Recurrent neural networks (RNNs) have been proven effective for sequence-based tasks such as speech recognition and machine translation. However, the implementation of deep RNNs on traditional hardware platforms is inefficient due to long-range temporal dependence and irregular computation patterns within RNNs. This inefficiency manifests itself in the proportional increase in the latency of RNN inference with respect to the number of layers of deep RNNs on CPUs and GPUs. Previous work has focused mostly on optimizing and accelerating individual RNN cells. To make deep RNN inference fast and efficient, we propose an accelerator based on a multi-FPGA platform called Flow-in-Cloud (FiC). In this work, we show that the parallelism provided by the multi-FPGA system can be taken advantage of to scale up the inference of deep RNNs, by partitioning a large model onto several FPGAs, so that the latency stays close to constant with respect to increasing number of RNN layers. For single-layer and four-layer RNNs, our implementation achieves 31x and 61x speedup compared with an Intel CPU.

key words: multi-FPGA, recurrent neural networks, LSTM

1. Introduction

Recurrent neural networks have been widely used to solve a variety of sequence-based tasks, such as speech recognition, machine translation, text generation, etc. There exist several types of RNN cells, including simple RNN [1], Long Short-Term Memory (LSTM) [2] and Gated Recurrent Unit (GRU) [3]. The LSTM RNNs can outperform simple RNNs in modeling very long sequences, and overcomes the vanishing gradient problem of simple RNNs. GRU has similar structure as LSTM, but is less computationally intense.

In contrast to convolutional layers in Convolutional Neural Networks (CNNs) which have high compute-to-data ratio, RNN layers perform less computation for fixed amount of weight data. Therefore RNNs are more challenging to be scaled up in hardware implementation due to their complicated dataflow and long-range temporal dependence. Comparing to the plethora of literature on accelerating CNNs with FPGAs [4]–[6], relatively few efforts have devoted to accelerating RNNs. Among the research on RNN acceleration, one prominent example is ESE [7], an FPGA-based accelerator dedicated to quantized and sparse LSTMs. [8] proposes a structured compression technique so that matrix-vector multiplications can be accelerated using the FFT algorithm. [9] focuses on accelerating GRUs by applying the zero-skipping technique to matrix-vector multiplications that are dominant in a GRU layer. The E-RNN framework [10] unites the efforts to accelerate LSTMs and GRUs and proposes an acceleration approach that can be applied to both cells.

All of the above-mentioned works focus on accelerating certain RNN cells such as LSTMs and GRUs, assuming sparsity in the model. However, little research has been done to accelerate deep RNNs, which have become widely adopted recently, especially for large-scale speech recognition systems [11], [12]. Accelerating deep RNNs is challenging because first, the data size increases proportionally with regard to the depth, and second, the dependency exists both time-wise and layer-wise.

The difficulty to fit a large model into a single FPGA, is often approached by partitioning the model to multiple FPGAs [13]–[15]. We adopt the idea of model partitioning and propose a multi-FPGA based accelerator for deep RNNs. We implement our solution on the FiC system [16], where custom designed FPGA boards which are connected with high-speed serial links.

In this paper, we propose FiC-RNN, an acceleration framework for deep RNN with the multi-FPGA platform FiC. This paper is an extension of previous work [17], and we develop a tool for model partition of deep RNN models and generation design files for each FPGA. We show that our solution can achieve near single-layer latency for arbitrarily deep RNN models by exploiting the parallelism of connected FPGAs.

2. Background

2.1 RNN Cells

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:

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

where $x_t$ denotes the input sequence, and $h_t$ denotes the hidden vector sequence.

LSTM-RNNs also have this chained structure, but with a more complicated repeating module:
\[ f_t = \sigma_g(W_f x_t + U_f h_{t-1} + b_f) \]  
\[ i_t = \sigma_g(W_i x_t + U_i h_{t-1} + b_i) \]  
\[ o_t = \sigma_g(W_o x_t + U_o h_{t-1} + b_o) \]  
\[ c_t = f_t \odot c_{t-1} + i_t \odot c_t(W_c x_t + U_c h_{t-1} + b_c) \]  
\[ h_t = o_t \odot \sigma_h(c_t) \]

\( f_t, i_t \) and \( o_t \) represent the outputs at a certain time step of the forget, input and output gates respectively, which have the same gating function \( \sigma_g \). \( W \) and \( U \) are weight matrices, and \( b \) represents bias vectors. In addition to the hidden state \( h_t \) which are typical in all RNN cells, LSTMs also keep track of its cell state \( c_t \), which enables an uninhibited flow of information from timestep to timestep.

A variation of LSTM, namely LSTM with a projection layer, is proposed by [11]. LSTMP differs from 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 with a lower dimension, so that LSTMP needs less parameters than standard LSTM.

2.2 Deep RNN Architectures

Deep RNNs can be constructed in a variety of ways [18]. 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.

Stacked LSTM-RNNs have been proven effective for acoustic modeling [11]. In stacked RNNs, the hidden states \( h_t \) from the lower RNN cell are input into the RNN cell above. Figure 1 demonstrates how the one LSTM cell can be stacked together. The inputs at timestep \( t \) \( (x_t) \) go through multiple layers, which allows the network to learn at different time scales over the input [19].

Bidirectional RNNs (BRNNs) represent another type of deep RNNs that are able to learn from not only the past, but also future input. This is accomplished by using a second layer where the input sequence is processed reversely in time, which results in the opposite information flow through time.

3. The Multi-FPGA Platform Flow-in-Cloud

Recently, FPGAs have received a lot of attention as a potential solution to many of the important issues in cloud computing, especially energy efficiency and flexibility. A lot of research efforts have been devoted to bringing FPGAs into data centers across academia and industry [20]. However, the performance gain of using FPGAs is limited by the onboard resources of a single FPGA. The Flow-in-Cloud (FiC) system addresses this challenge by offering large number of interconnected FPGAs.

3.1 The Prototype FiC System

The prototype FiC system consists of a number of FPGA-based boards, whose structure is shown in Fig. 2. A Xilinx Kintex Ultrascale (XCKU095) FPGA running at 100 MHz is mounted on each board, and the on-board Raspberry Pi 3 with Linux-based OS is in charge of controlling and configuring the FPGA.

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, where the application resides. The HLS module can be developed independently of the static area, and integrated using the partial reconfiguration technique [21].

The communication between the FiC system to the outer world is provided by an KCU1500-based I/O board, which is connected to a host machine via PCI Express bus. The I/O board can be connected up to 8 FiC boards using Firefly cables.
3.2 The FPGA Interconnection

The FiC boards are connected via STDM switches [16], each of which has up to five ports, one of which is for incoming data from the HLS module, and the others are for outgoing data packets to other FiC boards. The ports are connected with Firefly cables, which we refer to as lanes. One lane is composed of four links with a bandwidth of 8.5 Gbps each, amounting to a total bandwidth is 34 Gbps when all of the four lanes are used for inter-switch communication. The links are connected to the Xilinx Aurora IP, which provides connectivity with the FPGA logic via the AXI4-Stream interface. To enable inter-FPGA communication via the switches, the top-level function of the HLS module can be designed as follows:

```c
void hls_module(data_t input[N_INPUT],
    output_t output[N_OUTPUT], ...
    output_t output_n[N_OUTPUT])
{
    #pragma HLS INTERFACE axis port=input
    #pragma HLS INTERFACE axis port= output, ...,
    // data processing and writing to output ports
    }
```

The switch controls data communication following a routing table, which can be configured either by the on-board Raspberry Pi or via the I/O board. The routing table specifies the source port and destination port for the data packets, and their designated time slots under the STDM scheme.

4. Implementation

We implemented various deep RNN architectures on the FiC system by partitioning one layer to a FiC board. The design process can be divided into three stages: training and quantizing of RNN models, automatically generating optimized RNN cell and partitioning deep RNN models to individual FPGAs.

4.1 Quantization

We apply 8-bit quantization to the input data, parameters and activations of the RNN models to achieve better efficiency. To quantize an LSTM cell, quantization operations have to be inserted after every numeric operation in the LSTM computation graph, as shown in Fig. 3. These quantization operations can be expressed as following:

\[ q_k(x) = \text{round}(\text{clamp}(x, -n, n) \times 2^k) / 2^k, \]

where the input \( x \) is clamped to the range \([-n, n]\) and uniformly quantized to \( k \)-bit precision.

In our implementation, we set \( k \) to 8 and quantize the data tensors to 8-bit accuracy. After the usual training of the model, we modify the computation graph by injecting the quantization ops, which are retrainable, and retrain the model until the accuracy is recovered.

We apply this technique to different RNNs trained on the TensorFlow Speech Command dataset [22], and quantized the models to 8 bits with neglectable drop in accuracy, as shown in Table 1. The input acoustic feature vectors are mel-frequency cepstral coefficients (MFCC) extracted from the raw audio, and have a dimension of 16. For the LSTM models, the number of hidden units for all layers is 128. For the LSTMP units, the number of hidden units and the number of projection units are 128 and 64 respectively. It can also be observed that deeper RNNs can achieve higher accuracy than shallow ones.

![Fig. 3 Applying quantization to the computation graph of an LSTM.](image)

| # layers | LSTM | LSTMP |
|----------|
| original accuracy (%) | 92.7 | 94.1 | 93.1 | 94.2 |
| 8-bit accuracy (%) | 92.7 | 94.0 | 92.8 | 94.0 |

4.2 RNN Cell Implementation

We implemented LSTM and LSTMP cells using arbitrary-precision fixed-point numbers using the Vivado HLS tool [23]. Other RNN cells can be implemented in a similar way.

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. The number of parameters in a RNN cell can be computed as \( n_c \times n_c \times 4 + n_i \times n_c \times 4 + n_o \times 4 \), where \( n_c \) is the number of cell states, \( n_i \) and \( n_o \) are the input and output size respectively. With a precision being 8-bit, a large LSTM cell of size 1024 needs around 8.4MB of memory, which safely be stored in the on-board BRAM. LSTMP cells of the same cell size have even lower memory requirement. In our implementation, we therefore choose to pin all the weights to the BRAM, which saves time and energy overhead of loading parameters from external DRAMs. This is viable for deep RNNs thanks to the abundant BRAM resources provided by the multi-FPGA system.

The computation of an LSTM cell includes matrix-
vector multiplications, elementwise vector multiplication and addition, as well as nonlinear activation functions. The micro architecture of the HLS module is as shown in Fig. 4. The hidden states $h$ and cell states $c$ are stored in on-chip buffers. The packet encoder and decoder interface with the STDM switches are described in Sect. 3.2.

At each timestep, the decoded inputs and hidden states are fed into parallel matrix-vector multiplication units (MVM), each of which contains pipelined multiply-and-accumulate units. The elementwise-multiply is fully unrolled, and the accumulate is implemented as an adder tree. The outputs of MVM go to activation and addition units (AAU) which fuse element-wise addition with activation functions. The vector-vector multiplication (VVM) units implement elementwise vector multiplication and produce final output for the packet encoder.

The activation functions are implemented as look-up tables (LUTs). For RNNs, sigmoid and tanh functions are commonly used as activation functions. The otherwise computationally expensive nonlinear functions can be quantized and implemented as LUTs on an FPGA to reduce the computation to a single clock cycle. Figure 5 shows a uniformly quantized hyperbolic tangent function.

4.3 Model Partition

The trained and optimized RNN model is partitioned and each layer goes to one FiC board, as illustrated in Fig. 6.

To ease the design flow, we develop a model partitioning tool which reads the RNN model files generated by Keras, and creates C++ files for deep RNN inference from hand-optimized template. To help the tool identify different code section for different boards, the programmer should manually insert tags into the C++ code. Based on the inserted tags, the tool resumes to partition the original C++ files into smaller files, each containing a single RNN cell, and calls the command-line utility vivado_hls to automatically build the HLS projects. As the last step, each synthesized HLS project is imported to an independent Vivado project, which generates a bitstream for a single FPGA.

The computation of an inference is pipelined across multiple FPGAs. The first layer starts computing $h_0$ first, which is sent to the next FPGA as soon as the computation finishes. At the next timestep, $h_0$ is used by FPGA0 to compute $h_1$, and by FPGA1 to compute $g_0$ at the same time. This results in simultaneous computation of hidden states across all layers, except for an overhead of data packet processing and transfer.

Using this method, the computation of continuous stream of input data can also be 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.

Bidirectional RNNs are also suitable to be accelerated on the FiC system, by running a forward layer and a backward layer independently on two different FiC boards.

5. Evaluation

We compare the FiC implementation of deep RNNs against an Intel XEON CPU E5-2680 v2 @ 2.80GHz and an NVIDIA Tesla K20m GPU. The evaluated models are
LSTM-RNNs with depths of 1 to 4, each of which contains 128 hidden units. Each inference takes 98 timesteps.

The FPGA resource utilization of a single layer is summarized in Table 2. Note that DSPs are not used at all due to the usage of low-precision fixed-point values.

The CPU and GPU performance is measured using the Keras framework with the TensorFlow backend [24]. For the GPU evaluation, the highly optimized CudnnLSTM kernel is used. We compare the CPU and GPU performance against the results measured on the real FiC system. We consider small-batch inference, which is common in data center requests. For benchmarking the CPU and GPU implementation, the latency is averaged over a total of 10 inferences.

For measuring the latency of the FiC system, the wall time of RNN inference is read directly from the waveform with help of the ILA IP debug core. The results are as shown in Fig. 7.

The power performance in terms of GOPS per watt is evaluated. We measured the voltage and current consumed by the FiC system directly on the socket (see Fig. 8). While running a four-layer LSTM-RNN, the whole system with four working boards consumes around 16W in total. For measuring power on CPU and GPU, we use the Intel Power Gadget and the nvidia-smi utility respectively. The results are summarized in Fig. 7. We observe that the FiC implementation outperforms both CPU and GPU in terms of latency and power efficiency. While the latency on CPU and GPU increases almost linearly with regards to the number of layers, our implementation shows better scalability. For the single-layer RNN, our implementation is about 31x faster than CPU and 2x faster than GPU; for the four-layer RNN, the acceleration rate is 61x and 5x respectively. It should be mentioned that in the ideal case, the latency of FiC should stay constant for RNNs of 2 to 4 layers, whereas in reality the latency rises slightly due to data encoding and decoding, as well as the transmission latency in the network. [25] provides a more in-depth analysis of the communication cost of the FiC system.

In addition, our implementation is more energy efficient than CPU and GPU for all evaluated models. Note that the energy efficiency does not scale as well as the speed due to the fact that deep RNN implementation requires more FPGA resources and consumes therefore more power.

6. Conclusion

In this paper, we have presented a novel method to accelerate deep RNNs on a novel multi-FPGA platform. 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. In addition, the FiC system demonstrates superior scalability in comparison to CPUs or GPUs, where the latency of deep RNNs increases proportionally with the number of layers.

Acknowledgements

This work was supported by JST CREST Grant Number JP-MJCR19k1, Japan.
References

[1] J.L. Elman, “Finding structure in time,” COGNITIVE SCIENCE, vol.14, no.2, pp.179–211, 1990.
[2] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” Neural Comput., vol.9, no.8, pp.1735–1780, Nov. 1997.
[3] J. Chung, Ç. Gülçehre, K. Cho, and Y. Bengio, “Empirical evaluation of gated recurrent neural networks on sequence modeling,” CoRR, vol.abs/1412.3555, 2014.
[4] Y. Umuroğlu, N.J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, “Finn: A framework for fast, scalable binarized neural network inference,” Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA ’17, pp.65–74, 2017.
[5] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, “Optimizing fpga-based accelerator design for deep convolutional neural networks,” Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA ’15, pp.161–170, 2015.
[6] K. Abdelouahab, M. Pelcat, J. Sérot, and F. Berry, “Accelerating CNN inference on fpgas: A survey,” CoRR, vol.abs/1806.01683, 2018.
[7] S. Han, Y. Wang, H. Yang, W.B.J. Dally, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, and S. Yao, “Ese: Efficient speech recognition engine with sparse lstm on fpga,” Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA ’17, pp.75–84, 2017.
[8] S. Wang, Z. Li, C. Ding, B. Yuan, Q. Qiu, Y. Wang, and Y. Liang, “C-lstm: Enabling efficient lstm using structured compression techniques on fpgas,” Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA ’18, pp.11–20, 2018.
[9] C. Gao, D. Neil, E. Ceolini, S.-C. Liu, and T. Delbruck, “Deltarnn: A power-efficient recurrent neural network accelerator,” Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA ’18, pp.21–30, 2018.
[10] Z. Li, C. Ding, S. Wang, W. Wen, Y. Zhuo, C. Liu, Q. Qiu, W. Xu, X. Lin, X. Qian, and Y. Wang, “E-rnn: Design optimization for efficient recurrent neural networks in fpgas,” 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp.69–80, 2019.
[11] H. Sak, A.W. Senior, and F. Beaufays, “Long short-term memory based recurrent neural network architectures for large vocabulary speech recognition,” CoRR, vol.abs/1402.1128, 2014.
[12] A. Hannun et al., “Deep speech: Scaling up end-to-end speech recognition,” CoRR, vol.abs/1412.5567, 2014.
[13] C. Zhang, D. Wu, J. Sun, G. Sun, G. Luo, and J. Cong, “Energy-efficient CNN implementation on a deeply pipelined FPGA cluster,” Proceedings of the 2016 International Symposium on Low Power Electronics and Design, ISLPED 2016, San Francisco Airport, CA, USA, pp.326–331, Aug. 08 - 10, 2016.
[14] W. Zhang, J. Zhang, M. Shen, G. Luo, and N. Xiao, “An efficient mapping approach to large-scale dnn on multi-fpga architectures,” 2019 Design, Automation Test in Europe Conference Exhibition (DATE), pp.1241–1244, 2019.
[15] E. Chung and J. Fowers, “Accelerating persistent neural networks at datacenter scale,” Hot Chips 29, Session 6, 2017.
[16] K. Musha, T. Kudoh, and H. Amano, “Deep learning on high performance FPGA switching boards: Flow-in-cloud,” Applied Reconfigurable Computing, Architectures, Tools, and Applications - 14th International Symposium, ARC 2018.
[17] Y. Sun, A. Ben Ahmed, and H. Amano, “Acceleration of deep recurrent neural networks with an fpga cluster,” Proceedings of the 10th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies, HEART, pp.1–4, 2019.
[18] R. Pascau, C. Gulcehre, K. Cho, and Y. Bengio, “How to con-struct deep recurrent neural networks,” Proceedings of the Second International Conference on Learning Representations (ICLR 2014), arXiv:1312.6026, 2014.
[19] M. Hermans and B. Schrauwen, “Training and analysing deep recurrent neural networks,” in Advances in Neural Information Processing Systems 26, ed. C.J.C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K.Q. Weinberger.
[20] A. Putnam, A.M. Caulfield, E.S. Chung, D. Chiu, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G.P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P.Y. Xiao, and D. Burger, “A reconfigurable fabric for accelerating large-scale datacenter services,” Proceedings of the 41st Annual International Symposium on Computer Architecture, ISCA ’14, pp.13–24, 2014.
[21] M. Yamakura, K. Hironaka, K. Azegami, K. Musha, and H. Amano, “The evaluation of partial reconfiguration for a multi-board FPGA system ficsw,” Proceedings of the 10th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies, HEART, pp.1–4, 2019.
[22] P. Warden, “Speech commands: A dataset for limited-vocabulary speech recognition,” CoRR, vol.abs/1804.03209, 2018.
[23] Xilinx, Vivado Design Suite User Guide High-Level Synthesis, ug902, Feb. 2018.
[24] M. Abadi et al., “Tensorflow: A system for large-scale machine learning,” 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), 2016.
[25] K. Ito, K. Izuka, K. Hironaka, Y. Hu, M. Koibuchi, and H. Amano, “Implementing a Multi-ejection Switch and Making the Use of Multiple Lanes in a Circuit switched Multi-FPGA System,” CANDAR Workshop, 2020.

Yuxi Sun received B.Sc. degree from RWTH Aachen University, Germany, in 2017. She is a double-degree master student at Keio University and RWTH Aachen University in the presence.

Hideharu Amano received Ph.D. degree from the Department of Electronic Engineering, Keio University, Japan in 1986. He is currently a professor in the Department of Information and Computer Science, Keio University. His research interests include the area of parallel architectures and reconfigurable systems.