Scalable Multi-FPGA Acceleration for Large RNNs with Full Parallelism Levels

Dongup Kwon†, Suyeon Hur†, Hamin Jang†, Eriko Nurvitadhi‡, Jangwoo Kim†*

†*Dept. of Electrical and Computer Engineering*, *Seoul National University*
{dongup, suyeon339, hamin.jang, jangwoo}@snu.ac.kr

‡*Intel Corporation*
eriko.nurvitadhi@intel.com

***Abstract*—The increasing size of recurrent neural networks (RNNs) makes it hard to meet the growing demand for real-time AI services. For low-latency RNN serving, FPGA-based accelerators can leverage specialized architectures with optimized dataflow. However, they also suffer from severe HW under-utilization when partitioning RNNs, and thus fail to obtain the scalable performance.**

In this paper, we identify the performance bottlenecks of existing RNN partitioning strategies. Then, we propose a novel RNN partitioning strategy to achieve the scalable multi-FPGA acceleration for large RNNs. First, we introduce three parallelism levels and exploit them by partitioning weight matrices, matrix/vector operations, and layers. Second, we examine the performance impact of collective communications and software pipelining to derive more accurate and optimal distribution results. We prototyped an FPGA-based acceleration system using multiple Intel high-end FPGAs, and our partitioning scheme allows up to 2.4x faster inference of modern RNN workloads than conventional partitioning methods.

I. Introduction

Recurrent neural networks (RNNs) are used in a wide range of applications [1]–[3], and the growing demand for real-time AI services has stimulated neural network (NN) accelerators with diverse hardware platforms. In particular, FPGAs offer both high performance and easy programmability by utilizing their reconfigurable architectures. Recent studies show that FPGA-based NN accelerators can outperform GPUs by incorporating specialized architectures and low-precision floating-point formats [4], [5]. Modern GPUs offer high peak throughput, but it is still challenging to fully utilize their processing elements (PEs) with a small batch [4], [5]. Meanwhile, ASICs provide high energy efficiency by employing NN-specific architectures, but they cannot adopt new RNN models.

Recently, emerging large-scale RNNs [3] pose a new challenge as they require substantial on-chip memory size and many operations. Fig. 1a demonstrates the total number of weights in RNNs. Google's neural machine translation (GNMT) consists of more than 200M parameters with eight-layer LSTMs. Besides, modern RNNs consist of more than two layers (e.g., 2-8), and each layer includes 8M-32M multiply-and-accumulate (MAC) operations. Moreover, the increasing length of input sequences further aggravates the situation.

To overcome the challenges of fast-growing RNN models, modern acceleration architectures utilize multiple accelerators

*Corresponding author.

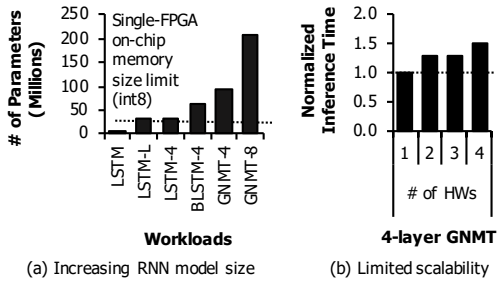

Fig. 1. Increasing model size and limited scalability of modern RNNs. The hidden dimensions of LSTM, BLSTM, GNMT are 1024. LSTM-Large (LSTM-L) has 2× hidden dimension. BLSTM denotes a bidirectional LSTM.

[4]. However, existing *model parallelism* methods suffer from severe *HW underutilization* when they distribute RNNs to multiple accelerators and thus fail to obtain the scalable performance. Fig. 1b shows GNMT gets slower even when using multiple accelerators since a simple layer-wise model parallelism scheme cannot appropriately exploit the parallelism of the GNMT model.

In this paper, we identify the performance bottlenecks of existing multi-accelerator RNN partitioning strategies. First, the existing layer-wise partitioning experiences low HW utilization when it is applied to RNN models with complex data dependencies. Also, the existing intra-layer operation-level partitioning does not guarantee the optimal distribution due to load imbalance among the accelerators. Moreover, existing schemes lead to sub-optimal partitioning decisions because they do not consider the performance overhead of collective communications as well as the performance impact of software pipelining.

To address the scalability challenges, **we introduce full parallelism levels of modern RNNs and propose a systematic strategy to obtain optimal parallelism degrees at each level.** First, we perform *two-dimensional* RNN graph analysis to obtain the exact layer-level parallelism degrees and determine the appropriate number of FPGAs dedicated to each layer. Second, we leverage *matrix-level row-wise* partitioning analysis in addition to the conventional operation-level partitioning. Third, given partitioned RNN dataflow graphs, we derive more detailed performance estimation results by considering *collective communications* and *software pipelining* which enables more optimal distribution results.

For evaluation, we prototyped an FPGA-based acceleration

978-1-7281-1085-1/20/$31.00 ©2020 IEEE

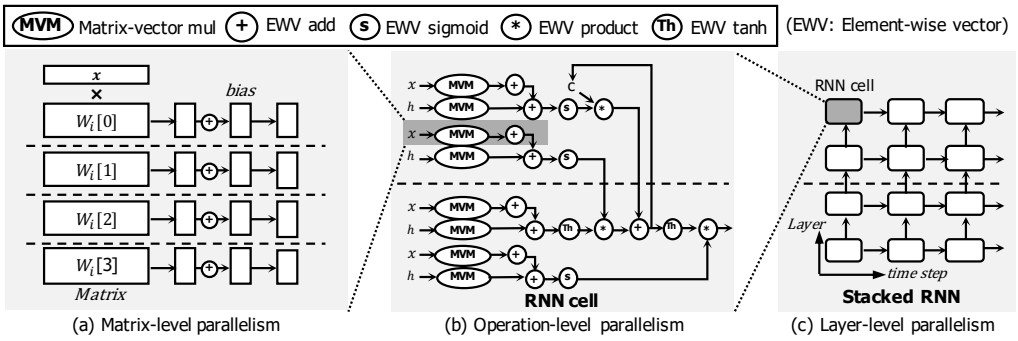

Fig. 2. Parallelism levels of large-scale and stacked RNNs from fine-grained (left) to coarse-grained (right) operations. (a) Row and column blocks in matrices, (b) matrix/vector instructions in gate operations, and (c) layers in stacked RNN models can be executed in parallel by assigning each to separate FPGAs. For example, parallel executable operations are separated by the dotted lines in this figure.

system using multiple Intel high-end FPGAs, and our partitioning scheme allows up to 2.4x faster inference of modern RNNs than conventional multi-FPGA partitioning methods.

II. Background

A. Recurrent Neural Network

1) Basic RNN Operation: An RNN (e.g., LSTM, GRU) consists of several weight matrices and performs *matrix/vector operations* on given input sequences. We use an LSTM as a main example throughout the paper. An LSTM contains eight separate weight matrices (W_i, W_f, W_o, W_c, U_i, U_f, U_o, U_c) and bias vectors (b_i, b_f, b_o, b_c), and following gate operations:

$$\begin{aligned} f_t &= \sigma(W_f \cdot x_{t-1} + U_f \cdot h_{t-1} + b_f) \\ i_t &= \sigma(W_i \cdot x_{t-1} + U_i \cdot h_{t-1} + b_i) \\ c_t &= f_t * c_{t-1} + i_t * tanh(W_c \cdot x_{t-1} + U_c \cdot h_{t-1} + b_c) \\ o_t &= \sigma(W_o \cdot x_{t-1} + U_o \cdot h_{t-1} + b_o) \\ h_t &= o_t * tanh(c_t) \end{aligned} \quad (1)$$

x_t, h_t, i_t, f_t, o_t, and c_t in Eq. (1) denote input, internal state, input gate, forget gate, output gate, and memory cell at any point t, respectively. $*$, $+$, σ, and $tanh$ indicate element-wise vector multiplication, addition, sigmoid, and hyperbolic tangent activation function, respectively.

2) Large-scale stacked RNNs: Modern RNNs comprise *multiple layers* as well as *large weight matrices*. For example, GNMT has a multi-layer *encoder-decoder* structure [3]. Modern RNNs also consist of different types of layers and complex dependencies between layers. GNMT's bottom encoder layers are bidirectional, which process input sequences in both forward and backward directions, and decoder layers have data dependencies between input sequences.

B. Parallelism Levels in Modern RNNs

Fig. 2 illustrates multi-layer RNNs and their parallelism levels existing from fine-grained to coarse-grained levels as follows.

Matrix. Matrix-vector multiplications can be partitioned into multiple row or column blocks (Fig. 2a).

Operation. Independent intra-layer matrix/vector instructions can also be partitioned and distributed (Fig. 2b).

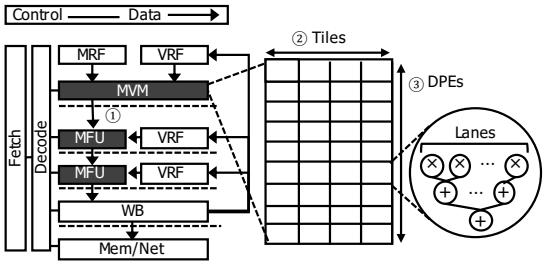

Fig. 3. Baseline FPGA-based RNN accelerator architecture.

Layer. Multiple layers in a stacked RNN can be executed in parallel by assigning each layer to separate FPGAs (Fig. 2c).

At each parallelism level, the parallelism degrees are defined by RNN models and continue to grow as models become larger and stacked for more high-quality services. However, existing partitioning schemes suffer from severe *HW under-utilization* when partitioning RNNs, and thus fail to obtain the scalable performance.

III. Accelerator Architecture

To identify the performance bottlenecks of existing RNN partitioning strategies, we prototyped an FPGA-based acceleration architecture using Intel Stratix10 FPGAs.

ISA. Our accelerator executes general matrix/vector operations similar to Microsoft's Brainwave [4]. It performs matrix-vector multiplications and various element-wise vector operations with the parameterized length of matrix and vector operands. It also executes necessary load and store operations to access operands in memory or through a communication interface.

Matrix/Vector Functional Unit. Fig. 3 illustrates a *decoupled* architecture and its hierarchical arithmetic units. In this architecture, matrix, vector, and memory units are decoupled and controlled independently. First, its *matrix-vector multiplier* (MVM) and *multi-function units* (MFUs) have direct datapaths between them for vector chaining (see ① in Fig. 3). Second, the MVM consists of many SIMD arithmetic units. The MVM has multiple tiles that split a matrix into sub-column blocks (see ② in Fig. 3), and each tile has many dot-product engines (DPEs) (see ③ in Fig. 3). Each DPE executes the same number of element-wise multiplications as the number of lanes. Third, the MFUs execute element-wise vector operations (i.e., add, subtract, multiply) and activation functions. Also, the MVM

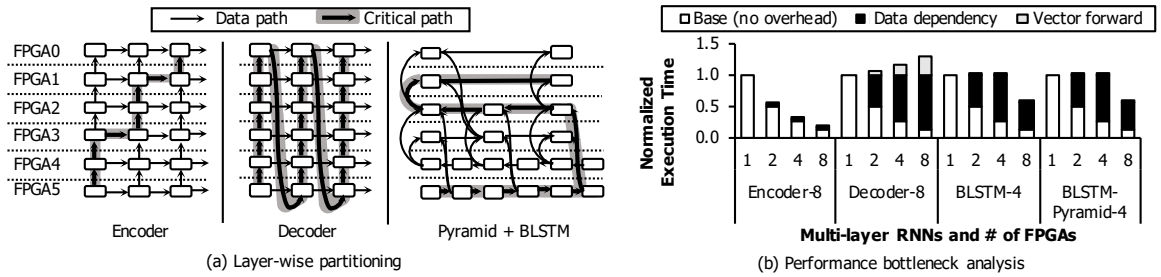

Fig. 4. Layer-wise partitioning and its performance bottleneck analysis. The critical paths across multiple layers make it difficult to fully utilize FPGAs with layer-wise partitioning. Directed arrows across dotted lines denote vector-forward communications.

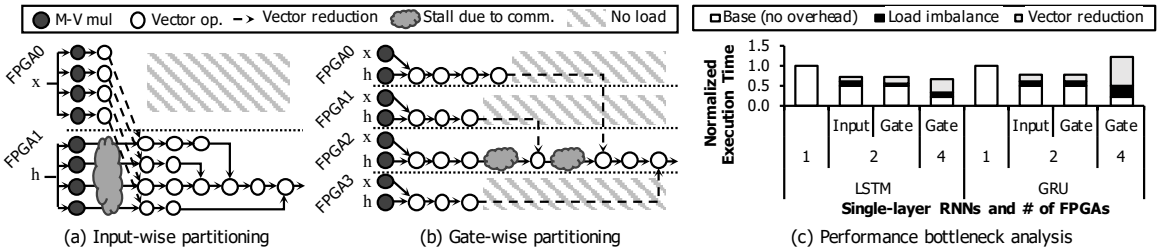

Fig. 5. Operation-level partitioning of an LSTM with (a) input-wise and (b) gate-wise dimensions and their performance bottlenecks.

and MFUs have separate vector register files (VRFs) and matrix register files (MRFs).

Vector Chaining. The architecture can easily take advantage of vector chaining due to the direct datapaths between the MVM and MFUs. A set of DPEs partitions a weight matrix into multiple row blocks and forwards the sub-block results to the MFUs. As a result, the MFUs can start independent instructions upon receiving the first sub-block result from the MVM. Since RNN variants consist of many subsequent matrix/vector operations, pipelining them with vector chaining can reduce pipeline bubbles significantly.

Persistent RNN. We implement persistent RNN by keeping entire weight matrices in FPGA's on-chip memory (e.g., M20K, BRAM). This approach has many benefits in real-time RNN inference by avoiding memory bottlenecks for accessing weights in memory. This approach has been implemented and evaluated by many previous studies [4]–[6].

IV. Performance Bottleneck Analysis

In this section, we identify the performance bottlenecks of existing RNN partitioning strategies. We see that *layer-wise partitioning* experiences severe under-utilization of HW resources when it is applied to RNN models having complex data dependencies. Besides, the existing *intra-layer operation-level* partitioning does not guarantee the optimal distribution due to the huge *load imbalance* and many *vector-reduction communications*. In the following subsections, we explain four performance bottlenecks of existing partitioning strategies.

A. *Complex Data Dependency*

The conventional *layer-wise* partitioning assigns a disjoint set of layers to separate FPGAs. Fig. 4a demonstrates that the multi-layer encoder has high concurrency between layers on different input data, which is more useful to the layer-wise partitioning. However, the layer-wise partitioning becomes ineffective when it handles multiple layers that have dependencies between different input sequences (e.g., bi-directional, decoder, pyramid). For instance, decoder-like layers have dependencies between the topmost output of the previous time step and the bottommost input of the current time step. In addition, *vector forwarding* happens between FPGAs for sending output data to the next layer as an input. These *non-uniform* and *strongly dependent* RNN models become prominent as they can find relationships between input/output sequences or reduce the number of required time steps.

Figure 4b shows the overhead of data dependencies and vector forwarding. Except for the encoder, other multi-layer benchmarks fail to obtain the scalable performance with multiple FPGAs due to severe hardware under-utilization. The bi-directional layers (i.e., BLSTM-4), on the other hand, achieve up to 2x speedup results with eight FPGAs. Their forward and backward directions can be assigned to different FPGAs when the number of FPGAs is more than the number layers.

B. *Load Imbalance and Vector Reduction*

The conventional *intra-layer* partitioning methods utilize a depth-first traversal algorithm when they distribute RNNs to multiple FPGAs [7]. Consequently, matrix/vector operations are likely to be partitioned into two dimensions: *input-wise* and *gate-wise*. Fig. 5a and 5b illustrate the operation-level partition results of an LSTM. The input-wise partitioning distributes operations based on input sources (i.e., x, h). The gate-wise partitioning is based on gate types and allows each gate to run on discrete FPGAs.

However, the conventional method has two major limitations for multiple FPGAs. First, assigning each input source or gate to discrete FPGAs increases the load imbalance of vector operations severely. Second, inter-FPGA communications for

vector reduction cause stall cycles due to dependencies between consecutive vector operations. Fig. 5c shows the input- and gate-wise partitioning fail to obtain the scalable performance as the overhead of load imbalance and vector reduction increases with the number of FPGAs. GRUs suffer from more serious performance degradation than LSTMs because they have fewer matrix/vector operations, but more dependencies between gate operations.

C. *Underestimated Inter-FPGA Communication Cost*

The existing schemes lead to sub-optimal partitioning decisions because they do not consider the performance loss of inter-FPGA collective communications. When partitioned RNNs *synchronize* output vectors across FPGAs, the operation-level and matrix-level partitioning require *broadcast* and *all-gather*, respectively (see ③ in Fig. 6), which becomes major performance bottlenecks for many FPGAs. However, previous studies demonstrated only point-to-point communication link latency and overlooked the performance loss of mandatory collective communications.

D. *Unoptimized Instruction Scheduling*

Previous studies have also overlooked the performance impact of instruction scheduling in the context of multi-FPGA acceleration. As LSTM and GRU models consist of many consecutive operations (see Fig. 2b), RNN execution has to wait for preceding time steps to complete, which often leads to sub-optimal distribution decisions.

V. Our Partitioning Strategy

To address the scalability challenges, we introduce three parallelism levels of modern RNNs and propose systematic strategies to obtain optimal parallelism degrees at each level. Fig. 6 summarizes our four key ideas to find an optimal partitioning result for multiple FPGAs by exploiting the full parallelism of large-scale RNNs. First, we leverage *two-dimensional RNN analysis* to find dependencies between input sequences and layers. By doing so, we determine the optimal number of FPGAs dedicated to each layer. Second, we perform *single-layer partitioning analysis* to explore both operation-level (i.e., input/gate-wise) and *matrix-level row-wise partitioning*, which cannot be explored by conventional approaches at the same time. This partitioning method provides optimal distributions that significantly reduce the load imbalance and vector reduction overheads. Third, we analyze the required *collective communications* and take into account their performance impact when we explore different partition cases. Fourth, we apply *software pipelining* to increase the utilization of FPGAs.

A. *Two-dimensional RNN Analysis*

Our two-dimensional RNN graph analysis builds an RNN model graph in the *input sequence* and *layer* dimensions. After that, it traverses through the graph and finds the exact number of parallelism degrees (PDs). For instance, with the three-layer encoder model, every layer is independent with others

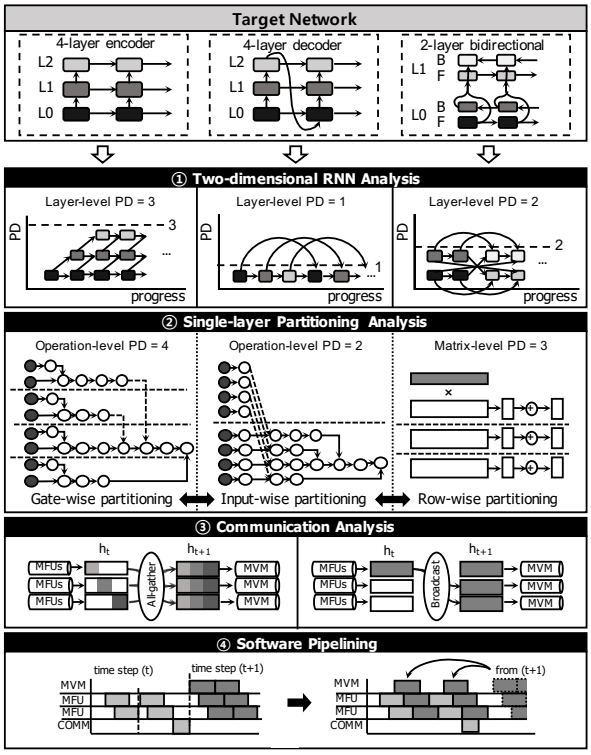

Fig. 6. Our multi-FPGA partitioning strategy.

for different input sequences (i.e., PD=3). On the other hand, with the three-layer decoder model, the layers are dependent on others and they have no concurrency in the input dimension (i.e., PD=1). A bidirectional layer has two parallelism degrees due to its forward and backward directions (i.e., PD=2).

B. *Single-layer Partitioning Analysis*

When we explore partitioning strategies of a single-layer RNN, we take account of matrix-level row-wise partitioning as well as the conventional operation-level partitioning (i.e., input/gate-wise). The matrix-level row-wise partitioning divides every weight matrix into multiple row blocks and then assigns them to separate FPGAs for concurrent execution. Also, the vector operations are split into multiple sub-blocks and executed independently (see Eq. 1). As a result, the row-wise partitioning does not require any vector reduction during gate operations. Also, as the vector operations get uniformly distributed, it does not cause any load imbalance problem.

C. *Cost Analysis of Collective Communication*

To derive more detailed communication cost of RNN partitioning, we incorporate collective communication overhead by considering communications required to execute the partitioned RNNs. Layer- and input-wise partitioning do not require communications for vector-synchronization, but gate- and row-wise partitioning demand collective communications across multiple FPGAs.

1) Row-wise Partitioning: Row-wise partitioning requires all-gather collective communications and gathers divided output vector elements ($h_{t-1} \rightarrow h_t$ in Eq. (1)). The *ring-based*

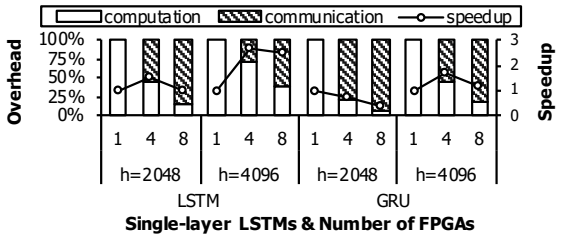

Fig. 7. Increasing communication overhead with row-wise partitioning (h: hidden dimension).

TABLE I
FPGA IMPLEMENTATION RESULTS ON STRATIX 10

Hardware Configuration	ALMs	M20Ks	DSPs	Freq. (MHz)
INT8, 4T-120D-40L	567,982 (61%)	9,018 (77%)	4,880 (85%)	275

TABLE II
SIMULATION PARAMETERS FOR MULTI-FPGA RNN EXECUTION

	Parameters
Workload	$Type = LSTM \vert GRU$ $h = 2048\ -\ 4096$, $L = 1, 8$, $ts = 32$
Architecture	$N_{tile} = 4$, $N_{dpe} = 120$, $N_{lane} = 40$
Communication	$\alpha = 1\mu s$, $\beta = 0.17ns/byte$

all-gather algorithm takes $(N-1)$ steps, where N is the number of FPGAs, and h is the hidden dimension of an RNN model. Fig. 7 shows marginal performance benefits due to all-gather communications. Although the vector operations of each FPGA are perfectly balanced, multiple devices fail to achieve the scalable performance due to the increased communication overhead. Eq. (2) summarizes the all-gather collective communication model with latency (α) and bandwidth (β).

$$Latency_{all-gather,ring} = (N-1)(\alpha + \frac{h}{N}\beta) \qquad (2)$$

2) Gate-wise Partitioning: Gate-wise partitioning requires collective broadcast communications to share output vectors with others. Similar to the all-gather communication, the *ring-based broadcast* algorithm takes $(N-1)$ steps, but it has different data traffic as they transfer whole vectors. Eq. (3) summarizes broadcast collective communication models.

$$Latency_{broadcast,ring} = (N-1)(\alpha + h\beta) \qquad (3)$$

D. Software Pipelining

We use software pipelining by unrolling iterations on input sequences. Since the input vectors of the following iterations are available in advance, the next iteration's matrix-vector multiplication can start before the current iteration is finished. We re-order matrix-vector multiplications when we find idle MVMs due to back-to-back dependencies or collective communications.

VI. EVALUATION

A. Methodology

We evaluate our proposed partitioning strategy using our FPGA prototype and cycle-accurate multi-FPGA simulation tools. We validate our simulation tool by comparing it against cycle-precise RTL simulation from the prior work [5]. Table I shows our accelerator implementation results using Intel programmable acceleration cards with Stratix 10 [8]. Our accelerator architecture supports an 8-bit integer data type, four tiles, 120 DPEs, and 40 lanes (i.e., 4T-120D-40L). Our design space exploration tools and FPGA implementation show 1%-2% marginal cycle differences. The confidence of our experimental results is high because the RNN dataflow and accelerator architectures are deterministic. We utilize a ring-based communication link model to simulate multi-FPGA processing. Table II shows our simulation and workload parameters based on recent RNN studies [3]. We report simulation results for RNN models larger than FPGA's on-chip memory to fully comprehend the trends of an increasing number of FPGAs.

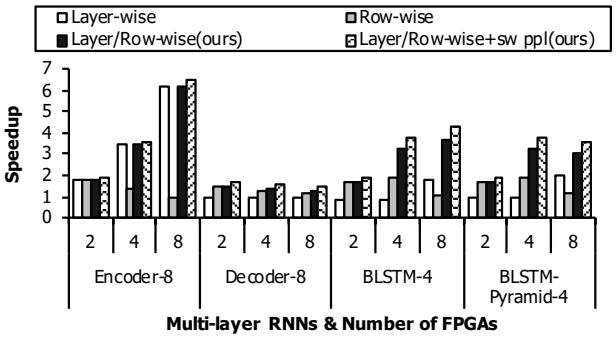

Fig. 8. Speedup of multi-layer LSTM benchmarks with different partitioning strategies (h = 2048, ts = 32). The speedup values are normalized to single-FPGA inference time.

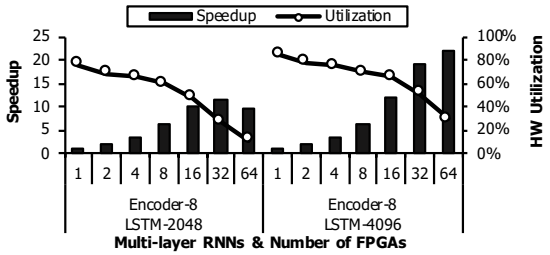

Fig. 9. Scalability results of layer- and row-wise partitioning with the increasing number of FPGAs.

B. Multi-layer LSTM Benchmarks

We first evaluate the performance of multi-layer RNNs having various data dependencies. Across all multi-layer workloads, our strategy shows the best distribution results. Especially, with BLSTM configurations, our scheme has huge benefits by combining two parallelism levels. Fig. 8 shows that combining layer- and row-wise partitioning methods derived from our strategy achieves the more scalable performance for BLSTM and pyramid models. For example, 2 (layer-wise) $\times$ 4 (row-wise) configuration shows the best performance for BLSTM-4 models. Moreover, our software pipelining scheme enables higher performance for all the multi-layer RNNs.

The layer-only partitioning has the highest load balance and no vector reduction during gate operations, so its performance nicely scales with the number of FPGAs with the 8-layer encoder. However, the networks having dependencies between input sequences (e.g., decoder and pyramid) cannot exploit the

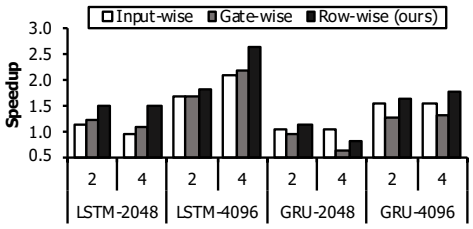

Fig. 10. Speedup of single-layer LSTM benchmarks with different partitioning strategies (ts = 32). The speedup values are normalized to single-FPGA inference time.

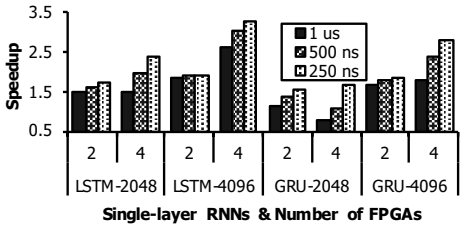

Fig. 11. Sensitivity test of row-wise partitioning with different link latency. The speedup values are normalized to single-FPGA inference time with row-wise partitioning.

advantages of the layer-only partitioning. On the other hand, row-only partitioning shows sub-optimal performance due to its heavy collective communications.

Our layer- and row-wise partitioning can further improve the performance beyond eight FPGAs by aggressively exploiting both parallelism degrees. Fig. 9 demonstrates the performance gain and hardware utilization with up to 64 FPGAs. Our technique has more benefits for larger LSTM models. However, its performance saturates with too many FPGAs because each FPGA gets too small workload.

C. Single-layer LSTM/GRU Benchmarks

To find the best single-layer RNN partition results, we also explore all the possible partitioning choices. Fig. 10 shows the speedup of different strategies over a single-FPGA acceleration. The matrix-level row-wise partitioning provides the more scalable performance for single-layer LSTMs and GRUs. However, its performance does not scale for small-size RNNs since the all-gather communication overhead becomes the performance bottleneck. GRU's performance saturates quickly because it demands more vector-reduction communications and fewer gate operations than LSTMs. On the other hand, conventional approaches suffer from load imbalance and vector reduction with many FPGAs. Fig. 11 shows that low-latency links allow the more scalable performance for multi-FPGA acceleration while reducing the row-wise partitioning's communication overhead.

D. Performance Impact of Software Pipelining

Fig. 12 illustrates the speedup of single-layer RNN with software pipelining. It allows higher performance gain depending on the size of the hidden dimension and the number of FPGAs. The performance gain is maximized when the execution latency of re-ordered matrix-vector multiplication is comparable to pipeline bubbles caused by back-to-back dependencies

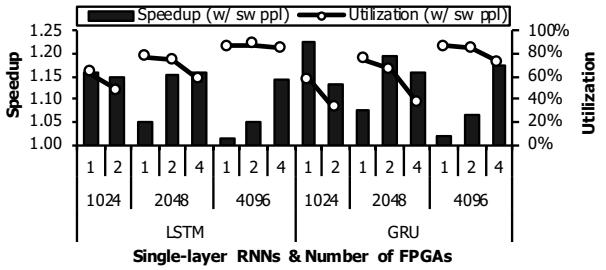

Fig. 12. Performance impact of software pipelining in single-layer RNNs. The speedup values are normalized to single-FPGA inference time.

as well as vector synchronization. As the communication latency increases with more FPGAs, the gap between time steps gets bigger. This leads to the highest performance gain as long as the increasing communication time is comparable to the re-ordered matrix-vector multiplication. However, the continuous growth of communication latency eventually reduces the gain when the model is distributed to many FPGAs.

VII. Conclusion

In this paper, we explored multi-FPGA acceleration strategies for large-scale RNNs. We utilized full parallelism levels of modern RNN workloads and examined the performance impact of collective communications and software pipelining. Our partitioning method allows up to 2.4x faster inference for large RNN workloads than conventional methods.

Acknowledgment

This work was supported by Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT1901-12. The EDA tool was supported by the IC Design Education Center (IDEC), Korea.

References

[1] D. Amodei, S. Ananthanarayanan, R. Anubhai, J. Bai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, Q. Cheng, G. Chen *et al.*, "Deep speech 2: End-to-end speech recognition in english and mandarin," in *International conference on machine learning*, 2016, pp. 173–182.

[2] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in neural information processing systems*, 2014, pp. 3104–3112.

[3] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," *arXiv preprint arXiv:1609.08144*, 2016.

[4] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi *et al.*, "A configurable cloud-scale dnn processor for real-time ai," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*. IEEE Press, 2018, pp. 1–14.

[5] E. Nurvitadhi, D. Kwon, A. Jafari, A. Boutros, J. Sim, P. Tomson, H. Sumbul, G. Chen, P. Knag, R. Kumar *et al.*, "Why compete when you can work together: Fpga-asic integration for persistent rnns," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2019, pp. 199–207.

[6] G. Diamos, S. Sengupta, B. Catanzaro, M. Chrzanowski, A. Coates, E. Elsen, J. Engel, A. Hannun, and S. Satheesh, "Persistent rnns: Stashing recurrent weights on-chip," in *International Conference on Machine Learning*, 2016, pp. 2024–2033.

[7] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman *et al.*, "Serving dnns in real time at datacenter scale with project brainwave," *IEEE Micro*, vol. 38, no. 2, pp. 8–20, 2018.

[8] "Intel® stratix® 10 fpgas overview - high performance stratix® fpga."