Large Scale Recurrent Neural Network on GPU

Boxun Li[1], Erjin Zhou[1], Bo Huang[1], Jiayi Duan[1], Yu Wang[1], Ningyi Xu[2], Jiaxing Zhang[2], Huazhong Yang[1]
[1]Dept. of E.E., Tsinghua National Laboratory for Information Science and Technology,
Tsinghua University, Beijing, China
[2]Microsoft Research Asia, Beijing, China
Email: yu-wang@mail.tsinghua.edu.cn

***Abstract*—Large scale artificial neural networks (ANNs) have been widely used in data processing applications. The recurrent neural network (RNN) is a special type of neural network equipped with additional recurrent connections. Such a unique architecture enables the recurrent neural network to remember the past processed information and makes it an expressive model for nonlinear sequence processing tasks. However, the large computation complexity makes it difficult to effectively train a recurrent neural network and therefore significantly limits the research on the recurrent neural network in the last 20 years. In recent years, the use of graphics processing units (GPUs) becomes a significant advance to speed up the training process of large scale neural networks by taking advantage of the massive parallelism capabilities of GPUs. In this paper, we propose an efficient GPU implementation of the large scale recurrent neural network and demonstrate the power of scaling up the recurrent neural network with GPUs. We first explore the potential parallelism of the recurrent neural network and propose a fine-grained two-stage pipeline implementation. Experiment results show that the proposed GPU implementation can achieve $2 \sim 11\times$ speed-up compared with the basic CPU implementation with the Intel Math Kernel Library. We then use the proposed GPU implementation to scale up the recurrent neural network and improve its performance. The experiment results of the Microsoft Research Sentence Completion Challenge demonstrate that the large scale recurrent network without class layer is able to beat the traditional class-based modest-size recurrent network and achieve an accuracy of 47%, the best result achieved by a single recurrent neural network on the same dataset.**

I. Introduction

The amount of data in our world have been exploding and we have entered an era of 'big data' [1]. Modern data acquisition routinely produces massive amounts of complex data that motivate a series of scientific discoveries ranging from the genomic analysis to user-behavior predictions of the social network [2]. Big data is changing our lives.

The analysis of big data requires efficient data processing methods. Artificial neural networks (ANNs) are among the most commonly used methods for data processing applications [3]. Artificial neural networks are computational models inspired by the nervous systems in nature and have found extensive utilization in solving many complex real-world problems [4]. In recent years, the large scale artificial neural networks, also known as the deep neural networks (DNNs) or deep learning, have demonstrated a great promise in many artificial intelligence tasks. State-of-the-art performance have been reported in many domains, ranging from computer vision, speech recognition, to nature language processing and information retrieval [5]. The deep neural network has become one of the most popular tools to process big data [6].

The recurrent neural network (RNN) is a special type of neural network that operates in the time domain [7]. Different from the deep neural network, where all the layers process the input data in a uniform direction, the recurrent neural network is equipped with additional recurrent connections that have important capabilities not found in feedforward networks [8]. These unique recurrent connections enable the recurrent neural network to store information for later use and capture the long-range dependencies between input data. Therefore, the recurrent neural network has been regarded as an expressive model to deal with nonlinear sequential processing tasks [7], such as speech recognition [9], text generation [7] and even SQL attack detection [10].

However, the difficulty of training the recurrent neural network makes the recurrent neural network fail to become a mainstream tool in machine learning [11]. The major challenge comes from the huge computation complexity of the recurrent neural network. For example, the recurrent neural network is usually used as the language model in nature language processing tasks. In these tasks, the number of output nodes in the recurrent neural network is usually required to be equal to the size of the vocabulary. Therefore, a task with 10K words in the vocabulary will demand a 'small' recurrent neural network with 100 nodes in the hidden layer to have at least $10,000 \times 100 = 10^6$ parameters. The traditional CPU-based computers will take hundreds of hours to train a recurrent neural network with such a large number of parameters. People have to divide the output layer into classes to reduce the computation complexity at the cost of performance reduction [12]. In addition, the unstable relationship between the parameters and the dynamics of the hidden states leads to a large number of epochs before convergence and huge time consumption for training a recurrent neural network [13]. As a result, although the recurrent neural network demonstrates a great potential for sequential data processing, there has been surprisingly limited research on the recurrent neural network, especially on the large scale recurrent neural network, over the past few decades [7].

In recent years, the use of graphics processing units (GPUs) has been a significant advance to speed up the training process of large scale deep neural network models [14]. GPUs achieve high performance by using single-instruction, multiple-data (SIMD) pipelines with minimal overhead incurred by control

hardware [15]. Recent work has demonstrated that, compared with the implementations on CPUs, the training process of large scale deep neural network can be significantly speeded up around $2 \sim 10\times$ on GPUs [16], [17]. And therefore, GPUs have become essential tools for studying deep neural networks.

In this paper, we propose an efficient GPU implementation of large scale recurrent neural networks. And we try to improve the network performance by increasing the scale of the recurrent neural network. The contributions of this paper include:

- We explore the natural parallelism within the calculations from hidden layer to output layer, and the one from hidden layer to the next timestep. Based on that, we propose an efficient GPU implementation of large scale recurrent neural networks with a fine-grained two-stage pipeline architecture. Experiment results show that the proposed GPU implementation can realize a $2 \sim 11\times$ speedup compared with the CPU implementation based on the Intel Math Kernel library.
- Based on the proposed GPU implementation, we remove the class layer, an additional layer used to significantly reduce the computation complexity at the cost of performance, and successfully train a large scale recurrent neural network with 1,000 nodes in the hidden layer and ∼10,000 nodes in the output layer. We then used the trained recurrent neural network as a language model to test its performance. And the experiment on the Microsoft Research Sentence Completion (MRSC) Challenge demonstrates that the GPU trained large scale recurrent neural network without class layer is able to achieve an accuracy of 47%, the best performance achieved by a single recurrent neural network model on the same dataset.

The rest of this paper is organized as follows: Section 2 provides the background knowledge and related work. Section 3 introduces the proposed GPU implementation of large scale recurrent neural networks. Section 4 studies the performance of the large scale recurrent neural network by taking a sentence completion task as a case study. And Section 5 concludes this work.

II. Background Knowledge and Related Work

A. RNN Architecture

A standard recurrent neural network can be illustrated in Fig. 1. It can be seen that there're unique feedback connections between the output and input nodes of the hidden layer. The relationship between the output and input of the recurrent neural network can be described as follows:

$$\vec{h}(t) = f(W_{ih}\vec{x}(t) + W_{hh}\vec{h}(t-1) + \vec{b}_h) \tag{1}$$

$$\vec{y}(t) = g(W_{ho}\vec{h}(t) + \vec{b}_o) \tag{2}$$

where

$$t = 1, 2, 3, ..., n$$

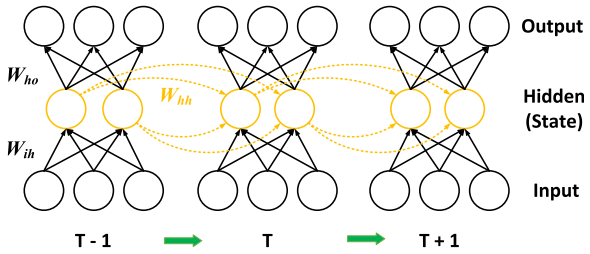

Fig. 1. RNN Architecture

where $\vec{x}(t)$ and $\vec{y}(t)$ are the input and output data of the network at timestep t. $\vec{h}(t)$ is the temporary state of the hidden layer at timestep t and $\vec{h}(0)$ is set to 0. W_{ih} and W_{ho} are the weight matrices between the input-hidden layer and hidden-output layer, respectively. W_{hh} is the recurrent weight matrix between the last hidden state at timestep $t-1$ and the current hidden state at timestep t. $\vec{b}_h$ and $\vec{b}_o$ are the bias of hidden and output layers, respectively. $f(x)$ and $g(x)$ are the active functions of the hidden and output layers. The most common functions are the sigmoid and softmax function as follows:

$$Sigmoid: y(x_i) = \frac{1}{1+e^{-x_i}} \tag{3}$$

$$Softmax: y(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{4}$$

A recurrent neural network must get trained for each specific task. The training algorithm of the recurrent neural network is backpropagation through time (BPTT) [18]. The key idea of BPTT is to truncate the infinite recursion and expand the network as a finite feedforward network. Then the expanded network can be trained like other feedforward network structures by (i). calculating the actual outputs of the network for the given input data, and (ii). updating the weights of each matrix through backpropagating the deviations between the actual and desired outputs layer by layer. The update of each weight (w_{ji}) 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 η is the learning rate and $\delta_j(t)$ is the error back propagated from the node j in the next neighbour layer at timestep t. T is the BPTT step for training the recurrent neural network. $x_i(t)$ is the input of the node i.

When the active function of the output layer is softmax, and we use the cross-entropy as the loss function, the error derivative ($\delta_p(t)$) of the node p in the output layer can be calculated as follows:

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

where $o_p(t)$ and $t_p(t)$ are the actual and desired outputs of the recurrent neural network, respectively.

And when the active function in the hidden layer is sigmoid, for the node k in the hidden layer, the error derivative ($\delta_k(t)$) is:

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

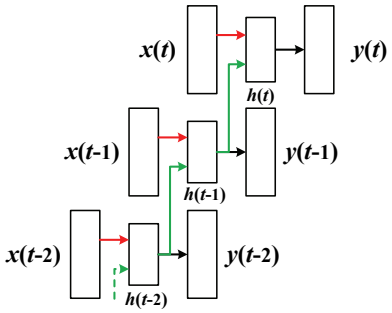

Fig. 2. Truncated RNN for Training

$$f'(x) = f(x) \cdot (1 - f(x)) \tag{8}$$

$$\delta_{BPTT}(t) = \sum_{o \in output} w_{ok}\delta_o(t) + \sum_{h \in hidden} w_{hk}\delta_h(t+1) \tag{9}$$

where $h_k(t)$ is the output of the node k in the hidden layer at timestep t and $f(x)$ is the active function of the hidden layer. δ_{BPTT} is the accumulation of the errors backpropagated through time. w_{ko} and w_{kh} are the weights of W_{ho} and W_{hh} in Eq. (1) and (2), respectively. $\delta_o(t)$ is the error of the output layer at the same timestep t. $\delta_h(t)$ is the error of the hidden layer backpropagated from the next neighbour timestep $t+1$ and $\delta_h(T+1)$ should be set to 0.

It should be noted that, although the expanded recurrent neural network looks like a deep neural network, there should be only 3 different weight matrices (W_{ih}, W_{hh} and W_{ho}) in the expanded recurrent neural network. In other words, the weight matrix between the corresponding layers at different timestep (the arrows with the same color in Fig. 2) must get updated together and stay the same because they come from the same recurrent neural network.

B. RNN Language Model

The recurrent neural network can be used as a language model in speech and nature language processing tasks. The language model is used to evaluate the correctness of a word sequence in many applications of text and speech, such as the speech recognition, spelling correction and machine translation [19].

The modern language model is based on statistics, instead of grammar, and the correctness of a sequence is represented by a probability. For a good language model, the probability of a meaningful sequence should be larger than an incorrect one as the following example:

$$P(I\ saw\ a\ dog) > P(Eye\ saw\ a\ dog)$$

The most commonly used language model is N-gram [20]. In the N-gram model, the probability of a word in a sequence depends on the last $N-1$ words before itself. For example, the probability of the following sequence is calculated through a 2-gram language model:

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

All the conditional probabilities in the equation are calculated through statistically analyzing the whole training data collected. However, there's a problem with these conditional probabilities in the N-gram language model: the number of parameters (the conditional probabilities) will increase exponentially with N. An N-gram model requires V^N conditional probabilities for a vocabulary with a size of V, which are difficult to storage. Moreover, the space of the training data will also be highly sparse for a larger N. In other words, most combinations of words, although they may be meaningful, won't exist in the training dataset. And therefore, it will become very difficult to statistically analyze the training data and calculate an efficient conditional probability. Experiment results have proven that the performance of N-gram is poor for a model with a larger N ($N \geq 5$) [21]. As a result, the N-gram can only realize a short-term perspective of the sequence and is clearly insufficient to capture semantics of sentences [22].

When the recurrent neural network is used as a language model, the sizes of the input and output layers will be set to be equal to the size of vocabulary (either full or compressed) and each node in the input or output layer will represent one or more words in the vocabulary. When calculating the correctness of a sequence, a series of words will be input into the recurrent neural network language model in sequence. For example, at the timestep T, the Tth word in the sequence will be input into the recurrent network. As a result, only the node corresponding to the Tth word in the input layer will be set to 1, and all the other input nodes will be set to 0. And then the value of the node corresponding to the $(T+1)$th word in the output layer will represent the probability of this word under the condition of all the T history input words. Finally, the probability of a sentence can be calculated by the product of all these conditional probabilities at different timesteps together.

Because of the additional recurrent connections of the recurrent neural network, the recurrent neural network language model (RNNLM) is able to calculate the conditional probabilities for any length of the input history and even represent the deep structure of the language itself [22]. Therefore, compared with N-gram, the recurrent neural network language model is able to realize a long-term perspective of the sequence. In addition, the scale of the recurrent neural network language model doesn't increase with N, the length of conditions (the input history), and there won't be a problem of the sparsity of the training data as the N-gram. Moreover, such a property also demonstrates that the recurrent neural language model is able to learn more patterns from less training data compared with the N-gram model [23].

C. RNN Training Acceleration

The training algorithm of the neural network model is an iterative process, and therefore, it's difficult to get parallelized. The hardware acceleration techniques mainly focus on utilizing the parallelism within each iteration in order to reduce the total time consumption for training.

A simple but efficient technique is to accelerate the matrix-vector multiplication operations in the algorithm. As shown in Eq. (1) and (2), the most basic operation of the recurrent neural network architecture is the matrix-vector multiplication. There have been several mature tools to accelerate this operation at different platforms, such as the Intel's MKL (Math Kernel Library) for CPU [24] and NVIDIA's CUBLAS (CUDA Basic Linear Algebra Subroutines) for GPU [25]. Many results have demonstrated that those tools are able to provide around an order of magnitude speed-up compared with the basic for-loop implementation and significantly reduce the time consumption for training.

However, although the matrix-vector multiplication can be significantly accelerated, the scale of the recurrent neural network may still be too large to realize an effective training, especially for the recurrent neural network language model. As discussed in Section II-B, the input and output layer sizes of the recurrent neural network language model are proportional to the vocabulary size. Usually, the vocabulary size can easily reach up to a size of 10K$\sim$ 200K. Therefore, a recurrent neural network language model with 100 hidden layers and 10,000 words in the vocabulary will require a 10,000$\times$100 weight matrix between the hidden and output layers. Such a large matrix is still too complicated to calculate, especially for CPU-based platforms.

A solution to significantly reduce the computation complexity of the recurrent neural network, especially the recurrent neural network language model, is to divide all the nodes in the output layer into classes and set up an additional class layer. Assuming that we need to calculate the result of node p in the output layer, which belongs to the class node c. And the recurrent neural network has V nodes in the output layer, H nodes in the hidden layer, C nodes in the class layer and there're N_c nodes belong to the class node c in total. Then when the class-based recurrent neural network needs to calculate the value of node p in the output layer, the model will first calculate the results of the class layer. Then the model will only calculate the output nodes belonging to the same class node c, instead of calculating all the nodes in the output layer. Therefore, when the class-based recurrent neural network is used as a language model, the conditional probabilities of the corresponding output node can be expressed as:

$$P(p|history) = P(p|c, history) \cdot P(c|history) \tag{10}$$

It can be seen that the computation complexity of the class-based model is proportional to:

$$C \times H \times H + N_c \times H \times H \tag{11}$$

while the the computation complexity of the intact recurrent neural network language model is proportional to [12]:

$$V \times H \times H \tag{12}$$

Therefore, with the help of the additional class layer, the computation complexity of the recurrent neural network can be significantly reduced. And the best choice of class layer size is $C = sqrt(V)$ [26]. However, the side effect of this technique is that the accuracy of the recurrent neural network will also decrease as a result of the extra deviations of the class layer.

III. GPU Implementation

A. Basic Implementation

We mainly focus on accelerating the training phase of the recurrent neural network. The training process can be abstracted as executing the following calculations repeatedly: 1) pick a labeled example from the training dataset; 2) calculate the actual output of the network; and 3) update the weights of the network with the differences between the actual and desired outputs as Eq. (5)-(9).

As mentioned in the Section II-C, we realize all the matrix-vector and matrix-matrix multiplication operations on GPUs with the CUBLAS library. In addition, we also implement other functions, such as the active functions in Eq. (3)-(4) and the calculations of errors in Eq. (6)-(7), on GPUs. Because the data transmission between CPU and GPU is very time consuming [27], we keep all the parameters (W_{ih}, W_{hh}, W_{ho}, b_h, and b_o) of a recurrent neural network in the GPU's global memory. And we also store all the states of hidden and output layers in the GPU's global memory. However, we store the training data in the main memory instead of the GPU's global memory. The reason is that in many applications of the recurrent neural network, such as the recurrent neural network language model, only one input node will be activated and only one output node will be monitored at each timestep. In other word, only a small group of index data are demanded to control the training process, and there's little benefit to store these data on GPU. However, it should be noted that a large training data, such as the images, should be stored on the GPU for efficient processing.

In addition, as only one node will be activated in the input layer, the operation of matrix-vector multiplication between the input and hidden layers can be realized by extracting the row in the weight matrix W_{ih} corresponding to the activated node. And therefore, the operation of $W_{ih} \cdot \vec{x}(t)$ in Eq. (1) can be expressed as:

$$W_{ih} \cdot \vec{x}(t) = \vec{w}_{ih}^k \tag{13}$$

$$W_{ih} = [\vec{w}_{ih}^1, ..., \vec{w}_{ih}^k, ..., \vec{w}_{ih}^n] \tag{14}$$

where $\vec{x}(t)$ is the input column vector. Only the kth element of $\vec{x}(t)$ is 1 and the other elements are 0. $\vec{w}_{ih}^k$ is the the kth column of W_{ih} as Eq. (14). And therefore, the the matrix-vector multiplication operation of $W_{ih} \cdot \vec{x}(t)$ can be simplified into an operation of copying the data of corresponding column of W_{ih} to the destination vector.

B. Pipeline Implementation

As discussed in Section II-A, the training process of the recurrent neural network includes two phases: (i). feedforward calculating the actual outputs of the network for the given input, and (ii). backpropagating the deviations between the actual and desired outputs to update the weights of the

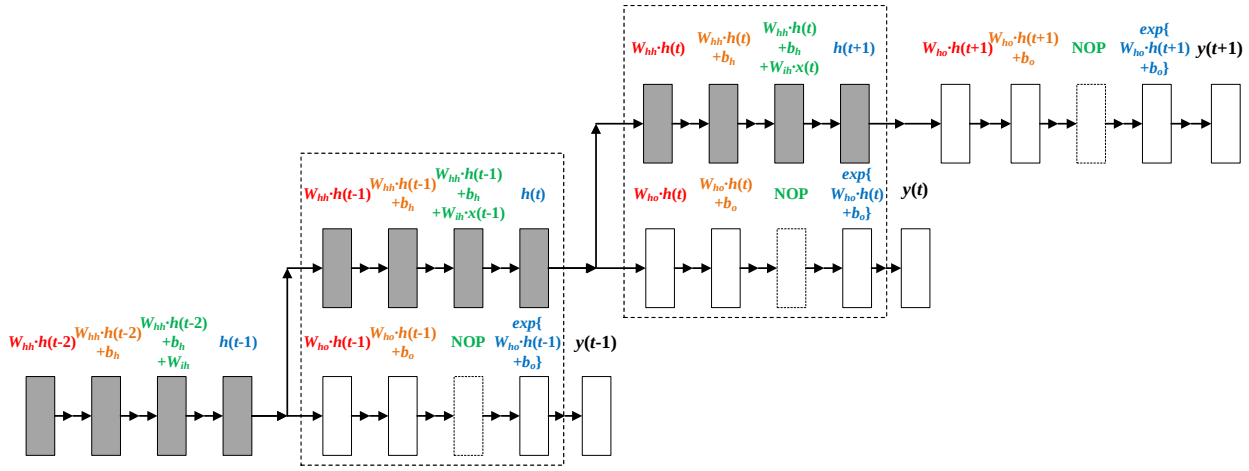

Fig. 3. Pipeline Implementation of the Feedforward Phase of RNN

network. The two phases are similar and both the phases can be implemented with two-stage pipeline architectures. Fig. 3 illustrates the pipeline architecture for the feedforward phase.

The pipeline implementation of the feedforward phase involves two stages. One stage completes the calculations of the results of hidden layer (the grey boxes in Fig. 3) and the other stage completes the calculations of the output layer's results (the white boxes in Fig. 3)). Each stage includes 4 serial steps:

1) Completing the matrix-vector multiplication;
2) Adding the bias up to the state of the layer
3) Adding the corresponding column of W_{ih} up to the hidden state, or calculating the exponent of the output layer;
4) Calculating the sigmoid function of the hidden layer, or scaling the exponential result of the output layer;

We implement the first step, second step and the steps of calculating nonlinear functions in the hidden and output layers in parallel. And the rest 2 operations (adding the corresponding row of W_{ih} up to the hidden layer, and scaling the output layer result) are implemented in serial. Therefore, there're totally 5 steps in the pipeline implementation and we label them with different colors in Fig. 3. Each pair of white and gray blocks in the same column is executed in parallel.

C. Matrix Combination

As discussed in the Section III-A, we use the CUBLAS library to implement the operations of matrix-vector multiplications. However, the functions in the CUBLAS are encapsulated. It's difficult to combine two kernels together when we want to implement the first two steps in the pipeline architecture. Therefore, in order to take full advantage of the massive parallelism of the GPU, we combine the two matrices, W_{ih} and W_{hh}, together, instead of combining the two kernels together (Fig. 4). And we also combine the two bias vectors, b_h and b_o, together. Such a technique of combining different matrices or vectors helps realize the steps in different pipeline stages in parallel.

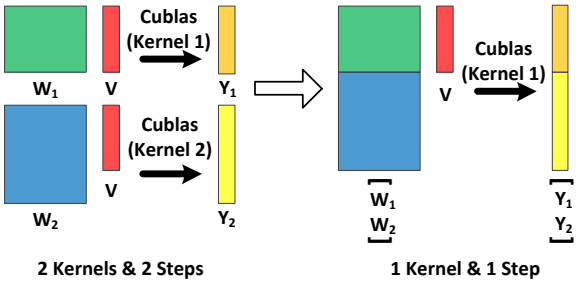

Fig. 4. Matrix Combination

A problem of the matrix combination technique is that the CUBLAS library requires the parameters of the matrices must be stored in column-major order. Such a regulation will cause a problem that the parameters of the two matrices will be crossed with each other in the combined matrix. As we must complete the matrix-vector multiplication between a single matrix, instead of the combined matrix, and the vector at both the beginning and the end of the pipeline, we have to extract a single matrix from the discontinuous combined matrix. There're two methods to extract the single matrix:

- LDA method: extract the single matrix with the `M`, `N`, and `LDA` (Leading Dimension Array) parameters in CUBLAS;
- TRANS method: store the transpositions of the combined matrix, i.e., a combination of W'_{hh} and W'_{ho}, to make the parameters of the two matrices continuous, and use the `TRANS` parameter to control the CUBLAS operations.

Both of the methods have a problem that the memory access of the matrix data may be discontinuous and may lead to a bad performance. However, the experiment results in the latter section will demonstrate that the pipeline implementation of RNN based on the LDA method performs better than the one based on the TRANS method, and can realize an additional $2 \sim 40\%$ speed-up compared with the original CUBLAS implementation without pipeline architecture.

D. Warp Formation of GPU Implementation

The warp formation in the proposed GPU implementation should be carefully considered to make a better use of the GPU architecture. The reason is that GPUs will group the

TABLE I
TIME CONSUMPTION (S) OF THE RNN 10K OUTPUT NODES.

Hidden Layer	BPTT = 2				BPTT = 5				BPTT = 10			
	Original[1]	LDA[5]	Trans[6]	MKL	Original[11]	LDA[15]	Trans[16]	MKL	Original[21]	LDA[25]	Trans[26]	MKL
100	38.38	38.14	42.93	108.29	77.64	75.88	80.89	185.93	124.41	120.71	125.58	254.12
200	47.12	46.88	68.37	145.10	92.61	89.22	128.15	223.38	144.17	137.03	199.02	371.52
500	77.60	77.41	149.45	315.45	147.03	143.65	282.14	876.56	217.28	209.27	436.51	1174.07
1000	131.58	132.01	145.78	1141.18	245.69	244.69	274.39	2019.25	353.61	350.40	399.22	4022.88

TABLE II
TIME CONSUMPTION (S) OF THE RNN WITH 10K OUTPUT NODES AND WARP FORMATION IN THE HIDDEN LAYER

Hidden Layer	BPTT = 2				BPTT = 5				BPTT = 10			
	Original[2]	LDA[7]	Trans[8]	MKL	Original[12]	LDA[17]	Trans[18]	MKL	Original[22]	LDA[27]	Trans[28]	MKL
128	40.04	38.66	47.28	96.63	80.71	76.50	89.03	150.59	128.75	120.99	137.84	264.07
256	51.29	49.53	74.66	128.42	99.48	94.77	139.01	240.56	153.27	144.00	213.21	389.44
512	75.61	74.04	78.51	429.09	142.30	137.90	145.70	871.02	209.62	201.05	210.34	1490.79
1024	132.49	131.01	135.27	1238.60	247.47	241.97	249.88	2385.18	355.82	345.07	354.39	3796.22

TABLE III
TIME CONSUMPTION (S) OF THE RNN WITH 10K OUTPUT NODES AND WARP FORMATION IN BOTH HIDDEN AND OUTPUT LAYERS

Hidden Layer	BPTT = 2				BPTT = 5				BPTT = 10			
	†Separate[4]	Original[3]	LDA[9]	Trans[10]	†Separate[14]	Original[13]	LDA[19]	Trans[20]	†Separate[24]	Original[23]	LDA[29]	Trans[30]
128	38.99	40.04	39.25	47.71	81.05	77.73	77.31	89.41	128.74	123.43	121.81	138.75
256	49.87	51.55	49.89	74.45	100.09	95.93	95.14	139.39	154.39	146.07	143.39	212.93
512	74.44	75.85	73.78	79.43	142.73	139.20	137.52	146.31	209.63	203.68	199.24	212.49
1024	131.46	133.15	130.45	137.09	247.32	243.16	239.78	253.30	353.78	346.65	338.92	359.63

†: There're 10,000 nodes in the output layer in this simulation. We divide the combination kernels in the pipeline implementation (one used for active functions, and the other one used for derivative calculations) into two separate kernels. And there won't be any branch in the separate kernels.

threads executing the same code into fixed sized SIMD batches known as *warps* and a warp can execute an instruction for all its threads in parallel. However, a warp can only have one active PC instruction at any given time. Conditional branch instructions may cause the threads to take different dynamic execution paths, or *diverge*, and therefore, those threads may execute in sequence and the performance may be bad [28]. As a result, there's a major consideration of formatting warps to avoid the conditional branch instructions in the programs (kernels) executed on the GPU.

In order to avoid the divergence in GPU kernels, the first solution is to divide the combination kernel in the pipeline implementation back into two separate kernels. For example, the kernel of calculating the sigmoid functions in the hidden layer and the kernel of calculating the exponential functions in the output layer are combined into one kernel for the pipeline implementation. But the combination kernel will be divide back into two separate kernels in order to make sure that there won't be any branch in the separate kernels and avoid the divergence. However, such an implementation cannot take full advantage of the GPU architecture. And a better solution to avoid the divergence is to try to fit in the warp size of the GPU. In other words, we will fill the nodes in the output layer with extra dummy nodes and make the length of the output vector an integral multiple of the warp size. For example, the current warp size of NVIDIA GPU is 32 [29]. If there're 10,000 nodes in the output layer in practice, we will set the length of the output vector to 10,016. The extra 16 nodes involve in computation just to avoid diverges and there will be no use of these nodes' results. In addition, we also suggest to set the length of hidden layer as an integral multiple of the warp size.

E. Experiment Results

We conduct a simulation to compare the performance of different implementations. We use the recurrent neural network as a language model as described in the Section II-B. We select 50,000 words from the Treebank corpus [30] as the training data for the test. The vocabulary size of the dataset is 10,000. The training depth (BPTT step) in the test is set to 2, 5 and 10. All the GPU implementations are tested on a NVIDIA GeForce GTX580 GPU with 1.5GB global memory. And the CPU implementations are tested on two Intel Xeon E5-2690s (@2.9GHz with 16 cores in total). The 'Original' represents the GPU implementation based on CUBLAS without pipeline architecture. The 'LDA' and 'Trans' represent the pipeline implementations based on the 'LDA Method' and 'TRANS Method' described in the Section III-C. The 'Separate' represents the GPU implementation based on Matrix Combination and separate kernels for the calculations of nonlinear functions. And the 'MKL' is the CPU implementation with Intel Math Kernel Library (MKL). And the results can be concluded as follows:

1) The proposed GPU implementation is able to realize a $2 \sim 11\times$ speed-up compared with the CPU implementation. And the accelerating rate (the times of speed-up) increases with the hidden layer size and BPTT step.
2) The pipeline implementation doesn't work well with a network with a small hidden layer. The reason is that there're too many extra nodes used for warp formation ($> 20\%$ extra computation when setting the hidden layer from 100 (200) to 128 (256)). However, when the size of the network grow up, the overhead of the extra nodes will decrease, and the pipeline architecture will become efficient.

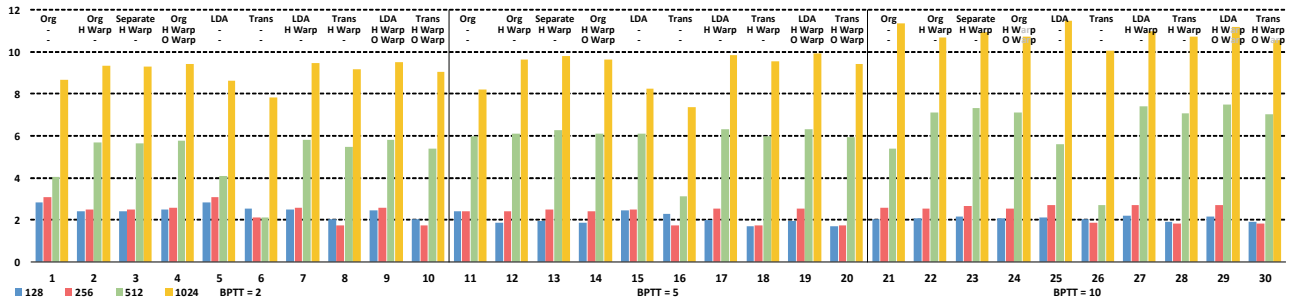

Fig. 5. Speedup results of GPU implementations. The indexes represent the superscripts in Table I, II, and III

3) The pipeline implementation based on the LDA method performs better than the one based on the TRANS method. And when there's no optimization on the GPU's warp formation, the pipeline implementation based on the TRANS method may even perform worse than the original implementation without pipeline.
4) There really exists a problem of divergence with GPU implementations. And the warp formation method of filling data up to the integral multiple of warp size is more efficient than the method of separating the combination kernels.
5) The warp formatted (both in the hidden layer and output layer) implementation based on the LDA method and pipeline architecture performs better than other implementations in most cases. Such a method is able to achieve an additional $2\% \sim 37.2\%$ speed-up compared with the original implementation without pipeline architecture or warp formation.

IV. A Case Study: Microsoft Research Sentence Completion Challenge

A. The Microsoft Research Sentence Completion Challenge and Experiment Setup

We use the large scale recurrent neural network as a language model as a case study to test its performance. And we use the Microsoft Research Sentence Completion (MRSC) Challenge to test the RNN language model. The MRSC challenge intended to stimulate research into language modeling techniques which are sensitive to overall sentence coherence [31]. The challenge consists of fill-in-the-blank questions similar to those widely used in the Scholastic Aptitude Test. Fig. 6 illustrates the sample questions in the challenge. And there're 1,040 tests in total in the challenge.

In the experiment, we use the recurrent neural network language model to calculate the score (probability) of the sentence filled with each given option. We choose the option that leads to the highest score of the complete sentence as the final answer of the model. The training data of the model are a large set of Nineteenth and early Twentieth Century novels and there're around 38M words in the training dataset. The vocabulary size of the training data is around 60K. However, many of the words in the training dataset only appear a few times. Therefore, we statistically analyze the frequencies of words in the training data. We retain the 9K words that

1. I have seen it on him , and could _____ to it.
   (a) write
   (b) migrate
   (c) climb
   (d) swear
   (e) contribute

2. They seize him and use violence towards him in order to make him sign some papers to make over the girl's _____ of which he may be trustee to them.
   (a) appreciation
   (b) activity
   (c) suspicions
   (d) administration
   (e) fortune

3. My morning's work has not been _____ , since it has proved that he has the very strongest motives for standing in the way of anything of the sort.
   (a) invisible
   (b) neglected
   (c) overlooked
   (d) wasted
   (e) deliberate

Fig. 6. The first three questions of the Microsoft Research Sentence Completion Challenge

appear frequently and merge the other low-frequency words into 1,583 'words'. And we try to make the frequencies of those merged 'words' be equal with each other. Therefore, there will be 10,583 nodes in the output layer of the recurrent neural network model. We set the size of the hidden layer to 1,000 and the BPTT step to 5 when training this large recurrent neural network.

B. Experiment Results

The results of different methods are illustrated in Table IV and Fig. 5. It can be seen that the the RNN language model can realize a loner-term perspective of the sentence compared with the N-gram model as discussed in the Section II-B. And an accuracy of 47% demonstrates that the proposed large scale RNN is able to beat the traditional class-based modest-size recurrent network. In addition, The Maximum Entropy RNN model (RNNME) trained on 50M tokens is able to achieve an accuracy of 49.3%. Therefore, both combining RNN with other models and increasing the training data may lead to an improvement on the model's performance. Finally, the vLBL+NCE5 model achieved the best result by

TABLE IV
PERFORMANCE OF DIFFERENT METHODS ON THE MRSC CHALLENGE

Method	Accuracy (%)
Chance	20
Smoothed 4-gram in [31]	39
RNN-100 with 100 classes[1]	40
Proposed	47
RNNME-300 in [32][2]	49.3
vLBL+NCE5 in [33][3]	60.8
Human	91

[1] The model is trained with the RNNLM Toolkit in [26].
[2] The model is trained on about 50M tokens using 200K vocabulary.
[3] The best result we have seen so far.

constructing a model of noise (to distinguish noise and real language) and analyzing a group of words at the same time (note that the RNNLM only considers 1 word at one timestep). And therefore, we may apply those methods to our RNNLM model to process more information and further improve the performance.

V. CONCLUSION

In this paper we propose an efficient GPU implementation of the large scale recurrent neural network, which achieves a $2 \sim 11\times$ speed-up compared with the basic CPU implementation with the Intel MKL. We then explore the effectiveness of scaling up the recurrent neural network with GPUs. The experiment results of the MRSC Challenge demonstrate that the large scale recurrent network is able to beat the traditional modest-size RNN and achieve an accuracy of 47%, the best result achieved by a single recurrent neural network on the same dataset. However, the performance of the model still falls far behind the human intelligence. There's still a lot of work to do to further improve the model's performance.

ACKNOWLEDGMENTS

This work was supported by the Microsoft Research Asia (MSRA), National Natural Science Foundation of China (No. 61373026), 973 project 2013CB329000, Tsinghua University Initiative Scientific Research Program, and Youth Talent Development Plan of Beijing (YETP0099). And we gratefully acknowledge the support of NVIDIA Corporation with the donation of the GPUs used for this research.

REFERENCES

[1] B. Brown, M. Chui, and J. Manyika, "Are you ready for the era of big data?" *McKinsey Quarterly*, vol. 4, pp. 24–35, 2011.
[2] N. . Workshop. Modern nonparametric methods in machine learning. [Online]. Available: http://nips2013.sched.org/2013-12-09/list/descriptions/type/workshops
[3] R. Menéndez de Llano and J. L. Bosque, "Study of neural net training methods in parallel and distributed architectures," *Future Generation Computer Systems*, vol. 26, no. 2, pp. 267–275, 2010.
[4] I. Basheer and M. Hajmeer, "Artificial neural networks: fundamentals, computing, design, and application," *Journal of microbiological methods*, vol. 43, no. 1, pp. 3–31, 2000.
[5] J. Dean *et al.*, "Large scale distributed deep networks," in *NIPS*, 2012.
[6] K. Yu, "Large-scale deep learning at baidu," in *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management*. ACM, 2013, pp. 2211–2212.
[7] I. Sutskever, J. Martens, and G. E. Hinton, "Generating text with recurrent neural networks," in *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, 2011, pp. 1017–1024.
[8] R. J. Williams and D. Zipser, "A learning algorithm for continually running fully recurrent neural networks," *Neural computation*, vol. 1, no. 2, pp. 270–280, 1989.
[9] L. Deng, J. Li, J.-T. Huang, K. Yao, D. Yu, F. Seide, M. Seltzer, G. Zweig, X. He, J. Williams *et al.*, "Recent advances in deep learning for speech research at microsoft," *ICASSP 2013*, 2013.
[10] J. Skaruz and F. Seredynski, "Recurrent neural networks towards detection of sql attacks," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*. IEEE, 2007, pp. 1–8.
[11] J. Martens and I. Sutskever, "Learning recurrent neural networks with hessian-free optimization," in *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, 2011, pp. 1033–1040.
[12] T. Mikolov, A. Deoras, D. Povey, L. Burget, and J. Cernocky, "Strategies for training large scale neural network language models," in *Automatic Speech Recognition and Understanding (ASRU), 2011 IEEE Workshop on*. IEEE, 2011, pp. 196–201.
[13] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," Technical Report, Tech. Rep., 2012.
[14] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, "Deep learning with cots hpc systems," in *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, vol. 28, no. 3, May 2013, pp. 1337–1345.
[15] W. W. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient gpu control flow," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2007, pp. 407–420.
[16] R. Raina, A. Madhavan, and A. Y. Ng, "Large-scale deep unsupervised learning using graphics processors." in *ICML*, vol. 9, 2009, pp. 873–880.
[17] J. Ngiam, A. Coates, A. Lahiri, B. Prochnow, A. Ng, and Q. V. Le, "On optimization methods for deep learning," in *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, 2011, pp. 265–272.
[18] M. Bodén, "A guide to recurrent neural networks and backpropagation," *The Dallas project, SICS technical report*, 2002.
[19] D. Jurafsky and J. H. Martin, *Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition*. MIT Press, vol. 2.
[20] A. Pauls and D. Klein, "Faster and smaller n-gram language models." in *ACL*, 2011, pp. 258–267.
[21] F. Jelinek, *Statistical methods for speech recognition*. MIT press, 1997.
[22] T. Mikolov, S. Kombrink, L. Burget, J. Cernocky, and S. Khudanpur, "Extensions of recurrent neural network language model," in *Acoustics, Speech and Signal Processing (ICASSP)*, 2011.
[23] T. Mikolov, M. Karafiát, L. Burget, J. Cernockỳ, and S. Khudanpur, "Recurrent neural network based language model," in *INTERSPEECH*, 2010, pp. 1045–1048.
[24] Intel, "Intel math kernel library," http://software.intel.com/en-us/intel-mkl/.
[25] C. Nvidia, "Cublas library," *NVIDIA Corporation, Santa Clara, California*, vol. 15, 2008.
[26] T. Mikolov, S. Kombrink, A. Deoras, L. Burget, and J. Černockỳ, "Rnnlm-recurrent neural network language modeling toolkit," in *Proc. IEEE workshop on Automatic Speech Recognition and Understanding*, 2011, p. 16.
[27] V. Volkov and J. W. Demmel, "Benchmarking gpus to tune dense linear algebra," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, 2008, p. 31.
[28] V. Narasiman *et al.*, "Improving gpu performance via large warps and two-level warp scheduling," in *Micro*. ACM, 2011, pp. 308–317.
[29] NVIDIA, "Gpu computing sdk," https://developer.nvidia.com/gpu-computing-sdk.
[30] M. P. Marcus, M. A. Marcinkiewicz, and B. Santorini, "Building a large annotated corpus of english: The penn treebank," *Computational linguistics*, vol. 19, no. 2, pp. 313–330, 1993.
[31] G. Zweig and C. J. Burges, "A challenge set for advancing language modeling," in *Proceedings of the NAACL-HLT 2012 Workshop: Will We Ever Really Replace the N-gram Model? On the Future of Language Modeling for HLT*. Association for Computational Linguistics, 2012, pp. 29–36.
[32] T. Mikolov, "Statistical language models based on neural networks," Ph.D. dissertation, Ph. D. thesis, Brno University of Technology, 2012.
[33] A. Mnih and K. Kavukcuoglu, "Learning word embeddings efficiently with noise-contrastive estimation," in *NIPS*, 2013, pp. 2265–2273.