmable logic using the dma module implemented in the block design. The dma module is also responsible for collecting the output spikes from the network. This communication is handled by the jupyter notebook running on the ARM processor. The final block diagram for the hardware implementation is as shown below:



Overall, we tried three hardware implementations out of which we were able to successfully implement two networks onto the actual PYNQ board while we got one of the networks to synthesize.

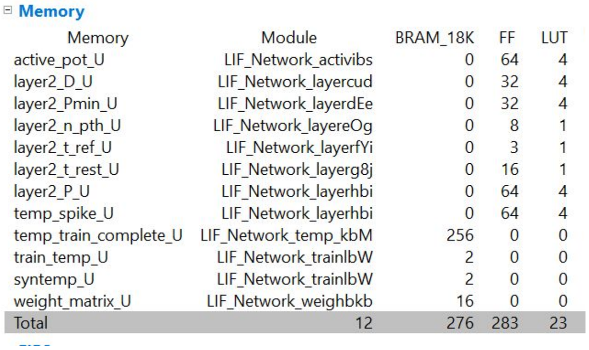
1. 2 Layer SNN Implementation for 6 digit classification
   We successfully obtained a working baseline hardware implementation for the 2 layer network. The overall FPGA utilization with specific focus on memory is outlined in the figure below.



Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	1	-	-
Expression	-	-	0	1140
FIFO	-	-	-	-
Instance	-	5	820	1743
Memory	276	-	283	23
Multiplexer	-	-	-	1121
Register	-	-	1047	-
Total	276	6	2150	4027
Available	280	220	106400	53200
Utilization (%)	98	2	2	7



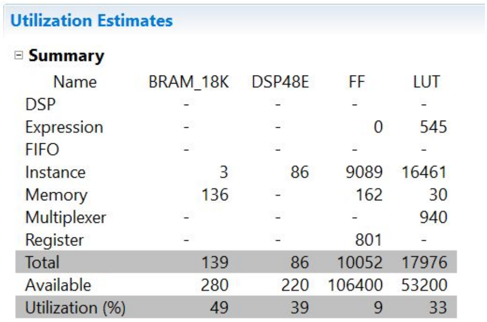
Memory

Memory	Module	BRAM_18K	FF	LUT
active_pot_U	LIF_Network_activibs	0	64	4
layer2_D_U	LIF_Network_layercud	0	32	4
layer2_Pmin_U	LIF_Network_layerdEe	0	32	4
layer2_n_pth_U	LIF_Network_layereOg	0	8	1
layer2_t_ref_U	LIF_Network_layerfYi	0	3	1
layer2_t_rest_U	LIF_Network_layerg8j	0	16	1
layer2_P_U	LIF_Network_layerhbi	0	64	4
temp_spike_U	LIF_Network_layerhbi	0	64	4
temp_train_complete_U	LIF_Network_temp_kbM	256	0	0
train_temp_U	LIF_Network_trainlbW	2	0	0
syntemp_U	LIF_Network_trainlbW	2	0	0
weight_matrix_U	LIF_Network_weighbkb	16	0	0
Total	12	276	283	23

   It can be seen that the maximum utilization is of the block RAM and the main reason behind this is the temporary array required in the design to store the input from the dma.
   The speedup attained in this implementation over software implementation on the ARM processor for all of the six classes has been summarized in the table below.

Class	Software Execution Time	Hardware Execution Time	Speedup
1	3.4366652966	0.0948166847	36.24537
2	3.4633243084	0.0945048332	36.64706
3	3.4377005100	0.0919346810	37.39286
4	3.3822371960	0.0874323845	38.68403
5	3.5381476879	0.0918786526	38.50892
6	3.4209511280	0.0865030288	39.54718

2. 3 layer SNN Implementation for 10 digit classification
   As we were unable to fit the 5 layer high accuracy SNN python implementation, we compromised accuracy for better performance and area efficiency. In the three layer configuration, we ported the network that consisted of a hidden layer of 64 neurons. From the utilization estimate outlined below, it can be seen that the network occupies less than half of the available resources, even in terms of memory. The average execution time for this implementation is 0.2 seconds which is almost double that of the two layer implementation. This implementation was further optimized by rearranging the different operations within the network to make use of the efficient dataflow existing in the implementation. Additionally, some of the loops within the design - mainly the portion responsible for computing matrix multiplication were pipelined to improve the overall performance of the hardware implementation.

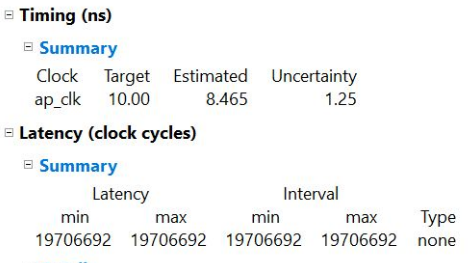


Utilization Estimates

⊟ Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	545
FIFO	-	-	-	-
Instance	3	86	9089	16461
Memory	136	-	162	30
Multiplexer	-	-	-	940
Register	-	-	801	-
Total	139	86	10052	17976
Available	280	220	106400	53200
Utilization (%)	49	39	9	33

The latency for the optimized and baseline implementation respectively are covered in the figure below. As the clock cycle time remains consistent, the main focus is on the latency. It can be seen that there is an improvement of 1.86 in the performance.



⊟ Timing (ns)

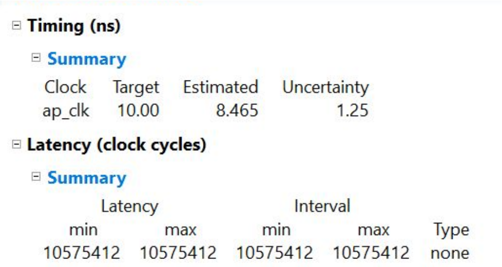
⊟ Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.465	1.25

⊟ Latency (clock cycles)

⊟ Summary

Latency		Interval		
min	max	min	max	Type
19706692	19706692	19706692	19706692	none



⊟ Timing (ns)

⊟ Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.465	1.25

⊟ Latency (clock cycles)

⊟ Summary

Latency		Interval		
min	max	min	max	Type
10575412	10575412	10575412	10575412	none

Milestones:

Milestone 0 - A report on comparison of SNN models with reasoning for the chosen model ✔
Progress: Done
Deliverable: Milestone 0 Report

Milestone 1 - Demonstrating working simulation of a simple block on Vivado & implementation on FPGA board! ✔
Progress: Done
Deliverable: PYNQ Board Setup Report

Milestone 2 - Building software implementation of SNN for binary image classification ✔
Progress: Done
Deliverable: Code, trained weights and output spikes - Software Implementation Report

Milestone 3 - Building software implementation of SNN for multiclass image classification ✔
Progress: Done
Deliverable: Software Implementation Report

Milestone 4- A C/C++ implementation of the SNN network architecture suitable for HLS ✔
Progress: Done
Deliverable: Final C/C++ Code and Classified Image Results

Milestone 5 - HLS and testing to attain working baseline implementation on FPGA board for binary class classification ✔
Progress: Done (Implemented 6-class implementation instead)
Deliverable: Demonstration of successful 6-class classification on PYNQ

Milestone 6 - HLS and testing to attain working implementation on FPGA board for multiclass classification (10-class) ✔
Progress: Done
Deliverable: Demonstration of successful multiclass classification on PYNQ

Milestone 7 - Performance improvement (10x speedup) of hardware implementation
Progress: Done ✔
Deliverable: Report and Graphs showing performance improvement

Milestone 8 - Final report and video ✔
Progress: Done
Deliverable: A report on all implemented network architectures

Assigned grades for milestones in mid quarter

Grade	Milestones	Completion
B+	2,4,5	Completed
A-	2,3,4,5,6	Completed
A	2,3,4,5,6,7	Completed

Conclusion:

We were able to successfully implement two python software implementations for the spiking neural network consisting of the LIF neuron model. Additionally, we were able to port these implementations to C++ which were then passed through the entire HLS flow successfully. We achieved two of our key objectives:

1. Attain greater than 95% accuracy for the python implementation of the spiking neural network
2. Achieve greater than 10x (almost 40x over ARM processor) speedup for hardware implementation

There is however a lot of scope for improvement in the design in general. Specifically, the hardware implementation can benefit from efficient tackling of the loop that iterates over the timesteps. It is the major performance bottleneck that currently exists in the design and hampers performance.

References:

1. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4522567/ - Unsupervised learning of digit recognition using spike-timing-dependent plasticity
2. https://github.com/Shikhargupta/Spiking-Neural-Network Pure Python implementation of Spiking Neural Network
3. https://towardsdatascience.com/spiking-neural-networks-the-next-generation-of-machine-learning-84e167f4eb2b - Background on Spiking Neural Networks
4. https://github.com/nengo/nengo Python library for simulating the spiking neural networks
5. https://bitbucket.org/akhodamoradi/pynq_interface/wiki/AXI_DMA Tutorial for DMA operation on PYNQ