Acceleration of LRCN Networks on a FPGA

Matthew Portnoy, Hrushikesh Patil
Department of Electrical and Computer Engineering
Stony Brook University, Stony Brook, NY, USA
{matthew.portnoy, hrushikesh.patil}@stonybrook.edu

I. Introduction

Long-term Recurrent Convolutional Networks (LRCNs), are a class of architectures usually used for visual recognition and description which combines convolutional layers and long-range temporal recursion and is end-to-end trainable. Such networks are combinations of CNNs and LSTMs. Since videos include both spatial as well as temporal components, using LRCN has proved to be much efficient than other available counterparts such as 3D CNNs. The LRCN networks aim to leverage the power of CNNs to recognize spatial patterns and use the LSTMs to identify the temporal changes in datasets. However, this comes at a huge computational cost. Making these networks unsuitable of many real time applications like driverless cars, autonomous robots.

Recurrent neural network (RNN), is a deep learning algorithm that takes advantage of previous outputs as inputs for prediction. RNNs show a strong ability to learn and predict sequential data. But there are still some flaws, and for that reason we use Long Short-Term Memory (LSTM), which is an extension of an RNN. LSTMs have gates that know which data to keep and the ones to forget. So, this method is better for more complex solutions. Unfortunately, machines such as CPUs have trouble dealing with the high performance and parallelism. So, for this reason we chose an FPGA, which is a device that is great for high performance. And would be good for acceleration of complex data.

Ideally our approach would be to accelerate both CNNs and LSTM networks used in the LRCN. However due to limited time we plan to accelerate an LRCN network by using an FPGA, but only accelerate the LSTM part of the LRCN. The motivation for this work is to understand how FPGAs can make LRCNs applicable in a real time scenario.

First, we took a pre-trained LRCN network on UCF101 dataset. Then, we extracted the weights of the LSTM layer, and transferred it to the DRAM of the FPGA. From there, the weights were transferred through stream the weights using DMA. Then we designed the architecture for 1 layer of LSTM on an FPGA. The FPGA does calculations to receive outputs that is sent back to the main processor. This process is done until there is a least amount of error, and therefore the data is accelerated. We used a MiniZed board for our design and is displayed in Figure 1.

Figure 1. MiniZed board that was used and includes a Zynq 7Z007S processor [1].

II. Background

In this section, we discuss in detail information about RNN, LSTM, and LRCN.

A. RNN

Recurrent Neural Network (RNN), was first invented to deal with sequential data. To do this, the network had to be able to learn from its previous states. This differs from a feed-forward network because there is a connection in the hidden layer from the output of each neuron to the input of each neuron. This creates a form of memory and allows the model to learn relationships in sequence. The output h_t of a hidden recurrent layer is shown as:

$$h_t = \sigma(W_{xh}x_t + W_{hh}h_{t-1} + b_h) \quad [2]$$

Where W_{xh} represents the weight matrix between the input layer and the hidden layer. W_{hh} represents the weight matrix of the recurrent connections between two hidden layer states at two consecutive time steps. The vector b_h represents the hidden bias vector. The function σ is the hidden layer activation function, an element-wise sigmoid () function.

B. LSTM

Although RNNs can learn from prior information, research later showed that it could not maintain long-term memory and have sustainable prediction accuracy. To overcome this problem a model called Long Short-Term Memory (LSTM) was invented. LSTMs uses gates that know

which data to keep and the ones to forget. This is one of the main reasons why LSTMs are good at learning from prior information. The architecture of LSTM is shown in Figure 2 below.

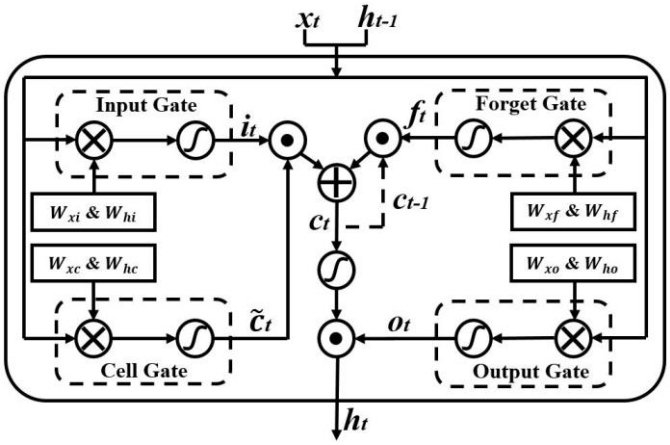

Figure 2. LSTM cell [2]

An LSTM model has four gates and those are Input Gate, Forget Gate, Output Gate, and Cell Gate. The Cell Gate is also known as the cell state, and it influences all the other gates. And takes in the parameters x_t and h_{t-1}. The forget gate decides based on x_t and h_{t-1} to determine which parts of the cell state to forget by multiplying with a 0 or small number. The input gate decides based on x_t and h_{t-1} to determine which parts of the cell state to update. The output gate decides based on x_t, h_{t-1}, and c_t to determine the output. The equations that describe these gates and how they function are shown below:

$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + b_i)$$

$$f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + b_f)$$

$$\tilde{c}_t = tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_f)$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$$

$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + b_o)$$

$$h_t = o_t \odot tanh(c_t)$$

Where σ is the sigmoid() function, and f, i, o, $\tilde{c}_t$ represent the output vectors of forget, input, cell, and output gates, respectively. The W terms denote weight matrices. W_{xi} is the weight matrix between input gate i and input vector x. The vector b denotes the bias vectors, where b_i is the bias vector of input gate i.

C. *LRCN*

A typical LRCN is implemented using AlexNet for CNN and LSTM portion. A LRCN uses 2.22 billion floating point operations and 86.56M synapse weights for processing just one video frame. Video frames are entered sequentially into the system and first processed by CNN for the extraction of visual features. The output is a vector with 1000 dimensions with each dimension representing a category of objects. This vector is passed to the RNN module to generate proper descriptions. The RNN module produces a separate word in each iteration. By combining these different neural networks together, LRCN becomes an end-to-end model for video content description. An example LRCN is shown in Figure 3.

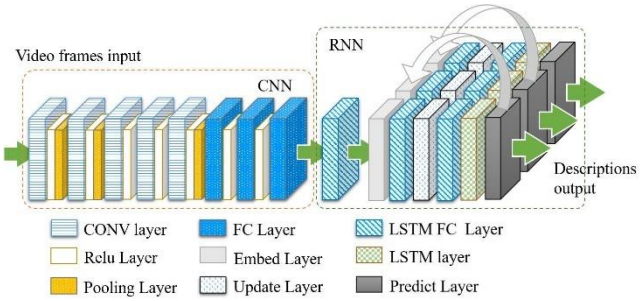

Figure 3. LRCN structure [3]

III. SOFTWARE IMPLEMENTATION

In software implementation we have a pretrained model of LRCN. The LRCN in question is a combination of Resnet152 CNN and LSTM network. We have used PyTorch to evaluate the LRCN. The input to the LRCN network is the videos from the UCF101 [6] dataset. The videos are shaped as (t-dim, channels, x-dim, y-dim) = (28, 3, 224, 224). Where 28 is the no of frames of the video, 3 is the number of channels (RGB) in each frame and 224x224 pixels is the size of each individual frame. This is done to match the input size ResNet152 which is tensors of size (3, 224, 224).

The software evaluation is divided into parts.

1. Computing the CNN outputs
2. Computation of LSTM output
3. FCN evaluations

The CNN computes the spatial features from each frame and outputs a vector of length 512 for each frame. The python script evaluates the spatial features for each frame and hence after evaluations of each frame we have the output as a tensor of size (28, 512).

The output of CNN is given to LSTM network. The LSTM network is followed by a Fully Connected Layer (FCN) which has 101 outputs.

In our system we propose accelerating the LSTM layer by use of FPGA hardware accelerator. Hence, the FPGA takes the o/p generated by the PyTorch python script in Step 1 and begins computing the output for LSTM layer.

The step 2 happens in the FPGA and is explained in the next section. The o/p from step 2 are returned back to the python program running on the CPU via UART.

In step 3 the FCN is evaluated with the o/p received from the accelerator as the i/p.

The complete system flowchart is given below.

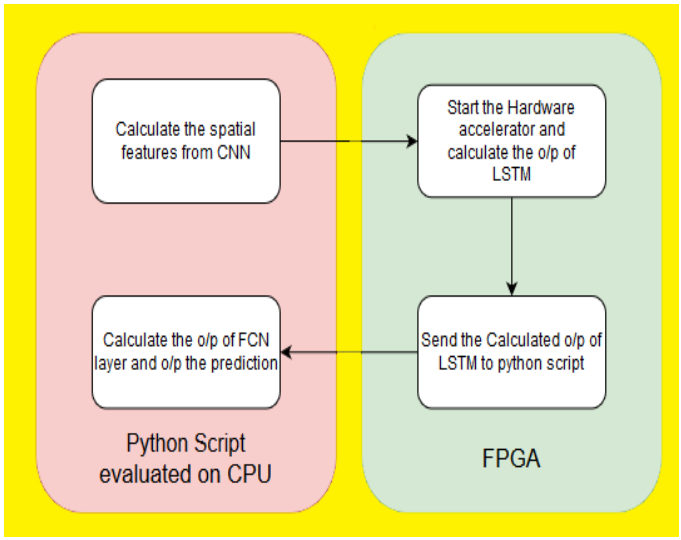

Figure 4. Flow diagram of the system

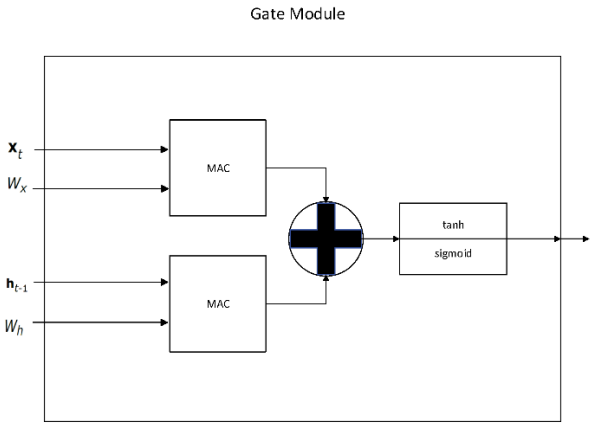

Figure 5. Gate Module

IV. Hardware Implementation

For the hardware design, we are implementing one layer of LSTM. Our design was inspired by [4]. The operations that were used in this design are matrix-vector multiplications and non-linear functions. The nonlinear functions that we chose are hyperbolic tangent and sigmoid. The matrix vector multiplication is computed by a MAC unit. And the nonlinear functions are implemented using a piecewise linear approximator.

We have two different types of modules implemented in this design and those are gate and Ewise. The gate module is shown in Figure 5. Our gate module comprises of 2 MAC units, an adder, and a sigmoid/tanh block. Our MAC takes in a vector stream and weight matrix row stream. The same vector stream is multiplied and accumulated with each weight matrix row to produce an output vector with same size of the weight's height. Then the MAC is cleared after computing each output element to avoid accumulating previous matrix rows computations. The bias b can be added in the multiply accumulate by adding the bias vector to the last column of the weight matrix. The results from the MAC units are added together. The adder's output goes to our sigmoid/tanh block, which is an element wise non-linear function.

The non-linear function is segmented into lines y = ax+b, with x limited to a particular range. Each line segment is implemented with a MAC unit and a comparator. The MAC multiplies a and x and accumulates with b. The comparison between the input value with the line range decides whether to process the input or pass it to the next line segment module. This algorithm is similar for sigmoid and tanh. And in our gate module it chooses between the two.

We break down the operation of gate in a LSTM into separate modules. In these modules, the MAC units perform 32W-bit multiplication that results into 32-bit values. The addition is performed using 32-bit values to. All that is left are some element wise operations. To do this, extra multipliers and adders were added into a separate module shown in Figure 6. The LSTM module has three gate blocks. The gates are pre-configured to have a non-linear function (tanh or sigmoid). The LSTM module is shown in Figure 7.

In the first and second stage, four gate modules (16 MAC units) are running in parallel to generate two internal vectors (i_t, $\tilde{c}_t$, f_t and o_t), which are stored into a FIFO for the next stages. Then we created an ewise module that consumes the FIFO vectors to output the h_t and c_t back to main memory. After that, the module waits for new weights and new vectors, which can be for the next layer or next time step.

In the driving software for the processor, because of the recurrent nature of LSTM, c_t and h_t becomes the c_{t-1} and h_{t-1} for the next time step. Therefore, the input memory location for c_{t-1} and h_{t-1} is the same for the output c_t and h_t. Each time step c and h are overwritten. This is done to minimize the number of memory copies done by the CPU.

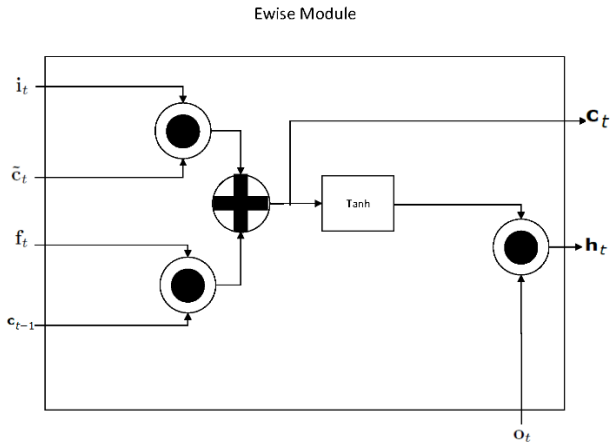

Figure 6. Ewise Module

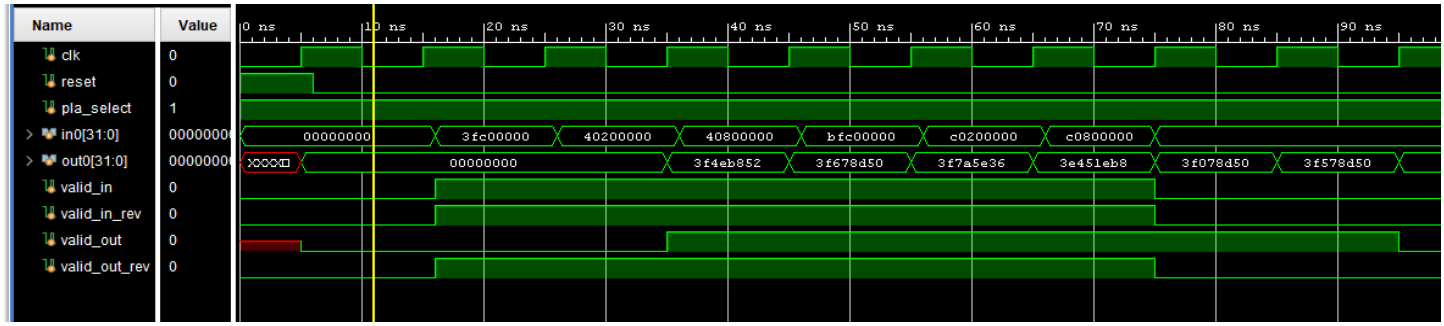

Figure 8. Sigmoid Simulation Results

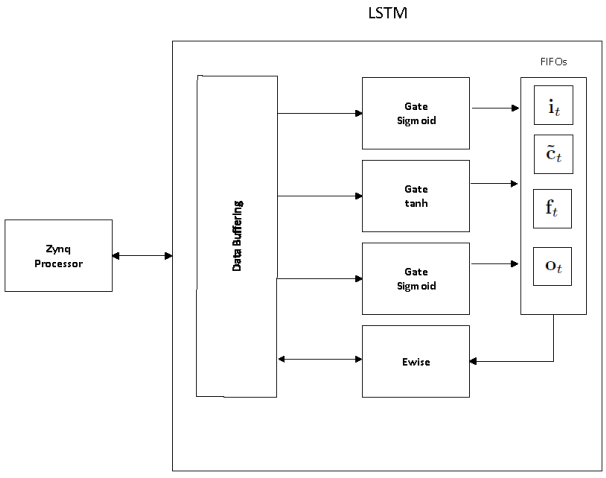

Figure 7. LSTM Module

V. Evaluation

A. *LRCN Model and Pytorch Results*

Initially we tried to train our own network but the data itself was in order of tens of Gbs even for the smallest of the dataset (UCF11). Hence, we found a LRCN pretrained on UCF101 dataset. The LRCN is a combination of ResNet152 and LSTM layer. The LRCN model is divided into two different sub models.

1. CNN encoder
2. LSTM decoder

The CNN encoder is a pretrained ResNet152 which is pretrained on ImageNet data. The top layer (FCN layer) of CNN is replaced with two fully connected layers of size 512 each. The output of the CNN encoder is a vector of length 512.

LSTM encoder has 512 nodes followed by FCN layer and an output layer of 101 outputs.

Actual Implementation on FPGA

B. *LSTM Model and FPGA Results*

Gate Module

From the planned gate module only the MAC units were implemented. The input datatype was 32-bit floating point numbers. The input weight matrices were 512*512, while input and output vectors were of length 512. Considering for every 512*512 multiplies and add, there were only 1 addition of outputs of matrix vector products of weights and vectors (x & h) and bias vector followed by application of tanh/sigmoid function. We decided to offload these computations to PS in favor of a "embarrassingly parallel" hardware acceleration. In this case we first implemented 2 MAC units in parallel and then we scaled up to 8 MAC units in parallel. In 8 MAC unit we were unable to fit the entire implementation on the MiniZed due to inadequate LUTs. The metric used to evaluate the implementation was the runtime of the entire program. To check the correctness of the implementation we set all input vector and weight matrix values to 1. We compared each implementation by run times, and included runtimes of embedded CPU which was simply the PS of Minized and Nvidia K80 GPU [Refer Figure 9]. We also ran an implementation where we simply loaded the weights to BRAMs [Refer Table 1]. From the table we can see that 68 seconds are taken just to load and read out the values and the PS takes less than 2 secs for 8 MAC implementations to calculate matrix vector product. Refer Table 2 for implementation details of 8 MAC unit design.

	Nvidia K80	Emebedded CPU (PS of MiniZed)	2 MAC Hardware Accelerator at 25 MHz	2 MAC Hardware Accelerator at 30 MHz	8 MAC Hardware Accelerator at 30 MHz	Only Loading of BRAMs with input matrices
Time in Seconds	0.251	5.02	88	71	69.56	68.11

Table 1

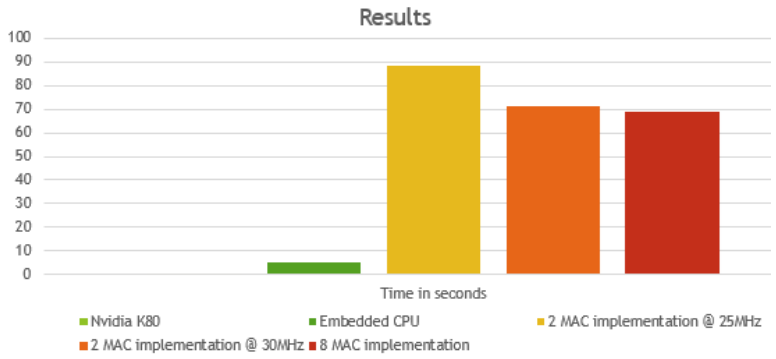

Figure 9

Utilization	LUT	FF	BRAM	DSP
Percentage	96.89	50.59	88	6.06

Table 2

Sigmoid

In the code we checked if the input was between the range values, and if it was, we assigned specific slope and bias values. This was then asserted into a floating point multiply and adder ip, which did the calculation for y =mx+b.

For this example we used two ranges c0,c1. Those ranges were assigned the values c0=1.67, c1=3.33. So there were three possibilities x<c0, c0<x<c1, and x>c1. The three different slope and bias values were the following m0 = 0.205, m1 = 0.075, m2 = 0.017, b0 = 0.5, b1 = 0.717, b2 = 0.9.

In the simulation we used 8 input values 1.5,2.5,4 ,-1.5,-2.5,-4 with the following output values 0.8075, 0.9045, 0.978, 0.192, 0.5295,0.649. Figure 8 shows our simulation and the output values in hexadecimal.

VI. Conclusion and Future Work

In this paper, we proposed an FPGA-based accelerator for an LRCN. We implemented this design on a AVNET MiniZed board. The performance of FPGA in accelerating the LSTM is directly dependent on memory. Since we are only able to fit 115 vectors of length 512 in BRAM the acceleration is hindered by the constant fetching of data from PS to PL. As from the data above it can be seen that 68 secs were required to constantly fetch and store data while PL runtime was only 1 seconds. If sufficient memory is available that prevents the frequent fetching of data from PS, we can see that our 8 MAC implementations (1 seconds of PL runtime) is faster than that of embedded CPU runtime. Considering the fact that our implementation did not have any sort of pipelining in Floating Point fused multiply and add (Latency set to 0), our clock frequency was only 30 MHz which can be improved by adding pipelining thereby significantly improving the performance.

The future work would be to continue the full LSTM module in the FPGA, the combination of the gate and ewise module. If we were to continue this project, we can add the other portions of the LRCN model.

VII. References

[1] AVNET. “MiniZed.” Zedboard, 2018, zedboard.org/product/minized.

[2] Y. Guan, Z. Yuan, G. Sun and J. Cong, "FPGA-based accelerator for long short-term memory recurrent neural networks," 2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC), Chiba, 2017, pp. 629-634.

[3] X. Zhang et al., "High-performance video content recognition with long-term recurrent convolutional network for FPGA," 2017 27th International Conference on Field Programmable Logic and Applications (FPL), Ghent, 2017, pp. 1-4.

[4] A. X. M. Chang, B. Martini, and E. Culurciello. “Recurrent neural networks hardware implementation on FPGA.”, *CoRR*, abs/1511.05552, 2015.

[5] Video Classification, https://github.com/HHTseng/video-classification

[6] Khurram Soomro, Amir Roshan Zamir and Mubarak Shah, UCF101: A Dataset of 101 Human Action Classes From Videos in The Wild, CRCV-TR-12-01, November, 2012.

VIII. Contributions

The software implementation (pytorch code, extraction of weight matrices and input vectors), MAC unit Verilog code, C code and Vivado block diagram for MAC unit was done by Hrushikesh Patil. Verilog code and simulation of Piecewise Linear approximator along with majority of report, initial Vivado block diagram for overall design (gate and ewise) was done by Matthew Portnoy. Literature review, presentation was done together.