FPGA Acceleration of Recurrent Neural Network based Language Model

Sicheng Li, Chunpeng Wu, Hai (Helen) Li
University of Pittsburgh
Pittsburgh, PA, USA
{sil27, chw127, hal66}@pitt.edu

Boxun Li, Yu Wang
Tsinghua University
Beijing, P.R. China
yu-wang@tsinghua.edu.cn

Qinru Qiu
Syracuse University
Syracuse, NY, USA
qiqiu@syr.edu

***Abstract*—Recurrent neural network (RNN) based language model (RNNLM) is a biologically inspired model for natural language processing. It records the historical information through additional recurrent connections and therefore is very effective in capturing semantics of sentences. However, the use of RNNLM has been greatly hindered for the high computation cost in training. This work presents an FPGA implementation framework for RNNLM training acceleration. At architectural level, we improve the parallelism of RNN training scheme and reduce the computing resource requirement for computation efficiency enhancement. The hardware implementation primarily targets at reducing data communication load. A multi-thread based computation engine is utilized which can successfully mask the long memory latency and reuse frequent accessed data. The evaluation based on the Microsoft Research Sentence Completion Challenge shows that the proposed FPGA implementation outperforms traditional class-based modest-size recurrent networks and obtains 46.2% in training accuracy. Moreover, experiments at different network sizes demonstrate a great scalability of the proposed framework.**

***Keywords*-recurrent neural network (RNN); language model; FPGA; acceleration;**

I. Introduction

Language models that approximate and evaluate the relative likelihood of phrases in text and speech applications are widely used in natural language processing. They are the critical components in systems for text and speech recognition [1][2][3], machine translation [4], *etc*. Inspired by routinely massive data acquisition from data centers [5], the user-behavior prediction starts gaining attentions. As an important part of it, language models are particularly useful in human understanding of languages [6].

Sequential data prediction has been considered as the key in language model development. Extensive studies were conducted to improve the ability of models over the simple but effective *n-gram* models [7]. Advanced techniques based on decision trees [8] and maximum entropy [9] were investigated. Rather than solely replying on the relationship of words, these works incorporate more features, such as part of speech tags and syntactic structure. *Deep neural networks* (DNNs) also demonstrated great potential in the domain of language models [10]. Prior research and experiments showed that *neural network based language models* (NNLM) can outperform many major advanced language modeling techniques [11].

Recurrent neural network (RNN) is a special type of neural network that operates in time domain. Unlike DNNs where all the layers process input data in a uniform direction, RNN uses additional recurrent connections to capture the long-range dependencies of input data and store such historical information in *hidden layer* for later use. Thus, it is regarded as an expressive model to deal with nonlinear sequential processing tasks [12]. However, its training procedure involves with very high computation cost. Assume the hidden layer has $1,000$ nodes, then $10,000 \times 1,000 = 10^7$ parameters are required to represent the weights between the hidden and output layers. Moreover, the unstable relationship of parameters and the dynamic temporal states of the hidden layer results in a large number of epochs for convergence and hence long training time.

These difficulties in training procedure severely increases the system complexity in RNN implementation. The situation becomes even worse in the case of routinely massive data acquisition. For example, to process the ever-increasing amount of text available on the Internet, Google used distributed systems to conduct daily training of *RNN based language model* (RNNLM). The implementation led to extremely high costs in system development and power consumption: a typical voice search language model for US English stream trained on 230B words consumes $1KJ$ energy per search [13]. The hardware acceleration, thus, is necessary and implementations in ASICs [14], GPUs [15] and FPGAs [16] have been explored. Among them, FPGA based accelerators have attracted great attentions for flexible reconfiguration capability and high energy efficiency.

FPGA-based platform, *e.g.*, Convey computer system [17] used in this work, offers a large number of configurable logic elements and high external memory bandwidth. To maintain the design scalability, it is crucial to optimize and balance the computation resources and memory accesses. In this work, we propose an FPGA implementation framework for RNNLM training acceleration. A holistic approach is adopted which combines the computation efficiency enhancement at architectural level and the memory access load reduction in hardware implementation. The main contributions of this paper include:

- At architectural level, the framework extends the inherent parallelism of RNN and adopts a mixed-precision

scheme. The approach enhances the utilization of configurable logics and improves computation efficiency.
- The hardware implementation integrates a groups of computation engines and a multi-thread management unit. The structure successfully conceals the irregular memory access feature in data back-propagation stage and reduces external memory accesses.
- Our framework is designed in a scalable manner to benefit the future investigation for ever-larger networks.

We realized the RNNLM on Convey HC-2^{ex} system [17]. The design was trained with a dataset of 38M words. It consists of $1,024$ nodes in hidden layer. Our design performs better than traditional class-based modest-size recurrent networks and obtains 46.2% in accuracy in *Microsoft Research Sentence Completion* (MRSC) challenge. The experiments at different network sizes on average achieve $14.1\times$ speedup over the optimized CPU implementation and a comparable performance with high-end GPU system, demonstrating a great system scalability.

The rest of our paper is organized as follows: Section II introduces the language model and RNN algorithm; Section III presents our analytical approach for accelerator design optimization; Section IV and V explain our proposed architecture and the corresponding hardware implementation, respectively; Experimental results and analysis are shown in Section VI; and Section VII concludes the paper.

II. Background

A. Language Models

Rather than checking linguistic semantics, modern language models based on statistical analysis assign a probability to a sequence of words by means of a probability distribution. Ideally, a meaningful word sequence expect to have a larger probability than an incorrect one, such as

$$P(\underline{I}\ saw\ a\ dog) > P(\underline{Eye}\ saw\ a\ dog). \tag{1}$$

Among developed language models, *n-gram model* is the most commonly used. In an n-gram model, the probability of observing the i^{th} word w_i in the context history of the preceding $i-1$ words can be approximated by the probability of observing it in the shortened context history of the preceding $n-1$ words. For example, in a *2-gram* (also called as *bigram*) model, the probability of "$I\ saw\ a\ dog$" can be approximated as

$$\begin{aligned} P(I\ saw\ a\ dog) =& P(I|-) \times P(saw|I) \times P(a|saw) \\ & \times P(dog|a) \times P(-|dog) \end{aligned}. \tag{2}$$

Where, $-$ represents the start or end of the sentence.

A conditional probability, *e.g.*, $P(a|saw)$ in Eq. (2), can be obtained through statistical analysis based on training data. The number of conditional probabilities required in n-gram increases exponentially as n grows: for a vocabulary with a size of V, an n-gram model need store V^n parameters. Moreover, the space of training data becomes highly sparse as n increases. In other words, a lot of meaningful word sequences will be missed in the training data set and hence statistical analysis cannot provide the corresponding conditional probabilities. Previous experiments showed that the performance of n-gram language models with a larger n ($n > 5$) is less effective [7]. N-gram model can realize only the short-term perspective of a sequence, which is clearly insufficient to capture semantics of sentences [18].

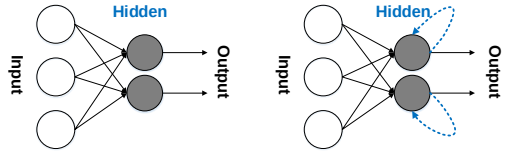

Figure 1. (a) Feedforward neural network; (b) Recurrent neural network.

B. RNN & RNN based Language Model

Figure 1 illustrates the structure of a standard *recurrent neural network* (RNN). Unlike feedforward neural networks where all the layers are connected in a uniform direction, a RNN creates additional recurrent connections to internal states (*hidden layer*) to exhibit historical information. At time t, the relationship of input $\vec{x}(t)$, the temporary state of hidden layer $\vec{h}(t)$, and output $\vec{y}(t)$ can be described as

$$\vec{h}(t) = f\left(W_{ih}\vec{x}(t) + W_{hh}\vec{h}(t-1) + \vec{b}_h\right), \text{ and} \tag{3}$$

$$\vec{y}(t) = g\left(W_{ho}\vec{h}(t) + \vec{b}_o\right). \tag{4}$$

Where, W_{ih} is the weight matrix connecting the input and hidden layers, W_{ho} is the one between the hidden and output layers. W_{hh} denotes the recurrent weight matrix between the hidden states at two consecutive time steps, *e.g.*, $\vec{h}(t-1)$ and $\vec{h}(t)$. $\vec{b}_h$ and $\vec{b}_o$ are the biases of the hidden and output layers, respectively. $f(z)$ and $g(z)$ denote the activation functions at the hidden and output layers, respectively.

The input/output layer of *RNN-based language model* (RNNLM) corresponds to the full or compressed vocabulary. So each node represents one or a set of words. In calculating the probability of a sentence, the words will be input in sequence. For instance, $\vec{x}(t)$ denotes the word at time t. And output $\vec{y}(t)$ represents the probability distribution of the next word, based on $\vec{x}(t)$ and the historical information stored as the previous state of network $\vec{h}(t-1)$.

RNNLM uses internal states at hidden layer to store the historical information, which is not constrained by the length of input history. Compared with n-gram models, RNNLM is able to realize a long-term perspective of the sequence. Note that the hidden layer usually has much less nodes than the input/output layer and its size shall reflect the amount of training data: the more training data are collected, the larger hidden layer is required. Moreover, the aforementioned sparsity of the training data in n-gram language model is not an issue in RNNLM, indicating that RNNLM has a stronger learning ability [12].

C. The RNN Training

When training a network of RNNLM, all data from training corpus are presented sequentially. In this work, we

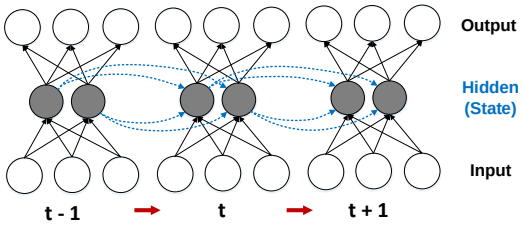

Figure 2. Unfold RNN for training through BPTT algorithm.

used *back-propagation through time* (BPTT) algorithm. As illustrated in Figure 2, the approach truncates the infinite recursion of a RNN and expands it to a finite feed-forward structure, which then can be trained by following the regular routine of feed-forward networks.

For a given input data, the actual output of network shall first be calculated. Then the weights of each matrix will be updated through back-propagating the deviations between the actual and desired outputs layer by layer. The update of weight w_{ji} between node i of the current layer and node j of the next layer at time t can be expressed as

$$w_{ji} \leftarrow w_{ji} + \eta \cdot \sum_{t=1}^{T} \delta_j(t) \cdot x_i(t), \tag{5}$$

where $x_i(t)$ is the input of node i; η is the learning rate; $\delta_j(t)$ is the error back-propagated from node j; and T is the BPTT step for RNN training.

At the output layer, we adopted *softmax* activation function $g(z) = \frac{e^z}{\sum_k e^{z_k}}$ as the cross-entropy loss function. The error derivative of node p $\delta_p(t)$ can be obtained simply from RNN's actual output $o_p(t)$ and the desired one $t_p(t)$:

$$\delta_p(t) = t_p(t) - o_p(t). \tag{6}$$

Sigmoid function $f(z) = \frac{1}{1+e^{-z}}$ is utilized at the hidden layer. The error derivative of node k $\delta_k(t)$ is calculated by

$$\delta_k(t) = f'(x)|_{f(x)=h_k(t)} \cdot \delta_{\mathrm{BPTT}}(t). \tag{7}$$

Where, $h_k(t)$ is the state of node k in hidden layer at time t. $\delta_{\mathrm{BPTT}}$is the accumulation of the errors back-propagated through time, that is,

$$\delta_{\mathrm{BPTT}}(t) = \sum_{o \in output} w_{ok}\delta_o(t) + \sum_{h \in hidden} w_{hk}\delta_h(t+1). \tag{8}$$

Here, $\delta_o(t)$ denotes the error of output layer at time t, while $\delta_h(t+1)$ is the error of hidden layer back-propagated from the following time step $t+1$. w_{ok} and w_{hk} are the transposed weights of W_{ho} in Eq. (3) and W_{hh} in Eq. (4), respectively.

III. Analysis for Design Optimization

We first analyze the utilization of computation and communication resources in RNNLM as these are two principal constraints in system performance optimization.

Computation resource utilization. To analyze the computation cost, we implemented RNNLM on a CUBLAS-based NVIDIA GPU [19] and profiled the runtime of every major function. The result in Table I shows that the matrix-vector

Table I
RNN Computation Runtime Breakdown in GPU

Matrix-vector Multi.	Activation Functions	Sum of Vector Elem.	Vector Scaling	Delta	Others
71.0%	21.4%	2.3%	1.9%	1.0%	2.4%

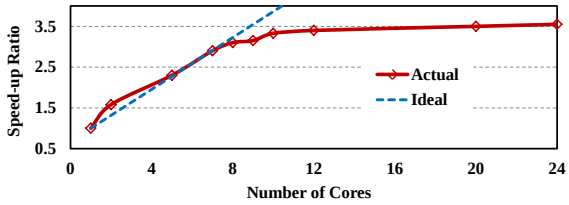

Figure 3. The system speedup is saturated with the increment of CPU cores as the memory accesses become the new bottleneck.

multiplication consumes most of computation resource. The activation functions, as the second contributor, consume more than 20% of runtime. So we mainly focus on enhancing the computation efficiency of these two functions.

Memory accesses. During training, the matrix-vector multiplication in the back-propagation phase requires the transposed form of weight matrices as shown in Eq. (8). Such a data access exhibits irregular behavior, making the further performance improvement very difficult. To explore this effect experimentally, we mapped RNNLM on a multi-core server with Intel's *Math Kernel Library* (MKL) [20]. Figure 3 shows the normalized system performance. As more cores are utilized, the major constrain changes from computation resource to memory bandwidth. Accordingly, the speedup becomes slower and eventually saturated when the memory bandwidth is completed consumed.

Scalability. A scalable implementation must well balance the use of computation units and memory bandwidth. As the configurable logic elements on FPGA grow fast, the implementation shall be able to integrate additional resources. Our approach is to partition a design into multiple identical groups and migrate the optimized development of a group to bigger and ore devices for applications in larger scale.

IV. Architecture Optimization

This section describes the optimization details at the architectural level. We propose a parallel architecture to improve the execution speed between the hidden and output layers. Moreover, the computation efficiency is enhanced by trading off data and function precision.

A. Increase Parallelism between Hidden and Output Layers

Previously, Li *et al.* proposed a pipeline architecture to improve the parallelism of RNN [15]. As illustrated in Figure 4, it partitions the feed-forward phase into two stages: the data flow from input to hidden layer represented by gray boxes and the computation from hidden to output layer denoted in white boxes. Furthermore, it unfolds RNN along time domain by tracing B previous time steps (usually $2 \sim 10$) and pipelines these calculations.

However, our analysis reveals that the two stages have extremely unbalanced throughputs. Assume a RNN with V nodes in the input and output layers and H nodes in the hidden layer. The input layer activates only one node at a time, so $W_{ih}\vec{x}(t)$ in Eq. (3) can be realized by extracting the row of W_{ih} corresponding to the activated node, that is, copying a row of W_{ih} to the destination vector. Thus, the computation complexity of $\vec{h}(t)$ is mainly determined

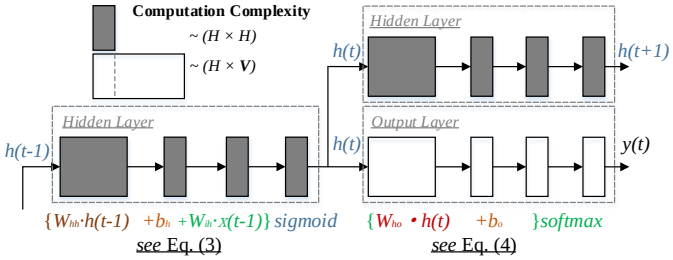

Figure 4. The data flow of a two-stage pipelined RNN structure [15]. The computation complexity of white boxes in output layer is much bigger than that of gray boxes in hidden layer.

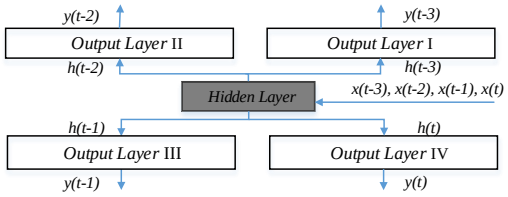

Figure 5. Our proposed parallel architecture for RNNLM.

by $W_{hh}\vec{h}(t-1)$, which is $O(H \times H)$. The calculation of $\vec{y}(t)$ has a computation complexity of $O(H \times V)$ because $W_{ho}\vec{h}(t)$ is dominant. Usually V is related to the vocabulary and can easily reach up to a size of 10K $\sim$ 200K while H can maintain at a much smaller scale like 0.1K $\sim$ 1K. Thus, the execution time of the second stage is much longer than that of the first one. Such a pipelined structure [15] is not optimal for the entire workload.

Our effort is dedicated in further improving the execution of the second stage. As illustrated in Figure 5, we duplicate more processing elements of the output layer. More specific, our proposed architecture conducts the calculation of the hidden layer in serial while parallelizing the computation of the output layer. For example, assume B is 4. At time step $t-3$, the result of the hidden layer goes to *Output Layer I*. While *Output Layer I* is in operation at $t-2$, the hidden layer will submit more data to the next available output layer processing element, e.g., *Output Layer II*. As such, the speed-up ratio of the proposed design over the two-stage pipelined structure [15] can be approximated by

$$\textit{Speed-up} = \frac{(t_V + t_H) + t_V \times (B-1)}{t_H \times B + t_V}, \tag{9}$$

where t_V and t_H are the latencies of the output layer and the hidden layer, respectively. For instance, assume V= 10K, H= 0.1K, and B= 4, the execution of our architecture is about 3.86× faster than the design of [15].

For the proposed design, B shall be carefully selected based upon application's scale. From the one hand, a bigger B indicates more time steps processed in one iteration and therefore requires more resources. From the other hand, the higher B is, the faster execution can be obtained. Moreover, by introducing more time steps, more historic information are sustained for better system accuracy too.

B. Computation Efficiency Enhancement

Through appropriately trading off data and function precision of RNNLM, we can greatly improve its computation efficiency without degrading the training accuracy.

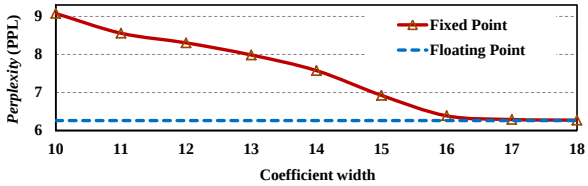

Figure 6. The impact of the reduced data precision of W_{ho} in RNNLM on the reconstruction quality of word sequences.

Fixed-point data conversion. The floating-point data are adopted in the original RNNLM algorithm and the corresponding hardware implementation, which demand significant computation resources and on-chip data space. The fixed-point operation is more efficient in FPGA implementation but the errors caused by precision truncation could accumulate iteratively. Fortunately, neural networks exhibit self-recovery characteristics, which refers to the tolerance to noise in decision making.

Mixed-precision data format. As the computation of output layer is more critical, lowering the data precision of W_{ho}, if possible, would be the most effective option. We analyze RNNLM using the Fixed-Point MATLAB Toolbox and evaluate the quality of different data format by examining the *perplexity* (PPL) of word sequences [12]. Figure 6 shows that when W_{ho} has 16 or more bits and keep the other training parameters as well as the states of hidden and output layers in original 64 bits, a fixed-point implementation can achieve the same reconstruction quality as a floating-point design. In other words, this scheme improves the runtime performance while maintaining the system accuracy to the maximum extent.

Approximation of activation functions. Our preliminary investigation in Table I reveals that the activation functions are the second contributor in runtime. This is because the complex operations in *sigmoid* and *softmax* functions, such as exponentiation and division (Section II-C), are very expensive in hardware implementation. Instead of precisely mapping these costly operations to FPGA, we adopt the *piecewise linear approximation of onlinear function* (PLAN) [21] and simplify the activation functions with the minimal number of additions and shifts. Our evaluation shows that on average, the error between our approximation and the real *sigmoid* calculation is only 0.59%, which doesn't affect much on the convergence properties in RNNLM training.

V. Hardware Implementation Details

The hardware implementation in FPGA will be presented in this section. We map the proposed architecture to *computation engines* (CEs), each of which is divided into a few *processing elements* (PEs). Moreover, the memory efficiency is improved through data allocation and reuse.

A. System Overview

Figure 7 presents an overview of our hardware implementation on the Convey HC-2^{ex} computer system. The CPU on the host side is used for accelerator startup and weight initialization. There are 16 DIMMs and 1024 banks in the

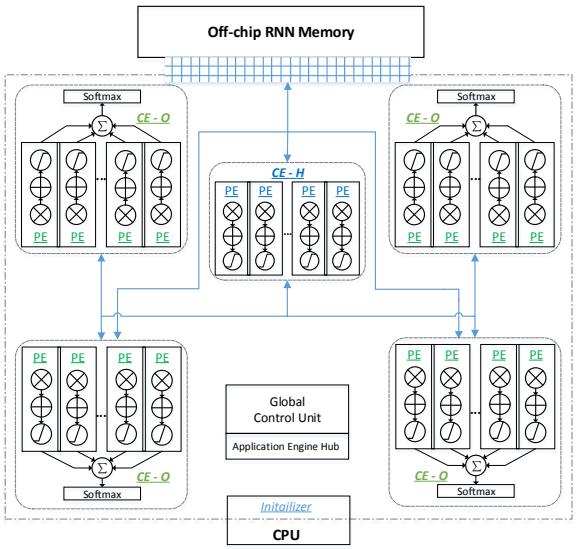

Figure 7. An overview of the RNNLM hardware implementation.

off-chip memory. Thus the chance of bank conflicts is low even the parallel accesses are random. The global control unit receives commands and configuration parameters from the host through *application engine hub* (AEH).

We map the proposed parallel architecture of RNNLM into two types of *computation engines* (CEs): *CE-H* for the hidden layer and *CE-O* for the output layer. According to Figure 5, one *CE-H* and multiple *CE-O* are required. The two types of CEs are customized for high efficiency, with the only difference in the design of activation function. The system configuration, *e.g.*, the number and scale of CEs, is upon users' decision. Moreover, each CE is segmented into several identical *processing elements* (PEs). Since the major of RNNLM execution is performed through these PEs, the proposed implementation can easily be migrated to a future device by instantiating more PEs.

The matrix-vector multiplication not only consumes the most runtime but also demands a lot of data exchange as shown in Table II. The situation in the feed-forward phase can be partially alleviated by data streaming and datapath customization. However, the multiplication operations of transposed matrices in the back-propagation phase exhibit very poor data locality, leading to nontrivial impact on memory requests. The long memory latencies potentially could defeat the gains from parallel executions. In addition, Table II implies that the data accesses in RNNLM have very diverse characteristics, each of which shall be considered specifically in memory access optimizations. These details will be presented in the following subsections.

Table II
MEMORY ACCESS REQUIREMENT

Dataset	Operation	Total #	Size (byte)
Training data	read only	38M	152M
W_{ih}, W_{ho}	read & write	V×H (10K × 1K)	40M
W_{hh}	read & write	H×H (1K × 1K)	4M
b_o, $\vec{y}(t)$	read & write	V (10K)	40K
b_h, $\vec{h}(t)$	read & write	H (1K)	4K

B. Data Allocation

From the one hand, the RNNLM implementation is associated with an extremely large data set, including a training data set (*e.g.*, 38M words in our test) as well as the weight parameters (*e.g.*, 40Mb for a vocabulary of 10K words and the hidden layer of 1K nodes). From the other hand, only a small amount of index data are required to control the RNNLM training process: at a time step, only one input node will be activated and only one output node will be monitored. Therefore, we propose to store the training data in the host main memory and feed them into the FPGAs during each training process.

Though FPGAs in the Convey HC-2^{ex} system (Xilinx Virtex6 LX760) has a large on-chip block RAM (about 26MB), not all the space is available for users. Part of it is utilized for interfacing with memory and other supporting functions. Therefore, we keep the intermediate data which are frequently access and update in the training process, such as all the parameters (W_{ih}, W_{hh}, W_{ho}, b_h, and b_o) and all the states of hidden and output layers, in the off-chip memory instead of on-chip memory. Only a subset of data is streamed into the limited on-chip memory at runtime for the best utilization and system performance.

C. Thread Management in Computation Engine

How to increase the effective memory bandwidth is critical in CE design. Previously, Ly and Chow proposed to remove the transpose of a matrix by saving all the data in on-chip block RAMs [22]. At a time, only one element per row/column of the matrix is read out through a carefully designed addressing scheme. As such, a column or row of the matrix is obtained from one memory thread. However, the approach requires that the weight matrix fits on-chip memory and the number of block RAMs for the weight matrix equals to the number of neurons in different layers. It is not applicable to our RNNLM in a much larger scale.

There are 16 channels in the system. To improve the memory efficiency, we introduce a hardware supported multi-threading architecture named as *thread management unit* (TMU). Figure 8 illustrates its utilization in CEs. To process all the elements of a matrix row through a single memory channel, TMU generates a thread for each matrix row and the associated start and end conditions. All the *ready* threads are maintained by TMU. Once a channel finishes a row, it

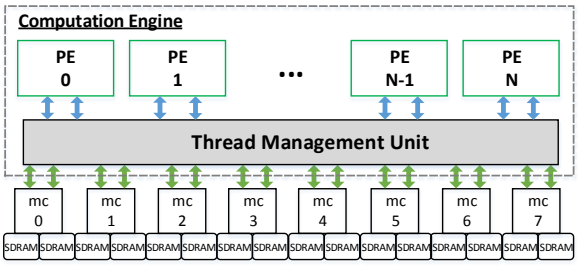

Figure 8. The thread management unit in a computation engine. PEs are connected through a crossbar to the shared memory structure.

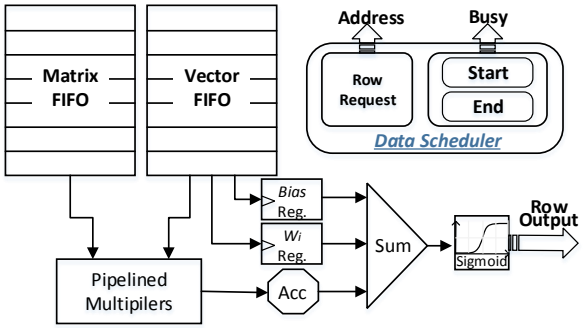

Figure 9. The multi-thread based processing element.

can switch to another ready tread, which usually has been prefetched from memory so the memory latency is masked. Each PE holds a *busy* flag high to prevent additional threads from being assigned. When all the PEs are busy, TMU backloads threads for later assignment.

TMU supports the data communication among a large number of PEs and improves the execution parallelism. Note that there is only one TMU in a CE. Increasing the number of PEs does not introduce more hardware overhead.

D. Processing Element Design

The computation task within a CE is performed through *processing elements* (PEs). These PEs operate independently, each of which takes charge of a subset of the entire task. For example, when realizing a matrix-vector multiplication, each PE is assigned with a thread that computes a new vector value based on a row of the weight matrix. Data transition can operate in the burst mode: based on the start and end addresses, a thread fetches all the requested data in a row from the off-chip memory, as shown in Figure 9.

CE controls the memory requests of the weight matrix and vector arrays. The Convey system supports the in-order return of all memory requests, so the reordering of memory accesses can be done through TMU assisted by the crossbar interface from FPGAs to memory modules. Data returned from memory can be buffered in *Matrix and Vector FIFOs*, using the corresponding thread id as the row index. When a new thread is assigned to a PE, it raises a *busy* flag and requests the weight and vector data from memory based on the start and end addresses. Once all the memory requests for the thread are issued, the flag is *reset*, indicating that the PE is ready for another thread even through the data of the prior thread is still in processing. As such, the memory access load can be dynamically balanced across all PEs.

E. Data Reuse

Off-chip memory accesses take long time. For example, the memory latency on our Convey platform is 125 cycles at 150MHz. To speed up the execution of RNNLM, we can reduce off-chip memory accesses through data reuse.

Algorithm 1 presents the data flow from input to hidden layer, during which the state of hidden layer is frequently accessed. Similar data access pattern has also been observed in the calculation of output layer. We propose reuse buffers

Algorithm 1 Data flow from input layer to hidden layer

1: **for** $t = 0; t < BPTT; t++$ **do**
2: **if** t != 0 **then**
3: $mvmulti\ (W_{hh}, hidden(t-1), hidden(t))$;
4: $vdadd\ (hidden(t), b_h, h(t))$;
5: $vdadd\ (hidden(t), \vec{w_{ih}^k}, hidden(t))$;
6: **else**
7: $vddadd\ (\vec{w_{ih}^k}, b_h, hidden(t))$;
8: **end if**
9: $sigmoid\ (hidden(t), hidden(t))$;
10: **end for**

for matrix and bias vector respectively named as *Wi Reg.* and *Bias Reg.* as shown in Figure 9. First, a row of weight matrix are fed into an array of multipliers that are organized in fine-grain pipeline and optimized for performance. While data goes into the accumulator and completes the matrix multiplication, the weight and bias are buffered registers. After data summation is completed, PE enables its activation function, *e.g.*, *sigmoid* in *CE-H* or *softmax* in *CE-O*, to obtain the state of hidden/output layer.

VI. Experimental Results

In this section, we present the experimental results of the RNNLM implementation and evaluate the proposed framework in terms of training accuracy, system performance, and efficiency of computation engine design.

A. Experiment Setup

We implemented the RNNLM on the Convey HC-2^{ex} platform [17]. We described the design in System C code, which then was converted to Verilog RTL using Convey Hybrid Threading HLS tool ver. 1.01. The RTL is connected to memory interfaces and the interface control is provided by Convey PDK. Xilinx ISE 11.5 is used to obtain the final bitstream. Table IV summarizes the resource utilization of our implementation on one FPGA chip. The chip operates at 150MHz after placement and routing.

B. Training Accuracy

Microsoft Research Sentence Completion (MRSC) is used to validate our FPGA based RNNLM. The challenge consists of fill-in-the-blank questions [23]. The experiment calculates the score (probability) of a sentence filled with each given option and takes the option that leads to the highest score as the final answer of the model.

A set of the 19^{th} and early 20^{th} Century novels were used in training. The dataset has 38M words and the vocabulary of the training data is about 60K. In the implementation, we merge the low-frequency words and map the remaining $10,583$ words to the output layer. For better accuracy, we set the hidden layer size to $1,024$ and BPTT to B= 4.

Table IV
Resource Utilization

	LUTs	FFs	Slice	DSP	BRAM
Consumed	176,355	284,691	42395	416	280
Utilization	37%	30%	35%	48%	39%

Table III
RUNTIME (IN SECONDS) OF RNNLMS WITH DIFFERENT NETWORK SIZE

Hidden Layer Size	BPTT = 2				BPTT = 4				BPTT = 8			
	CPU-Single	*CPU-MKL*	*GPU*	*FPGA*	*CPU-Single*	*CPU-MKL*	*GPU*	*FPGA*	*CPU-Single*	*CPU-MKL*	*GPU*	*FPGA*
128	404.89	139.44	39.25	35.86	627.72	182.4	61.84	43.93	1313.25	370.37	97.448	90.74
256	792.01	381.39	49.89	69.72	1317.34	625.656	76.11	86.86	2213.64	770.02	114.71	146.46
512	1485.56	764.29	73.78	139.44	2566.80	1218.70	110.01	160.72	4925.50	2173.68	159.39	290.97
1024	2985.31	1622.56	130.45	278.88	5767.08	2242.24	191.82	327.44	10842.51	4098.85	271.13	625.94

Table V
ACCURACY EVALUATION ON MSRC

Method	Accuracy
Human	91%
vLBL+NCE5 [25]	60.8%
RNNME-300 [24]	49.3%
RNNLM (this work)	**46.2%**
RNN-100 with 100 classes [12]	40%
Smoothed 3-gram [23]	36%
Random	20%

Table V compares the training accuracy of various language models. Our FPGA-based RNNLM effectively realizes long-term perspective of the sentence and beats the n-gram model. RNNME that integrates RNN with maximum entropy model [24] is able to further improve the training accuracy to 49.3%. Though vLBL+NCE5 [25] obtains the best training effect, it has far more computation cost than our RNNLM because vLBL+NCE5 used a much larger dataset (47M), integrated a data pre-processing technique called *noise-contrastive estimation* (NCE), and analyzed a group of words at the same time.

C. System Performance

We evaluated the performance of the proposed design by comparing it with implementations on different platforms. The configuration details are summarized in Table VI. A simplified training set of $50K$ words was selected because the training accuracy is not the major focus. The vocabulary size of the dataset is $10,583$.

To conduct a fair comparison, we adopted the well-tuned CPU and GPU-based design from [15]. Furthermore, we tested different network sizes by adjusting the BPTT depth and the hidden layer size. The results are shown in Table III. Here, **CPU-Single** represents the single-thread CPU implementation; **CPU-MKL** is multi-thread CPU version with Intel *Math Kernel Library* (MKL); **GPU** denotes the GPU implementation based on CUBLAS; and **FPGA** is our proposed FPGA implementation.

Compared to CPU-single, the performance gain of CPU-MKL is mainly from the use of MKL, which speeds up the matrix-vector multiplication. However, general-purpose processor has to follow the hierarchical memory structure so the space for hardware-level optimization is limited. Our FPGA design customizes the architecture and datapath specific to RNNLM's feature. On average, it obtains $14.1\times$ and $4\times$ speedups over CPU-single and CPU-MKL, respectively.

At relative small network scales, our FPGA implementation operates faster GPU because of GPU's divergence issue. Besides, GPU spends significant runtime on complicated activation functions, while the approximation in FPGA requires only a small number of additions and shifts. However, as the hidden layer and BPTT increase, the limited memory bandwidth of the Convey system constrains the speedup of FPGA. The GPU implementation, on the contrary, is less affected because GTX580 offers $10\times$ memory bandwidth. This is why GPU performs better than FPGA at large scale networks. By augmenting additional memory bandwidth to system, the performance of FPGA shall be greatly improved.

Table III also demonstrates the effectiveness of our proposed parallel architecture. Let's take the example of $1,024$ nodes in hidden layer. As BPTT increases from 2 to 4, implying the doubled timesteps within an iteration, the FPGA runtime increases merely 17.6%. This is because the CEs operate in a parallel format when calculating the output layer results at different time steps. When increasing BPTT from 4 to 8, the runtime doubles because only four CEs were implemented.

Besides performance, the power efficiency is also an important metric. Currently, we do not have a setup to measure the actual power. So the maximum power rating is adopted as a proxy. Table VII shows the power consumption comparison when implementing a modest size network with 512 nodes in hidden layer and BPTT depth of 4. Across the three platforms, our FPGA implementation achieves the best energy efficiency.

Table VI
CONFIGURATION OF DIFFERENT PLATFORMS

Platform	Cores	Clock	Memory Bandwidth
NVIDIA GeForce GTX580	512	772 MHz	192.4 GB/s
Intel® Xeon® CPU E5-2630 @ 2.30 GHz	12	2.3GHz	42.6 GB/s
Convey HC-2ex	-	150 MHz	19.2 GB/s

D. Computation Engine Efficiency

The memory access optimization is reflected by the design of CEs. As a CE is partitioned into multiple individual PEs and each PE executes a subset of the entire workload, the peak performance can be calculated by [26]

$$Throughput = PE \cdot Freq \cdot Width \cdot Channel. \quad (10)$$

Where, *PE* is the number of PEs in each layer; *Width* represents the bit width of weight coefficients; *Freq* is the system frequency in *MHz*; and *Channel* denotes the number of memory channels.

Table VII
POWER CONSUMPTION

	Multi-core CPU	GeForce GPU	FPGA
Run time (s)	2566.80	110.01	160.72
Power-TDP (W)	95	244	25
Energy (J)	243846 (60.69×)	26842 (6.68×)	4018 (1×)

Table VIII
COMPUTATION ENGINE EFFICIENCY

Platform	Cores #	Clock	Peak GOPS	Feed-forward	BPTT	Average Efficiency
Single-core CPU	1	2.3 GHz	2.3	1.03	0.83	**40.43%**
Multi-core CPU	6*2	2.3 GHz	27.6	3.6	2.6	**11.23%**
FPGA-Hidden	8	150 MHz	2.4	1.68	1.11	**58.10%**
FPGA-Output	8*4	150 MHz	9.6	5.2	3.5	**45.52%**

Table VIII compares the computation energy efficiency of FPGA and CPU implementations, measured in *giga operations per second* (GOPS). The *actual sustained performance* of the feed-forward and BPTT phases are calculated by the total number of operations divided by the execution time. Note that a PE is capable of two or more fixed-point operation per cycle.

Though CPU runs at a much faster frequency, our FPGA design obtained higher sustained performance. By masking long memory latency through multi-thread management technique and reduce external memory accesses by reusing data extensively, the computation engine exhibits a significant efficiency that is greater than 45%.

VII. CONCLUSION

In this work, we proposed a FPGA acceleration framework for RNNLM. The system performance is optimized by improving and balancing the computation and communication. We first analyzed the operation condition of RNNLM and presented a parallel architecture to enhance the computation efficiency. The hardware implementation maps neural network structure with a group of computation engines. Moreover, a multi-tread technique and a data reuse scheme are proposed to reduce external memory accesses. The proposed design was developed on the Convey system for performance and scalability evaluation. The framework shall be easily extended to large system and other neural network applications, which will be the focus of our research.

ACKNOWLEDGMENT

This work was supported in part by NSF 1337198, NSF 1311747, DARPA D13AP00042, NSFC 61373026, and Tsinghua Univ. Init. Sci. Res. Pgm.. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of grant agencies or their contractors.

REFERENCES

[1] I. Sutskever, J. Martens, and G. E. Hinton, "Generating text with recurrent neural networks," in *Proc. of the 28th Int. Conf. on Machine Learning (ICML)*, 2011, pp. 1017–1024.

[2] Q. Qiu, Q. Wu, M. Bishop, R. Pino, and R. Linderman, "A parallel neuromorphic text recognition system and its implementation on a heterogeneous high-performance computing cluster," *IEEE Trans. on Comp.*, pp. 886–899, 2013.

[3] L. Deng, J. Li, J.-T. Huang, K. Yao, D. Yu, and F. Seide, "Recent advances in deep learning for speech research at microsoft," in *IEEE Conf. on ICASSP*, 2013, pp. 8604–8608.

[4] *A Recursive Recurrent Neural Network for Statistical Machine Translation*. ACL, 2014. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=215646

[5] H. Sak, O. Vinyals, G. Heigold, A. Senior, E. McDermott, R. Monga, and M. Mao, "Sequence discriminative distributed training of long short-term memory recurrent neural networks," in *Interspeech*, 2014.

[6] J. T. Goodman, "A bit of progress in language modeling," *Comp. Speech & Language*, vol. 15, no. 4, pp. 403–434, 2001.

[7] A. Pauls and D. Klein, "Faster and smaller n-gram language models," in *Proc. ACL*, 2011, pp. 258–267.

[8] G. Potamianos and F. Jelinek, "A study of n-gram and decision tree letter language modeling methods," *Speech Communication*, vol. 24, no. 3, pp. 171–192, 1998.

[9] S. Khudanpur and J. Wu, "A maximum entropy language model integrating n-grams and topic dependencies for conversational speech recognition," in *IEEE Conf. on ICASS*, vol. 1, 1999, pp. 553–556 vol.1.

[10] Y. Bengio, R. Ducharme, P. Vincent, and C. Janvin, "A neural probabilistic language model," *J. Mach. Learn. Res.*, vol. 3, pp. 1137–1155, 2003.

[11] T. Mikolov, A. Deoras, S. Kombrink, L. Burget, and J. H. Cernocky, "Empirical evaluation and combination of advanced language modeling techniques," in *Interspeech*. ISCA, 2011.

[12] T. Mikolov, A. Deoras, D. Povey, L. Burget, and J. Cernocky, "Strategies for training large scale neural network language models," in *IEEE Workshop ASRU*, Dec 2011, pp. 196–201.

[13] C. Chelba, J. Schalkwyk, T. Brants, V. Ha, B. Harb, W. Neveitt, C. Parada, and P. Xu, "Query language modeling for voice search," in *Proc. SLT*, 2010, pp. 127–132.

[14] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," *Micro, IEEE*, vol. 33, no. 3, pp. 16–27, 2013.

[15] B. Li, E. Zhou, B. Huang, J. Duan, and Y. Wang, "Large scale recurrent neural network on gpu," in *IJCNN*, 2014, pp. 4062–4069.

[16] Y. Maeda and M. Wakamura, "Simultaneous perturbation learning rule for recurrent neural networks and its fpga implementation," *IEEE Trans. on Neural Networks*, vol. 16, no. 6, pp. 1664–1672, 2005.

[17] *Convey Reference Manual*, Convey computer, 2012, http://www.conveycomputer.com/.

[18] T. Mikolov, S. Kombrink, L. Burget, J. Cernocky, and S. Khudanpur, "Extensions of recurrent neural network language model," in *IEEE Conf. on ICASSP*, 2011, pp. 5528–5531.

[19] *NVIDIA CUDA C Programming Guide*, NVIDIA Corp.

[20] *Intel math kernel library*, Intel, http://software.intel.com/en-us/intelmkl/.

[21] H. Amin, K. Curtis, and B. Hayes-Gill, "Piecewise linear approximation applied to nonlinear function of a neural network," *Circuits, Devices and Systems, IEEE Proc.*, vol. 144, no. 6, pp. 313–317, 1997.

[22] D. L. Ly and P. Chow, "A high-performance fpga architecture for restricted boltzmann machines," in *Proc. FPGA*, 2009, pp. 73–82.

[23] G. Zweig and C. J. Burges, "A challenge set for advancing language modeling," in *Proc. NAACL-HLT 2012*, pp. 29–36.

[24] T. Mikolov, "Statistical language models based on neural networks," Ph.D. dissertation, Brno Uni. of Technology, 2012.

[25] A. Mnih and K. Kavukcuoglu, "Learning word embeddings efficiently with noise-contrastive estimation," in *Proc. NIPS*, 2013, pp. 2265–2273.

[26] M. Pietras, "Hardware conversion of neural networks simulation models for neural processing accelerator implemented as fpga-based soc," in *Proc. FPL*, 2014, pp. 1–4.