Accelerating Deep RNN Inference with multiple FPGAs

Yuxi Sun[1,a)] Akram Ben Ahmed[1] Hideharu Amano[1]

Abstract: In this paper, we propose an acceleration methodology for deep recurrent neural networks (RNNs) implemented on a multi-FPGA platform called Flow-in-Cloud (FiC). RNNs have been proven effective for modeling temporal sequences, such as human speech and written text. However, the implementation of RNNs on traditional hardware is inefficient due to their long-range dependence and irregular computation patterns. This inefficiency manifests itself in the proportional increase of run time with respect to the number of layers of deep RNNs when running on traditional hardware platforms such as a CPUs. Previous works have mostly focused on the optimization of a single RNN cell. In this work, we take advantage of the multi-FPGA system to demonstrate that we can reduce the run time of deep RNNs from $O(k)$ to $O(1)$.

1. Introduction

Recurrent neural networks have been for a long time the state-of-art for many sequence-based tasks, such as speech recognition, machine translation, text generation, etc. There exist several types of RNN cells, including simple RNN [15], Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU). Simple RNN cells are capable of modeling character-level languages [1] as well as recognizing hand-written characters. The LSTM architecture outperforms simple RNNs in modeling long sequences, and overcomes the vanishing gradient problem of simple RNNs.

In contrast to convolutional neural networks (CNNs), whose architectures allow massive parallelism by reuse of filter weights, RNNs are harder to be accelerated on hardware due to their high complexity and temporal dependence. Compared to the plethora of literature on accelerating CNNs with FPGAS, relatively few FPGA-based approaches have been proposed to accelerate RNNs. One prominent example include *ESE* [2], an efficient hardware architecture dedicated to LSTMs designed by *H. Song et. al*, which utilizes parameter pruning and quantization. Authors in [3] also proposed an FPGA-based acceleration of LSTMs using compression techniques and claimed to outperform ESE. In addition to LSTMs, there is also another work that focuses on accelerating GRUs [4].

However, all of the works mentioned above took only single-layer RNNs into consideration, whereas deep RNN architectures are becoming more widely adopted recently. For example, *H. Sak et. al.* demonstrated that a multi-layer LSTM RNN with projection layers outperforms its single-layer counterpart when evaluated with the Google Voice Search task [5].

Starting from the aforementioned facts, we propose in this paper a multi-FPGA based approach for accelerating deep RNNs which achieves single-layer speed for arbitrarily deep RNN architectures. The system is composed of instances of our custom designed FiC boards which are connected by high-speed serial links. The data flows among the boards via STDM (Static Time Division Multiplexing) switches [6]. To accelerate deep RNNs on the FiC system, each layer of a RNN can be implemented on one FiC board, so that different layers are computed simultaneously.

2. Deep RNN Architectures

Unlike CNNs, deep RNNs can be constructed in a variety of ways [7]. In this work, we only consider stacked RNNs and bidirectional RNNs. We use LSTM to illustrate the deep architectures of RNNs, which should apply also for other types of RNN cells.

2.1 RNN Background

All recurrent neural networks have the form of a chain of repeating modules, each depending on the output of its predecessor. In simple RNNs, the repeating module is a single layer composed of a hyperbolic tangent function ($tanh$):

$$h_t = \sigma_h(W_{xh}x_t + W_{hh}h_{t-1} + b_h), \qquad (1)$$

where x_t denotes the input sequence, and h_t denotes the hidden vector sequence. LSTM also has this chained structure, but with a more complicated repeating module:

[1] Keio University
3-14-1 Hiyoshi, Yokohama,223-8522, Japan
[a)] hikari@am.ics.keio.ac.jp

$$f_t = \sigma_g(W_f x_t + U_f h_{t-1} + b_f) \quad (2)$$
$$i_t = \sigma_g(W_i x_t + U_i h_{t-1} + b_i) \quad (3)$$
$$o_t = \sigma_g(W_o x_t + U_o h_{t-1} + b_o) \quad (4)$$
$$c_t = f_t \circ c_{t-1} + i_t \circ \sigma_c(W_c x_t + U_c h_{t-1} + b_c) \quad (5)$$
$$h_t = o_t \circ \sigma_h(c_t) \quad (6)$$

f_t, i_t and o_t represent the outputs at a certain time step of the forget, input and output gates, respectively. These three gates have the same gating function σ_g. LSTMs keep two types of states: hidden state h_t and cell state c_t. The hidden state is typical of RNNs, and simple RNN and GRU also use hidden states to keep previous information. The cell state is the key to LSTMs. The LSTM regulates what information should be added or removed to the cell state by using the three gates. It can also allow an uninhibited flow of information through the states [8]. LSTM with recurrent projection layer (LSTMP) is a variation of the standard LSTM proposed by Google, which is also known as Google LSTM [5]. LSTMP has demonstrated success in large-scale acoustic modeling tasks while saving a lot of parameters comparing to the standard LSTM. LSTMP differs from the standard LSTM in that the hidden state vectors h_t are multiplied with a projection matrix W_p of dimension $n_p \times n_c$, where n_p is the number of units in the projection layer, which is usually smaller than n_c.

$$h_t = o_t \circ \sigma_h(c_t) \circ W_p \quad (7)$$

A standard LSTM cell has $4 \times n_c \times n_c + 4 \times n_i \times n_c + 4 \times b_c$ parameters, while an LSTMP cell has $4 \times n_c \times n_p + 4 \times n_i \times n_c + n_p \times n_c + 4 \times b_c$. For inner layers in a deep RNN architecture, where n_i equals n_c and n_p for LSTM and LSTMP cells respectively, LSTMP is able to reduce the number of parameters by a factor of $\frac{n_p}{n_c}$.

Stacked RNNs

When stacking RNNs, the hidden states h_t from the lower RNN cell are input into the RNN cell above. Fig. 1 demonstrates how one LSTM cell can be stacked on top of another. Stacking LSTMs has been proven effective for acoustic modeling [5]. The inputs to a deep RNN go through multiple non-linear layers, so that at each layer, the information is processed by the network unrolled in time. Authors in [9] stated that deep layers in RNNs allow the network to learn at different time scales over the input.

Bidirectional RNNs

Bidirectional RNN (BRNN) is another RNN architecture that has the ability to learn from not only the past input sequence, but also future input. It accomplishes this with a second layer where the input sequence is reversed, which results in the reversed flow through the hidden states

3. The FiC platform

Recently, FPGAs have been receiving a lot of attention thanks to their power efficiency and flexibility. Since energy efficiency is one of the most important issues in recent cloud computing, a lot of research using FPGAs in the cloud

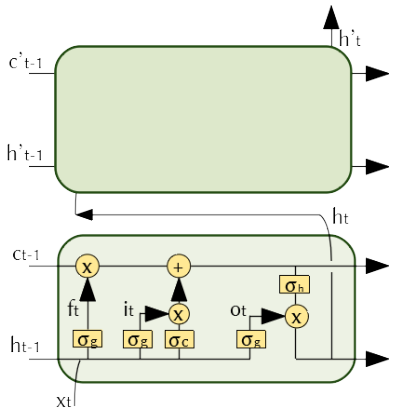

Fig. 1: A Stacked LSTM RNN

have been conducted [10], and commercial systems including Amazon's F1 instance have become available. However, the performance improvement that can be gained by using FPGAs is limited by the on-board resources of a single FPGA. To deal with this problem, we have been developing a multi-FPGA system called Flow-in-Cloud (FiC). FiC is consisting of a number of middle-scale economical FPGAs interconnected with high communication bandwidth network.

3.1 The prototype FiC system

The prototype FiC system, depicted in Fig. 2, is consisted of a number of FPGA-based boards connected with each other, and an I/O board with high speed serial links. A Xilinx Kintex Ultrascale (XCKU095) FPGA running at 100 MHz is mounted on each board, and GTH serial links each of which provides 8.5Gbps are used for interconnection.

The entire system is connected with a single host computer through an I/O board using Xilinx KCU1500. The I/O board is connected to PCIe gen3 of the host computer and 8 serial links to some FiC-SW boards. The data for the computation are transferred through the I/O board and distributed to multiple boards in FiC. Large amounts of data can be stored in on-board 16GB of DDR4-SDRAM modules. The on-board Raspberry Pi 3 with Linux based OS is in charge of controlling and configuring the FPGA. The total system is designed considering cost and energy efficiency by excluding expensive components.

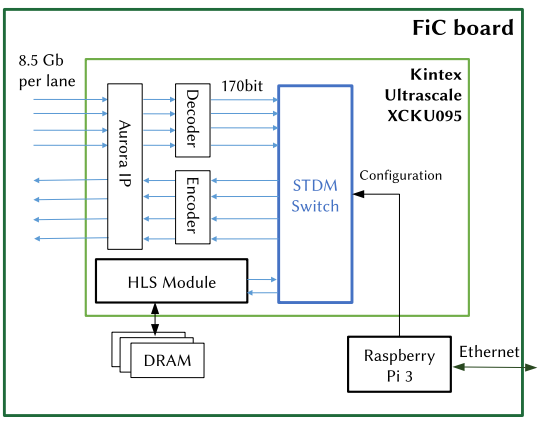

Fig. 2: Block diagram of FiC board

Fig. 3: The structure of a FiC data packet.

3.2 The FPGA Interconnection

The FiC boards are connected using STDM switches [6]. Currently, each switch has three ports, one of which is for incoming data from the HLS module, and the other two are for outgoing data packets to other FiC boards. The ports are connected with *link*s (Firefly cables). One serial link is composed of four lanes. As previously mentioned, each lane has a bandwidth of 8.5 Gbps, so the total bandwidth is 34 Gbps when all of the four lanes are used for inter-switch communication. One incoming and one outgoing link is connected to the Xilinx Aurora IP (Intellectual Property), which communicates with the switch logic via the AXI4-Stream interface.

The data packets that go through the switch must follow the format shown in Fig. 3. When generating the data packets, the HLS module can specify when the packets should be sent by programming the `Local Slot` field. A consistency check between the `Local Slot` value and the routing table setting is performed in the hardware. The `UDID` (User-defined ID) and `Data` fields can be used for application data. The highest bit is a valid bit and is set by the switch logic.

3.3 Partial Reconfiguration

The FPGA on a FiC board is divided into two areas: the static area consisting of the Xilinx Aurora IP and the STDM switch, and the dynamic area consisting of the HLS module. The application resides in the dynamic area, i.e. the HLS module.

The static and dynamic areas share the resources of a single FPGA, so that a switch with a larger number of ports will result in reduced resources for the application in the dynamic area.

4. Implementation

We implemented various deep RNN architectures on the FiC system by dividing each layer to a FiC board. The RNN cells are implemented using the Vivado HLS tool. The synthesized RNN cells are exported as IPs and integrated into the rest of the FPGA design using Vivado IP Integrator.

4.1 HLS implementation of RNN Cells

We implemented simple RNN cells and LSTM cells using arbitrary-precision fixed-point numbers [11]. In this paper, we only discuss the results of the fixed-point LSTM implementation. Compared to CNNs, RNNs have relatively few parameters, so that the parameters of a not-so-large RNN can be stored directly on the FPGA.

There are 8 matrix-vector multiplications in a single LSTM cell, as previously explained in Section 2.1. On an FPGA, these can be computed simultaneously using 8 instances of matrix-vector multiplication modules. Inside the multiplication function, the top level-loop is pipelined

In a typical LSTM-RNN, *sigmoid* and *tanh* functions are commonly used for activation functions. These computationally expensive nonlinear functions can be quantized and implemented as a look-up table, as represented in Fig. 4. in this fashion, computing the activations only takes one clock cycle.

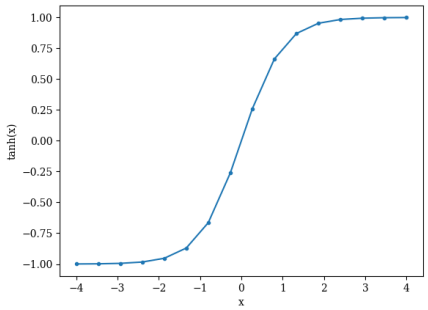

Fig. 4: Quantized tanh activation function

For a deep LSTM with an input vector size of 16 (which can be the number of acoustic features in the case of speech recognition) and 128 hidden states. The post-implementation utilization reported by Vivado is shown in 1. The DSP48E resources are underutilized due to Vivado HLS' constraint that computation of under 10 bits can be implemented as FF and LUT only.

Table 1: Post-implementation Resource Utilization of an LSTM cell with projection

	BRAM_36K	DSP48E	FF	LUT
Avail.	1680	768	1075200	537600
Used	1010	8	68348	323753
Utili.	60%	1%	8%	60%

4.2 Data Switching

The computation pattern of a two-layer stacked RNN is shown in Fig. 5. Each x_t in the input sequence is a vector of size 16. In Fig. 5, `A` and `B` stand for the RNN cells, which are in our case LSTMs.

The first layer starts computing first. As soon as the first vector of hidden states h_0 is ready, it goes to the next FiC board via the switch. At the same time, h_0 is used with x_1 to compute the next hidden state vector h_1. This results in a nearly simultaneous computation of h'_{t-1} and h_t, except for the overhead of data packet processing in the switch. The initial overhead of computing h_0 can be covered if the inference of continuous input is pipelined.

An arbitrary amount of layers can be stacked by increasing the number of FiC boards used, and the total computation time will stay almost constant. The data switching patterns of a four-layer RNN is illustrated in Fig. 6.

Bidirectional RNNs are also suitable to be accelerated with FiC, since the forward and backward layers can run independently on two different FiC boards To build a deep bidirectional RNN such as the one described in [12], the forward and backward layers can be stacked in a similar way as Fig. 5. For classification tasks, the fully-connected layer can

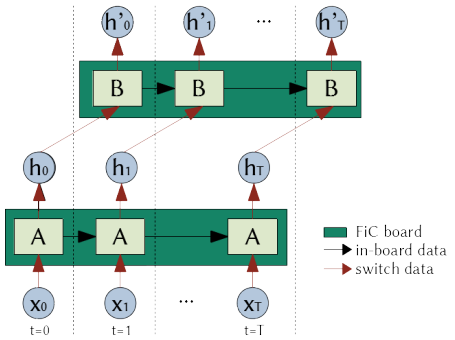

Fig. 5: Stacked RNN implemented on multiple FiC boards. The hidden states h_t are routed from one FiC board to another via the switch.

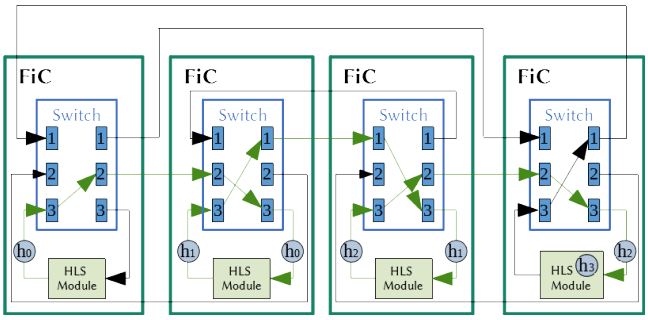

Fig. 6: An overview of data path of a 4-layer RNN. The black arrows represent the possible connections between boards, and the green ones are the connections where the data (hidden states of the RNN) flows through.

be placed on another FiC board, which receives the forward and backward states from two other boards.

4.3 Quantization

The input data, parameters and activations of the RNN models are quantized to 8 bit numbers. To quantize an LSTM cell, quantization operations have to be inserted after every numeric operation in the LSTM computation graph. These quantization operations are provided by the quantization module [13] of Tensorflow, and can be expressed as following:

`fake_quant_with_min_max_vars` quantizes the input tensor to a specified range and bit length, and stores the results in floating point. It is retrainable, so that if a drop in accuracy is observed after applying quantization, the quantized model can be retrained to achieve desired accuracy. The quantization operation corresponds to the following equation:

$$q_k(x) = \frac{round(clamp(x, min, max) * (2^k))}{2^k},$$

which quantizes the inputs x to the range $[-2^{k-1}, -2^{k-1})$.

We applied this quantization techniques to two datasets: MNIST handwritten digit dataset and TensorFlow Speech Command dataset [14], and found out that 8-bit implementation results in neglectable drop in accuracy, as shown in Table 2. For the TensorFlow Speech Command dataset, mel-frequency cepstral coefficients (MFCCs) are used to extract speech features from the raw audio data.

Table 2: Original and 8-bit Quantized Accuracy Comparison for Two Datasets

	MNIST	TF Speech Command			
Accuracy (%)	1xLSTM	1xLSTM	4xLSTM	1xLSTMP	2xLSTMP
original	97.9	92.7	94.09	93.1	94.2
8-bit fake quant.	97.5	92.9	94.03	92.8	94.0
8-bit fixed-point	97.3	92.7	-	-	94.0

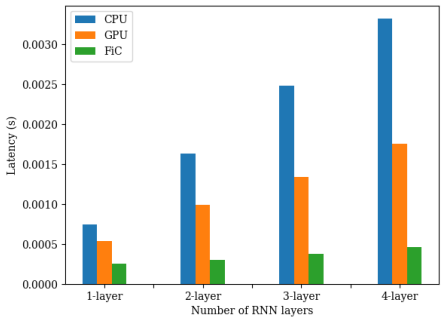

Fig. 7: Speed of 8-bit LSTM Inference on different platforms.

5. Evaluation

We compare the FiC implementation of deep RNNs against an Intel i7-7500U CPU (laptop) and a NVIDIA GeForce 940 MX GPU. The CPU and GPU performance is measured using the Keras framework with the TensorFlow backend. For measuring the speed of LSTM on GPU, the highly optimized `CudnnLSTM` implementation is used. The FiC performance is measured on the real FiC system prototype. The time to run an inference can be read directly from the waveform in the Vivado Hardware Manager, and the results are shown in Fig. 7.

We evaluated the performance of four stacked LSTM architectures of different depths, using 8-bit quantized model. On a CPU, the runtime of an inference increases proportionally with the number of layers. The GPU does a better job at scaling-up than CPU, but a significant increase in runtime can still be observed for 3-layer and 4-layer LSTMs. In contrast, on the FiC system, computation of different layers can be parallelized on separate FiC boards; thus, the latency is almost not affected by the number of layers. For the FiC implementation, the latency increases slightly with the increase of layers. This is largely due to the transmission latency between switches, plus the overhead of the conversion between the 16-bit data and the packet data, as previously demonstrated in Fig. 3.

Table 3 compares FiC against CPU in terms of GOPS and GOPS/Watt performance. The GOPS performance of CPU only scales up slightly when the RNN gets deeper. In contrast, FiC demonstrates superior scalability because of the increase of hardware resources. The power performance (GOPS/W) of the FiC system is evaluated by measuring the voltage and current on the socket (Fig. 8). While running

a 4-layer LSTM RNN, the four FPGAs on the FiC boards consumes around 16 W in total. This power performance is comparable to that of a modern laptop. It is important to mention that the power consumption of the Raspberry Pi is also taken into consideration when measuring the entire FiC system power.

Table 3: Performance comparison between CPU and FiC

	1-layer	2-layer	3-layer	4-layer
CPU (GOPS)	1.046	1.226	1.252	1.269
FiC (GOPS)	0.778	2.190	3.202	3.884
CPU (GOPS/W)	0.074	0.088	0.089	0.091
FiC (GOPS/W)	0.195	0.274	0.267	0.243

For CPU the number of floating-point operations is counted, and for FiC fixed-point operations. The CPU power is measured using Intel Power Gadget.

6. Conclusion and Future Work

In this paper, we presented a method to accelerate deep RNNs with a multi-FPGA system called FiC. By taking advantage of the parallelism provided by multiple FPGAs and the time dependence of RNNs, the latency of a deep RNN can be reduced to that of a single-layer RNN. This is a considerable speed-up in comparison to CPUs or GPUs, where the latency of deep RNNs increases proportionally with the number of layers. The FiC system shows good scalability in terms of GOPS performance, and is superior to CPUs in terms of power consumption.

Our future work will focus on two different aspects: the optimization of the HLS implementation of RNN cells (such as LSTMs) and the reduction of communication overhead among HLS modules and switches.

Acknowledgments This paper is based on results obtained from a project commissioned by the New Energy Industrial Technology Development Organization (NEDO).

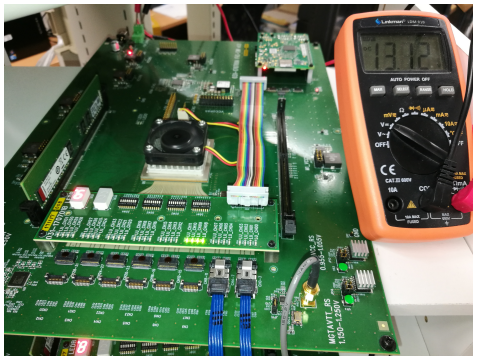

Fig. 8: Power measurement while FiC boards are running

References

[1] A. Karpathy: *The Unreasonable Effectiveness of Recurrent Neural Networks*, `http://karpathy.github.io/2015/05/21/rnn-effectiveness/` (2019.02.08).
[2] Song Han et al: *ESE: efficient speech recognition engine with compressed LSTM on FPGA*
[3] Sh. Wang, Z. Li, C. Ding, B. Yuan, Y. Wang, Q. Qiu, and Y. Liang: *C-LSTM: enabling efficient LSTM using structured compression techniques on FPGAs*
[4] Ch. Gao, D. Neil, E. Ceolini, Sh.-Ch. Liu, and T. Delbruck: *Deltarnn: A power-efficient recurrent neural network accelerator*, FPGA '18, pages 21–30. ACM, 2018.
[5] Hasim Sak, Andrew W. Senior, and Françoise Beaufays: *Long short-term memory based recurrent neural network architectures for large vocabulary speech recognition*
[6] Kazusa Musha, Tomohiro Kudoh, and Hideharu Amano. *Deep learning on high performance FPGA switching boards: Flow-in-Cloud*, Applied Reconfigurable Computing, 2018.
[7] Razvan Pascanu, Çaglar Gülçehre, Kyunghyun Cho, and Yoshua Bengio: *How to construct deep recurrent neural networks*,
[8] Christopher Olah: *Understanding LSTM Networks*, `http://colah.github.io/posts/2015-08-Understanding-LSTMs/` (2019.02.08).
[9] Michiel Hermans and Benjamin Schrauwen: *Training and analysing deep recurrent neural networks*, Advances in Neural Information Processing Systems 26, pages 190–198 (2013).
[10] Andrew Putnam et al: *A reconfigurable fabric for accelerating large-scale datacenter services*, ISCA '14, pages 13–24, 2014.
[11] Xilinx: *Vivado Design Suite User Guide High-Level Synthesis.*
[12] Alex Graves, Navdeep Jaitly, and Abdel rahman Mohamed: *Hybrid speech recognition with deep bidirectional lstm*, In IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU), 2013.
[13] *TensorFlow 1.13*, `https://www.tensorflow.org/api_docs/python/tf/quantization/quantize`
[14] *Speech commands: A public dataset for single-word speech recognition* Pete Warden
[15] Jeffrey L. Elman: *Finding structure in time*, COGNITIVE SCIENCE (1990).