Accelerating Recurrent Neural Networks in Analytics Servers: Comparison of FPGA, CPU, GPU, and ASIC

Eriko Nurvitadhi, Jaewoong Sim, David Sheffield, Asit Mishra, Srivatsan Krishnan, Debbie Marr
Intel Corporation
Contact: eriko.nurvitadhi@intel.com

Recurrent neural networks (RNNs) provide state-of-the-art accuracy for performing analytics on datasets with sequence (e.g., language model). This paper studied a state-of-the-art RNN variant, Gated Recurrent Unit (GRU). We first proposed memoization optimization to avoid 3 out of the 6 dense matrix vector multiplications (SGEMVs) that are the majority of the computation in GRU. Then, we study the opportunities to accelerate the remaining SGEMVs using FPGAs, in comparison to 14-nm ASIC, GPU, and multi-core CPU. Results show that FPGA provides superior performance/Watt over CPU and GPU because FPGA's on-chip BRAMs, hard DSPs, and reconfigurable fabric allow for efficiently extracting fine-grained parallelisms from small/medium size matrices used by GRU. Moreover, newer FPGAs with more DSPs, on-chip BRAMs, and higher frequency have the potential to narrow the FPGA-ASIC efficiency gap.

I. Introduction

Machine learning (ML) algorithms are at the heart of many analytics workloads, which extract knowledge from the plethora of digital data available today. Recurrent Neural Network (RNN) is an ML algorithm that can provide state-of-the-art results for analytics of datasets with sequences. It is widely adopted in various use cases. For example, in language modeling [1], RNN is used to analyze sequences of words and sentences for applications such as sentence completion, speech recognition, machine translation, etc.

Analytics workloads, including RNN-based ones, are typically deployed on large-scale servers in data centers, which demand extreme energy efficiency in addition to high performance. To this end, recent server systems integrate hardware accelerators alongside general purpose CPUs to deliver significant execution efficiency for hotspots offloaded to these accelerators, while maintaining generality to execute the rest of the workloads on CPUs.

FPGAs, GPUs, and ASICs are the well-known options for accelerators available in the market today. For FPGAs, recently there have been major efforts from technology leaders to better integrate FPGA accelerators within data center servers (e.g., Microsoft Catapult, IBM CAPI, Intel Xeon+FPGA). There is also a growing number of GPU and ASIC accelerator solutions offered commercially, such as NVIDIA GPU and IBM PowerEN processor with edge network accelerators.

In this paper, we investigate the opportunities for optimizing and accelerating a state-of-the-art RNN algorithm variant, Gated Recurrent Network (GRU) [2], for language model classification tasks on analytics servers. First, we propose a memoization optimization to remove 50% of the SGEMV operations, which is key bottleneck in GRU algorithm, at the cost of more memory space usage. Our optimization does not functionally alter the GRU algorithm. Next, we developed FPGA accelerators for the SGEMV hotspot, targeting multiple generations of Altera FPGAs (Stratix V and Arria 10). Finally, we compare our FPGA accelerator against state-of-the-art 14nm ASIC implementation, and optimized SGEMV software on multi-core CPU and GPU.

The rest of the paper is organized as follows. The next section gives background on ML for analytics using recurrent neural networks. Section III presents the proposed memoization optimization. Section IV details our FPGA and ASIC accelerators. Section V reports our evaluation results. Section VI and VII offer related work and concluding remarks.

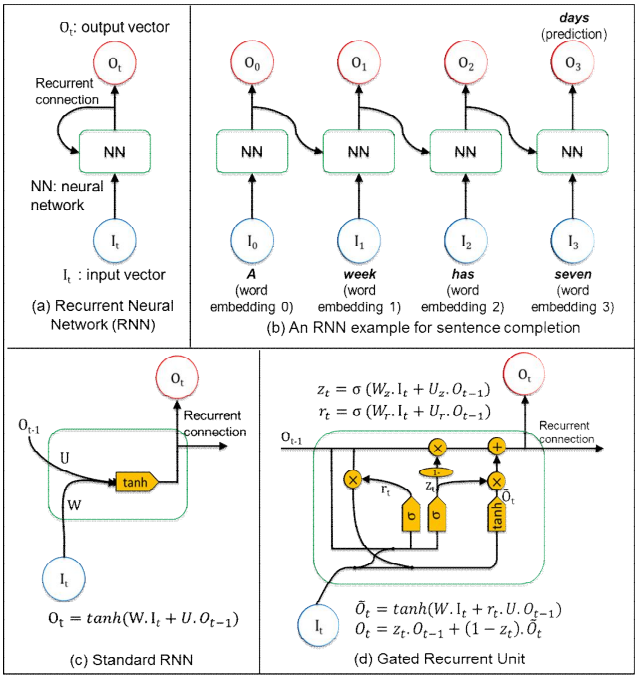


Fig. 1. Recurrent Neural Networks overview. (a) illustrates an RNN, which is a neural network that has recurrent connections. (b) shows an example of RNN for sentence completion. (c) shows the details of RNN. (d) shows the details of Gated Recurrent Network, a state-of-the-art variant of RNN.

II. Background

Many data analytics workloads rely on machine learning (ML) algorithms. A typical ML setup for data analytics consists of two phases. In *training phase*, a known set of data

samples is fed into an ML algorithm, which then *creates a model* with predictive power. Then, in the *classification phase*, this model is used by the ML algorithm to *make predictions* for any new given data. This paper focuses on RNN algorithms used in classification phase for language modeling.

Recurrent Neural Networks (RNNs). RNNs are neural networks that contain recurrent connections (i.e., loops) within the network. Figure 1(a) provides an illustration. Due to such recurrent connections, RNNs are especially useful in analyzing sequences. While a typical feedforward (non-recurrent) neural network produces its output solely based on its current input, an RNN produces its output by considering not only its current input, but also based on the history of its previous inputs.

RNNs provide state-of-the-art results in language modeling [1]. Figure 1(b) shows an example that predicts the next word for a given sequence of words in a sentence. For a 4-word input sequence "A week has seven", the RNN calculate probabilities of the next word and predicts that the next word is "days".

The details of a standard RNN is provided in Figure 1(c). It consists of a fully connected tanh() layer with recurrent connections. It accepts as inputs: (1) data at step t, I_t, and (2) the output of previous step O_{t-1}. W and U are dense matrices containing the network weights. I_t and O_{t-1} are represented as dense vectors. I_t can be represented in a simple one-hot encoding format, or in a format produced using advanced word embedding methods (e.g., [4]). This study used the later since it is more efficient, and is the existing state-of-the-art.

One known weakness of the standard RNN is its poor ability in *learning long-term dependencies*. Consider the sentence completion task for the following example input sentences, "I grew up in France… I speak fluent *French*". In the last sentence, recent information (i.e., "I speak fluent") suggests that the next word would be a name of a language. But, to predict that the language is "*French*", the network has to consider information from much earlier part of the input sentences ("I grew up in France").

Gated Recurrent Unit (GRU). A standard RNN provides fixed weights between the current input (I_t) and the previous history (O_{t-1}) in producing the current output (O_t), which tends to forget information from much earlier part of the input sequence. This makes standard RNN ineffective in learning long-term dependencies. GRU [5] is an RNN variant that addresses this issue. Unlike a standard RNN, a GRU has the ability to dynamically adjust the weights on current input and history to determine how much long-term history to keep and new information to carry forward.

Figure 1(d) depicts GRU details. z_t is an "update gate", which takes the current input I_t and the output of the previous step O_{t-1} and determines how much old information to carry through to the output (O_t). r_t is a "reset gate" which determines how much old information should be ignored in coming up with a candidate output ($\tilde{O}_t$). The lower r_t is, the more the old information is ignored. Lastly, the final output O_t is determined by taking into account the candidate output $\tilde{O}_t$ and old information scaled by the update gate z_t.

There are other RNN variants that have been studied in the literature. Among these variants, Long Short Term Memory (LSTM) and GRU provide the best accuracies [5]. We focus on GRU since it is more computationally efficient than LSTM.

III. GRU Algorithm Optimization

Algorithmically, most of the computation happens in the dense matrix vector multiplications (SGEMVs) of the weight matrices (i.e., W and U matrices in Figure 1(d)). For the W, Wr, and Wz matrices, the number of rows are based on the number of hidden units (hsz) used in the GRU and the number of columns are based on the width of the word encodings (esz). For the U, Ur, and Uz matrices, the number of rows and columns are based on the number of hidden units (hsz). Thus, in overall, the multiplication on these 6 matrices demand (3 x hsz x esz) + (3 x hsz x hsz) multiply-accumulate operations.

Since we focus on classification use case (instead of training), these weight matrices are already learned and do not change over the course of the computation. Therefore, to improve the efficiency of the GRU computation, we propose to memoize the results of the multiplications for the W, Wr, and Wz matrices. For all possible word encodings in the vocabulary (i.e., all I_t values in Figure 1(d)), we calculate $W.I_t$, $Wr.I_t$, and $Wz.I_t$ ahead of time and memoize the results in memory. While we use 3 times the memory space, we can now avoid 3 out of the 6 SGEMVs in GRU since we do not have to perform the SGEMVs on the W, Wr, and Wz. More precisely, we avoid computing 3 x hsz x esz multiply-accumulate operations.

We evaluate our approach using GRU software from Yandex [6] as reference, on Penn Treebank dataset containing ~79K test data words. We used suggested algorithm parameters [6] to provide good results on this dataset. (i.e., sigmoid hidden layer of size 256, hierarchical softmax, 10K words vocabulary). Since the hidden layer size dictates weight matrix size that affects computational demand, we further experimented with twice smaller (128 units) and larger (512 units) hidden layer sizes. Existing RNN/GRU/LSTM studies have suggested this range of hidden layer sizes (e.g., 30-500 units in [1], 36-400 units in [5], and 128 units in [9]). We ran our experiments on a 2.3 GHz Intel® Xeon E5-2699v3 server and collected runtime profile using gprof and Intel® Vtune.

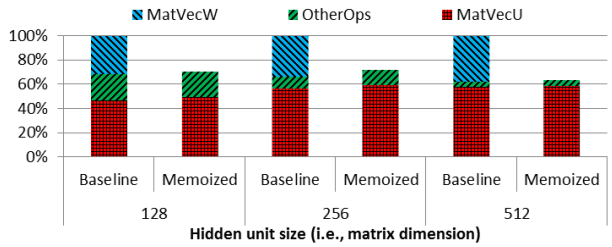


Fig. 2. Runtime breakdown of GRU with and without memoization. The y-axis shows runtime normalized to a baseline implementation. Memoization improves performance by 46% on average, by avoiding the multiplications on W, Wr, and Wz matrices (MatVecW). Even with memoization, most of the runtime is still spent on SGEMVs (i.e., in the MatVecU).

Figure 2 shows the evaluation results. The y axis shows the runtime normalized to the baseline GRU version. Each bar is

broken down into runtime spent on multiplications on W, Wr, and Wz matrices (MatVecW), multiplications U, Ur, and Uz matrices (MatVecU), and the rest of the GRU computations (OtherOps). The memoization optimization eliminates the MatVecW operations, and results in 46% improvements on average. In both baseline and memoized versions, SGEMV operation is the hotspot that dominates majority of runtime. As such, the rest of the paper focuses on improving the SGEMV execution efficiency using hardware accelerators.

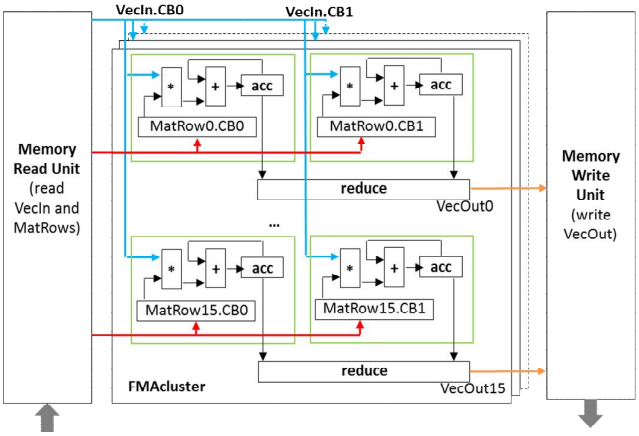


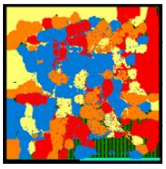
Fig. 3. Accelerator Design. The matrix is divided into column blocks (e.g., CB0, CB1). Each floating-point multiply-accumulate (FMA, in green box) processes a row in a column block, by multiplying input vector (VecIn) elements against matrix row elements (MatRow) kept in on-chip RAMs. Input/output vectors are read/written by memory read/write units.

Accelerator Designs	FPGA	ALMs	DSPs	M20Ks
8 clusters, 256 FMAs	Stratix V	296K	256	4MB
32 clusters, 1024 FMAs	Arria 10	224K	1024	4MB

(a) FPGA accelerators resource utilization

Frequency	1GHz
Process Technology	14nm
Number of FMA units	64
On-chip RAMs	4 MB, 64 banks
Peak performance	128 GFLOP/s

(b) ASIC accelerator specification



(c) ASIC design.

Fig. 4. FPGA and ASIC accelerators under study. (c) shows place-and-routed ASIC design, where each color higlights a cluster with 16 FMA units.

IV. Custom Hardware Accelerator

We designed a custom hardware (in Figure 3) to accelerate the SGEMV hotspots in GRU. The design consists of a memory read unit, a memory write unit, and clusters of multiple floating-point multiply-accumulate units (FMAs).

In this design, the matrix is divided into column blocks (e.g., CB0, CB1). Each FMA unit (green box in Figure 3) is responsible for processing a row in a column block by multiplying elements of the input vector (VecIn) against row matrix elements kept in the on-chip RAMs (MatRow). By having an FMA responsible for a row, the accumulate operation can be done in place in an FMA unit's accumulator register (acc).When the FMA produces the final accumulated value for a row in a column block, it is provided to the reduction unit (reduce). This unit adds the two FMA results for both column blocks to produce the final output vector element value (VecOutN). Multiple FMA units are grouped together as an FMAcluster. Numbers of FMAs and/or FMAclusters can be adjusted to scale the design. Arbitrarily large matrix is divided into blocks, which are processed by the accelerator one at a time. The accelerator computes rows and columns in a block in parallel. Blocks are determined based on available accelerator resources (FMAs, RAMs) and the matrix size.

FPGA Implementations. For our experiments, we target Altera Stratix V and Arria 10 FPGAs. Both FPGAs contain ~6MBs of on-chip RAMs (i.e., M20s, MLABs). Stratix V has 352 DSPs, while Arria 10 has 1518 DSPs. We implemented our accelerator in Verilog RTL and used Quartus for synthesis. FMAs and RAMs in the design are mapped to DSPs and M20Ks. We obtained FPGA power estimate using Altera's PowerPlay Early Power Estimator tool and memory power estimate using DRAMsim. Figure 4 details the resource utilizations for the designs we studied.

ASIC implementation. For the ASIC implementation, we set a comparable peak performance target as the Stratix V FPGA implementation, which is 126.7 GFLOP/s. As such, we included 4 FMAclusters with no column blocking, with 16 FMAs/cluster. Hence, there are 64 FMA units in our ASIC implementation. At 1GHz, the 64 FMAs can provide 128 GFLOP/s peak performance. We synthesized and place-and-routed the design for Intel's 14nm process technology. This gave us ASIC area and power numbers. The ASIC design met our target 1 GHz frequency. Figure 4 provides the summary of the ASIC implementation. It also shows a place-and-routed design, where each of the four FMA clusters is highlighted with a different color. We model RAM structures using CACTI. Our design uses a 4MB RAMs with 64 banks.

V. Evaluation Results

The comparison of performance, performance utilization, and performance/watt among CPU, GPU, FPGA, and ASIC are shown in Figures 5-7. We provide results for NoBatch mode (1 input at a time) and Batch10 mode (group 10 inputs at a time, as used in [3]). The CPU and GPU results are from optimized software using MKL and cuBLAS libraries on Intel Xeon E5-2699v3 CPU and NVIDIA GTX Titan X GPU.

FPGA vs. CPU/GPU. Without batching, FPGA performs better than CPU and GPU, except for Stratix V for the larger 512x512 matrix (Figure 5). Accordingly FPGA is more efficient with better utilization of its peak performance (Figure 6), resulting in superior ~10x performance/Watt (Figure 7). CPU and GPU cannot extract enough fine-grained parallelism from small/medium size GRU matrices, while FPGA utilize custom hardware design to carefully place and distribute matrix data on the many on-chip RAMs (M20Ks), and to orchestrate data movements to utilize the available FMA units (DSPs).

Batching improves CPU/GPU utilization, which leads to better performance. Nevertheless, in overall CPU/GPU are still underutilized (Figure 6). Batching improves CPU more so than GPU. Interestingly, even without batch mode, Arria 10

provides competitive performance relative to batched CPU and GPU. This shows the potential for FPGA to offer the best of both worlds, achieving competitive performance and efficiency without the shortcomings of batch mode (e.g., increased processing latency, implementation complexity overheads).

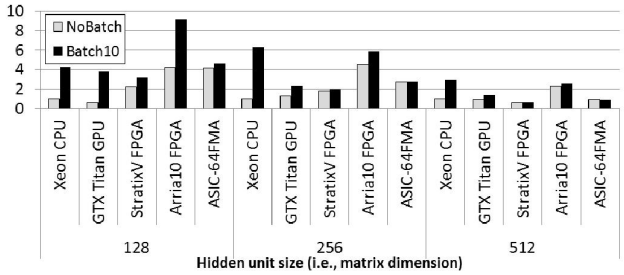


Fig. 5. Performance for all the accelerators under study, relative to CPU performance with no batching.

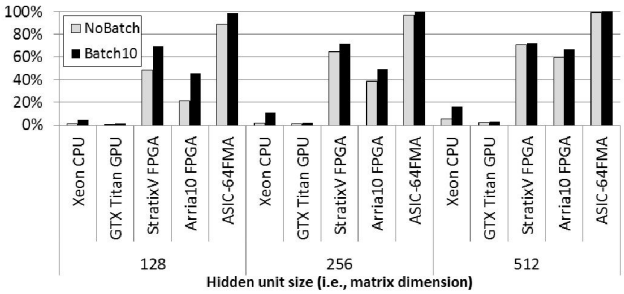


Fig. 6. Achieved performance relative to peak performance. E.g., 10% means the system is underutilized, where the achieved GFLOP/s is only at 10% of the available peak GFLOP/s. On the other hand, 100% means full utilization.

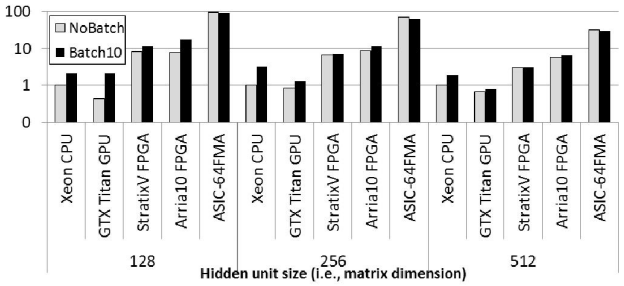


Fig. 7. Performance/Watt for all the accelerators under study, relative to the CPU performance with no batching. The y-axis is in log-scale.

FPGA vs. ASIC. The ASIC accelerator has better performance and utilization than Stratix V accelerator (Figures 5-6). Even though the ASIC accelerator was designed to be comparable in peak performance to Stratix V design, ASIC's truly custom implementation leads to better efficiency. E.g., our Stratix V design only used 256 DSPs out of 352 DSPs available on the FPGA due to routing constraints, while our ASIC contains just the necessary FMA hardware units.

Typically, one would expect ASIC to provide an order of magnitude efficiency improvements over FPGA, which is consistent with our results (Figure 7). However, note that the FPGAs are fabricated using an older technology than the ASIC. For example, the Stratix V FPGAs are fabricated using 28-nm TSMC technology, while our accelerator uses 14-nm Intel technology. If we were to normalize to consider the process technology difference, we can expect conservatively that the FPGA is at most only ~7x less efficient than the ASIC. Next-generation Stratix 10 FPGAs will have significantly more DSPs and M20Ks, along with more advanced fabric to enable higher frequency. For accelerator designs such as one studied here, which relies heavily on FMA operations and on-chip RAMs, the FPGA-ASIC efficiency gap will narrow even more.

VI. Related Work

To the best of our knowledge, there are only two prior RNN FPGA accelerators [8][9]. The first [8] is for training a standard RNN. It uses 1-hot word encoding. The second [9] is an LSTM FPGA accelerator that also uses 1-hot word encoding. In contrast, we focus on GRU, an RNN variant that is as accurate as LSTM [5], but require less computation. Furthermore, we use word embedding approach [4] that is more efficient and scalable than 1-hot encoding. There are studies on RNN for GPUs (e.g., [7]), but they do not consider FPGAs and ASIC. Finally, there are many other accelerator proposals for ML (e.g., [10][11]), but they are not for RNN and its variants.

VII. Conclusion

This paper studied opportunities to optimize and accelerate Gated Recurrent Unit, an advanced variant of the Recurrent Neural Networks. We first proposed a memoization optimization to avoid 3 out of 6 SGEMV operations that are the key bottleneck in GRU. Then, we studied an accelerator design for SGEMV mapped to Stratix V and Arria 10 FPGAs as well as a 14-nm ASIC, comparing them against optimized GPU and GPU implementations. Results show that FPGAs offer significant efficiency improvements over CPU and GPU. Furthermore, while ASIC is still more efficient than FPGA, the efficiency gap may become closer with newer FPGAs with more hard DSPs, on-chip BRAMs, and higher frequency.

References

[1] T Mikolov, M Karafiát, L Burget, J Cernocký, S Khudanpur, "Recurrent Neural Network based Language Model," INTERSPEECH, 2010.

[2] K. Cho, B. Merrienboer, C. Gulcehre, et al., "Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation," Empirical Methods in Natural Language Processing, 2014.

[3] D. Amodei, et al., "Deep Speech 2: End-to-End Speech Recognition in English and Mandarin," arXiv:1512.02595 [cs.CL].

[4] T. Mikolov, I. Sutskever, K. Chen, et al., "Distributed Representations of Words and Phrases and their Compositionality," NIPS, 2013.

[5] J. Chung, C. Gulcehre, K. Cho, Y. Bengio, Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling," NIPS Deep Learning and Representation Learning Workshop, 2014.

[6] Yandex RNN software. https://github.com/yandex/faster-rnnlm

[7] B. Li, E. Zhou, B. Huang, J. Duan, Y. Wang, "Large Scale Recurrent Neural Network on GPU," IJCNN, pp. 4062-4G69, 2014.

[8] S. Li, C. Wu, H. Li, B. Li, Y. Wang, Q. Qiu, "FPGA Acceleration of Recurrent Neural Network Based Language Model," FCCM 2015.

[9] A. X. M Chang, B. Martini, E. Culurciello, "Recurrent Neural Networks Hardware Implementation on FPGA," arXiv:1511.05552 [cs.NE].

[10] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun. "Cnp: An fpga-based Processor for Convolutional Networks," FPL 2009.

[11] E. Nurvitadhi, A. K. Mishra, et al., "Hardware Accelerator for Analytics of Sparse Data," Design Automation and Test in Europe (DATE), 2016.