η-LSTM: Co-Designing Highly-Efficient Large LSTM Training via Exploiting Memory-Saving and Architectural Design Opportunities

Xingyao Zhang
Bespoke Silicon Group (BSG)
University of Washington
Seattle, USA
xingyaoz@cs.washington.edu

Haojun Xia
Future System Architecture Lab
University of Sydney
Sydney, Australia
xhjustc@gmail.com

Donglin Zhuang
Future System Architecture Lab
University of Sydney
Sydney, Australia
dzhu9887@sydney.edu.au

Hao Sun
Future System Architecture Lab
University of Sydney
Sydney, Australia
hsun2147@uni.sydney.edu.au

Xin Fu
ECE Department
University of Houston
Houston, USA
xfu8@central.uh.edu

Michael Taylor
Bespoke Silicon Group (BSG)
University of Washington
Seattle, USA
prof.taylor@gmail.com

Shuaiwen Leon Song
Future System Architecture Lab
University of Sydney
Sydney, Australia
leonangel991@gmail.com

Abstract*—Recently, the recurrent neural network, or its most popular type—the Long Short Term Memory (LSTM) network—has achieved great success in a broad spectrum of real-world application domains, such as autonomous driving, natural language processing, sentiment analysis and epidemiology. Due to the complex features of the real-world tasks, current LSTM models become increasingly bigger and more complicated for enhancing the learning ability and prediction accuracy. However, through our in-depth characterization on the state-of-the-art general-purpose deep-learning accelerators, we observe that the LSTM training execution grows inefficient in terms of storage, performance, and energy consumption, under an increasing model size. With further algorithmic and architectural analysis, we identify the root cause for large LSTM training inefficiency: massive intermediate variables. To enable a highly-efficient LSTM training solution for the ever-growing model size, we exploit some unique memory-saving and performance improvement opportunities from the LSTM training procedure, and leverage them to propose the first cross-stack training solution, *η-LSTM*, for large LSTM models. *η-LSTM* comprises both software-level and hardware-level innovations that effectively lower the memory footprint upper-bound and excessive data movements during large LSTM training, while also drastically improving training performance and energy efficiency. Experimental results on six real-world large LSTM training benchmarks demonstrate that *η-LSTM* reduces the required memory footprint by an average of 57.5% (up to 75.8%) and brings down the data movements for weight matrices, activation data and intermediate variables by 40.9%, 32.9% and 80.0%, respectively. Furthermore, it outperforms the state-of-the-art GPU implementation for LSTM training by an average of $3.99\times$ (up to $5.73\times$) on performance and 63.7% (up to 76.5%) on energy saving. We hope this work can shed some light on how to design high logic utilization for future NPUs.

***Index Terms*—Machine Learning, Recurrent Neural Network, Accelerator**

I. Introduction

In recent years, machine learning and its special set of algorithms—artificial neural networks—have been experiencing an unprecedented growth in terms of adaptation and social impact. The recurrent neural network, or one of its most popular types—the Long Short Term Memory (LSTM) network—has achieved great success in a broad range of application domains including autonomous driving [1], [2], natural language processing [3], [4], business process management [5], sentiment parsing [6] and even recent tasks addressing the COVID-19 pandemic [7], [8]. However, due to the complex features of the real-world tasks, the LSTM models become increasingly bigger for enhancing the learning ability and prediction accuracy [9], [10]. Previous studies [11]–[13] mainly focus on improving the execution efficiency of LSTM inference, but enabling highly-efficient training for large LSTM models has been an open problem.

In this work, we first provide a detailed characterization on state-of-the-art large LSTM training and analysis for its inefficiency and design challenges (Sec. III). We have observed that training larger LSTMs frequently results in lower hardware throughput and energy efficiency. To identify the root causes, we further profile the GPU unit utilization and locate the memory-subsystem related overheads. Based on these high-level clues from these general-purpose accelerators, we conduct further algorithmic and architectural analysis, and discover that the root cause of large LSTM training inefficiency is the massive intermediate variables.

This work was partially supported by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement number FA8650-18-2-7863, and by the DARPA/SRC JUMP ADA Center. This research was also partially supported by Facebook Faculty Award and University of Sydney faculty startup funding, Australia Research Council (ARC) Discovery Project DP210101984. This research was also partially supported by NSF grants CCF-1900904, CCF-1619243, and CCF-1537085 (CAREER).

These variables are generated by forward (FW) propagation, stored and then reused by backpropagation (BP) for the gradient calculations. Due to their long reuse distances, they are typically stored in the DRAM during FW to release on-chip resources for other live variables that have shorter reuse distances. After further analysis, we find that these intermediate variables produced by state-of-the-art LSTM training flows pose a negative impact on storage (large memory footprint), performance (training latency and throughput) and energy efficiency. There have been several recent studies focused on reducing the large memory footprint induced by large intermediate variables during forward- and backward-propagation types of execution [14]–[16], e.g., vDNN [14] and SuperNeurons [15] for CNNs, and Echo [16] for the Transformer. However, LSTM training exhibits a unique and complex computation pattern that precludes these previous approaches from being applied (Sec.III).

To enable a highly-efficiency LSTM training solution with ever-growing model sizes, we exploit several unique memory-saving and performance improvement opportunities from LSTM training, and leverage them to propose a cross-stack training solution, *η-LSTM* (pronounced /ˈā-tə/ LSTM), which comprises both software-level and hardware-level innovations.

At the software-level, *η-LSTM* primarily focuses on reducing the large intermediate variables generated during FW propagation via tackling two major factors that determine the size of the intermediate variables: LSTM cell-level variable reduction (Sec.IV-A) and BP layer length reduction (Sec.IV-B). The corresponding designs for them are based on our *two key observations* about LSTM training. First, the cell-level intermediate variables possess very limited opportunities for compression; but with our execution reordering design, both the memory footprint and training latency can be effectively reduced. Second, not all the BP cells produce significant gradients for weight updating; thus, we can skip the execution for these "insignificant" BP cells to further reduce memory footprint, data movement and training latency.

At the hardware-level, we propose new hardware designs and optimizations to effectively support our software-level innovations and provide further architecture-level enhancements on performance and energy efficiency. Based on design space exploration, we discover that our proposed memory-saving optimizations will cause low hardware utilization when performed on the state-of-the-art NN accelerators [11]–[13], [17], resulting in inefficient execution and increased training time (Sec. V-A). To address this design challenge, we propose a novel PE design named *Omni-PE* (Sec. V-B) that can perform all the operations in LSTM training and its runtime resource allocation scheduler (Sec. V-C) that dynamically distributes the computational resources according to the LSTM workload requests, resulting in high hardware utilization. Finally, we offer a detailed overall hardware design along with its optimizations for *η-LSTM* (Sec. V-D). This study makes the following contributions:

- We conduct a comprehensive characterization study for large LSTM training on modern GPUs and identify its root cause for training inefficiencies;
- We exploit several unique memory-saving and performance improvement opportunities in LSTM training and leverage them to propose *the first highly-efficient cross-stack training solution* for large LSTM models, which comprises both software-level and hardware-level innovations;
- At the software level, we introduce several key observations regarding the unique LSTM training data patterns, which are then leveraged to enable large reduction on memory footprint, data movements and training latency;
- At the hardware level, we propose novel architecture designs that enable high logic utilization on customized NPUs, while supporting our software-level optimizations and further enhancing LSTM training performance and energy efficiency;
- The experimental results show that for large LSTM training scenarios, our pure software-level memory-saving optimizations reduce the memory footprint by an average of 57.52% (up to 75.75%) and reduce data movement for weight matrices, activation data and intermediate variables by 40.85%, 32.89% and 80.04% respectively. In return, our memory-saving techniques improve the state-of-the-art GPU (Nvidia V100 32GB) implementation by $1.56\times$ (up to $1.79\times$) on performance and $1.54\times$ (up to $1.78\times$) on energy with no convergence speed issues and negligible accuracy impact. And our pure hardware architecture design can achieve an average of $1.67\times$ (up to $2.69\times$) better energy over the state-of-the-art GPU design. Combining them together, our overall *η*-LSTM design surpasses the state-of-the-art GPU implementation by an average of $3.99\times$ (up to $5.73\times$) on performance and $2.75\times$ (up to $4.25\times$) on energy.

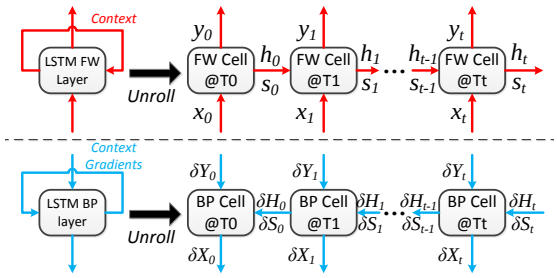


Fig. 1: The execution flow for one LSTM layer during the forward- (FW) and back-propagation (BP) phases. T represents timestamps.

II. Background: LSTM Training

The Long-Short Term Memory (LSTM) network is a popular type of recurrent neural network (RNN) which has been adopted in a wide range of real-world applications. Similar to other supervised learning networks such as convolution neural networks (CNNs), LSTM training comprises both forward- (FW) and backpropagation (BP) procedures. However, common LSTM networks exhibit a unique execution flow that is different than CNNs or CNN-like networks. For example, as

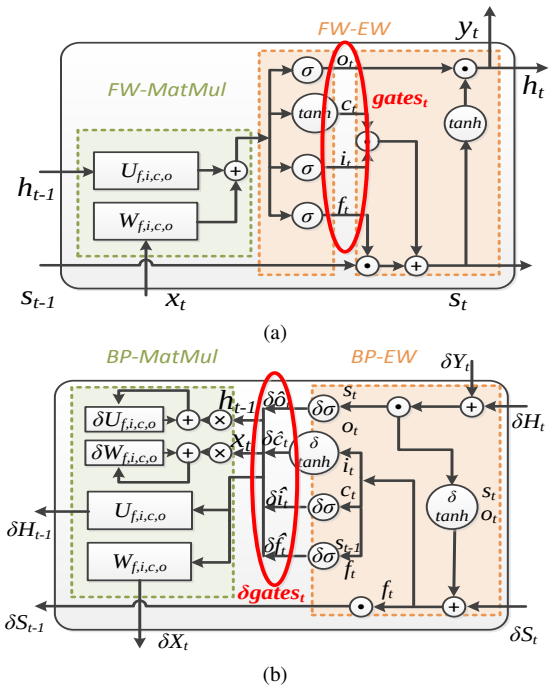


Fig. 2: The zoom-in view of the LSTM cell execution during training: (a) forward- (FW) and (b) back-propagation (BP).

illustrated on the left, "rolled", side of Fig. 1, within each layer, the operations of mapping the inputs to the outputs are integrated into one FW cell, while the operations of mapping the output gradients to the weight gradients and input gradients are integrated into one BP cell. Both cells are executed *recurrently* because they need to adopt the historic self-output information. This feature helps model the context dependency within the input activation for the modeling sequence tasks (e.g., language modeling, trajectory prediction, etc). It also forces the layer's input to be consumed step-by-step to form a *sequential* execution. Such sequential execution of one LSTM layer can be unrolled into a sequence of cells for both FW and BP to represent the cell states at different timestamps[1], as shown on the right, "unrolled", side of Fig. 1, where all the unrolled cells within the same layer share the same weight matrices. Since the adjacent cells in both FW and BP exhibit context dependency [18], they can only be processed sequentially.

Fig. 2 also shows the zoom-in view of the actual computation in both FW and BP LSTM for the cells located at the tth timestamp. In the FW cell (Fig. 2a), the matrix-multiplication (FW-MatMul) and the element-wise operations (FW-EW) leverage the layer input x_t and the context output of its previous cell h_{t-1} to produce the values for the multiple LSTM $gate_t$ in the cell (circled in red), including the forget gate f_t, the input gate i_t, the cell gate c_t and the output gate o_t. Their computation share a similar function:

$$gate_t = AF(W_{f,i,c,o}x_t + U_{f,i,c,o}h_{t-1} + b_{f,i,c,o}) \quad (1)$$

[1]In this paper, we focus on the LSTM unrolling analysis [18], in which a cell in a layer is an unrolled cell at a certain timestamp.

where AF represents the activation functions including sigmoid (σ) for f_t, i_t and o_t; and the hyperbolic tangent ($tanh$) for c_t. $W_{f,i,c,o}$ and $U_{f,i,c,o}$ are the weight matrices for x_t and h_{t-1}, respectively, which are different when computing towards different gate values. $b_{f,i,c,o}$ represents the offsets. By leveraging another element-wise operation, $gate_t$ generates the cell state s_t, the context link h_t for the next cell, and the output y_t for the cell in the next layer. Note that x_t, h_{t-1}, and s_t are saved into memory as intermediate parameters for future usage during FW.

On the other hand, in the BP cell shown in Fig. 2b, the gates gradients ($\delta gate_t$) and the cell state gradients for the previous cell (δS_{t-1}) are generated through performing element-wise operation (BP-EW) on the following parameters: the feedback of the output gradients (δY_t) from the next layer, the context gradients (δH_t) from the next timestamp, the cell state gradients (δS_t) and the gate values for this timestamp calculated previously during FW ($gate_t$). Then, these gradients will be applied to generate the gradients for the two cells through inner-product ($\cdot$) matrix-multiplication (BP-MatMul): the cell in the previous layer with the same timestamp (δX_t), and the cell at the current layer with the previous timestamp (δH_{t-1}).

$$(\delta X_t,\ \delta H_{t-1}) = (W^T_{f,i,c,o},\ U^T_{f,i,c,o}) \cdot \delta gates_t \quad (2)$$

And these gradients will also be used to produce the weights gradients (W, U) through the outer product ($\bigotimes$) matrix-multiplication (BP-MatMul):

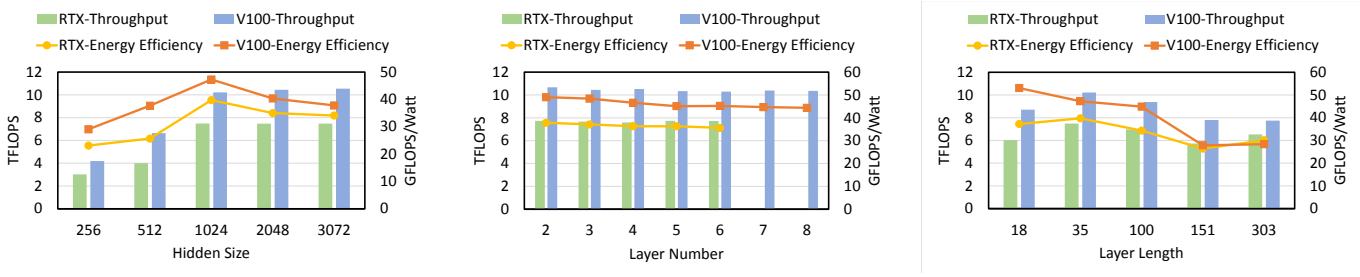
$$(\delta W_{f,i,c,o},\ \delta U_{f,i,c,o}) \mathrel{+}= \delta gates_t \bigotimes (x^T_t,\ h^T_{t-1}) \quad (3)$$

III. Challenges for Large LSTM Training

Due to the complex features of the real world tasks, neural network models have been growing exponentially larger, enhancing learning ability and prediction accuracy [9], [10]. Additionally, today's network compression and pruning techniques [19]–[21] require training of large NN models at the beginning for better non-linearity fitting, before pinpointing and reducing the model redundancy. Thus, it is inevitable that LSTM models are growing large, e.g., expanding the hidden size (i.e., determines the weight matrices size), layer length (i.e., represents the number of cells per layer) and layer number (i.e., indicates the model depth). Previous studies [11]–[13] mainly focus on improving the execution efficiency of LSTM inference, but techniques for enabling highly-efficient training for large LSTM models remain unexplored. Next, we provide a detailed characterization on the state-of-the-art large LSTM training and an analysis of its inefficiency and design challenges.

A. Initial Characterization: LSTM Training on General-purpose Accelerators

As GPUs become increasingly popular as general-purpose accelerators for performing neural network training, we conduct a comprehensive characterization on two state-of-the-art GPUs (i.e., 16GB Nvidia Quadro RTX 5000 with the



(a) (b) (c)

Fig. 3: The performance and energy efficiency comparison of training LSTM on the latest GPUs (Nvidia RTX5000 with Turing architecture and V100 with Volta architecture) when scaling up the LSTM model size by increasing (a) the hidden size, (b) the layer number, or (c) the layer length.

Turing architecture [22] and 32GB Nvidia Tesla V100 with the Volta architecture [23]) to evaluate the LSTM training efficiency with the ever-increasing model sizes. We first use PyTorch [24] to implement the LSTM training algorithms with various model configurations, and then apply Nvidia Profiler [25] to extract GPU runtime information, e.g., execution latency, power consumption and unit utilization. More detailed experimental setup can be found in Sec. VI-A.

There are three typical ways for increasing LSTM model size: increasing the hidden size, increasing the number of layers, and increasing the layer length. Fig. 3 illustrates the comparison in throughput and energy efficiency of the LSTM training when scaling up the LSTM model size by increasing the hidden size, the layer number, or the layer length. We have the following observations:

Impact of varying the hidden size. To study the impact of hidden size on LSTM training efficiency, we implement different LSTM models for the real-world task of the language modeling using the Penn TreeBank (PTB) dataset [26] with a fixed layer number of 3 and layer length of 35 (determined by the dataset). Fig. 3a shows that when the hidden size is increased, the GPU throughput will first increase and then plateau. Since matrix multiplications (MatMul) are the main computation tasks for LSTM training, this throughput increment directly originates from the MatMul optimizations on modern GPUs. Specifically, when model is small (e.g., the hidden size is < 1024 [27]), the increased model size demands higher thread number, which creates opportunities for better parallel execution and high ALU utilization. However, once the model reaches a certain size, the ALU units are saturated and the throughput is not further improved; however, the energy efficiency starts to decline when further increasing the model size beyond the throughput saturation point. This indicates an increasing energy consumption due to the memory activities.

Impact of varying the layer number. To demonstrate the impact of the varying number of layers on LSTM training efficiency, we also implement different LSTM models for the PTB language modeling with the fixed hidden size of 2048 and layer length of 35. Fig. 3b shows that with the hidden size set large, although the GPU throughput varies little when scaling the number of layers, the corresponding energy efficiency decreases. This suggests that larger number of layers may not significantly impact the overall throughput but it can cause extra energy overhead for training. Note that due to the native memory size limitation, the 7- and 8-layer LSTM models cannot be trained on the 16G RTX 5000 GPU.

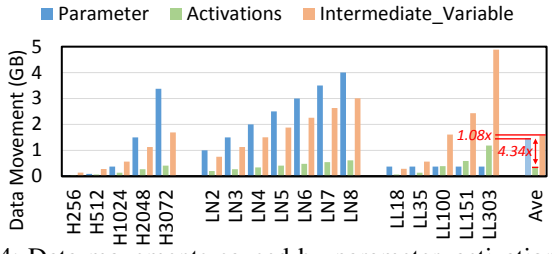


Fig. 4: Data movements caused by parameter, activation data, and intermediate variables.

Impact of varying the layer length. The LSTM layer length is associated with the dataset, which cannot be easily tuned like other configurations, e.g., the hidden size and layer number. To illustrate the impact of the increasing layer length (i.e., indicating the model depth) on LSTM training efficiency, we implement different LSTM models for different datasets shown in Table. I with the fixed hidden size of 1024 and layer number of 3. Fig. 3c demonstrates that when increasing the layer length, the overall throughput tends to decrease, resulting in energy efficiency drop. This suggests that longer layer lengths can cause negative impact on both throughput and energy efficiency.

In summary, we have observed that training larger LSTMs frequently results in lower hardware throughput and energy efficiency. To identify its root causes, we profile the GPU unit utilization with Nvidia Profiler and observe that the Load/Store (LDST) unit utilization significantly increases for large LSTM models, indicating the memory-subsystem related overheads. Next, based on these high-level hints, we further investigate the inefficient LSTM training and its root cause.

B. Root Cause for Large LSTM Training Inefficiency

During the forward computation phase, the FW-EW operations of each LSTM cell generates a group of intermediate variables, including i_t, f_t, c_t, o_t and s_t. When conducting LSTM inference, these intermediate variables are abandoned shortly after the current (i_t, f_t, c_t, o_t) or the next cell (s_t) computation. However, for the LSTM training processing,

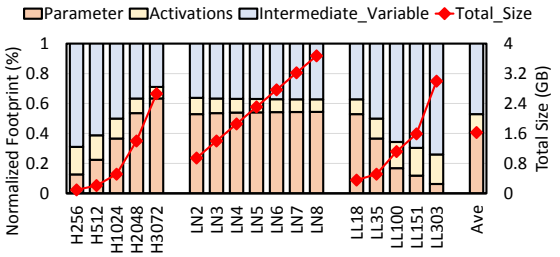


Fig. 5: The total size (line) and the breakdown (bar) of the GPU memory footprint.

due to the chain rule, these variables (i.e., i_t, f_t, c_t, o_t, s_t and s_{t-1}) are reused in the BP cell computation during backpropagation. Since the typical BP execution does not start until the completion of the FW computation, the reuse distances of these variables can be quite long. In the state-of-the-art LSTM implementations, they are commonly stored in the DRAM to release on-chip resources for other live variables that have short reuse distances. After further characterization and analysis, we discover that these intermediate variables produced by the state-of-the-art training flow pose several major negative impacts on large LSTM training, including storage, performance and energy inefficiencies.

Storage Inefficiency: Fig. 5 illustrates the breakdown of the memory footprint for the aforementioned LSTM training scenarios in Fig. 3, representing three major portions of runtime data: the weight matrices, the activation data, and the intermediate variables. Here, H256-H3072 correspond to the LSTM training models from Fig. 3a with different configurations of the hidden sizes; LN2-LN8 correspond to the LSTM training models from Fig. 3b with different configurations of layer numbers; and LL18-LL303 correspond to the LSTM training models from Fig. 3c with different configurations of layer lengths. We observe that on average 47.18% (up to 74.01%) of the memory footprint is contributed by the intermediate variables, which can easily surpass the GPU's on-chip memory capacity. Frequent accesses to them during the training may cause a large number of random memory accesses, resulting in large sub-memory system access overhead. Additionally, due to this storage overhead, large LSTM training could be easily bounded by the native hardware's storage limitation. For instance, as shown in Fig. 3b, the 7- and 8-layer LSTM models cannot even be trained on a 16G RTX 5000 GPU. This inefficiency substantially hinders real-world LSTM model design and implementation.

Performance Inefficiency: In Fig. 3c, we observe that GPU throughput declines for the LSTM training with a larger layer length. As shown in Fig. 5, we find that the memory footprints of the intermediate variables can grow significantly for these LSTM models with longer layer lengths. Since the memory accesses to these variables are usually on the critical path of training, they cause not only the end-to-end training performance degradation but also the decreased hardware throughput due to the significant memory access overhead. Note that the layer length is associated with the specific training dataset. As the real-world tasks become increasingly complex, LSTM's layer length is somewhat inevitable to be expanded to better capture long-term context between inputs.

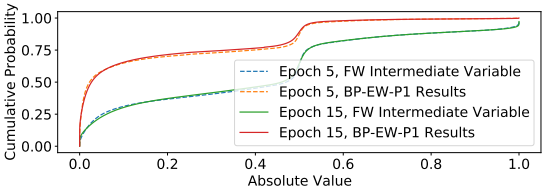


Fig. 6: The cumulative absolute value distribution for the FW intermediate variables and the BP-EW-P1 results at different training epochs.

Energy Inefficiency: We also find that that the large data movements between the on-chip and off-chip memories caused by accessing the intermediate variables during training directly contribute to the aforementioned energy efficiency declines shown in Fig. 3. For instance, Fig. 4 quantifies the amount of GPU-DRAM data movement from the activation data and the intermediate variables, respectively. Their DRAM accesses are much more difficult to reduce than the weight matrices because they are produced at runtime and often do not exhibit a certain data pattern which will be benefited from compression. We observe an average of $4.34\times$ additional data movements (up to $4.81\times$) caused by the intermediate variables over that from the activation data. Moreover, the size of the intermediate variables also grows faster than activation data.

C. Ineffectiveness of the State-of-the-art Techniques

There have been some studies focusing on reducing the large memory footprint caused by the intermediate variables during forward- and backward-propagation types of execution, e.g., vDNN [14] for the CNNs, SuperNeurons [15] for the CNNs, and Echo [16] for the Transformer. The core concept of these works is trading-off memory footprint with re-computation to carefully balance between the memory consumption upper-bound and the training performance. For instance, to reduce memory footprint, Echo [16] makes a key observation that in the attention layers of Transformers some partial intermediate variables during forward propagation can be used to derive the other key intermediate variables when needed in the back-propagation. So Echo only needs to store a small amount of these partial variables for the backpropagation re-computation to calculate the gradients. However, as discussed in Section II, the LSTM training exhibits a unique computation pattern that the stored intermediate variables (e.g., i_t, f_t, c_t, o_t and s_t) are independent of each other and cannot be derived from each other. Without the essential memory-saving opportunities, Echo becomes ineffective for LSTM training. On the other extreme, to minimize the memory footprint without storing any intermediate variables, the entire FW cell needs to be recomputed from scratch during the BP cell processing. This is infeasible in practice since it will substantially extend the back-propagation latency and cause serious performance overhead for LSTM training.

Additionally, the state-of-the-art LSTM acceleration approaches mainly focus on weight compression (e.g., pruning) for LSTM inference. For example, S.Han et al. [11] and

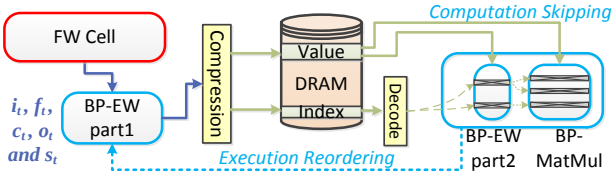


Fig. 7: The diagram of cell-level variables reduction with execution reordering.

S.Wang et al. [17] reduce the number of weight columns in MatMul computation to cut down the LSTM inference workload; and S.Wang et al. [13] propose to transform the weight matrices into a smaller block-circulate format for weight compression. However, these approaches cannot reduce the size of the intermediate variables in large LSTM training and address the aforementioned inefficiencies caused by them.

IV. Unique LSTM Memory-Saving Opportunities and Optimizations

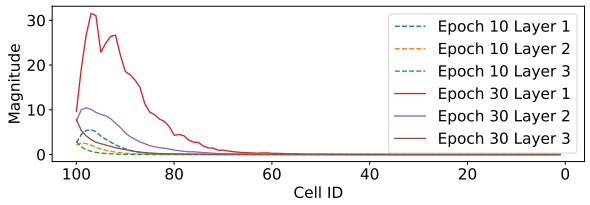
To enable a highly-efficiency LSTM training solution with ever-growing model sizes, we exploit some unique memory-saving and performance improvement opportunities in the LSTM training procedure and leverage them to propose a cross-stack training solution, η-*LSTM*, which comprises both software-level (Sec. IV) and hardware-level (Sec. V) innovations. In this section, we introduce key observations and software-layer designs in η-LSTM which effectively reduce the large intermediate variable size and the massive data movements between FW and BP cells during LSTM training. Specifically, we focus on optimizing two major factors that determine the intermediate variables' size: the number of variables in each cell (Sec. IV-A) and the number of cells within one layer (i.e., layer length; Sec. IV-B).

A. Cell-level Reduction for Intermediate Variables

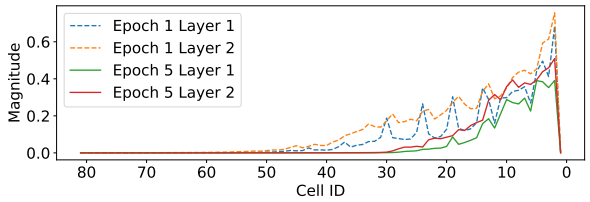
Basic idea. First, we attempt to reduce the size of cell-level intermediate variables. Specifically, we focus on compressing the intermediate variables (e.g., f_t, i_t, c_t,o_t, and s_t) generated by the FW cells since these variables typically cannot be immediately consumed and have to be stored for BP to accelerate the calculation for the gradients in the state-of-the-art implementations. They comprise the memory footprint upper-bound of the entire LSTM training. Through further investigation on the computation flow, we make a *key observation* that the cell-level intermediate variables generated by FW possess very limited opportunities for effective compression (discussed next). One intuitive solution for this is to quickly consume these variables during FW for certain important computation outcomes, instead of fully storing them during the training process to cause different efficiency issues (Sec.III). Guided by this intuition, we find that it is possible to reorder the LSTM training execution flow to quickly and effectively consume these FW intermediate variables while maintaining computation correctness. More surprisingly, we discover that this reordering creates a new set of variables that exhibit much higher data compression opportunities and can be effectively compressed to travel between FW and BP cells, resulting in significant memory footprint and latency reduction for LSTM training. Next, we provide more detailed discussion on the cell-level variable reduction.

Enhancing Data Compression Opportunities via Execution Reordering. To explore the opportunities of cell-level variable reduction, we collect the FW's intermediate variables from six large LSTM training benchmarks evaluated in this paper, shown in Table. I and investigate their value distributions. Note that the BP-EW computation can be divided into two computation stages: BP-EW-P1, which performs computation relying on only FW intermediate variables, and BP-EW-P2, which takes the outputs from BP-EW-P1 to calculate gradients during backpropagation. For example, the formula to calculate the input gate gradient is: $\delta\hat{i_t} = \delta S_t \bigodot c_t \bigodot i_t \bigodot (1 - i_t)$. Since c_t and i_t are generated by the FW procedure, $c_t \bigodot i_t \bigodot (1-i_t)$ belongs to BP-EW-P1, while the computation involving gradients belongs to BP-EW-P2. Fig. 6 shows the cumulative absolute value distribution for the FW generated intermediate variables and the calculated results from BP-EW-P1 phase, at different training epochs. The x-axis represents the absolute value range of the FW intermediate variables and the results from BP-EW-P1 phase, which is [0,1]. Note that both FW intermediate variables and BP-EW-P1 results are within this range based on formulation (Fig. 2 in Sec.II), e.g., FW intermediate variables are produced by the activation functions.

From Fig. 6, we can observe that only around 25% of the FW intermediate variables have values smaller than 0.1. On the other hand, BP-EW-P1 phase (which only depends on the FW intermediate variables for computation) generates approximately 65% of the data outputs with values smaller than 0.1, representing a very different data value pattern than FW intermediate variables. Also, this interesting data pattern is not impacted by different training epochs either. Since BP-EW-P1 computation only relies on FW intermediate variables, if hardware resources are available, it is possible to reorder the LSTM training by bringing the BP-EW-P1 phase forward into the FW phase to execute concurrently, immediately consuming all the FW intermediate variables. There are two direct benefits associated with this design. First, based on Fig. 6, BP-EW-P1 results' data value pattern can significantly enhance the data compression opportunities for reducing LSTM's training memory footprint upper-bound bounded by FW intermediate variables, e.g., experiments show that applying a *near-zero pruning* at the value threshold around 0.1 provides both large memory savings and little training accuracy loss (See Sec. VI-B4 for more discussion.). Second, after the reordering, BP-EW-P1 will generate much smaller compressed results which replace the FW intermediate variables to be requested by the BP processing. In other words, these large cell-level FW generated intermediate variables are no longer needed to be stored for BP processing, resulting in large memory footprint reduction. Additionally, the rest of the BP process now takes the decoded format of the outputs from BP-EW-P1 in FW, which also helps skip some unnecessary computation in both



(a)



(b)

Fig. 8: The weight gradients magnitude for different timestamp BP cells when implementing (a) single loss LSTM - IMDB [28] and (b) per-timestamp loss LSTM - WMT(MLPerf) [29].

BP-EW-P2 and BP-MatMul (e.g., near-zero operands). The overall execution reordering strategy is illustrated in Fig. 7 and its hardware support is discussed in Sec. V.

B. BP Layer Length Reduction

Basic idea. After the cell-level variable reduction, we further extend our exploration to the layer-level for additional memory footprint reduction and latency improvement opportunities. Specifically, we focus on analyzing whether or not the demands from BP process for the intermediate variables produced in FW can be reduced. After all, if BP does not demand such reuse for fast computation, there is no need to store FW-generated intermediate variables. Based on this intuition and further evaluation, we make a *key observation* that *not all the BP cells produce significant gradients for weight updating*. Therefore, we propose to predict these insignificant BP cells and skip their execution. This skipping will require no intermediate variables to be stored in the corresponding FW process as if performing LSTM inference (calculating loss only), further reducing the memory footprint and training latency. We elaborate the observations and proposed techniques as follows.

Key observation: Since the LSTM layer is unrolled into a sequence of LSTM cells, the BP cells in the same layer generate the gradients (δW and δU) for updating the same weight matrices. To explore the characterization of these weight gradients generated by different BP cells of the same layer, we collect the gradients from the different layers of two real-world LSTM training benchmarks (IMDB and WMT (MLPerf) from Table. I) and compare magnitudes (i.e., the accumulated value for all their absolute scalar data), shown in Fig. 8.

Interestingly, we find that how the loss is calculated is determined by LSTM model designers and it impacts which BP cell can be skipped. For instance, we have identified two types of LSTM models based on how the loss is computed:

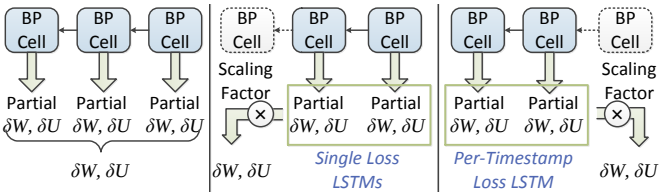


Fig. 9: The diagram of BP cell Skipping. The left figure represents the original BP cell execution, the middle and right figures represent our BP cell skipping approach for single loss LSTM and per-timestamp loss LSTM, respectively.

single loss LSTM and *per-timestamp loss LSTM*. For the former, LSTM models are designed to calculate the loss based on the last timestamp cell at the final layer, e.g., IMDB model for reviewing attitude classification. For the latter, LSTMs calculate the loss per timestamp BP cell at the final layer, e.g., LSTM model from MLPerf performing machine translation. Fig. 8 further demonstrates these two types of LSTM models: different models exhibit different patterns of gradients magnitude. For the single loss LSTMs (e.g., IMDB), the gradients magnitude per layer decreases from the last to the first cell. This is because the single loss vanishes with the increased propagation distance [30]. For per-timestamp loss LSTMs (e.g., WMT), the gradients magnitude grows at each layer from the last cell to the first cell. This is because each cell will receive the loss of the corresponding timestamp and this loss information gets accumulated from the last cell to the first cell. Note that the per-timestamp loss LSTM generates insignificant gradients at the beginning of BP process.

Prediction for the Insignificant BP Cells. By leveraging the key observation above, we propose to skip the execution for those insignificant BP cells. First, we need to build an effective mechanism to identify which BP cells produce insignificant gradients for weight updating. Since it is a futile attempt to perform this identification after the BP cells complete their execution, we aim to identify these BP cells ahead of their execution via the loss information as it determines the magnitude of the gradients for each BP cell. Furthermore, we find that the gradients exhibit a certain correlation with the propagation distance: (1) for the cells at certain timestamp, the gradients increase from the last layer to the first layer; (2) for the cells from the same layer, the weight gradients decrease for single loss LSTMs and increase for per-timestamp loss LSTMs, from the last timestamp to the first timestamp. Based on these, we can build a model with the information of loss, layer and timestamp as inputs to predict the weight gradients magnitudes for each BP cell as follows:

$$\delta W\_Mag = \frac{\alpha \cdot \sum loss \cdot (LN - layerID)}{(LL - timeStamp)^{\beta}} \quad (4)$$

Here, $\sum loss$ represents the loss accumulation from the last timestamp to the current timestamp. For single loss LSTMs, $\sum loss$ is only the loss from the last timestamp cell at the final layer. LN and LL represent the layer number and the layer length of the LSTM model, respectively. $layerID$ and $timeStamp$ indicate the BP cell location in the LSTM graph.

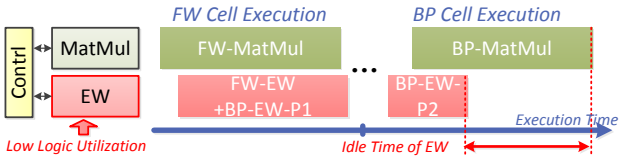


Fig. 10: The diagram of inefficient computational resource allocation when implement LSTM training on the architecture designs with static computational resource allocation.

β indicates gradients magnitude trend according to the LSTM type: for single loss LSTMs, β equals to "1", and for per-timestamp loss LSTMs, β equals to "-1". Finally, α is the factor determined by the LSTM model and the dataset, which can be calculated using the results of the first training epoch. These predicted weight magnitude will be used to compare with the threshold to determine the importance of the BP cells.

However, the basic method above can only reduce the intermediate variables' loading without reducing the actual memory footprint caused by these variables. This is because this basic strategy requires loss information to determine the importance of BP cell, and the loss is be produced after all FW cells' execution. Therefore, to obtain BP cells' importance prior to the FW execution, we propose to predict the loss for current training epoch using historic loss results:

$$pred\_loss_n = loss_{n-1} - \frac{(loss_{n-2} - loss_{n-1})^2}{loss_{n-3} - loss_{n-2}} \quad (5)$$

Here $pred\_loss_n$ represents the predicted loss results for nth epoch, and $loss$ represents the actual loss generated by previous epochs. Note that the first three epochs will not perform the prediction as the prediction requires the historic loss from previous three epochs. With the loss predicted, the importance of BP cells can be predicted before the FW cells' execution, thus, avoiding data movements and memory footprint caused by storing these insignificant variables.

Convergence-Aware BP Execution Skipping. To reduce the impact of skipping BP execution and maintain the convergence speed, we propose to offset the value loss. As shown in the Fig. 9, since the BP cells from the same LSTM layer generate the gradients for the same weights, the value loss can be potentially offset via amplifying the weight gradients results from those significant BP cells: $\delta W_{layer} = \delta W_{partial} \times factor$, here the scaling factor is determined by the predicted weight gradients and the number of the skipped BP cells.

V. η-LSTM Architecture Design

In this section, we discuss our hardware designs and optimizations to effectively support our software-level innovations and provide further enhancements for enabling highly-efficient large LSTM training. Based on design space exploration, we discover that our proposed memory-saving optimizations will induce low hardware utilization when performed on the state-of-the-art NN accelerators, resulting in inefficient execution and increased training time (Sec. V-A). To address this design challenge, we propose a customized highly-efficient LSTM training architecture comprised of a novel PE design

Cycle	1	2	3	4	5	6	7	8	9	10	11	12
Adder input1	A	A	C	C	E	C+D	G	C+D+F	A+B+F+G	A+B+F+G		
Adder Input2		B		D	A+B	F	A+B+E	H		C+D+F+H		
Adder Output				A+B		C+D	A+B+E	C+D+F	A+B+F+G	C+D+F+H		Sum (A~H)

Streaming Processing

Fig. 11: The timing chart for our streaming processing adder-based accumulator. "A" $\sim$ "H" represent the floating-point values to be accumulated.

(Sec. V-B) and its matching runtime scheduler (Sec. V-C). Finally, the overall η-LSTM architectural design is illustrated in Sec. V-D.

A. Hardware Design Challenge Induced by our Memory-Saving Optimizations

Irregular Workload Pattern: As discussed in the previous section, our memory-saving optimizations move some computation from the BP cells to the FW cells which further reduces the computation workload per BP cell. As discussed in Fig. 2 in Sec.II, as a result, the amount of MatMul and EW operations varies across cells and even becomes different for the same cell at different computation phases (FW or BP). Additionally, the reduction of the computation workload in each BP cell relies on runtime information, making the amount of the MatMul and the EW operations vary across different training iterations, training epochs, and datasets. Thus, the workload pattern becomes more irregular under our memory-saving optimizations which requires special hardware support.

Ineffectiveness of the State-of-the-Art NN Accelerators: There have been several architectural designs for accelerating NN execution, e.g., NPUs [31], [32] and LSTM inference accelerator [11]–[13], [17], [33]. Some designs, e.g. [33], adopt a unified processing element design to include a large number of logic for performing all the essential functionalities (e.g., multiplication, add, accumulation, activation), resulting in low logic utilization and high area cost. More commonly, some designs, e.g. [11], [13], [17] statically distribute the computation resources into several hardware modules (e.g., MatMul and EW modules) with each hardware module conducting different operations. The logic resources per hardware module are determined by the amount of the operations. However, this kind of static allocation approach may cause inefficient execution as the workload pattern becomes more irregular under our memory-saving optimizations for highly-efficient LSTM training. As shown in Fig. 10, when the EW execution in the BP cell becomes sparse, the corresponding hardware logic will be idle for a long period of time, causing low logic utilization and degraded performance.

B. Omni-Processing Element Design

To achieve high hardware utilization and corresponding performance benefit, we propose to dynamically distribute the computational resources according to the workload requests. To this end, we first group the hardware computational resources into *fine-grained* pieces, i.e., processing elements (PEs), to enable flexible resource allocation. We then propose

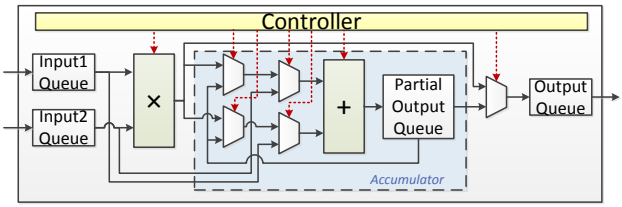


Fig. 12: The Omni-PE Design.

a novel universal PE design, *Omni-PE*, that is able to conduct the streaming processing for all the operations involved in the LSTM training. To be specific, we aim to reduce the PE resource consumption via maximizing the logic reuse between MatMul and EW operations.

The MatMul computation requires the multiplier and accumulator to process the streaming input each cycle, while the EW computation requires the multiplier, adder and activation function unit for the streaming input. Since the activation function unit can be implemented using the on-chip storage to build the look-up table [11], the major logic difference between MatMul and EW lies in the differences between the accumulator and the adder units.

Though both the accumulator and the adder perform the addition operation, their designs are very different. At each cycle, the accumulator sums up the current cycle's input with the accumulated value produced by the previous cycle [34]. However, when using the adder to implement this execution flow, it is unable to accept input at every cycle to support the streaming processing as performing addition in the adder takes multiple cycles (e.g., 8 cycles in our study), which periodically stalls the accumulation to wait for the addition output. Therefore, the accumulator and adder usually exhibit different low-level designs for enabling the streaming processing and they often cannot directly perform each other's functionality.

In this study, we enable the streaming accumulation in the adder by rearranging the computation flow. Instead of always waiting for the full accumulation results from the previous cycles, our design leverages partial outcomes to achieve the streaming processing, as illustrated by an example in Fig. 11. To simplify the discussion, we assume the *add* latency takes 2 cycles in this example. As it shows, at the beginning of the accumulation, the computation is conducted every 2 cycles given the streaming inputs since it takes 2 cycles to finish one addition. With the partial outputs ready later (e.g., at 4th cycle in the figure), the partial outputs (e.g., A+B) will be sent back to the adder while waiting for the next steaming input (e.g., F). As can be seen, the adder is effectively pipelined which is able to accept input at every cycle to support the streaming processing. Finally, when there is no more data from the input stream (e.g., from 9th cycle), all the partial accumulated results will be summed up to produce the final result.

Given that the adder is able to conduct the streaming accumulation, we then propose our Omni-PE design which can perform both MatMul and EW operations, as illustrated in Fig. 12. Our Omni-PE contains one multiplier and one adder. There are four multiplexers (MUX) inserted between the multiplier and adder, where the left two MUXs will flip per cycle to ensure the data evenly transmitted to the two input ports of the adder, and right two MUXs select the input for the adder as either the PE's input or the multiplier's output. The adder's output is then sent to a partial output queue for accumulation. Finally, a MUX is employed before the output queue to select the appropriate output from either the multiplier or the adder. With the control signals generated by the controller, our Omni-PE can be dynamically configured to perform the following operations for LSTM training:

- For matrix-vector multiplication ($\cdot$), both the multiplier and adder will be activated; the adder acts as the accumulator and the output MUX collects the results from the partial output queue;
- For element-wise multiplication ($\odot$) and outer-production ($\otimes$), only the multiplier will be activated, and its output will be sent to the output queue through the final MUX to skip the adder;
- For element-wise addition ($+$), only the adder will be activated which will take both inputs to the PE. The final MUX will select results from the partial output queue.

C. Runtime Resource Allocation

We then explore a Runtime Resource Allocation (R^2A) scheduler that concurrently launches the data dependence operations (i.e., MatMul and EW) and intelligently assigns our omni-PEs to these operations for the optimal performance and energy-efficiency. Our R^2A scheduler first estimates the amount of workloads (i.e., the operation types and their number) through the entire LSTM training, which is determined by the model configuration. Based on the estimated workloads of MatMul and EW, the scheduler will then divide the PEs into two groups with each group configured to perform MatMul and EW operations, respectively. During the LSTM training, some PEs become idle when assigned operation is not ready to start due to the data dependency on another operation. To improve the PE utilization, those idle PEs will be firstly re-assigned to conduct operations with ready inputs and then switch back later to perform the operations originally assigned by the scheduler.

For example, PEs are divided to perform MatMul and EW operations. During the forward propagation, since the EW operations depend on the results from MatMul, the PEs assigned for EW will first support MatMul (i.e., swing PEs) and then return to process EW once the adequate inputs have been generated from MatMul. Note that there exists no pipeline stalls as the swing PEs design can effectively avoid dependency waiting by reassigning the necessary number of PEs from EW for MatMul outputs generation. This results in efficiently overlapped execution between these two kernels. Both the PE assignment and the functionality switch are performed by the controllers illustrated in Fig. 12.

D. The Overall η-LSTM Architecture

As shown in Fig. 13a, the η-LSTM accelerator groups the Omni-PEs into channels for the parallelism enhancement

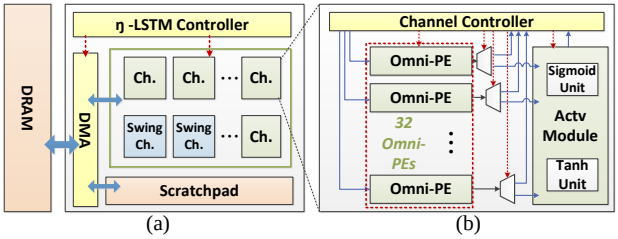


Fig. 13: The overview of our η-LSTM architecture and the micro-architecture diagram of Channel architecture.

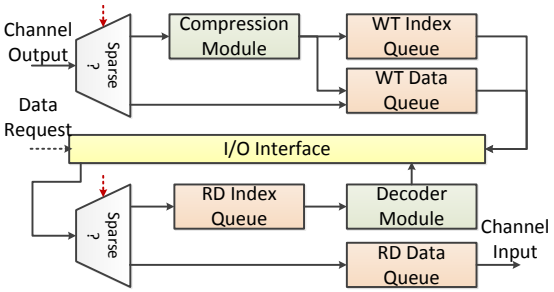


Fig. 14: The Customized DMA Module

and the control complexity reduction. We expand the concept of swinging individual PE's functionalities in our runtime resource allocation, to swing channels' functionalities as a group of PEs (i.e., *swing channel design*). Additionally, it also includes a customized direct memory access (DMA) module, a scratchpad and an η-LSTM controller (performing R^2A scheduler). The detailed designs are described as follows:

Channel Architecture: As Shown in Fig. 13b, one channel is composed of 32 Omni-PEs and a *channel controller* which manipulates all PEs' controllers within the channel. The channel controller is also responsible for performing data communication with the on-chip scratchpad. Note that for the operation like outer-product, all the PEs could request the same input data. To improve the data distribution efficiency, we insert a broadcasting queue inside the channel controller to provide data to all the PEs. Additionally, we apply an activation module inside each channel for calculating the activation functions, i.e., sigmoid (σ) and hyperbolic tangent ($tanh$). To reduce the area overhead, the activation module only contains one sigmoid unit and one hyperbolic tangent unit for all the 32 Omni-PEs as the workloads of activation operations are much lower than other operations in the LSTM. We further adopt a lookup table design to avoid the complex logic design for either the sigmoid (σ) or hyperbolic tangent ($tanh$) unit.

Customized DMA Module: To enable the cell-level intermediate variables' reduction and skipping execution for insignificant BP cells, we customized our DMA module that includes a data compression module and a decoder module. When the DMA module receives data from the channels, it will first identify the received data as dense or sparse; the dense data will be sent to the WT data queue, and the sparse data will be sent to the compression module to perform pruning according to their value magnitudes. The compression module will output the important data and their indices to the WT data queue and WT index queue, respectively. Finally, the data in the WT data queue and WT index queue will be stored either in the on-chip Scratchpad or off-chip DRAM via the I/O interface. When the DMA module receives the data requests from the channels, the dense data will be loaded via I/O interface and stored into the RD data queue waiting to be distributed by the η-LSTM controller. On the other hand, the sparse data value and their indices will be stored in the RD data queue and RD index queue, respectively. Note that the sparse intermediate variables can help locate the unimportant computation, providing the opportunities in further reducing the data requests for involving dense variables. In order to only load the data for important computation from the dense variables, we introduce the decoder module using the index information of the sparse operand to locate the corresponding address of the operand. Finally, the operands in the RD data queue will be sent to the channels for the training execution.

TABLE I: Large LSTM Training Benchmarks.

Name	Abbr.	Hidden_Size	Layers	Length
TREC-10 [35]	QC	3072	2	18
PTB [26]	LM	1536	4	35
IMDB [28]	SA	2048	3	100
WAYMO [36]	AD	1024	3	128
WMT [29], [37], [38]	MT	1024	4	151
BABI [39]	QA	1280	5	303

Scalability Discussion: Since our channel architecture supports SIMT execution (e.g., 1 channel supports 32 PEs) and the explicit execution dependency between channels is eliminated by our swing channel design, η-LSTM supports highly parallel execution. Thus, by adding more channels, η-LSTM can achieve linearly increasing throughput within the constraints of thermal, power and area. Additionally, since our co-design strategy enables both drastic intermediate data reduction (software) and their fast consumption (hardware), the memory cost is not required to linearly grow with the number of channels. Thus, η-LSTM exhibits good design scalability.

VI. Evaluation

A. Experimental Methodology

Training Benchmarks: We employ six representative large LSTM applications listed in the Table. I as our training benchmarks. Each application has its unique LSTM model configuration in terms of the hidden size (Hidden_Size), layer number (Layers), and layer length (Length). TREC-10 [35] performs question segmentation (QC) that classifies the questions into 10 categories. PTB [26] represents Penn TreeBank, which is widely used for word-level language modeling (LM). IMDB [28] performs the sentiment analysis (SA) that predicts the positive or negative attitude from texts. WAYMO [36] is a commercial LSTM model to perform object tracking for autonomous driving (AD). WMT [29] is a representative LSTM model included in the MLperf benchmark [38] which performs German-English machine translation (MT) [37]. BABI [39]

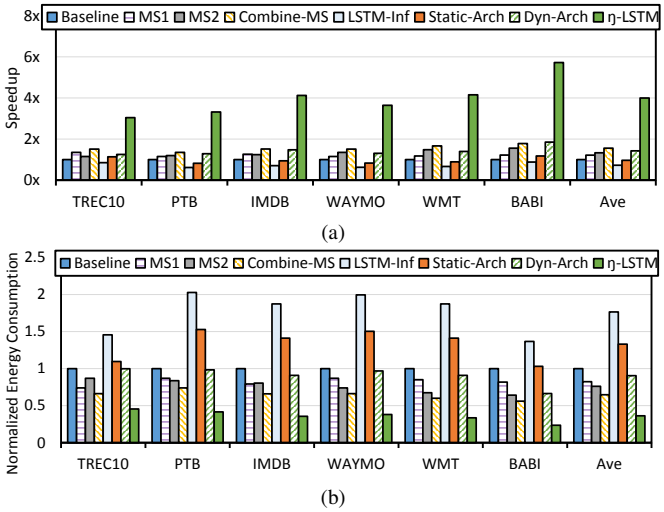


Fig. 15: The (a) speedup and (b) normalized energy consumption of η-LSTM design compared against other designs.

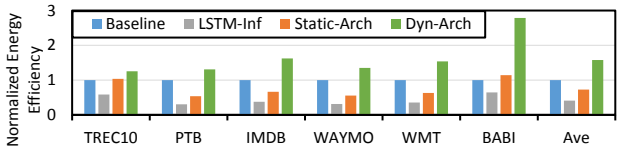


Fig. 16: The energy efficiency comparison across different hardware design scenarios.

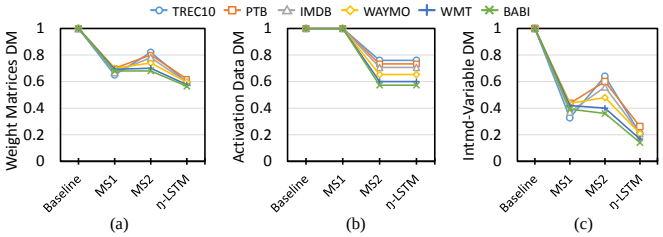


Fig. 17: The effectiveness of our memory-saving techniques on the data movement reduction for (a) weight matrices, (b) activation data, (c) intermediate variables.

performs the question answering (QA) for automatic text understanding and reasoning.

Environment Setup: The baseline and our software-level optimizations for the LSTM training are implemented using Pytorch [24], a popular open-sourced machine learning framework that supports dynamic computation graphs on the state-of-the-art GPUs. We conduct 32-bit floating-point LSTM training with a moderate batch size (i.e., 128) for our evaluation. On the hardware side, we implement all the hardware design optimizations discussed in Sec. V in Verilog and synthesized on the Xilinx Vertex-UltraScale+ HBM VCU128 FPGA evaluation board [40]. The design is operated at 500MHz. For the off-chip memory access evaluation, we adopt the HBM IP interface [41] connected to the on-board external HBM memory. The execution latency results are obtained from the post-synthesis design, and the energy consumption is evaluated using the Vivado Power Analysis [42] at the post-route level. Note that there are hardware resource differences between the NVIDIA GPUs and the FPGA board. For example, VCU128 only has an 8GB HBM memory while V100 has a 32GB HBM with double bandwidth. To provide fair evaluation, we implemented 40 channels on the VCU128 and set the HBM bandwidth as 224GB/s, which provides nearly a quarter of NVIDIA V100's computational capability and memory resources; and then we assemble our LSTM training accelerator with four VCU128 boards for performing the LSTM training (e.g., 1/4 training workload per board). We collect the latency and energy consumption accordingly.

Comparison Cases: To evaluate the effectiveness of η-LSTM, we compare it with the following design scenarios: (1) Baseline: the state-of-the-art GPU accelerated LSTM training execution; (2) MS1: our cell-level variables' reduction optimization (Section IV-A) implemented on top of the state-of-the-art GPU; (3) MS2: our BP cell computation reduction mechanism (Section IV-B) implemented on GPU; (4) Combine-MS: our combined software-level memory-saving optimizations from Sec. IV implemented on GPU; (5) LSTM-Inf: state-of-the-art LSTM inference accelerator design [11]; (6) Static-Arch: the architectural design for LSTM training with computational resources statically distributed across several hardware modules for conducting different operations (the distribution is based on the TREC10 configuration); and (7) Dyn-Arch: our architectural design of η-LSTM with dynamic resource allocation but without software-level memory-saving optimizations.

B. Experimental Results

1) Effectiveness on Performance and Energy Consumption: Fig. 15 illustrates the normalized performance and energy consumption of our η-LSTM design compared with other design scenarios. We first evaluate the effectiveness of memory-saving optimizations. From the figure, we observe our cell-level variable reduction (MS1) achieves on average $1.21\times$ (up to $1.35\times$) speedup and saves 17.66% (up to 26.01%) energy, i.e., $1.21\times$ (up to $1.35\times$ energy improvement), comparing with the baseline case. Our BP cell computation reduction (MS2) achieves on average $1.32\times$ (up to $1.56\times$) speedup and 23% (up to 35.75%) energy saving, i.e., $1.30\times$ (up to $1.56\times$) energy improvement, comparing with the baseline case. We observe that MS1 is more effective for LSTM training with larger hidden size, and MS2 is more effective for the LSTM training with larger layer length. This is because the LSTM with large hidden size provides more opportunities for MS1 to locate the trivial intra-cell variables, and more trivial BP cells can be identified in the LSTM with a long layer length that help achieve better performance and energy saving. Overall, our optimizations achieve on average $1.56\times$ (up to $1.79\times$) speedup and 35.26% (up to 43.94%) energy saving, i.e., $1.54\times$ (up to $1.78\times$) energy improvement, comparing with the baseline case.

We then evaluate the effectiveness of our architectural design. From the figure, we observe that the LSTM-Inf on average decreases the performance by 27.52% and incurs 76.56% additional energy cost. This is because LSTM-Inf

suffers from the resource-consuming PE design which results in significantly lower throughput under the resource limitation. Additionally, LSTM-Inf adopts a static computational resource allocation scheme, which is based on the TREC10 dataset in our experiment. This allocation strategy makes it hard for LSTM-Inf to be efficient for different LSTMs. By using Omni-PE in Static-Arch, the average performance is decreased by 3.36% with 33.03% additional energy overhead over the baseline. Although improved from LSTM-Inf, this still suffers from the same static computational resource allocation issue as that in LSTM-Inf. On the other hand, our Dyn-Arch design outperforms the baseline case by 1.42× (up to 1.85×) on performance and 9.5% (up to 33.66%) on energy saving, i.e., 1.10× (up to 1.51×) energy improvement. We further compare the energy efficiency among baseline, LSTM-Inf, Static-Arch, and Dyn-Arch as shown in Fig. 16. From the figure, the energy efficiency of LSTM-Inf is always lower than the baseline. The energy efficiency of Static-Arch varies for different LSTM benchmarks, and only when the LSTM workload matches the on-chip resource distribution, the energy efficiency of Static-Arch can outperform the baseline case. However, our Dyn-Arch can always outperform the baseline case and achieve on average 1.67× (up to 2.69×) energy efficiency.

Finally, our η-LSTM takes the advantages of both the software and hardware optimizations, and outperforms the baseline case by 3.99× (up to 5.73×) on speedup and 63.70% (up to 76.48%) on energy saving, i.e., 2.75× (up to 4.25×) energy improvement.

2) Effectiveness on Data Movement Reduction: Fig. 17 shows the effectiveness of our memory-saving optimizations on the data movement reduction. From the figure, we observe that MS1 on average reduces the data movement of weight matrices and intermediate variables by 31.79% and 60.27%, respectively. But MS1 has no impact on reducing the activation data movement. The MS2 on average reduces the data movement of weight matrices, activation data, and intermediate variables by 24.67%, 32.89%, and 49.34%, respectively. The MS1 outperforms MS2 on data movement for weight matrices and intermediate variables, but MS2 provides unique opportunity of deducting the activation data movement. Overall, these two optimizations can reduce the data movement involved in the LSTM training from different levels (i.e., cell-level and layer-level). Our η-LSTM outperforms the baseline case by on average 40.85%, 32.89% and 80.04% on the data movement reduction for weight matrices, activation data and intermediate variables, respectively.

3) Effectiveness on Memory Footprint Reduction: We then evaluate the effectiveness of our memory-saving optimizations on the memory footprint reduction. As shown in Fig. 18, all our optimizations efficiently reduce memory footprint. Specifically, the MS1 and MS2 achieve on average 32.37% (up to 39.09%) and 41.65% (up to 61.68%) memory footprint reduction, respectively. Overall, our integrated memory-saving optimizations achieve on average 57.52% (up to 75.75%) reduction in memory footprint.

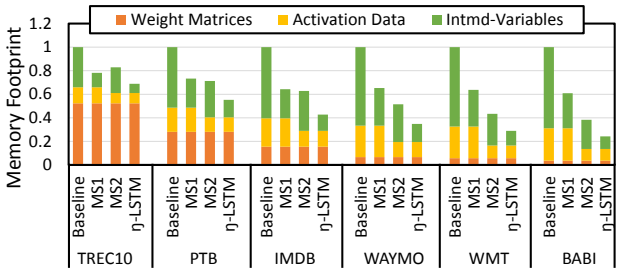


Fig. 18: The effectiveness of our memory-saving techniques on the memory-footprint reduction.

TABLE II: Accuracy Impact

	TREC10	PTB	IMDB	WAYMO	WMT	BABI
Baseline	78.82%	217.19 PPL	76.78%	0.138 MAE	3.13 BLEU	68.75%
Combined-MS	78.80%	218.36 PPL	76.78%	0.138 MAE	3.13 BLEU	68.69%
PPL: Perplexity (Lower is Better); MAE: Mean Absolute Error (Lower is Better); BLEU: BiLingual Evaluation Understudy (Higher is Better)						

4) Accuracy Impact: Note that our memory-saving optimizations perform approximate computing. Therefore, we conduct the accuracy analysis in terms of the convergence speed and the final accuracy difference. We find that the convergence speed is not affected for all our LSTM benchmarks. This is because the neural network training exhibits superior ability for tolerating noise and small changes, and more importantly, we adopt the convergence speed aware scaling for our BP cell skipping technique that compensates the negative impact on the convergence speed. As illustrated by Table. II, our memory-saving optimizations only incur $< 1\%$ accuracy difference compared to the baseline case.

5) Analysis on Adder-Based Accumulator Design: We compare our adder-based accumulator design with the Xilinx accumulator IP [34]. As the results demonstrated by Table. III, we first observe that our design saves 43.61% LUT and 37.25% FF compared with the Xilinx IP design. Besides, our design reduces the energy consumption by 17% compared with Xilinx IP design. This is because the Xilinx IP targets translating the 32-bit floating-point accumulation into 64-bit fixed-point accumulation, thus consuming more logic resources and causing more dynamic power overhead. Finally, we observe that the Xilinx IP exhibits lower latency compared with our design. However, since we are targeting the large LSTM training, the streaming input for accumulation is long. Our design only causes $< 2.87\%$ latency overhead for conducting accumulation with more than 1024 streaming inputs. Note that we already include this latency overhead in our overall evaluation.

VII. Related Works

Neural Network Acceleration: There have been massive numbers of studies focusing on the acceleration of the NN training. For example, Procrustes [43] exploits the sparsity in the CNN training and conducts the customized accelerator design. TensorDash [44] also designs an accelerator for sparse CNN training with a smart workload distribution for the dynamic variable sparsity. However, the LSTM training exhibits significantly different execution pattern from the CNN

TABLE III: The comparison between the Xilinx accumulator IP and our adder-based accumulator design.

	Utilization		Dynamic Power (W)				Latency
	LUT	FF	Clock	Signal	Logic	Total	(Cycle)
Xilinx IP	821	969	0.026	0.031	0.043	0.1	20
Our Design	463	608	0.014	0.039	0.03	0.083	50

training, the previous works can hardly reduce the intermediate variable size.

There are also several studies that focus on exploiting the software and architectural co-designs for accelerating the LSTM inference execution [11]–[13], [17], [45], [46]. For example, ESE [11] and DeltaRNN [12] compress the weight matrices and propose accelerator designs to improve the inference performance. Since the massive intermediate variable movement is caused by the unique features of LSTM training processing, these LSTM inference works fail to address this challenge. Additionally, the static computational resource allocation employed in these accelerator designs causes inefficient logic utilization to memory-saving optimizations that decreases the overall performance.

Memory Footprint Reduction: Several studies have explored trading the memory footprint reduction with the re-computation [14]–[16], [21]. For example, SmartExchange [21] proposes to encode the weight matrices into a small format and decode once they are loaded on-chip. However, there is very limited opportunity to compress or encode the FW intermediate variable involved in the LSTM training as discussed in Section IV-A. On the other hand, Echo [16] proposes to store partial intermediate variable for attention layer during the FW processing and perform the re-computing based on partial variables to generate the entire variables. However, the LSTM training exhibits different computation pattern that the generated intermediate variables are independent with each other, hence, Echo can hardly be applied for LSTM training.

VIII. Conclusion

To enable a highly-efficient LSTM training solution for the ever-growing model size, we propose the first cross-stack training solution, *η-LSTM*, for large LSTM models. *η-LSTM* comprises both software-level and hardware-level innovations that effectively lower the memory footprint upper-bound and excessive data movements during large LSTM training, while also drastically improving training performance and energy efficiency. Experimental results on six real-world large LSTM training benchmarks demonstrate that *η-LSTM* significantly reduces the memory-footprint and the data movement for the LSTM training. Furthermore, it outperforms the state-of-the-art GPU implementation for LSTM training on performance and energy efficiency.

References

[1] A. Girma, X. Yan, and A. Homaifar, "Driver Identification Based on Vehicle Telematics Data using LSTM-Recurrent Neural Network," in *2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)*, pp. 894–902, IEEE, 2019.

[2] S. H. Park, B. Kim, C. M. Kang, C. C. Chung, and J. W. Choi, "Sequence-to-Sequence Prediction of Vehicle Trajectory via LSTM Encoder-Decoder Architecture," in *2018 IEEE Intelligent Vehicles Symposium (IV)*, pp. 1672–1678, IEEE, 2018.

[3] S. Ghosh, O. Vinyals, B. Strope, S. Roy, T. Dean, and L. Heck, "Contextual LSTM (CLSTM) Models for Large Scale NLP Tasks," *arXiv preprint arXiv:1602.06291*, 2016.

[4] Z. Huang, W. Xu, and K. Yu, "Bidirectional LSTM-CRF Models for Sequence Tagging," *arXiv preprint arXiv:1508.01991*, 2015.

[5] N. Tax, I. Verenich, M. La Rosa, and M. Dumas, "Predictive Business Process Monitoring with LSTM Neural Networks," in *International Conference on Advanced Information Systems Engineering*, pp. 477–492, Springer, 2017.

[6] R. Jia and P. Liang, "Data recombination for neural semantic parsing," *arXiv preprint arXiv:1606.03622*, 2016.

[7] M. Z. Islam, M. M. Islam, and A. Asraf, "A Combined Deep CNN-LSTM Network for The Detection of Novel Coronavirus (COVID-19) using X-Ray Images," *Informatics in Medicine Unlocked*, vol. 20, p. 100412, 2020.

[8] S. Polyzos, A. Samitas, and A. E. Spyridou, "Tourism Demand and The COVID-19 Pandemic: An LSTM Approach," *Tourism Recreation Research*, pp. 1–13, 2020.

[9] O. I. Abiodun, A. Jantan, A. E. Omolara, K. V. Dada, N. A. Mohamed, and H. Arshad, "State-of-The-Art in Artificial Neural Network Applications: A Survey," *Heliyon*, vol. 4, no. 11, p. e00938, 2018.

[10] W. Liu, Z. Wang, X. Liu, N. Zeng, Y. Liu, and F. E. Alsaadi, "A Survey of Deep Neural Network Architectures and Their Applications," *Neurocomputing*, vol. 234, pp. 11–26, 2017.

[11] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, *et al.*, "ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 75–84, 2017.

[12] C. Gao, D. Neil, E. Ceolini, S.-C. Liu, and T. Delbruck, "DeltaRNN: A Power-Efficient Recurrent Neural Network Accelerator," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 21–30, 2018.

[13] 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," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 11–20, 2018.

[14] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "vDNN: Virtualized Deep Neural Networks for Scalable, Memory-Efficient Neural Network Design," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–13, IEEE, 2016.

[15] L. Wang, J. Ye, Y. Zhao, W. Wu, A. Li, S. L. Song, Z. Xu, and T. Kraska, "SuperNeurons: Dynamic GPU Memory Management for Training Deep Neural Networks," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 41–53, 2018.

[16] B. Zheng, N. Vijaykumar, and G. Pekhimenko, "Echo: Compiler-Based GPU Memory Footprint Reduction for LSTM RNN Training," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 1089–1102, IEEE, 2020.

[17] S. Wang, P. Lin, R. Hu, H. Wang, J. He, Q. Huang, and S. Chang, "Acceleration of LSTM with Structured Pruning Method on FPGA," *IEEE Access*, vol. 7, pp. 62930–62937, 2019.

[18] A. Sherstinsky, "Fundamentals of Recurrent Neural network (Rnn) and Long Short-Term Memory (LSTM) Network," *Physica D: Nonlinear Phenomena*, vol. 404, p. 132306, 2020.

[19] J.-H. Luo, J. Wu, and W. Lin, "Thinet: A Filter Level Pruning Method for Deep Neural Network Compression," in *Proceedings of the IEEE international conference on computer vision*, pp. 5058–5066, 2017.

[20] K. Ullrich, E. Meeds, and M. Welling, "Soft Weight-Sharing for Neural Network Compression," *arXiv preprint arXiv:1702.04008*, 2017.

[21] Y. Zhao, X. Chen, Y. Wang, C. Li, H. You, Y. Fu, Y. Xie, Z. Wang, and Y. Lin, "SmartExchange: Trading Higher-Cost Memory Storage/Access for Lower-Cost Computation," *arXiv preprint arXiv:2005.03403*, 2020.

[22] "NVIDIA TURING GPU ARCHITECTURE." https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[23] "NVIDIA TESLA V100 GPU ARCHITECTURE." https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[24] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, "Pytorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in neural information processing systems*, pp. 8026–8037, 2019.

[25] N. V. Profiler, "User's guide, 2014."

[26] M. Marcus, G. Kim, M. A. Marcinkiewicz, R. MacIntyre, A. Bies, M. Ferguson, K. Katz, and B. Schasberger, "The Penn Treebank: Annotating Predicate Argument Structure," in *HUMAN LANGUAGE TECHNOLOGY: Proceedings of a Workshop held at Plainsboro, New Jersey, March 8-11, 1994*, 1994.

[27] Y. Yu, X. Si, C. Hu, and J. Zhang, "A Review of Recurrent Neural Networks: LSTM Cells and Network Architectures," *Neural computation*, vol. 31, no. 7, pp. 1235–1270, 2019.

[28] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts, "Learning Word Vectors for Sentiment Analysis," in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, (Portland, Oregon, USA), pp. 142–150, Association for Computational Linguistics, June 2011.

[29] M. Johnson, M. Schuster, Q. V. Le, M. Krikun, Y. Wu, Z. Chen, N. Thorat, F. Viégas, M. Wattenberg, G. Corrado, *et al.*, "Google's Multilingual Neural Machine Translation System: Enabling Zero-Shot Translation," *Transactions of the Association for Computational Linguistics*, vol. 5, pp. 339–351, 2017.

[30] S. Hochreiter, Y. Bengio, P. Frasconi, J. Schmidhuber, *et al.*, "Gradient Flow in Recurrent Nets: The Difficulty of Learning Long-Term Dependencies," *A field guide to dynamical recurrent neural networks. IEEE Press*, 2001.

[31] D. A. Palmer and M. Florea, "Neural Processing Unit," Feb. 18 2014. US Patent 8,655,815.

[32] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural Acceleration for General-Purpose Approximate Programs," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 449–460, IEEE, 2012.

[33] F. Silfa, G. Dot, J.-M. Arnau, and A. Gonzalez, "E-PUR: An Energy-Efficient Processing Unit for Recurrent Neural Networks," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, pp. 1–12, 2018.

[34] "Accumulator v12.0 LogiCORE IP Product Guide (PG119) - Xilinx." https://www.xilinx.com/support/documentation/ip_documentation/accum/v12_0/pg119-c-accum.pdf.

[44] M. Mahmoud, I. Edo, A. H. Zadeh, O. M. Awad, G. Pekhimenko, J. Albericio, and A. Moshovos, "TensorDash: Exploiting Sparsity to Accelerate Deep Neural Network Training," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 711–724, IEEE, 2020.

[35] E. M. Voorhees and D. M. Tice, "Building A Question Answering Test Collection," in *Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 200–207, 2000.

[36] Z. Gu, Z. Li, X. Di, and R. Shi, "An LSTM-Based Autonomous Driving Model Using a Waymo Open Dataset," *Applied Sciences*, vol. 10, no. 6, p. 2046, 2020.

[37] M.-T. Luong and C. D. Manning, "Achieving Open Vocabulary Neural Machine Translation with Hybrid Word-Character Models," *arXiv preprint arXiv:1604.00788*, 2016.

[38] P. Mattson, C. Cheng, G. Diamos, C. Coleman, P. Micikevicius, D. Patterson, H. Tang, G.-Y. Wei, P. Bailis, V. Bittorf, *et al.*, "MLPerf Training Benchmark," *Proceedings of Machine Learning and Systems*, vol. 2, pp. 336–349, 2020.

[39] J. Weston, A. Bordes, S. Chopra, A. M. Rush, B. van Merriënboer, A. Joulin, and T. Mikolov, "Towards AI-Complete Question Answering: A Set of Prerequisite Toy Tasks," *arXiv preprint arXiv:1502.05698*, 2015.

[40] M. Wissolik, D. Zacher, A. Torza, and B. Da, "Virtex UltraScale+ HBM FPGA: A Revolutionary Increase in Memory Performance," *Xilinx Whitepaper*, 2017.

[41] "AXI High Bandwidth Memory Controller v1.0 Product Guide (v1.0)." https://www.xilinx.com/support/documentation/ip_documentation/hbm/v1_0/pg276-axi-hbm.pdf.

[42] T. Feist, "Vivado Design Suite," *White Paper*, vol. 5, p. 30, 2012.

[43] D. Yang, A. Ghasemazar, X. Ren, M. Golub, G. Lemieux, and M. Lis, "Procrustes: a Dataflow and Accelerator for Sparse Deep Neural Network Training," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 711–724, IEEE, 2020.

[45] Y. Guan, Z. Yuan, G. Sun, and J. Cong, "FPGA-Based Accelerator for Long Short-Term Memory Recurrent Neural Networks," in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 629–634, IEEE, 2017.

[46] X. Zhang, C. Xie, J. Wang, W. Zhang, and X. Fu, "Towards Memory Friendly Long-Short Term Memory Networks (LSTMs) on Mobile GPUs," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 162–174, IEEE, 2018.