FPGA Accelerator for RNN-based Prefetcher

EE-599 Spring 2020 Project Final Report

Pengmiao Zhang
ECE Department
University of Southern California
Los Angeles, CA, USA
pengmiao@usc.edu

Sasindu Wijeratne
ECE Department
University of Southern California
Los Angeles, CA, USA
kangaram@usc.edu

Viktor K. Prasanna
ECE Department
University of Southern California
Los Angeles, CA, USA
prasanna@usc.edu

ABSTRACT

This project aims to design an RNN-based prefetcher accelerated by FPGA that can achieve a high hit rate, a high computation speed, on-line update, and generalizability. While machine learning-based prefetching appeared recently, the low computation speed makes it impractical. FPGA accelerator provides us with a good direction for this problem. To achieve generalizability and reduce prefetcher size, the applications are clustered using a combination of A Variational Recurrent Auto-Encoder and k-means. One model will be trained for each cluster. LSTM (Long-Short Term Memory) is selected as the recurrent layer of the model. To implement the RNN-based prefetching on FPGA, model compression is necessary. The input and output of the model is encoded into binary which results in $O(n/\log n)$ compression factor. The compressed RNN models will be trained offline, then the models should be designed as specific hardware circuits on FPGA and the trained weights should be stored in LUT RAM for further updates when hit rate drops. Experiments involving prefetcher performance, FPGA resource utilization, FPGA speedup are conducted. Our model shows a 73.1% accuracy on memory prediction and its FPGA implementation achieves an acceleration in the order of 6000× comparing to CPU computation.

CCS CONCEPTS

• Computer Systems Organization • COMPUTER SYSTEM IMPLEMENTATION

KEYWORDS

RNN, FPGA, prefetching, accelerating

ACM Reference format:

Pengmiao Zhang, 2020. FPGA Accelerator for RNN-based Prefetcher: EE595 Spring2020 Project Final Report. In *Proceedings of ACM Woodstock conference (WOODSTOCK'18). ACM, New York, NY, USA, 2 pages.* https://doi.org/10.1145/1234567890

1 Introduction

Prefetching techniques can reduce memory latency as well as cache misses through fetching data before needed by a program. Prefetching can be achieved by both hardware and software. Software-based prefetching is typically accomplished by having the compiler analyze the code and place additional "prefetch" instructions into appropriate places in code.[8]. Compiler directed prefetching is widely used within loops with a large number of iterations. Software prefetchers usually work well only for "regular access patterns" and can cause waste of instruction execution bandwidth. Hardware methods prefetch data based on past access behavior using specialized hardware that observes access patterns. Classical methods include Stream Buffers [9] and Stride Prefetching [1]. While hardware prefetching requires more complex hardware mechanisms, it avoids waste of instruction execution bandwidth caused by inserting prefetching instructions in codes and has less CPU overhead [2].

Recently, machine learning-based prefetching methods are gaining increasing attentions [6]. Inspired by the extraordinary performance in time series data prediction, some efforts of applying recurrent neural network (RNN) models, especially LSTM (Long short-term memory) [4], as prefetcher algorithms have been conducted [10]. However, due to the high computation complexity of neural networks, it is hard to perform timely prefetches. Though an RNN based prefetcher can achieve higher memory access prediction accuracy, it may result in late prefetch, which not only defeats the purpose of fetching data before it is needed but causes even lower IPC by calling prefetching function.

One way to solve this dilemma is by using FPGA to accelerate the computation of RNN based prefetchers. FPGAs have shown a great improvement in both power consumption and performance in Deep Neural Networks (DNNs) applications compared to GPU [7]. Due to the feature of programmability, reconfigurability, and parallelizability of FPGA, it is possible to achieve complex software prefetching algorithms using hardware with higher speed and lower power consumption, thus making the RNN-based prefetching algorithm practical. Especially, for a data center application, a specific memory control unit can be applied using an FPGA-based prefetcher with RNN model to predict memory accesses.

Some related topics have been studies involving RNN prefectures and the application of RNN models on FPGA. Srivastava et al. [10] explored the performance of the compressed LSTM model in predicting memory performance, but they did not apply this method on real prefetchers. Guan et al. [5] designed an

LSTM accelerator on FPGA without compression. Wang et al. [11] implemented an LSTM model on FPGA compressed using Block-Circulant Matrix.

2 Problem Definition

2.1 Objective

This project aims to design an RNN-based prefetcher on FPGA, which can:

1. Predict memory access with high accuracy;
2. Achieve generalizability by applying one model to several applications;
3. Achieve high computing speed on FPGA compared to CPU implementation;
4. Update model weights online when application pattern changes.

The proposed prefetcher can be applied to a data center memory control unit. Under this situation a separate FPGA system for prefetching is practical and a data center is sensitive to both latency and power consumption.

2.2 Hypothesis

Implementing LSTM-based neural network on FPGA will significantly accelerate its inference computation time compared to CPU implementation.

2.3 Key Idea

1. Cluster the PARSEC traces using Variational Recurrent Autoencoder;
2. Train a double-compressed LSTM-based neural network model for each cluster of traces which could be applied to all applications in the cluster;
3. Implement the trained models on FPGA:
   - The weight of model can be stored in LUT RAM;
   - Mapping the trained embedded vector to input data;
   - Parallelly process the four gates in a LSTM cell using Horner's Rule;
   - Using binary adder tree to complete the matrix multiplication in the last dense layer;
   - Activation functions are approximated through piecewise polynomial expressions.
4. Design a finite state machine for the system to update the weights online when hit rate drops.

3 Design Details

3.1 Dataset

PARSEC benchmark [14] provides 13 applications that we can use for memory pattern study. The Intel Pin [15] tool was used to obtain memory access traces for each application. The features of input will be the jump of the virtual memory addresses between two adjacent instructions, referred to as delta. The output will be the next memory delta.

3.2 Architecture Overview

Figure 1 illustrates the overview architecture of the proposed FPGA accelerated prefetcher. There are two parts in the framework. Part 1 is responsible for the offline trace and model preparation. It consists of the application traces and the RNN-based predictive model. Part 2 involves the online working hardware in a computer system, which consists of not only the processor, memory hierarchy and the controller, but also an FPGA implanted into the system. This framework in part 2 is emerging in state-of-the-art technologies including 3D-stacked DRAM tightly integrated with FPGAs, such as in Xilinx VU37P FPGA and Intel Stratix 10 MX FPGA [10].

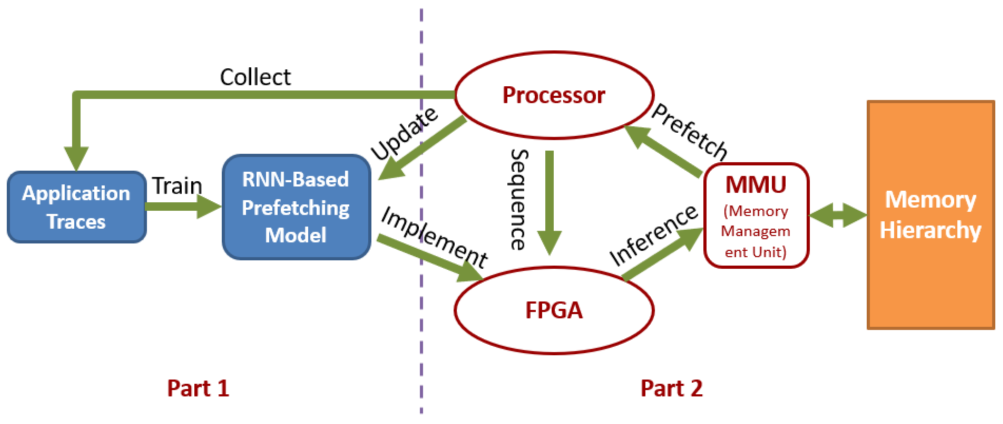


Figure 1: Architecture of Proposed Prefetcher

The proposed prefetcher uses an RNN-based memory predictive model as the prefetching algorithm. First, the application traces should be extracted from the running processor using Intel Pin tool. These traces then can be used to analyze the memory pattern. Based on the memory traces, an RNN predictive model can be trained offline. To achieve generalizability, the RNN model is supposed to be trained using several applications that share similar patterns. In this way, the weight and structure of an inference model is ready for prefetching job.

This inference model is supposed to be implemented on an FPGA chip (or on embedded FPGA) which accelerates the computation process and makes the RNN-based prefetching practical. The model on FPGA receives the information of previous memory accesses sequence and predict the potential trend of the memory trace, then transmit the forecasting result to the memory management unit (MMU). MMU can help fetch the data according to the FPGA prefetch's prediction before the data is requested by the program.

In addition to the one-way implementation, the feedback and update of the model is important to keep the system working well. The pattern of the system may change by time due to software update, a shift of user habit or other external influence factors. Therefore, the new traces should continue to be collected and the model should be updated when hit rate drops.

3.3 Model Design

3.3.1 Clustering. Since the raw sequence is long (10k), we use feature-based methods to cluster the application traces instead of directly dealing with the raw data. Therefore, we reduce the dimension first through learning from the long access sequence, then use basic clustering method to acquire several clusters.

A Variational Recurrent Auto-Encoder (VRAE) [12] model is used to reduce the dimension of the raw traces. VRAE is a generative model for unsupervised learning of time-series data. It combines the strengths of recurrent neural networks (RNNs) and stochastic gradient variational bayes (SGVB) [13]. Since VRAE uses stateful LSTM, it is capable to deal with long sequence time series data by splitting the long sequence into pieces of batches. We split each of the 13 PARSEC traces into 10 batches and use VARE to reduce the dimension from 10000 to 25. K-means is used to further cluster all these traces, the PCA on the VRAE dimension reduction result is shown in the Figure 2.

Table 1: Application Clustering

Cluster	Applications
Cluster 1	vips, dedup, streamcluster, freqmine swaptions, bodytrack, raytrace, canneal, fluidanimate, facesim
Cluster 2	Blackscholes, ferret
Cluster 3	X264

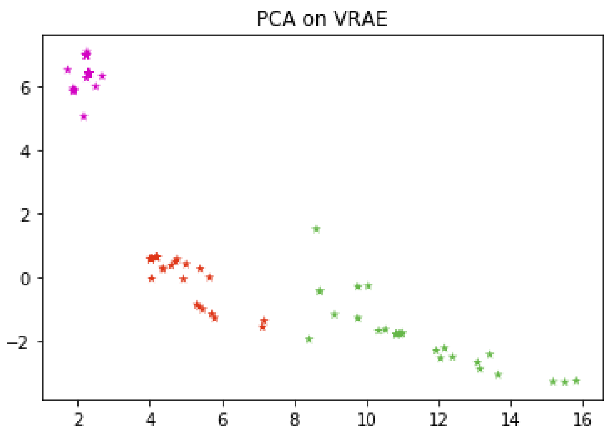


Figure 2: PCA on VRAE Trace Dimension Reduction

Through voting strategy, the applications can be divided into tree clusters as is shown in table 1. In this paper, we select the Cluster 1 to train a model and implement the model on FPGA.

3.3.2 Compressing. For an LSTM model to be realistically used for prefetching, it needs to have low latency and should require small amount of computation. These factors are closely related to the size (number of parameters) of the model. As shown in [10], the size of the simple LSTM model for memory access prediction is dominated by the dense last layer. Few thousands of output layers may lead to slowing down of inference due to large number of parameters in the final layer. Instead of using the deltas (jumps in memory accesses) directly as labels, the approach in [10] predicts the binary representation of deltas, converting the problem from a single label (1 out of n) prediction problem to a multi-label prediction problem (log*n* labels).

3.3.3 Model Training. The model is designed as a combination of embedding layer, an LSTM layer with dropout and a dense layer. This simple structure is necessary to keep the model size small and to achieve fast inference computation. The model structure and the number of parameters for each layer is shown in table 2. The training process of the model is shown in figure 3. After 20 epochs of training, the validation accuracy reaches 0.731.

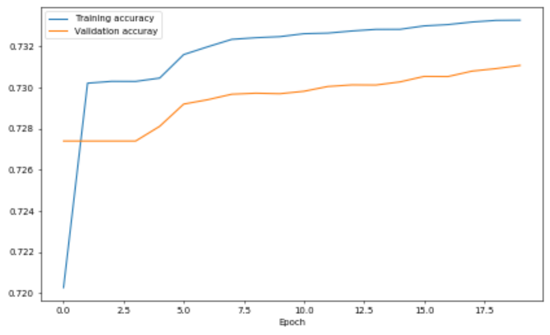


Figure 3: Model Training and Validation Accuracy

Github link: https://github.com/pengmiao-usc/EE-599_PengmiaoZhang_7865959675/tree/master/Final_Project

Table 2: Model Structure

Layer	Output Shape	Param #
Embedding	(None, 48, 8)	16
LSTM	(None, 32)	5248
Dropout	(None, 32)	0
Dense	(None, 16)	528

3.4 Model Implementation on FPGA

3.4.1 Platform and language. This project is designed on the ZYNQ-7 ZC702 Evaluation Board (xc7z020clg484-1) using Verilog. The number representation system used for this network is a signed fixed point. The are 7 integer bits (one of which is the sign bit), and 11 fractional bits, adding to a total of 18 bits.

3.4.2 Model framework. The framework is shown in figure 4, which is consistent with the model in table 2. However, one major modification is conducted for the convenience of FPGA implementation. Instead of passing the input vector of 48 dimensions at one stroke to the embedding layer, the vector is split into 48 one-dimension one-bit data then sent to the model sequentially. This is because the LSTM cells process the time-series data recurrently, one-stroke vector data cannot help but increases the burden of embedding layer who need to handle both data storage and clock control. Through this structure modification, the embedding layer does not need to worry about sending the data at each time step to the LSTM layer. The whole model will work in a pipeline by constantly receiving single time-step input. This modification involves the notification of the complete transmission of a full input vector in 48 time-steps, which can be easily achieved by counting the input bits.

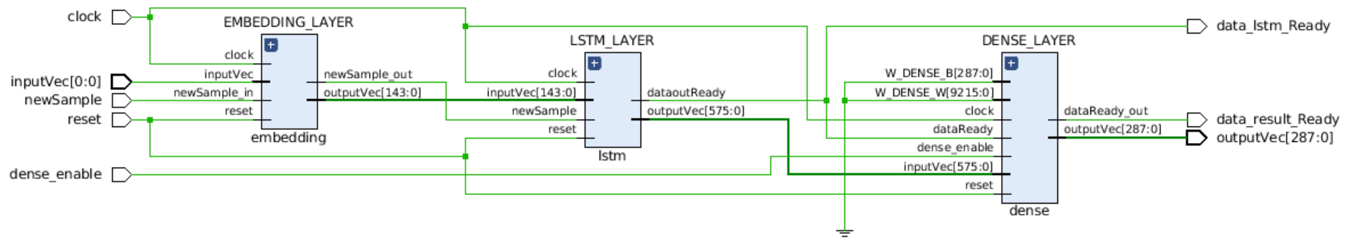


Figure 4: Model Framework

3.4.3 Weight Storage. Since the parameter number is small, they can be stored in LUT RAM. The weight of embedding layer and LSTM layer are stored in LUT RAMs distributed in each block. The RAM is designed to be column by column, each consists one vector of the weights in the shape of layer hidden dimension. This structure facilitates the vector-matrix multiplication in neural network. The weights of the dense layer are stored in normal register in order to achieve parallel acquisition.

3.4.4 Embedding layer. Since the input of access deltas are encoded into binary bits, the vocabulary is only two: 0 and 1. Each word maps to a trained embedding vector. Thus, the LUT RAM for embedding weight is in a shape of two columns each contains a 10-dimension vector. When an input bit is received, this layer will output the corresponding embedded vector.

3.4.5 LSTM layer. The structure of an LSTM cell is presented in [14]. The operation of each set of gates of the layer is given by the following set of equations, where vectors are represented by bold, lower-case letters, and matrices are bold, upper-case letters. $\mathbf{W}$, $\mathbf{R}$ and $\mathbf{b}$ are weight matrix. $\mathbf{x}^{(\mathbf{t})}$ and $\mathbf{y}^{(\mathbf{t-1})}$ are the input of the LSTM cell which represent new time-step input vector and the present time-step output vector respectively.

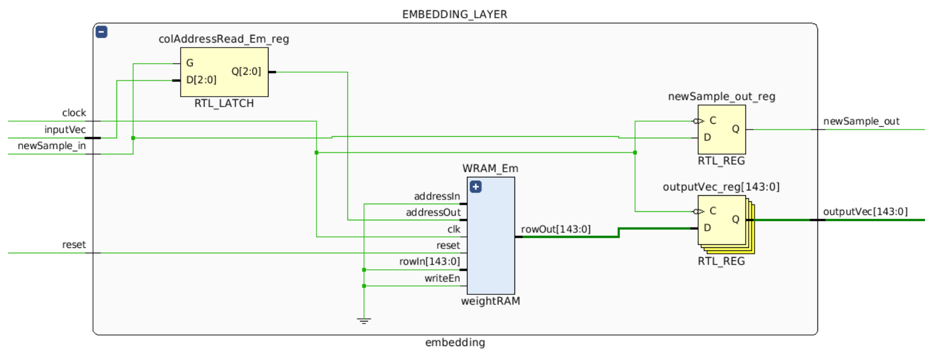


Figure 5: Embedding Layer

$$
\begin{aligned}
\mathbf{i}^{(\mathbf{t})} &= \sigma\left(\mathbf{W_i}\mathbf{x}^{(\mathbf{t})} + \mathbf{R_i}\mathbf{y}^{(\mathbf{t-1})} + \mathbf{p_i} \odot \mathbf{c}^{(\mathbf{t-1})} + \mathbf{b_i}\right) \\
\mathbf{z}^{(\mathbf{t})} &= g\left(\mathbf{W_z}\mathbf{x}^{(\mathbf{t})} + \mathbf{R_z}\mathbf{y}^{(\mathbf{t-1})} + \mathbf{b_z}\right) \\
\mathbf{f}^{(\mathbf{t})} &= \sigma\left(\mathbf{W_f}\mathbf{x}^{(\mathbf{t})} + \mathbf{R_f}\mathbf{y}^{(\mathbf{t-1})} + \mathbf{p_f} \odot \mathbf{c}^{(\mathbf{t-1})} + \mathbf{b_f}\right) \\
\mathbf{o}^{(\mathbf{t})} &= \sigma\left(\mathbf{W_o}\mathbf{x}^{(\mathbf{t})} + \mathbf{R_o}\mathbf{y}^{(\mathbf{t-1})} + \mathbf{p_o} \odot \mathbf{c}^{(\mathbf{t})} + \mathbf{b_o}\right) \\
\mathbf{c}^{(\mathbf{t})} &= \mathbf{i}^{(\mathbf{t})} \odot \mathbf{z}^{(\mathbf{t})} + \mathbf{f}^{(\mathbf{t})} \odot \mathbf{c}^{(\mathbf{t-1})} \\
\mathbf{y}^{(\mathbf{t})} &= \mathbf{o}^{(\mathbf{t})} \odot h\left(\mathbf{z}^{(\mathbf{t})}\right)
\end{aligned}
\tag{1}
$$

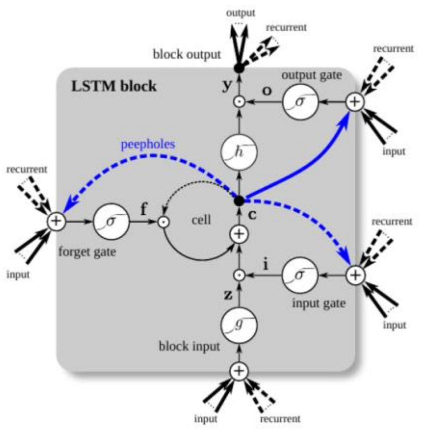


Figure 6 LSTM Cell Structure

3.4.5.1 LSTM components. The role and relevance of the main components can be summarized as follows.

- *Input Gate.* The relative importance of each feature of the input vector at time **t**, $\mathbf{x}^{(\mathbf{t})}$, and the output vector at the previous time step, $\mathbf{y}^{(\mathbf{t-1})}$,are weighed, producing an output $\mathbf{i}^{(\mathbf{t})}$;
- *Block Input Gate.* This gate controls the flow of information from the input gate to the memory cell. It also receives the input vector and the previous output set, producing an output $\mathbf{z}^{(\mathbf{t})}$;
- *Forget Gate.* Its role is to control the contents of the Memory Cell, either to set or reset them, using the Hadamard vector multiplication of its output at time **t**, $\mathbf{c}^{(\mathbf{t-1})}$, producing an output $\mathbf{f}^{(\mathbf{t})}$;
- *Output Block Gate.* It controls the information flow out of the LSTM cell, producing an output $\mathbf{o}^{(\mathbf{t})}$;
- *Memory Cell.* This is the memory element of the neuron, where the previous state is kept, and updated according to the dynamics of the gates that connect to it.
- *Output Activation.* Logistic sigmoid σ and hyperbolic tangent h is two commonly used activation functions. For Block Input Gate, both functions work so the activation function is represented as g.
- *Peepholes.* Direct connections from the memory cell to the gates, that allow them to 'peep' at the states of the memory cells. Their absence was proven to have a minimal performance impact, so they were omitted in this architecture.

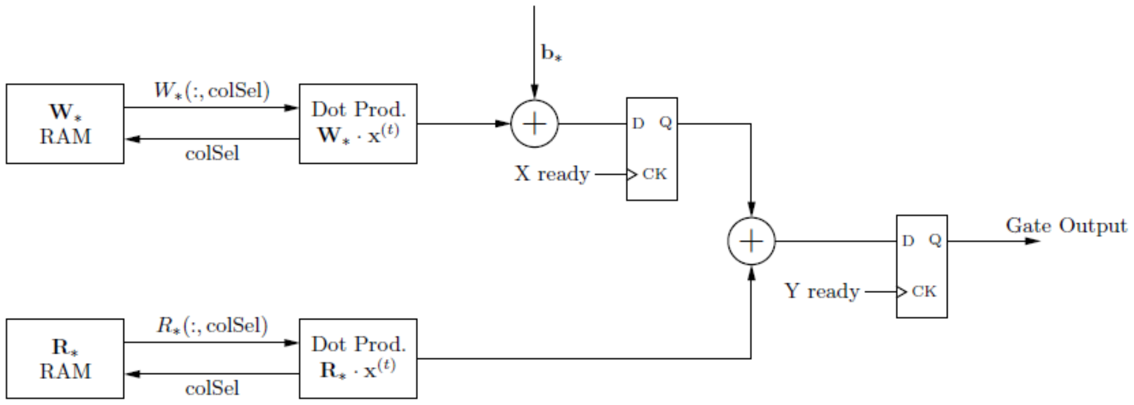

$\mathbf{b}_*$
$\mathbf{W}_*$ RAM
$W_*(:, \mathrm{colSel})$
colSel
Dot Prod. $\mathbf{W}_* \cdot \mathrm{x}^{(t)}$
D Q
CK
X ready
Gate Output
D Q
CK
Y ready
$\mathbf{R}_*$ RAM
$R_*(:, \mathrm{colSel})$
colSel
Dot Prod. $\mathbf{R}_* \cdot \mathrm{x}^{(t)}$

Figure 7: Gate Module

3.4.5.2 Gates module. After omitting the peephole, all the four gates can be represented as the format of $\mathbf{Wx}^{(\mathbf{t})} + \mathbf{Ry}^{(\mathbf{t-1})} + \mathbf{b}$. Therefore, a Gate module can be reused if only different weight and input vectors passing to the module. The matrix-vector multiplication of $\mathbf{Wx}^{(\mathbf{t})}$ and $\mathbf{Ry}^{(\mathbf{t-1})}$ can be processed parallelly. $\mathbf{x}^{(\mathbf{t})}$ comes from the output of embedding layer and $\mathbf{y}^{(\mathbf{t-1})}$ comes from the result of last time-step computation process. The structure of the Gate module is shown in figure 7.

3.4.5.3 Dot Product. The weight matrices **W** and **R** are multiplied by the input vector $\mathbf{x}^{(\mathbf{t})}$ and the layer output vector $\mathbf{y}^{(\mathbf{t-1})}$, respectively. This way, we need an HDL block that implements matrix-vector multiplication. If the input vector x has length M (the number of inputs to the layer), and so W has size N ×M, while y has length N (the number of neurons in the layer), and so R now has size N×N. Therefore, the dot product is always a vector-matrix multiplication which can be processed parallelly by calculate the vector with each column of the matrix at the same time.

3.4.5.4 LSTM module. The implementation of an LSTM cell in FPGA is shown in figure 8. The four gates can be processed parallelly and the structure is strictly consistent with the formula (1). In addition, the memory element of the LSTM cell is the array of flip-flops that keep the value of the vector $\mathbf{c}^{(\mathbf{t})}$.

3.4.6 Activation Functions. The hardware module that implements these activation functions is, essentially, a 2nd degree polynomial calculator, where the coefficients of the polynomial are chosen accordingly with the value of the input **x**. The coefficients achieved for the Sigmoid and Hyperbolic Tangent functions are reported in Table 3 and Table 4.

$$g(x) = p_0 + p_1 x + p_2 x^2 \tag{2}$$

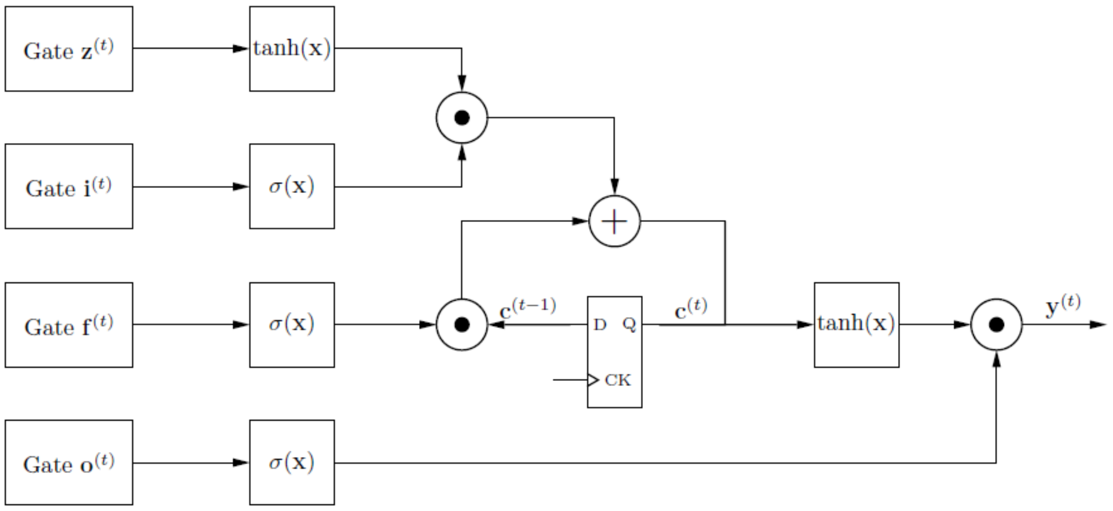


Figure 8: LSTM Module

Table 3: Polynomial Coefficients for Sigmoid

p_0	p_1	p_2	Interval
0	0	0	x < -6
0.20323428	0.0717631	0.00642858	$-6 \le x \le -3$
0.50195831	0.27269294	0.04059181	$-3 \le x \le 0$
0.49805785	0.27266221	0.04058115 0	$0 \le x \le 3$
0.7967568	0.07175359	0.00642671	$3 \le x \le 6$
1	0	0	$x \ge 6$

Table 4: Polynomial Coefficients for tanh

p0	p1	p2	Interval
-1	0	0	x < -3
-0.39814608	0.46527859	0.09007576	$-3 \le x \le -1$
0.0031444	1.08381219	0.31592922	$-1 \le x \le 0$
-0.00349517	1.08538355	-0.31676793	$0 \le x \le 1$
0.39878032	0.46509003	0.09013554	$1 \le x \le 3$
1	0	0	$x \ge 3$

3.4.7 Dense Layer. After all the time-steps completed in LSTM layer, the output vector will be transmitted to the final dense layer. In the dense layer, the weights are stored separately in registers which could be accessed parallelly. The dense layer module shows in figure 9. A binary adder tree (BAT) is designed for this layer to achieve the final matrix multiplication as shown in figure 10. The product should be processed through sigmoid activation function then concatenated as the final model output vector.

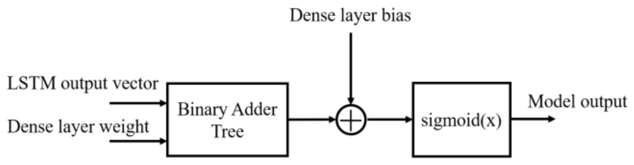


Figure 9: Dense Module

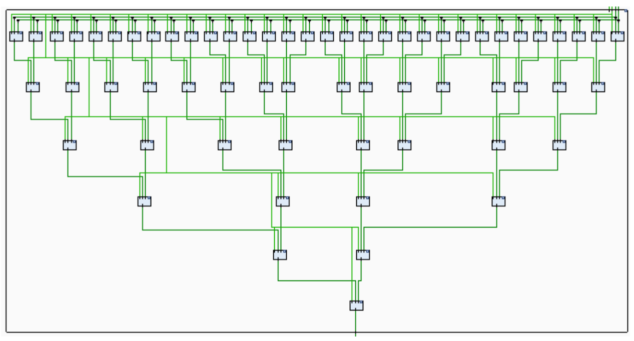
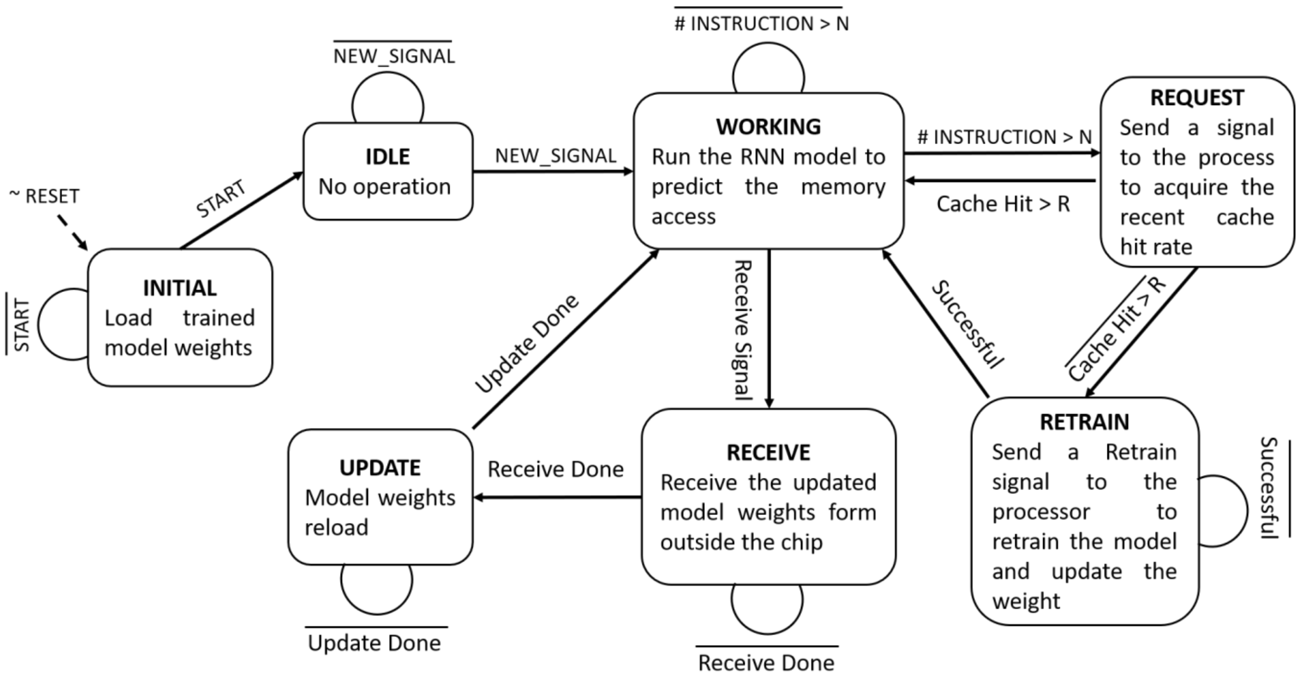


Figure 10: BAT of One Dimension in Dense Layer

3.5 Online Update State Machine

The state machine of the FPGA prefetcher is shown in Figure 11. There are six states: Initial, Working, Request, Retrain, Receive and Update.

- *Initial*. Load the trained model weights while reset. When receiving START signal, the prefetcher transfers to Idle state;
- *Idle*. This state waits for the signal coming and does no operation;
- *Working.* The RNN model works in this state to predict the memory access. When the accumulated number of instructions achieves N, the prefetcher will transfer to Request state;
- *Request.* The prefetcher sends a signal to the processor for recent information of cache hit. If the cache hit is larger than a threshold rate R, the state goes back to working, otherwise transfers to Retrain state.
- *Retrain.* The prefetcher sends the processer a signal to retrain the offline model using recent memory access data. If the signal is successfully sent, the state will transfer to Working state so that the prefetcher can still make prediction though in a less satisfactory performance.
- *Receive.* After offline model training, the prefetcher will receive the signal and start to receive the new weights.
- *Update.* Reload the model weight to the prefetcher which keeps the prefetcher working in a high performance.



$\overline{\text{\# INSTRUCTION > N}}$
$\overline{\text{NEW\_SIGNAL}}$
IDLE
No operation
NEW_SIGNAL
WORKING
Run the RNN model to predict the memory access
# INSTRUCTION > N
Cache Hit > R
REQUEST
Send a signal to the process to acquire the recent cache hit rate
~ RESET
START
$\overline{\text{START}}$
INITIAL
Load trained model weights
Update Done
Receive Signal
Successful
$\overline{\text{Cache Hit > R}}$
UPDATE
Model weights reload
Receive Done
RECEIVE
Receive the updated model weights form outside the chip
RETRAIN
Send a Retrain signal to the processor to retrain the model and update the weight
$\overline{\text{Successful}}$
$\overline{\text{Update Done}}$
$\overline{\text{Receive Done}}$

Figure 11: State Machine

4 Experimental Setup

4.1 Prefetcher Performance

The model prediction performance is shown in Figure 12. The model trained with application traces in the cluster 1 is shown in Table 1.

Due to some difficulties in working on ChampSim simulator, experiments involving the cache hit rate of the trained model are not conducted yet. However, the LSTM prefetcher performance has been proven in [10], which shows the LSTM based prefetcher outperforms traditional prefetcher algorithms, including Best Offset Signature Path, Variable Length Delta and Last-Delta Prefetchers.

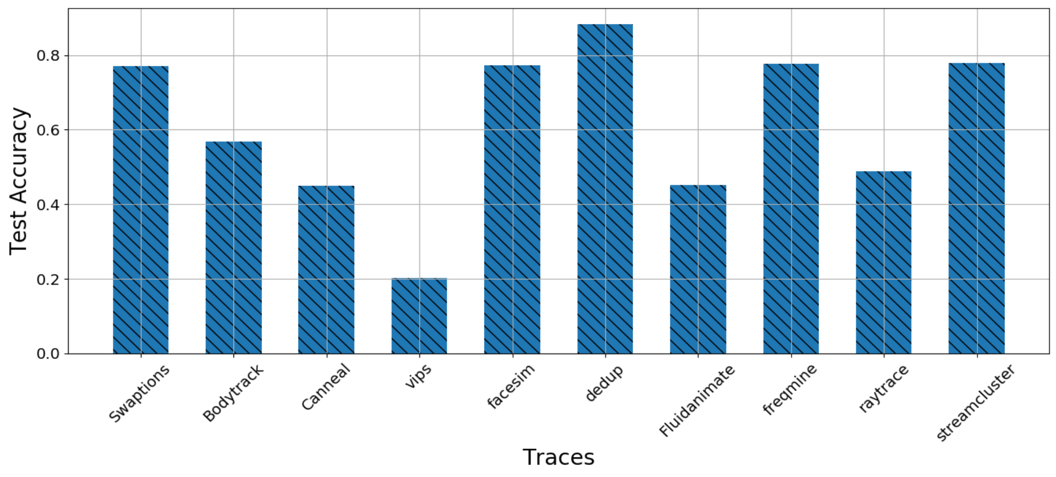


Figure 12: Model Performance

4.2 Vivado Simulation

Figure 13 shows the simulation result of the model implemented on FPGA in the configuration of table 2, with LSTM layer size at 32. After encoding the 3 previous memory access deltas into 16-bit binary, the time-step becomes 48. Though in Python the input is fed into the model as one vector and then processing each time step when the data arrives LSTM layer, in this FPGA implementation the input is given sequentially before the embedding layer to simplify the work in embedding layer without performance compromise. Thus, the InputVec is one-bit data given into the model. The Embedding_outputVec is the mapping result from a single bit to an 8-dimension vector, each number is of 18 bits with 11 fixed point bits. LSTM_outputVec produces the LSTM layer result for each time step. While the 48th new sample is received, the input vector transmission completes, and the dense layer is enabled, so dense_enable signal is set. The output of LSTM layer will serve as the input of the dense layer. After the computation of binary search tree in the dense layer, the 18-bit numbers concatenate into the model output vector.

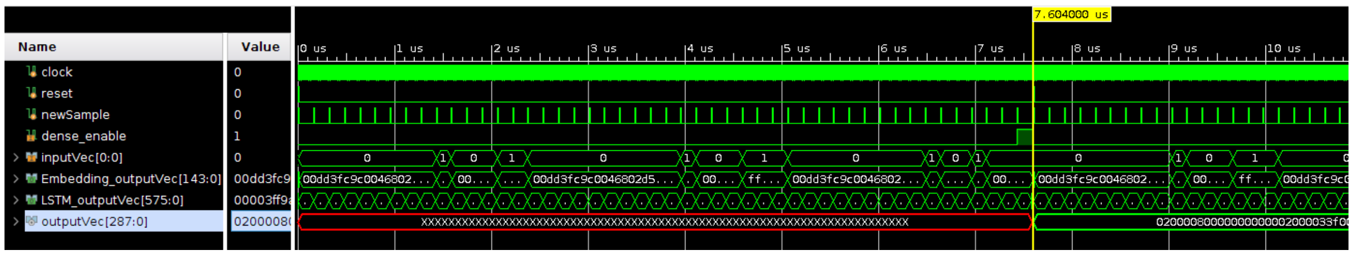


Figure 13: Waveform on Vivado Simulation

4.3 FPGA Speedup

An Experiment regarding to how the LSTM size (number of cells in the LSTM layer) influences the model performance and computation speed is conducted. For LSTM at size 16, 32,64 and 128, four models are both trained offline and simulated on Vivado. The models' accuracy is tested, the CPU and FPGA inference computation times are recorded. The experiment results are shown in Table 5. The trend is plotted in figure 14. The information of CPU processor and FPGA are shown as follows:

- CPU: Intel Xeon Gold 5120 CPU @ 2.20GHz
- FPGA: ZYNQ-7 ZC702 (xc7z020clg484-1)

Table 5 Accuracy and Speed Depend on LSTM Size

LSTM Size	Accuracy	CPU (s)	FPGA (μs)	Speed-up
16	0.3482	0.044	4.54	9692
32	0.7311	0.047	7.605	6180
64	0.8024	0.048	13.748	3491
128	0.8148	0.045	26.194	1718

4.2 FPGA Resource Utilization

The resource utilization with the variation of LSTM size is shown in Table 6. The utilization for size 16 and 32 is lower than 100% so the system can the handle these two LSTM sizes. larger LSTM size such as 64 will exceed the capability of the FPGA board.

When the size is set to be 128, the synthesis takes long time without success thus the utilization result in not available.

Table 6 Utilization Depends on LSTM Size

LSTM Size	Param #	LUT %	FF %
16	1,888	11	26
32	5,792	44	87
64	19,744	668	312
128	72,224	X	X

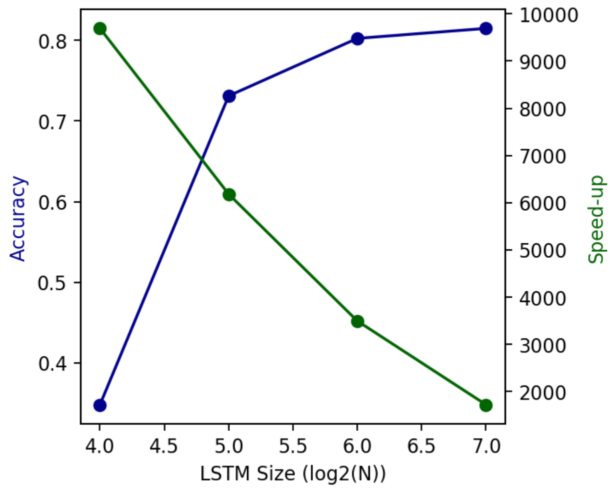


Figure 14: Accuracy and Speed-up Depend on LSTM Size

5 Analysis of Results

Table 5 and Figure 14 shows the accuracy and timing results. With the increase of model LSTM layer size, the prediction accuracy improves significantly from 16 to 32 but slightly from 64 to 128. This shows unnecessary large network can provide limited help to the improvement of prediction.

One can observe that the implemented hardware model is significantly faster than the CPU even running at lower clock frequency. While the given CPU calculates the four model in different LSTM size in a nearly constant time slot, FPGA computation time grows nearly linearly in the factor of log2(LSTM size). This leads to the decrease of speed-up with the increase of LSTM size.

Some other researchers show that the FPGA implementation of LSTM can achieve a speed-up at the level of 20x to 200x while this experiment shows a 1000x to 9000x improvement. This is caused by the accumulative advantages for each time step. Since the time step after encoding is long (48) in this prefetching model, which is forty times to the demo models in other researches.

Table 6 shows the resource utilization under different model size. According to the accuracy result in Table 5, raising LSTM size from 32 to 64 can only acquire 0.1 accuracy improvement, but brings 3.4 times parameter number, 3.6 times flip-flops and 15 times LUT utilization. Thus size 32 is the best choice for the prefetching model implementation on this FPGA board. This high increase of LUT is caused by both the model weight increase and the binary adder tree in the final dense layer. The dense layer parallelly computes all dimensions of LSTM output, which requires corresponding increase of adder, multiplier and activation function blocks when LSTM size increases.

6 Conclusion

This paper designs a prefetcher that is an RNN-based memory prediction model implemented and accelerated on FPGA. An RNN-based model is trained from a cluster of application's memory traces offline to predict the next access delta. This model is compressed by encoding the input and output into binary, which results in $O(n/\log n)$ compression. This trained model is implemented on FPGA using Verilog HDL. The model weights are stored in LUT RAM and the structure is designed to be suitable for parallel computation. Experiments show the prediction accuracy achieves 73.1%. This model is implemented on FPGA and surpassed the performance of the software CPU implementation by 6000×. This high speed-up is a combination of FPGA parallel computing and an accumulative effect of long time-steps for this specific task. Through the analysis of the experiment on LSTM size, a conclusion can be drawn that even larger size of LSTM size can increase the memory prediction accuracy, the improvement raise slows when size is larger than 32 while the burden for hardware implementation increases fast, which make the implementation impractical.

ACKNOWLEDGMENTS

This work is a final report for EE-599 Spring 2020. Thanks to Professor Viktor Prasanna's advices and comments on this work. Thanks to TA Sasindu Wijeratne's support on technique detail discussion of this research.

REFERENCES

[1] Jean-Loup Baer and Tien-Fu Chen. An effective on-chip preloading scheme to reduce data access penalty. In Proceedings of the 1991 ACM/IEEE conference on Supercomputing, pages 176–186, 1991.
[2] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. ACM SIGARCH Computer Architecture News, 19(2):40–52, 1991.
[3] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. A survey of model compression and acceleration for deep neural networks. arXiv preprint arXiv:1710.09282, 2017.
[4] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. 1999.
[5] Yijin Guan, Zhihang Yuan, Guangyu Sun, and Jason Cong. Fpga-based accelerator for long short-term memory recurrent neural networks. In 2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC), pages 629–634. IEEE, 2017.
[6] Shih-wei Liao, Tzu-Han Hung, Donald Nguyen, Chinyen Chou, Chiaheng Tu, and Hucheng Zhou. Machine learning-based prefetch optimization for data center applications. In Proceedings of the Conference on High Perfor- mance Computing Networking, Storage and Analysis, pages 1–10, 2009.
[7] Eriko Nurvitadhi, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason Ong Gee Hock, Yeong Tat Liew, Krishnan Srivatsan, Dun- can Moss, Suchit Subhaschandra, et al. Can fpgas beat gpus in acceler- ating next-generation deep neural networks? In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Ar- rays, pages 5–14, 2017.

[8] Allan Kennedy Porterfield. Software methods for improvement of cache performance on supercomputer applications. PhD thesis, 1989.
[9] Alan Jay Smith. Cache memories. ACM Computing Surveys (CSUR), 14(3):473–530, 1982.
[10] Ajitesh Srivastava, Angelos Lazaris, Benjamin Brooks, Rajgopal Kannan, and Viktor K Prasanna. Predicting memory accesses: the road to compact ml-driven prefetcher. In Proceedings of the International Symposium on Memory Systems, pages 461–470, 2019.
[11] Shuo Wang, Zhe Li, Caiwen Ding, Bo Yuan, Qinru Qiu, Yanzhi Wang, and Yun Liang. C-lstm: Enabling efficient lstm using structured compression techniques on fpgas. In Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pages 11–20, 2018.
[12] Fabius, Otto, and Joost R. van Amersfoort. "Variational recurrent auto-encoders." arXiv preprint arXiv:1412.6581(2014).
[13] Kingma, Diederik P., and Max Welling. "Auto-encoding variational bayes." arXiv preprint arXiv:1312.6114 (2013).
[14] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber," LSTM: A search space odyssey." CoRR, vol. abs/1503.04069, 2015. [Online]. Available: http://arxiv.org/abs/1503.04069.
[15] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, PACT '08, (New York, NY, USA), pp. 72–81, ACM, 2008
[16] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," SIGPLAN Not., vol. 40, pp. 190–200, June 2005.