Boosting LSTM Performance Through Dynamic Precision Selection

Franyell Silfa
Universitat Politècnica de Catalunya
Barcelona, Spain
fsilfa@ac.upc.edu

Jose Maria Arnau
Universitat Politècnica de Catalunya
Barcelona, Spain
jarnau@ac.upc.edu

Antonio González
Universitat Politècnica de Catalunya
Barcelona, Spain
antonio@ac.upc.edu

***Abstract*—The use of low numerical precision is a fundamental optimization included in modern accelerators for Deep Neural Networks (DNNs). The number of bits of the numerical representation is set to the minimum precision that is able to retain accuracy based on an offline profiling, and it is kept constant for DNN inference.**

In this work, we explore the use of dynamic precision selection during DNN inference. We focus on Long Short Term Memory (LSTM) networks, which represent the state-of-the-art networks for applications such as machine translation and speech recognition. Unlike conventional DNNs, LSTM networks remember information from previous evaluations by storing data in the LSTM cell state. Our key observation is that the cell state determines the amount of precision required: time-steps where the cell state changes significantly require higher precision, whereas time-steps where the cell state is stable can be computed with lower precision without any loss in accuracy.

We propose a novel hardware scheme that tracks the evolution of the elements in the LSTM cell state and dynamically selects the appropriate precision on each time-step. For a set of popular LSTM networks, it chooses the lowest precision for 57% of the time, outperforming systems that fix the precision statically. We evaluate our proposal on top of a modern highly-optimized LSTM accelerator, and show that it provides 1.46x speedup and 19.2% energy savings on average without degrading the model accuracy. Our scheme has an overhead of less than 8%.

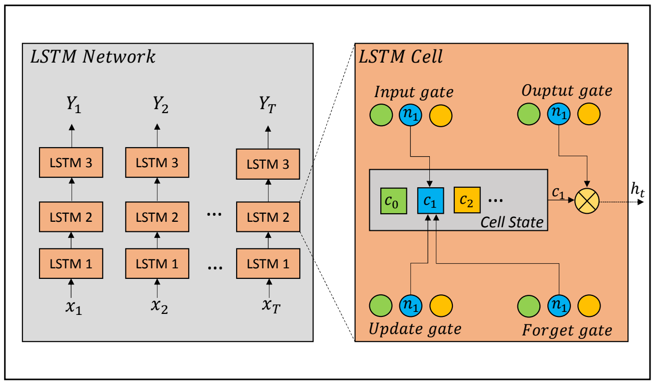
***Keywords*-RNNs; Long Short Term Memory; Accelerators; Quantization;**

I. Introduction

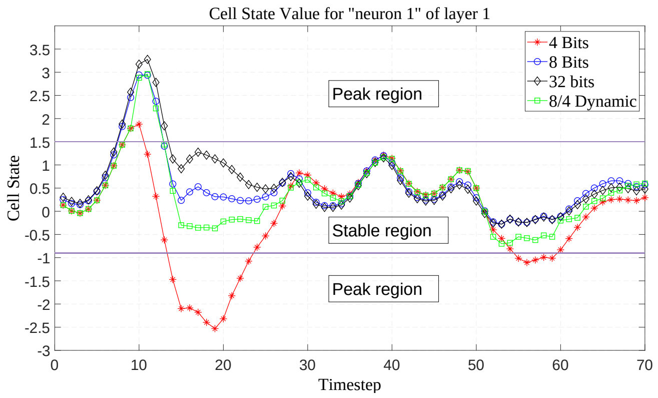
Long Short Term Memory (LSTM) neural networks represent a state-of-the-art solution for sequence-to-sequence problems such as machine translation [1], automatic caption generation [2] or speech recognition [3]. Unlike conventional DNNs, LSTMs store information from previous executions to improve the accuracy of future prediction. In addition, they can handle input and output sequences of variable length. However, their recurrent nature severely constrains the amount of parallelism that can be exploited when evaluating the different elements of an input sequence, hence, making it challenging to achieve low latency LSTM inference on CPUs [4] and GPUs [5]. Not surprisingly, accelerators to boost LSTM performance have been recently presented [6]–[8].

Perhaps the most popular and effective optimization for LSTMs is the use of reduced precision via linear quantization, where precision means the number of bits employed to encode inputs and weights. TPU [6] employs 8-bit weights and inputs for LSTM inference. Other proposals, such as Stripes [9] and BitFusion [10], support variable precision to further improve performance and energy efficiency for LSTM networks that can be computed with less than 8 bits. Despite the additional flexibility of these accelerators, the precision for each LSTM network is determined offline and it is fixed during inference. In other words, different LSTM networks can be evaluated at different precision, but a given LSTM is always computed at the same precision for all the inputs. In this work, we propose a mechanism to dynamically select precision during inference of each individual LSTM to boost performance without any loss in accuracy.

To find a practical scheme to set the precision online, we analyzed the impact of the precision on the state of the LSTM cell. The cell state is the critical component of an LSTM network as it stores information from previous inputs that will be used for future predictions. It consists of an array of N elements, where each element is computed by four neurons in different gates, i.e., fully-connected layers. Figure Ib shows the evolution of one element in the cell state in a speech recognition network [11], at three different levels of precision (32-bit floating-point, 8-bit integer, and 4-bit integer). As can be seen, 8-bit quantization closely tracks the 32-bit full-precision version's behavior, resulting in the same accuracy. However, 4-bit quantization introduces significant errors in some time-steps, resulting in noticeable accuracy loss. Previous schemes would conclude that this LSTM network layer cannot be evaluated using 4 bits. However, a more detailed look at Figure I reveals that the 4-bit version can mimic the behavior of the 32-bit version for a large percentage of time-steps. More specifically, for regions where the cell state is stable, the 4-bit version is quite accurate, whereas, for regions where the cell state changes rapidly, i.e., peaks/valleys, it tends to exhibit a more significant error. A more extensive analysis by using different LSTM networks and their respective training datasets shows that this behavior is quite prevalent. Precisely, for stable regions, the 4-bit version adds a small error of less than 25%, whereas, for peaks/valleys, it introduces a more significant error of 77% on average. For the sake of brevity, we will use the term peak to refer to both peaks and valleys.



(a)



(b)

Figure 1. Evolution of one element (n_1) in the cell state of a speech recognition LSTM network [11]. As it can be seen in (a), on each time-step the elements in the cell state vector are updated using a combination of the gates' output and the previous cell state. Also, for each LSTM Cell, the cell state vector is different. Shown in (b) is the value for element n_1 of the cell state. As it can bee seen, in stable regions the 4-bit version evaluation accurately tracks the behavior of the full-precision (FP32) version. However, a large error is introduced when the tracked element is on a peak.

In this work, we propose a scheme that dynamically selects the appropriate precision by monitoring the LSTM cell state. Our system keeps track of the values of each element in the cell state in recent time-steps. If the value is stable, the lowest precision supported by the hardware is selected to evaluate the next time-step. Otherwise, higher precision is used to avoid significant errors during peaks. For our benchmarks, this simple scheme employs the lowest precision for more than 57% of the time without losing accuracy. As can be seen in Figure Ib, the value of the cell state when applying our scheme (labeled 8/4 dynamic) follows the cell state of the 32-bits version closely. Note that in the peak regions, the error of *8/4 dynamic* is smaller than the error of the 4-bits version.

We implement our scheme on top of E-PUR [7], a recent accelerator highly optimized for LSTM inference. In order to support variable precision, the parallel dot product units in E-PUR are changed to multipliers that support mixed-precision. Then, we implement our dynamic precision selection scheme to decide the precision level for each element of the cell state on each time-step. Our scheme provides 1.46x speedup and 19.2% energy savings on average over the baseline, without affecting the accuracy.

The main focus of this paper is high-performance and energy-efficient LSTM inference. We claim the following contributions:

- We analyze the behavior of the LSTM cell state for a set of popular LSTM networks. We conclude that for time intervals where an element of the cell state is stable, it can be evaluated with lower precision without any impact on accuracy, whereas peaks require higher precision to prevent accuracy loss.
- We propose a novel mechanism that uses the cell state in LSTM cells to dynamically select the appropriate precision for each time-step and each cell element. Our scheme selects the lowest precision for 57% of the time.
- We implement our technique on top of E-PUR, a state-of-the-art accelerator for LSTM inference. Our system improves performance by 1.46x and energy consumption by 19.2% without loosing accuracy, while introducing an area overhead of less than 8%.

II. Background

A. Recurrent Neural Networks

A Recurrent Neural Network (RNN) is a state-of-the-art machine learning algorithm that is very successful in sequence-to-sequence problems such as machine translation [1] and speech recognition [12]. RNNs include loops that allow them to have more context information enabling them to make better predictions. Also, since they are executed recurrently for each input sequence element, they can handle problems with variable input and/or output sequence length. Capturing long term dependencies is a challenging task for basic RNNs (Vanilla RNNs) because the information tends to dilute over time. The Long Short Term Memory (LSTM) [13] networks were proposed to solve this issue.

1) LSTM Cell: An LSTM network is composed of several LSTM cells stacked together to create a deep LSTM network. For each LSTM cell, the principal component is the cell state, which stores the context information from previous input sequence evaluations. Also, an LSTM cell uses four gates to modulate how the cell state is updated. Among these gates, the updater gate (g_t), whose computations are shown in Equation 3, modulates the amount of input information that is being considered a candidate to update the cell state (c_t). The input gate(i_t), shown in Equation 1, controls what information will be added to the cell state. Shown in Equation 2 is the forget gate used to determine what information will be deleted from the current cell state (c_{t-1}). Finally, the output gate (o_t), shown in Equation 5, decides

$$i_t = \sigma(W_{ix}x_t + W_{ih}h_{t-1} + b_i) \quad (1)$$

$$f_t = \sigma(W_{fx}x_t + W_{fh}h_{t-1} + b_f) \quad (2)$$

$$g_t = \phi(W_{gx}x_t + W_{gh}h_{t-1} + b_g) \quad (3)$$

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t \quad (4)$$

$$o_t = \sigma(W_{ox}x_t + W_{oh}h_{t-1} + b_o) \quad (5)$$

$$h_t = o_t \odot \phi(c_t) \quad (6)$$

Figure 2. LSTM cell computations. $\odot$, ϕ, and σ denote element-wise multiplication, hyperbolic tangent and sigmoid function respectively.

what information from the cell is emitted to create the cell output (h_t) for the current time-step.

Each gate has two types of connections: the forward and the recurrent connections. The forward connections operate on the input coming from a previous cell (x_t). In contrast, the recurrent connections operate on the input coming from the cell output in the previous time-step (h_{t-1}). Note that the output of each gate is a vector. Thus, we call the elements of this vector *neurons* for the sake of simplicity. In order to evaluate each of these neurons, an inner product between the weights for the forward connections and the input vector x_t is done. Then, the result is added to the inner product of the weights in the recurrent connections and the vector h_{t-1}. Finally, an activation function is applied, which is usually a sigmoid or a hyperbolic tangent.

Most of the storage requirements of LSTM cells are due to the weight matrices and the output sequences. Regarding computations, most of the execution time is due to the evaluation of the matrix-vector multiplications of the various gates.

In this work, we applied linear quantization to the weight matrices and the input vectors x_t and h_{t-1}, as it is commonly done to reduce storage and computing requirements. The activation functions are evaluated in FP32.

B. Linear Quantization

Linear quantization is a commonly used technique to reduce memory footprint and computational cost. The main idea consists of approximating a full precision value (y) to a value y_q that is computed using an integer index and a quantization step (q) as shown in the following equations:

$$q = \frac{\alpha}{2^{n-1}} \quad (7)$$

$$i_k = round(y/q) \quad (8)$$

$$y_q = q * i \quad (9)$$

where n is the bit width of the integer index i_k, e.g., 8 bits, and α is the maximum absolute value of y.

In the case of an LSTM gate, once the weights and inputs have been quantized, the computations for the output of a neuron (z_k) are done using integer arithmetic and the result is converted back to floating point as shown in Equation 10 and Equation 11:

$$z_i = \sum w_i * x_i \quad (10)$$

$$z_k = z_i * q_x * q_w \quad (11)$$

where w_i and x_i are the quantized index for each element of the weight and input vector. Moreover, q_x and q_w are the quantization steps for the weight and inputs respectively. Multiplications are performed using 8-bits multipliers while summations and accumulations are normally performed with a higher number of bits (e.g., 24 bits).

C. Multi-Precision Multipliers

Multi-precision multipliers are commonly employed on works that use mixed-precision for their computations. These multipliers perform multiplications in parallel [10] or serial [9]. Regarding multi-precision parallel multipliers, they can be configured for a low precision (i.e., 4-bits) or a high precision (i.e., 8-bits) mode. When operating on low precision mode, they either work on half of the cycles needed to complete the high-precision multiplication, or they can provide twice the throughput.

For Serial Inner Products units (SIPs) an inner product is computed by serially feeding the bits of one of the operands while the bits of the other are feed-in parallel. On a cycle, a SIP unit performs the element-wise multiplications between a vector with *1-bit* elements and a vector with *n-bits* elements. Then, the summations are performed by accumulating the partial products computed on each cycle.

One advantage of SIP units over multi-precision parallel multipliers is that they allow a finer granularity when setting the precision. Nonetheless, our proposal is independent of either multi-precision parallel multipliers or SIP units. Thus, we evaluated both of them.

III. Dynamic Precision Selection

In this section, we describe our scheme to dynamically adjust the bit-width used to encode and operate the LSTM networks. First, we discuss the main bottlenecks on state-of-the-art hardware accelerators for LSTM inference. Next, we present the key idea for our proposal. Finally, we describe the hardware implementation of our technique.

A. Motivation

LSTM cells are composed of four gates, each with two matrices containing the weights for the forward and the recurrent connections, respectively. Since these weight matrices tend to be quite large, most of the energy consumed by state-of-the-art hardware accelerators for LSTM inference is

Table I
ACCURACY FOR SEVERAL LSTM MODELS USING DIFFERENT SCHEMES. 8 BITS AND 4 BITS ARE CONFIGURATIONS THAT STATICALLY SET THE BIT-WIDTH TO 8 AND 4 BITS FOR ALL THE TIME-STEPS, RESPECTIVELY. 8/4 IS OUR SCHEME FOR DYNAMICALLY SELECTING THE PRECISION.

App Domain	FP32 Accuracy	8-bit Accuracy	4-bit Accuracy	8/4 Accuracy	Error in Stable	Error in Peaks
Image Description [2]	32.2 Bleu	32.2 Bleu	27 Bleu	32 Bleu	20.3%	75%
Speech Recognition [11]	23.82 WER	23.8 WER	26.48 WER	23.8 WER	19.5%	74%
Machine Translation [1]	26 Bleu	26 Bleu	22.1 Bleu	25.9 Bleu	23.3%	82%
Speech Recognition [12]	10.24 WER	10.24 WER	13.24 WER	10.25 WER	20.1%	78%

due to the static and dynamic energy consumed by the on-chip memories employed to store the weights and intermediate results. Not surprisingly, the energy consumption of these on-chip memories accounts for up to 80% of the total energy in state-of-art solutions for LSTM [7].

An effective way to decrease memory footprint and thus static and dynamic energy without affecting accuracy is using Linear Quantization. Usually, a static profiling of the network is done to determine the minimum precision that could be used to quantize an LSTM model without degrading its accuracy. A common approach is to set a fixed precision (i.e., 8 bits) for the whole network. However, while this solution covers the worst case, it ignores cases where a lower precision could be employed for a subset of computations without losing accuracy.

Table I shows the accuracy for several applications employing LSTM networks, evaluated using 8 and 4 bits, and a mix of both. Regarding the mixed-precision, for each neuron (e.g., inputs and weights), we dynamically set the precision to 8 or 4 bits as described later in Section III-B. As it can be seen, when worst-case bit-width for the whole network is assumed, and 8-bit precision is employed, none of the models incurs in any accuracy loss. On the contrary, using 4-bit precision results in a significant degradation in all the networks' accuracy. In Section V, we show that for all the applications in Table I, more than 40% of computations can be done using 4-bits without incurring in any accuracy loss. Consequently, assuming that the required hardware is provided, the execution time can be reduced significantly. Also, energy consumption could be decreased since only half of the information has to be fetched when computing with 4 bits. We described our hardware solution to support mixed-precision execution in Section III-C1.

A primary challenge to dynamically change the precision is deciding when to use a high or low precision. In this work, we propose to use the state of the LSTM cell as an indicator of the required precision. Specifically, we aim to use high precision (i.e., 8 bits) for evaluations performed when a cell state element is in a peak and low precision (i.e., 4 bits) for the rest.

Table I shows the cell state error in the peaks and stable regions for several LSTM applications. In this regard, the cell state error is computed as the relative difference between the cell state computed in full-precision, and the cell state evaluated using 4-bits and 8-bits. We observe that when the cell state is stable (i.e., is changing slowly) the cell state

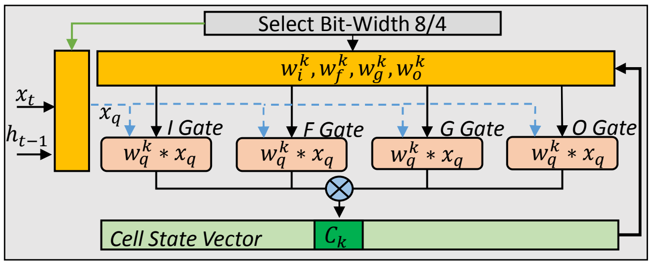


Figure 3. Relationship among the neurons on each gate and the elements of the cell state. The value C_k of the cell state is computed using the outputs of the k^{th} neuron in each gate. Then, a precision is chosen based on the evolution of C_k. Finally, using the selected bit-width, x_t and w^k are quantized to x_q and w_q^k, respectively.

error is less than 25% for all the networks. On the contrary, when the cell state is changing fast (i.e., in a peak) the cell state error tends to be very large (i.e., more than 70%), which largely degrades the model accuracy. This behavior is illustrated in Figure I, as described in SectionI.

As it can be seen in Table I, using 4-bits severely affect the final model accuracy. Therefore, introducing a larger error into the cell state results in a more substantial accuracy loss. Consequently, aiming to reduce the cell state error, we propose to evaluate the peaks using high precision while performing the computations outside the peaks using a lower bit-width. To exploit this observation, we design a scheme that monitors the evolution of the cell state at runtime for each element and selects high precision during peaks and low precision for stable phases. For this work, we use 8 bits for the high precision since it provides zero accuracy loss for all tested LSTM networks. On the other hand, we use 4 bits for the lower precision, which would have a significant loss in accuracy if it was used for all the time-steps. In the following sections, we detail this scheme and describe its hardware implementation on top of a state-of-the-art accelerator.

B. Overview

The main idea of our proposal is to set the precision at each time-step of execution for the input vectors x_t and h_{t-1} and their corresponding weights for each single element of the cell state individually. For a given LSTM cell, the k^{th} element of the cell state vector is computed using a combination of the output value of the k^{th} neuron on each gate. We refer to these four neurons simply as neuron n_k of the LSTM cell and set the precision for the four of them in tandem, since all of them are associated with the same

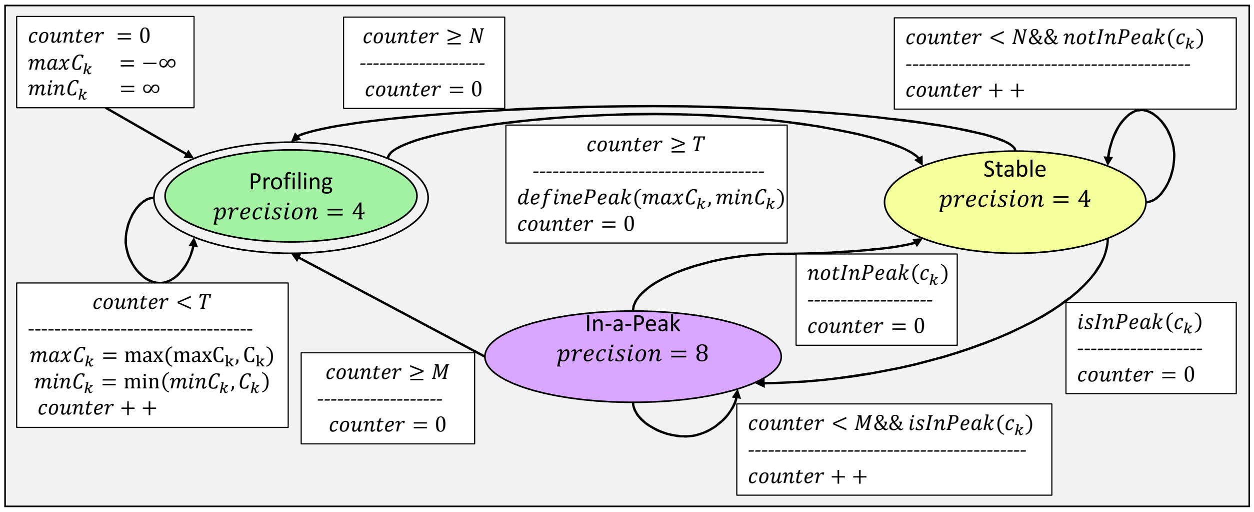


Figure 4. State machine employed to dynamically select the precision for an element c_k of the LSTM cell state.

$$r = maxC_k - minC_k \tag{12}$$

$$upperLimit = r + r * \beta \tag{13}$$

$$lowerLimit = r - r * \beta \tag{14}$$

$$isInPeak = lowerLimit \leq c_k \leq upperLimit \tag{15}$$

Figure 5. Positive and negative peak region definition for the cell state of a given neuron (i.e., n_k).

element of the cell state. This relationship is shown in Figure 3.

To determine when an element c_k of the cell state is on a peak, we employ the state machine depicted in Figure 4. Then, to track the evolution of c_k, we divide the process into three phases. First, the system starts in a *profiling state* that samples c_k for a certain number of time-steps. This profiling is done to determine the peak characteristics of c_k. Then, we have the *stable state* that indicates that c_k has had a stable value for the previously evaluated time-steps. Finally, the *in-a-peak state* tracks when c_k is in a peak.

As shown in Figure 4, the *profiling state* is performed for T time-steps. On each profiling step, we keep track of the maximum and minimum value of c_k. Note that the profiling is done using low precision because we assume that while profiling the cell state is inside a stable region. Finally, after T time-steps, we use the maximum and minimum value of c_k to set the limit values that define when a peak begins or ends, and then we move to the *stable state*.

The system remains in the *stable state* until a peak is detected. A peak is found using the values $minC_k$ and $maxC_k$, obtained previously in the profiling state, as shown in Figure 5. To determine that the value of c_k has entered a peak, we require that it exceeds the $minC_k$ and $maxC_k$ found during the profiling stage by a given margin to increase the detection's confidence. To this end, we use the parameter β in Equation 13 and Equation 14 to establish the upper and lower thresholds. If the c_k value in the cell state exceeds one of these thresholds, a peak is detected, and the system transitions to the *in-peak-state* to use high precision. It remains in this state until we detect that c_k is no longer in a peak using Equation 15. If the end of the peak is detected, we move to the *stable state* to switch back to low precision, as the value of the cell state has entered a stable region.

If the system stays in a peak for a large number of time steps (e.g., M in Figure 4), the profiling stage is triggered again. Note that this profiling is required since c_k may become stable at a value outside the thresholds of the original profiling. In this case, our scheme would stay indefinitely in the high precision state if profiling is not repeated to set new upper and lower thresholds. In other words, the information of the initial profiling may become outdated, since the range of values of the cell state may shift over time. In contrast, the system may be stuck in the *stable state* when the range of the cell state's values become narrower over time, as they will never exceed the minimum and maximum thresholds set in the initial profiling. To prevent this issue, we force a profiling stage when the system stays in the *stable state* for more than N time-steps. By doing this, we take into consideration newer values of the cell state and more adequate thresholds are set.

Our overall scheme for dynamic precision selection is summarized in Figure 6. Considering an input sequence with elements x_0 to x_{n-1}, for a given cell state element n_k, the scheme works as follows. First, the value c_k of the element n_k is computed using low precision and the *profiling state* is executed for T time-steps, marked as 1) in Figure 6. Then, after the profiling stage, the system moves to the *stable state* and performs all computations associated with n_k using low precision. Then, we detect that c_k is smaller than its previously profiled lower threshold and, thus, the

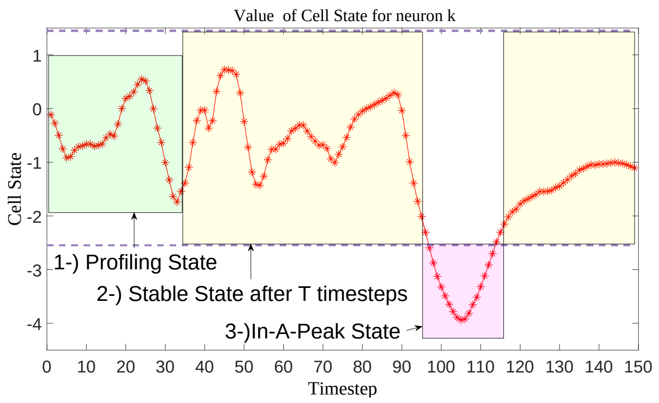


Figure 6. Evolution of the LSTM cell state for a given neuron. At each time-step, its stability is tracked to decide the precision for the next time-step. Dash lines (- -) represents the lower and upper limits that define the stable and peak region after the profiling is completed.

system changes to the *in-a-peak state*, as seen in Figure 6. Next, it stays in this state until the value of c_k comes back to its previously profiled range and, then, it switches back to *stable state*, where it waits for the occurrence of another peak or the triggering of another profiling stage. Note that this process is performed for each element in the LSTM cell state individually and, hence, our system may select a different precision for several neurons in the same time-step.

1) Supporting GRUs: To show the broad applicability of our dynamic precision scheme, we evaluated it on a Gated Recurring Unit (GRU) network. GRU cells represent a widely used alternative to LSTM networks. A GRU cell includes gates to control the flow of information inside and out of the cell. However, GRU cells do not include a cell state as LSTMs do. Nevertheless, we observed that we could track the evolution of the cell output (h_t) to change the precision dynamically effectively. In GRU networks, h_t output represents the feedback information that is recirculated in the cell, and its function is similar to the cell state in LSTMs. Therefore, we apply our scheme to GRU networks as we do for LSTMs (see Figure 4), but we dynamically set the precision by tracking the stability of the GRU cell output (h_t) in different time-steps.

2) State Machine Parameters: The state machine for dynamic precision selection, shown in Figure 4, requires three different parameters: M and N control the maximum number of time-steps that the system may remain in states *In-a-Peak* and *Stable* respectively before triggering the profiling. On the contrary, β is used to set the upper and lower thresholds to decide whether the value of the cell state is inside a peak. We performed a design space exploration for these parameters and found values that provide good results across the four LSTM networks used. Therefore, these parameters do not have to be manually tuned for each new LSTM network, as we empirically determined that the values shown later in Table III provide excellent results for a wide variety of networks. Furthermore, to prove that these values generalize well for new unseen inputs, we performed the design space exploration using the training datasets, whereas the evaluation of the technique is performed using the test datasets.

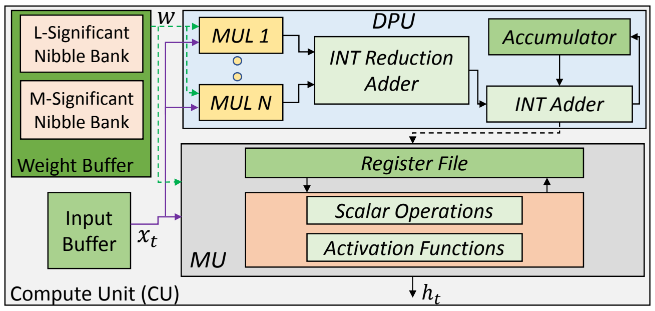


Figure 7. Compute Unit (CU). Multi-precision multipliers or SIP units are included to support variable precision.

C. Hardware Implementation

We implement our dynamic precision selection scheme on top of E-PUR [7], a state-of-the-art low power accelerator for LSTM networks. E-PUR consists of several computational units that evaluate the four different gates in an LSTM cell. Furthermore, it includes on-chip storage for weights and intermediate results. In E-PUR, computations are done using 8-bit parallel multipliers. To support variable precision, we replace the 8-bit parallel multipliers by either multi-precision parallel multipliers or SIP units. In the next subsections, we describe the overall architecture of E-PUR and explain how it can be extended to implement our scheme for dynamic precision selection.

1) Hardware Baseline: Figure 7 shows the structure of a computational unit (CU) which is tailored to the evaluation of a gate in an LSTM cell. A CU is composed of a dot product unit (DPU), a Multi-Functional Unit (MU) and several buffers to store the weights and inputs. DPUs are used to compute the matrix-vector multiplications in the four gates. Moreover, MUs are used to evaluate activation functions and scalar operations using floating-point numbers. Also, they are used to quantize the cell output (h_{t-1}) and to convert the DPU output to floating point. Note that weights are quantized offline.

In E-PUR, all the gates are evaluated in parallel using a fixed-precision of 8-bits. For a given time-step (x_t) of an input sequence (X), the following steps are performed to compute the output vector (h_t). First, for each output element (i.e, n_k) of h_t, the input and weight vectors are split into *K* sub-vectors of size *N*. Then, two sub-vectors of size N are loaded from the input and weight buffers, respectively, and the dot product between them is computed by the DPU, which also accumulates the result. Next, the steps are repeated for the next k^{th} sub-vector, and its result is added to the previously accumulated partial dot product.

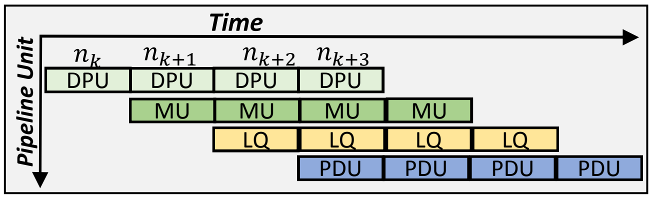


Figure 8. Computations are overlapped in the accelerator to hide latency.

This process is repeated until all K sub-vectors are computed and added together.

After a DPU computes its output value (y_t), it is sent to the MU, where it is converted to floating-point, and the activation function is computed. After each gate is evaluated, the cell state is computed and stored in the input buffer by the output gate. Also, the output value (h_t) is computed and quantized. Finally, the MU stores the final result in the on-chip memory for intermediate results. Note that the operations to compute the dot product, the activation function and to quantize the result are overlapped, as seen in Figure 8. Hence, once the DPU sends a result to the MU, it will continue with the next neuron. Similarly, once an activation function is computed, we proceed with the computation of the subsequent activation while applying the quantization steps to the previously computed activation. These steps are repeated until all the neurons in the LSTM cell are evaluated.

2) Mixed Precision with Multi-precision parallel multipliers: To support different precisions with parallel multipliers, we change the 8-bit parallel multipliers in the baseline implementation of E-PUR by multi-precision parallel multipliers (MPP). More specifically, we employ MPPs that operate in two precision modes high (i.e., 8-bits) and low (i.e., 4-bits). When working in high precision mode, an LSTM cell is evaluated as explained in the previous section. However, two multiplications are done per multiplier on each cycle when operating in low precision mode. Therefore, for each multiplier, we double the number of operands that are fed to them. Since the precision is lower (i.e., half), the bandwidth requirements do not increase.

3) Mixed Precision with SIP: To support mixed-precision using SIP units, the steps described in Section III-C1 are performed as follows. First, to maintain the same throughput as the baseline, 8 SIP units are included per DPU. Then, on each CU, eight sub-vectors of weights with N elements are fetched from the weight buffer and dispatched to each SIP unit. Furthermore, an 8*N-bit vector (v_0), corresponding to the most significant bit of each element in x_t, is fetched from the input buffer and dispatched to each SIP unit to perform the multiplication of v_0 with the corresponding weights. After this, each SIP accumulates its output. Next, this process is repeated until all the bits in x_t are multiplied and added together. Finally, the accumulated values on each SIP unit are added together, and the process is repeated for the remaining sub-vectors.

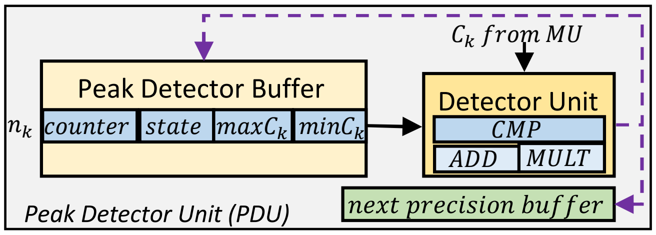


Figure 9. Structure of the Peak Detector Unit (PDU).

D. Dynamic Precision Selection

To set the precision for each neuron at each time-step, we extend E-PUR with a Peak Detector Unit (PDU), as shown in Figure 9. This unit tracks the evolution of each element in the LSTM cell state. The PDU contains a buffer to store the information needed by the state machine shown in Figure 4. Furthermore, it includes a detector unit that is employed to detect when the tracked cell state is inside or outside a peak, according to Equation 15. The PDU updates the state for a given element of the cell state after the MU computes its value and sets the precision to be used in the next time-step in the *next precision buffer*. Note that the computations performed by the PDU are overlapped with the MU evaluations in a pipelined manner, as shown in Figure 8. Also, the storage requirements of these buffers are small as described in Section IV.

1) Storing mixed-precision indexes : One major challenge to dynamically set the precision is storing the quantized integer indices for the weight matrix and input vectors in an energy-efficient manner. This challenge arises because, at each time-step, an index can be fetched in either low or high precision. Therefore, a mechanism that can store and fetch both indices efficiently is needed. One possible solution is that for a given floating-point value, the indices for high and low precision are store separately. The main drawback of this approach is that the memory footprint increases by 50%, which significantly increases the energy consumption of the system.

We address this challenge by only using one byte to store the high and low precision indices for a single weight. In this approach, for a given floating-point value, which is quantized in low and high precision, the most significant nibble of its high precision index can also be used as a low precision index. Therefore, the memory footprint of the baseline system is not increased. Furthermore, if the most and least significant nibbles of a given index are stored separately, only half of the memory accesses are needed to fetch the low precision indices, hence dramatically decreasing the system's dynamic energy consumption.

One drawback of using the most significant nibble of a high precision index as low precision index is that they are not always equal. Figure 10 shows a mapping of some

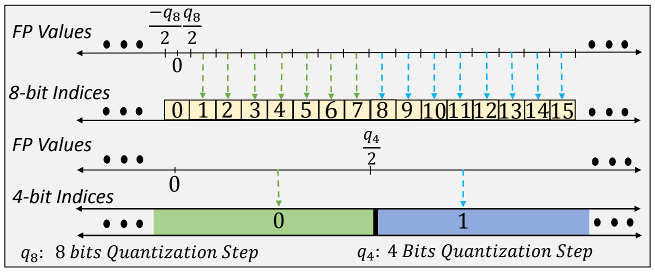


Figure 10. Linear Quantization of floating-point values for 8 and 4 bits. In some cases, using the most significant nibble of an 8-bit index to represent the corresponding 4-bit index yields an incorrect mapping.

floating-point numbers to integer indices using 8 bits (top of the figure) and 4 bits (bottom of the figure). As can be seen, using just the most significant nibble of the 8-bit index to obtain the 4-bit index is incorrect for half of the cases. As an example, a floating-point value mapping to index 11 using 8 bits is quantized as 1 when using 4 bits. However, using the most significant nibble of the 8-bit index would give an incorrect mapping of 0. A key observation is that values that are incorrectly mapped to its low precision counterpart always have a difference of one with the correct representation. Therefore, the proper index can be obtained by adding one to the most significant nibble of the corresponding high precision index. Furthermore, note that only the upper half of the high precision indices (indices 8 to 15) have an incorrect mapping. For the sake of simplicity, we only show in Figure 10, the first 16 indices and negatives values are omitted. However, the same issue would arise for the rest of the indexes.

In our scheme, after a given weight is quantized for low and high precision, the low precision index and the least significant nibble of the high precision index are stored in memory separately. Specifically, we employ the *l-significant nibble bank* and the *m-significant nibble bank* to store the low and high precision indices, respectively, as shown in Figure 7. Then, we use the low precision index to obtain the most significant nibble of its high precision counterparts by either using it directly or subtracting one from it when an incorrect mapping is found.

To evaluate an LSTM cell, in addition to the steps described in Section III-C1, we perform some extra tasks to set the appropriate precision for each neuron. First, for a given neuron (n_k), the bit-width to be used is read from the *next precision buffer* in the PDU. If low precision is selected, we only fetch the most significant nibble. Otherwise, if high precision is chosen, we fetch the bytes corresponding to each element in the weight vector of n_k. Then for each byte fetched, we use the least significant nibble to detect if the most significant nibble is correct or not. For incorrect cases, the most significant is adjusted by decreasing it by one unit. Once the values have been fetched and fixed, we send them to each of the DPUs as outlined in Section III-C1.

Regarding the input vector, we first fetch all its elements in high precision and low precision since the same input vector is used for all the neurons. Then, we proceed to feed them to each DPU, depending on the selected bit-width.

Finally, once we compute the cell state in the MU, it is sent to the PDU to determine the precision to be used on element n_i in the next time-step. Also, the output value is sent to the quantization unit, where it is quantized. Note that these operations are overlapped, as shown in Figure 8, and their latency is hidden by the computations in the DPU.

IV. Evaluation Methodology

We evaluate our proposal employing a diverse and representative set of modern LSTM networks, as shown in Table II. We include four LSTM networks from popular real-world applications: speech recognition [11], machine translation [1], image description [2], and sentiment classification [14]. Also, we include Deepspeech2 [12], which uses GRU cells. Our benchmarks largely differ in the number of layers and the dimensions of the cell size. For inference, we feed the networks with inputs from their respective test datasets, which include thousands of input sequences for each network. The length of the input sequences ranges from 20 time-steps to a few thousand. The models were implemented in Tensorflow [15]. Finally, a batch size of one is used.

To assess execution time and energy consumption, we employ a cycle-level simulator that models E-PUR [7] and the dynamic precision selection scheme described in Section III-D. The simulator provides the execution time and the activity factors of the different hardware components. The pipeline components are implemented in Verilog and synthesized using the Synopsys Design Compiler to obtain their energy consumption, area and delay. We use a typical process corner with a voltage of 0.78V. Also, we employ CACTI [16] to estimate the delay and energy consumption (static and dynamic) of the on-chip memories. Finally, to estimate the timing and energy consumption of main memory, we use MICRON's power model [17]. We model 4 GB of LPDDR4 DRAM. Regarding the frequency, we use the delays reported by Synopsys Design Compiler and CACTI to set it: we specified a clock frequency that allows most hardware structures to operate at one clock cycle. Regarding the Peak Detector Unit (PDU), note that the cell state is a vector of up to 2048 elements in our set of RNNs. Hence, the cost of tracking it is relatively cheap: 8KiB are needed assuming 4 bytes/element. In Table III, we show the configuration parameters for our system.

V. Experimental Results

This section presents the evaluation of our proposal. The baseline system is E-PUR using 8-bit parallel multipliers, labeled as E-PUR+PAR. We evaluate our scheme on top of E-PUR using SIP and parallel multi-precision multipliers. The

Table II
LSTM NETWORKS USED FOR THE EXPERIMENTS. 4-BITS USAGE REFERS TO THE PERCENTAGE OF THE EVALUATIONS PERFORMED AT LOW PRECISION.

Network	App Domain	Layers	Cell Size	4-bits Usage	Dataset	Time-step Range
IMDB Sentiment [14]	Sentiment Classification	1	128	100%	IMDB dataset	80
SHOW TELL [2]	Image Description	3	512	52%	MSCOCO	20-100
EESEN [11]	Speech Recognition	10	320	55%	Tedlium V1	60-2000
MNMT [1]	Machine Translation	8	1024	44%	WMT'15 En $\rightarrow$ Ge	30-100
DeepSpeech2 [12]	Speech Recognition	5	800	48%	LibriSpeech	100-1900

Table III
HARDWARE CONFIGURATION.

Parameter	Value
Technology	28 nm
Frequency	500 MHz
Intermediate Memory	6 MiB
Weight Buffer	2 MiB per CU
Input Buffer	8 KiB per CU
DPU Width	16 operations
Peak Detector Buffer	8 KiB
M	5% of time-steps
N	5% of time-steps
β	0.1

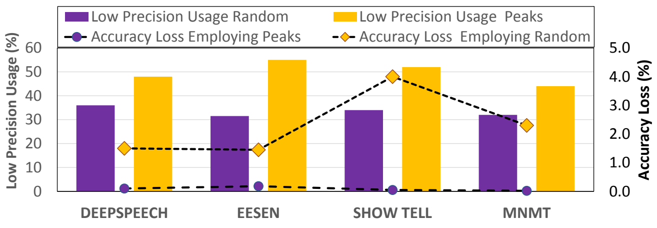


Figure 11. Comparison between our scheme ("Peaks") and a system that randomly chooses the evaluations done at low precision ("Random"). The random scheme produces a significant degradation in accuracy. Our scheme achieves higher low precision usage without any accuracy loss.

system implementing our scheme and SIP units is labeled as SIP+DYN. In contrast, the system with the multi-precision multipliers and our technique is marked as PAR+DYN. First, we present an evaluation of the effectiveness of our scheme. Second, we provide the performance and energy results. Finally, we evaluate our scheme using three levels of precision (8, 4, 2).

A. Effectiveness of tracking the cell state

In Figure 11, we compare our proposal with a scheme that randomly selects the precision level for each element of the cell state. On average, for the random system 34% of the evaluations are performed using 4 bits, whereas 66% are done using 8 bits. However, it produces a significant accuracy loss for all networks. In our proposal 49% of the evaluations are performed using 4 bits, without any accuracy loss. Therefore, tracking the stability of the LSTM cell state provides valuable information to select the precision.

B. Performance and Energy Improvements

Figure 12 shows that our scheme provides consistent and significant speedups, achieving 1.46x speedup on average. SIP+DYN and PAR+DYN exhibit the same speedup since

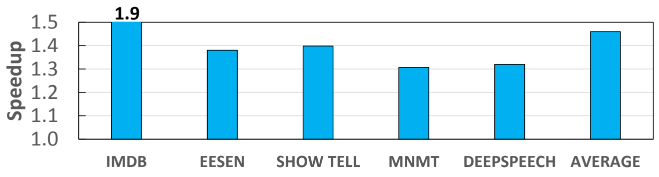


Figure 12. Speedups achieved by our scheme. Baseline configuration is E-PUR with 8-bit parallel multipliers.

both have the same throughput. The reduction in execution time is due to using 4-bits for more than 57% of the time. Note that the baseline employs 8 bits for all the computations to maintain the accuracy. Furthermore, the smaller the bit-width the higher the performance of the E-PUR+DYN and E-PUR+SIP: switching from 8 bits to 4 bits doubles the performance of the dot product units. Hence, for time-steps and cell state elements where 4-bit precision is used (stable regions), the latency of the dot product is reduced by a factor of 2x. This represents around 60% of the evaluations for our benchmarks. On the contrary, 40% of the evaluations are still done using 8 bits to maintain accuracy (peaks regions of the cell state).

Our scheme's latency is largely hidden due to overlapping the DPU and MU computations. Note that most of the execution time is due to the dot product calculations to evaluate each *neuron* on each gate using either 4 or 8 bits. In both cases, the latency of the dot product calculations is larger than the latency of the MU and LQ units. Only a small execution time overhead is added when the evaluation of an LSTM cell starts (i.e., first neuron and first time-step). The IMDB network has a speedup of 1.99x since it can be evaluated entirely using 4 bits. EESEN, SHOWTELL, and NMT achieve a speedup of 1.38x, 1.40x, and 1.31x, respectively, since they require high precision for some of their evaluations.

Figure 13 shows the energy savings, including both static and dynamic energy. On average, the energy savings for PAR+DYN and SIP+DYN are 19.2% and 17%, respectively. These savings come from several sources. First, using 4 bits reduces the dynamic energy of the weight buffer, since we only fetch the least significant nibble of the weights and inputs when using low precision. Second, the energy of dot product computation is reduced when employing 4 bits, since the activity in the DPU is reduced. Finally, the speedups reported in Figure 12 provide a reduction in static

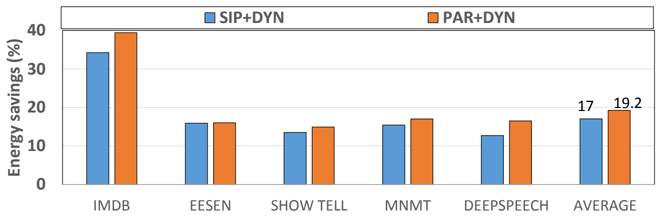


Figure 13. Energy savings achieved by our proposal compared to E-PUR-PAR. SIP+DYN and PAR+DYN refers to our scheme using SIP units and multi-precision parallel multipliers, respectively.

energy.

Figure 13 shows that using our scheme with parallel multipliers (DYN-PAR) provides 2% higher energy savings than the SIP implementation, since the SIP unit includes extra components to perform shifting and accumulation.

The LSTM networks EESEN and IMDB exhibit the most substantial energy savings: 15.9% and 34.2%, respectively, when using SIP+DYN. For PAR+DYN, their savings are 16% and 39.4%. For these networks, a large percentage of the computations are evaluated using low precision and, thus, the energy savings are significant. For the networks SHOWTELL and NMT evaluated on SIP+DYN, the energy savings are 13.5% and 15.4%, respectively, whereas when evaluated on PAR+DYN, their energy savings are 14.9% and 17%. Regarding DeepSpeech, the energy savings are 12.7% and 16.5% for SIP+DYN and PAR+DYN, respectively. Finally, the area overhead of our scheme is around 8%.

C. Multi-level Precision

Our scheme can be extended to support more than two levels of precision. To assess this, we evaluated EESEN and DeepSpeech using 8, 4 and 2 bits. A smaller range inside the stable region (shown in Figure 4) is defined, such that values of the cell state inside this new range are computed using 2 bits, otherwise they are done using 4 bits. For this scheme, 8.5% of the EESEN computations are performed using 2-bits, whereas, for DeepSpeech, 10% are done with 2-bits. Consequently, the energy consumption and execution time of these networks are improved by 3.3% and 1.02x, on average.

VI. Related Work

Hardware acceleration for LSTM networks has attracted the architectural community's attention in recent years. The Tensor Processing Unit (TPU) [6] is an ASIC that supports convolutional, fully-connected, and LSTM networks. TPU employs a fixed bit-width of 8 bits for weights and inputs. Our proposal is different as it selects the precision dynamically at runtime, using 4 bits for more than 57% of the time-steps.

Stripes [9] and BitFusion [10] support variable precision. Stripes uses SIP units and variable bit-width for the operands. BitFusion is a bit-flexible accelerator with an array of bit-level processing elements that can be dynamically merged or split to match the bit-width of individual DNN layers. Although Stripes and BitFusion offer more flexibility, each layer's bit-width is determined offline and kept constant during inference. We show that higher performance and energy efficiency can be achieved by dynamically selecting the precision based on the LSTM cell state. Our scheme can change the bit-width for every element of the cell state in every time-step.

Architectural techniques such as pruning, compression, and computation reuse are commonly employed during RNN inference. They are completely orthogonal to our scheme. ESE [18] exploits pruning and sparsity on FPGAs. C-LSTM [19] leverages structured compression, reducing LSTM model size while eliminating irregular computations. DeltaRNN [20] and the work in [21] exploit temporal coherency of the LSTM data to reuse computations and avoid redundant memory accesses. Work in [22] improves RNN energy efficiency by skipping computations, since previous calculations are memoized and reused in future evaluations. We focus on changing the bit-width at runtime, and computations are never skipped.

Regarding software-based RNN quantization, some proposals such as HitNet [23], Precision Gating (PG) [24], and Binary Forget and Input gate (BFIG) [25] have been introduced. These schemes apply very aggressive quantization statically. However, they introduce a significant accuracy loss, whereas our scheme does not degrade accuracy.

VII. Conclusions

In this paper, we present a novel scheme to dynamically select the precision for LSTM computation. We observe that the values of the LSTM cell state determine the required precision: time-steps where the cell state value changes rapidly (i.e., peaks) require higher precision to avoid errors, whereas time-steps where the cell state is relatively stable can be evaluated with a lower bit-width. We propose a scheme that monitors recent values of the cell state and selects the appropriate precision for the next time-step. Unlike previous schemes that fix the precision for each DNN layer offline, our system can change the precision for every cell state element and every time-step. We evaluate our proposal on top of E-PUR. Our scheme selects the lowest precision for more than 57% of the time without any accuracy loss, providing 1.46x speedup and 19.3% energy savings on average.

Acknowledgment

This work has been supported by the CoCoUnit ERC Advanced Grant of the EU's Horizon 2020 program (grant No 833057), the Spanish State Research Agency under grant TIN2016-75344-R (AEI/FEDER, EU), the ICREA Academia program, and the Fundación Carolina and PUCMM by a scholarship.

References

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

[2] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan, "Show and tell: Lessons learned from the 2015 mscoco image captioning challenge," *IEEE transactions on pattern analysis and machine intelligence*, vol. 39, no. 4, pp. 652–663, 2016.

[3] A. Graves, A. r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, May 2013, pp. 6645–6649.

[4] M. Zhang, S. Rajbhandari, W. Wang, and Y. He, "Deepcpu: Serving rnn-based deep learning models 10x faster," in *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, 2018, pp. 951–965.

[5] J. Appleyard, T. Kocisky, and P. Blunsom, "Optimizing performance of recurrent neural networks on gpus," *arXiv preprint arXiv:1604.01946*, 2016.

[6] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 1–12.

[7] F. Silfa, G. Dot, J.-M. Arnau, and A. González, "E-pur: an energy-efficient processing unit for recurrent neural networks," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, 2018, pp. 1–12.

[8] Y. Guan, Z. Yuan, G. Sun, and J. Cong, "Fpga-based accelerator for long short-term memory recurrent neural networks," in *Design Automation Conference (ASP-DAC), 2017 22nd Asia and South Pacific*. IEEE, 2017, pp. 629–634.

[9] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[10] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, V. Chandra, and H. Esmaeilzadeh, "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 764–775.

[11] Y. Miao, M. Gowayyed, and F. Metze, "Eesen: End-to-end speech recognition using deep rnn models and wfst-based decoding," in *Automatic Speech Recognition and Understanding (ASRU), 2015 IEEE Workshop on*. IEEE, 2015, pp. 167–174.

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

[13] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[14] A. M. Dai and Q. V. Le, "Semi-supervised sequence learning," *CoRR*, vol. abs/1511.01432, 2015. [Online]. Available: http://arxiv.org/abs/1511.01432

[15] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 2016, pp. 265–283.

[16] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP Laboratories*, pp. 22–31, 2009.

[17] Micron Inc., "TN-53-01: LPDDR4 System Power Calculator," https://www.micron.com/support/tools-and-utilities/power-calc.

[18] 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*. ACM, 2017, pp. 75–84.

[19] 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*. ACM, 2018, pp. 11–20.

[20] 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*. ACM, 2018, pp. 21–30.

[21] M. Riera, J.-M. Arnau, and A. González, "Computation reuse in dnns by exploiting input similarity," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*. IEEE Press, 2018, pp. 57–68.

[22] F. Silfa, G. Dot, J.-M. Arnau, and A. González, "Neuron-level fuzzy memoization in rnns," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 782–793.

[23] P. Wang, X. Xie, L. Deng, G. Li, D. Wang, and Y. Xie, "Hitnet: Hybrid ternary recurrent neural network," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, ser. NIPS'18. Red Hook, NY, USA: Curran Associates Inc., 2018, p. 602–612.

[24] Y. Zhang, R. Zhao, W. Hua, N. Xu, G. E. Suh, and Z. Zhang, "Precision gating: Improving neural network efficiency with dynamic dual-precision activations," in *International Conference on Learning Representations*, 2020.

[25] Z. Li, D. He, F. Tian, W. Chen, T. Qin, L. Wang, and T.-Y. Liu, "Towards binary-valued gates for robust lstm training," 2018.