E-BATCH: Energy-Efficient and High-Throughput RNN Batching

FRANYELL SILFA, Universitat Politècnica de Catalunya, Spain
JOSE MARIA ARNAU, Universitat Politècnica de Catalunya, Spain
ANTONIO GONZÁLEZ, Universitat Politècnica de Catalunya, Spain

Recurrent Neural Network (RNN) inference exhibits low hardware utilization due to the strict data dependencies across time-steps. Batching multiple requests can increase throughput. However, RNN batching requires a large amount of padding since the batched input sequences may vastly differ in length. Schemes that dynamically update the batch every few time-steps avoid padding. However, they require executing different RNN layers in a short timespan, decreasing energy efficiency. Hence, we propose E-BATCH, a low-latency and energy-efficient batching scheme tailored to RNN accelerators. It consists of a runtime system and effective hardware support. The runtime concatenates multiple sequences to create large batches, resulting in substantial energy savings. Furthermore, the accelerator notifies it when the evaluation of an input sequence is done. Hence, a new input sequence can be immediately added to a batch, thus largely reducing the amount of padding. E-BATCH dynamically controls the number of time-steps evaluated per batch to achieve the best trade-off between latency and energy efficiency for the given hardware platform. We evaluate E-BATCH on top of E-PUR and TPU. E-BATCH improves throughput by 1.8x and energy efficiency by 3.6x in E-PUR, whereas in TPU, it improves throughput by 2.1x and energy efficiency by 1.6x, over the state-of-the-art.

CCS Concepts: • **Computer systems organization** → **Neural networks**.

Additional Key Words and Phrases: Hardware Accelerators, Long Short Term Memory, Recurrent Neural Network, Batching

1 INTRODUCTION

Recurrent Neural Networks (RNNs) are a key technology for sequence-to-sequence applications such as machine translation [4] and speech recognition [3]. Their connections include feedback loops that allow them to remember information from previous executions and handle input and output sequences of variable length (i.e., number of time-steps). Modern RNNs feature a large number of parameters and, hence, their memory requirements are in the order of tens and even hundreds of megabytes [29]. Also, they impose severe data dependencies, and as a result, RNN inference exhibits a limited amount of parallelism. Not surprisingly, state-of-the-art systems such as TPU [15] or Brainwave [8] exhibit a low resource utilization for RNN inference: 18% and 3.5% respectively. On the other hand, other accelerators such as E-PUR [29] achieve nearly 100% utilization, but they are tailored to mobile environments. GPUs/CPUs using state-of-the-art RNN libraries also exhibit extremely low resource utilization. For example, the high-performance library cuDNN [6] shows an average utilization of 13.5% on an NVIDIA Titan V GPU for RNN inference.

Servers handling multiple requests (i.e., cloud services) from a large number of edge devices, such as smartphones, employ batching to increase parallelism and throughput. During inference, batching merges several requests and feeds them to the system simultaneously, so that all their computations are done in parallel. Therefore, the high energy cost of accessing the model parameters is shared by all the requests in a batch. Note that batching works best when the batched requests

Authors' addresses: Franyell Silfa, fsilfa@ac.upc.edu, Universitat Politècnica de Catalunya, Jordi Girona 1-3 , Barcelona, Spain, 08034; Jose Maria Arnau, jarnau@ac.upc.edu, Universitat Politècnica de Catalunya, Jordi Girona 1-3 , Barcelona, Spain, 08034; Antonio González, antonio@ac.upc.edu, Universitat Politècnica de Catalunya, Jordi Girona 1-3 , Barcelona, Spain, 08034.

The final publication is available at ACM via http://dx.doi.org/10.1145/3499757

are identical in length, i.e., their number of time-steps are the same. However, this is particularly difficult in RNNs since their input sequences usually have different number of time-steps. For instance, Deepspeech2 [3] has input sequences with several time-steps ranging from 60 to 1700 (Librispeech test set).

State-of-the-art deep learning systems [1, 23] handle this issue by padding the batched sequences such that their number of time-steps are identical to the number of time-steps of the longest sequence. The main drawback of this approach is that the latency of all the batched sequences increases since their evaluation cannot be completed until the longest sequence has been evaluated. Besides, energy is wasted performing computations on the extra added time-steps. Our experiments on E-PUR, an RNN accelerator, show that 30.2% of the energy consumption is due to padding, whereas the latency overhead is 28.5% on average for a set of RNNs.

For Deep RNN models, weight reuse is severely affected by the number of requests in a batch and the number of time-steps in the batched sequences. The reason is that, in order to evaluate a new layer of an RNN model, the weights are first brought to on-chip memory, hence evicting the weights of the previous layer. Henceforth, creating batches with short sequences incurs a large amount of weight swapping, and, as a consequence, energy consumption increases. This issue is evident in batching strategies such as Cellular Batching [9], where batches with short sequences are created, and not surprisingly, it is inefficient energy-wise. For instance, cellular batching on top of E-PUR consumes, on average, 4.5x more energy per request than sequence padding for DeepSpeech [3].

Motivated by the inefficiencies of current batching schemes, we propose E-BATCH, an RNN batching scheme that improves energy efficiency by avoiding padding and by increasing the temporal and spatial locality of the weights. In E-BATCH, sequences are allowed to join a batch while it is being evaluated. Also, when a new batch is created, all the available requests are distributed among all the hardware *processing lanes* (i.e., a processing element that can evaluate one or more sequences sequentially). More specifically, for a system with *n* processing lanes, we partition all the available requests among processing lanes in a way that the difference between the total number of time-steps evaluated by each lane is minimal. If the number of available requests is larger than the number of lanes, several requests are assigned to a given processing lane and are processed sequentially. Also, when the number of requests is too low, the system waits for some time before starting a new batch to increase its size. Furthermore, while a batch is being evaluated, specific new requests can join the batch to increase its efficiency. Note that assigning more than one request to a processing lane increases the length of the batched sequences. Furthermore, to meet Service-Level-Agreement (SLA), we limit the maximum number of time-steps in a given processing lane. Our experiments show that, on average, E-BATCH improves energy efficiency and throughput by 2.6x and 1.9x, respectively. For the rest of the paper, we refer to processing lanes simply as lanes.

To summarize, in this paper, we focus on batching for RNN inference. More specifically, batching for LSTM and GRU networks on DNN accelerators. Its main contributions are the following:

- We analyze the trade-off between energy and latency while batching RNN sequences. Also, we identify the excessive padding and the poor weight locality as the primary sources of inefficiencies in current RNN batching approaches.
- We propose a novel batching scheme that largely improves temporal and spatial locality of the weights and minimizes padding, which results in significant throughput and energy improvements.
- We implement our scheme on top of E-PUR and TPU, two state-of-the-art accelerators for RNNs. Our system improves energy efficiency by 3.6x/2.1x and throughput by 1.8x/1.6x for E-PUR/TPU, respectively, compared to a state-of-the-art RNN batching strategy.

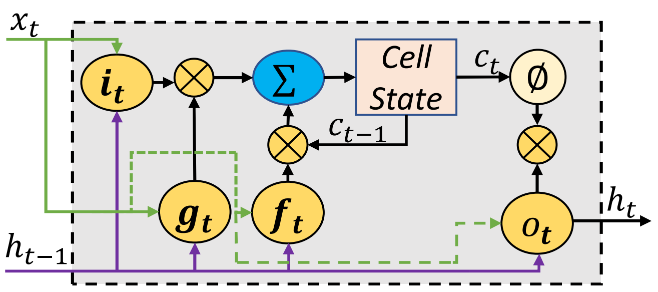

Fig. 1. Structure of an LSTM cell.

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

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

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

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

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

$$h_t = o_t \odot \phi(c_t) \tag{6}$$

Fig. 2. Computations of an LSTM cell. $\odot$, ϕ, and σ denote element-wise multiplication, hyperbolic tangent and sigmoid function respectively.

2 BACKGROUND

2.1 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a machine learning algorithm for sequence-to-sequence problems [3, 4, 19, 31]. RNNs include feedback loops that allow them to use past information from previous executions. Furthermore, RNNs are evaluated recurrently for each time-step of the input sequence. Therefore, they can handle input and output sequences of variable length.

Basic RNN (i.e., vanilla RNN) cannot capture long dependencies in the input sequence because information tends to dilute over time. For this reason, the Long Short Term Memory (LSTM) [13] and Gated Recurrent Unit (GRU) [7] architectures were proposed. Since they can exploit long and short-term dependencies, they represent the most successfully RNN architectures.

In an RNN, an input sequence is composed of N time-steps, i.e. $X = [x_1, x_2, ..., x_N]$. In an LSTM or GRU network, time-steps are processed sequentially, from x_1 to x_n. Note that the number of time-steps is normally different for each input sequence that is evaluated. Deep RNNs are created by stacking together several layers, where each of these layers consists of an LSTM or GRU cell.

2.1.1 *Basic structure of an LSTM Cell.*

Figure 1 shows an LSTM cell. The key component is the cell state (c_t, in Equation 4), that is computed as a function of four gates in charge of modulating the amount of information added or deleted from it on each time-step. The input gate (i_t, in Equation 1) modulates how much of the input information is added, whereas the forget gate (f_t, in Equation 2) controls how much information is removed. The updater gate (g_t, in Equation 3) modulates the amount of information that is considered as a candidate to update the cell state. Finally, the output

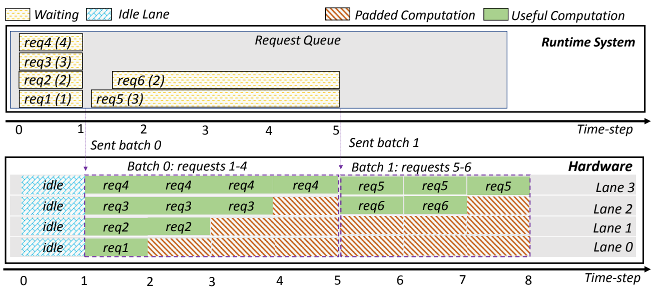

Fig. 3. Sequence Padding. Requests are shown in the request queue from their arrival time until they are dispatched to the hardware for evaluation. The number inside the parenthesis next to each queued request is the number of time-steps for that request. The batch size is 4.

gate (o_t, in Equation 5) controls how much information from the cell state is exposed as the cell output (h_t, in Equation 6).

The computations performed by an LSTM cell are shown in Figure 2. For each gate, there are two types of inputs: the current time-step (x_t) and the previous output (h_{t-1}). To compute gate's output, two matrix-vector multiplications are performed: one between the input x_t and W_x, and another between h_{t-1} and W_h. After these multiplications, an activation function is applied, which is typically a sigmoid or hyperbolic tangent. The output of each gate is a vector, and for the sake of simplicity, we call its elements *neurons*. Most of the execution time in an LSTM cell is due to the computation of the matrix-vector multiplication, whereas most of the energy consumption is due to accessing the weight matrices [29].

On the other hand, GRU cells are similarly to LSTM but they do not include a cell state. A GRU cell is composed of two gates: the update gate (z_t) and the reset gate (r_t). The update gate (z_t) modulates how much information from the previous output (h_{t-1}) will be carried over the current output (h_t), whereas the reset gate (r_t) controls how much information from the previous output is removed. In the rest of the paper, we refer to both LSTM and GRU cells as RNN cells.

2.2 Sequence Batching

Sequence Batching (we refer to it as batching for short) is a technique used to increase throughput. Inference machine learning systems handling multiple requests (e.g., data centers providing service to many users) batch various requests so that their computations are done in parallel. In this context, a batch is a set of one or more requests (i.e., input sequences). The number of batched requests (i.e., batch size) is usually limited by the number of hardware resources available in the system (e.g., number of processing lanes). Arriving requests are grouped into batches of size N, and batches are evaluated sequentially. Once a batch of requests is sent to the hardware, each of the batched requests is assigned to a processing lane. The evaluation of those requests is not completed until all of them are computed. Henceforth, batching tends to work best when the batched requests have the same length. However, this is an issue in RNNs since the number of time-steps on each input sequence may be significantly different. To mitigate this problem, the following strategies are employed.

2.2.1 Sequence Padding. Sequence Padding is used in TensorFlow [1] and PyTorch [23] to batch sequences with different number of time-steps. The sequence with the largest amount of time-steps (i.e., m) in a batch is first found. Then, the number of time-steps for the remaining sequences is increased to match the maximum length m, filling the extra time-steps with zeros.Note that the

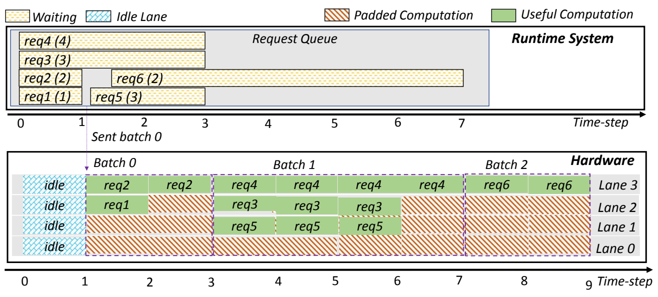

Fig. 4. Sequence Bucketing. For this example, the maximum difference in time-steps for requests that are batched together is 1.

larger the difference among batched input sequences, the larger the number of useless computations. Consider the batches created in Figure 3, where *requests 1-6* have 1, 2, 3, 4, 3, and 2 time-steps, respectively. In order to create a batch of size 4, *requests 1-3* are padded. Note that *request 1* completes its execution long before *request 4* (i.e. the longest request in the batch), but it is not returned to the user until *request 4* is finished. Furthermore, although *request 5* is available when *request 1* is already finished, its computation cannot start until the whole batch is computed.

2.2.2 Sequence Bucketing. Sequence Bucketing [16] is implemented on top of sequence padding in systems such as MXNet [5] and TensorFlow to reduce the amount of padding. To this end, different sequences are clustered together into a logical group, a.k.a. *bucket*, if they have similar length. Similarity is defined as the maximum difference in time-steps among all the sequences in any given *bucket*. In addition, it is constrained to be below a given threshold (i.e., the *bucket width*).

Only sequences from the same *bucket* can be batched together. Note that some sequences in a given batch may require padding. The maximum amount of time-steps padded is the *bucket width*. For instance, consider the example in Figure 4, assuming that *requests 1-6* have 1, 2, 4, 5, 3, and 2 time-steps respectively, and the *bucket width* is one. *Requests 1, 2, and 6* are assigned to a *bucket* whereas *request 3-5* are assigned to another *bucket*. Then, if batches are created using a batch size of 4, *requests 1-2* will be batched together. Similarly, *requests 3-5* will go into the same batch. Note that, although *request 6* is available when *requests 3-5* are batched, it is not included since it belongs to a different *bucket*. Batches are evaluated sequentially, and new requests are not allowed to join a given batch while it is being evaluated in the hardware. We refer to this optimization as bucketing.

2.2.3 Cellular Batching. Cellular Batching batches requests at the granularity of cells (i.e., a few time-steps) instead of whole sequences [9]. New requests are allowed to join a batch whose execution is ongoing. Also, completed requests are returned to the user immediately. Figure 5 shows how requests are scheduled in cellular batching. We assume a cell represents one time-step, a batch size of 4, and an LSTM model of one layer. The first time-steps of *requests 1-4* are batched together. Once the batch is evaluated, *request 1* is completed and returned to the user. After this, a new batch is created using the second time-step from *requests 2-4* and first time-step from *request 5*. Once this batch is evaluated, *request 2* is completed. Then, batching and evaluation continues for the remaining time-steps of *requests 3-6*. Since batches are created using a finer granularity, new requests can start execution as soon as the processing hardware becomes available, and completed requests are sent to the user immediately.

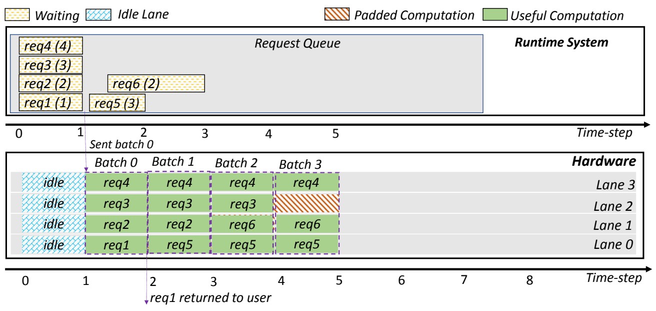

Fig. 5. Cellular Batching.

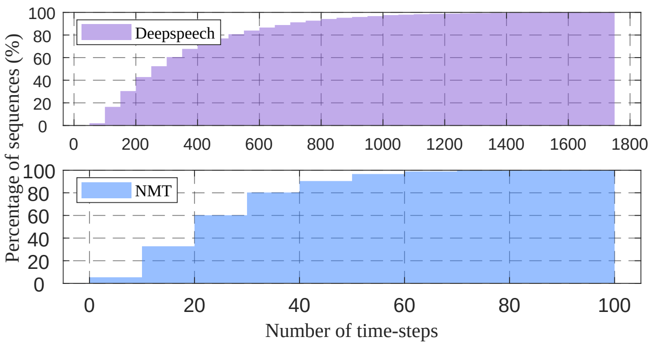

Fig. 6. Time-steps distribution for Deepspeech and NMT. The number of time-steps ranges from 10 to a few hundred.

3 SOURCES OF BATCHING INEFFICIENCIES IN RNNS

Both Bucketing and Cellular batching provide an improvement over sequence padding by reducing the amount of wasted computations. Cellular batching also improves latency by batching requests using a finer granularity. However, these solutions do not take into account energy consumption and the spatial and temporal locality of the weights for large RNN models (i.e., more than one layer), which is typically exploited in RNN accelerators. In this section, we identify the sources of inefficiencies in RNN batching and present a detailed analysis.

3.1 Number of Times-steps Variability

Figure 6 shows the cumulative distribution for the number of time-steps in the input sequences of two popular RNN models. As can be seen, both models have a wide range of sizes in the length of their input sequences. This variability severely affects batching systems that employ sequence padding or bucketing strategies since many useless computations are introduced. For instance, we have seen in our experiments that nearly 40% of the calculations evaluated by the hardware are unnecessary for sequence padding. On the contrary, when bucketing is applied, the number of wasteful computations evaluated decreases to nearly 5% since the number of time-steps among sequences that are batched together is quite similar. Regarding cellular batching, less than one percent of the computations executed are useless since batches are created and evaluated at a fine granularity (i.e., five time-steps).

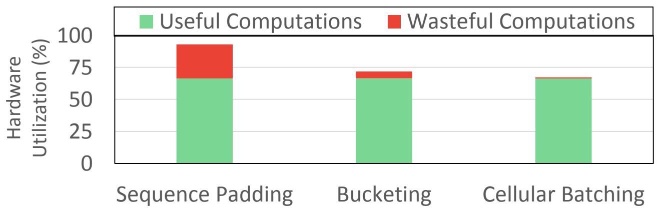

Fig. 7. Percentage of hardware utilization for useful and wasteful computations for Deepspeech.

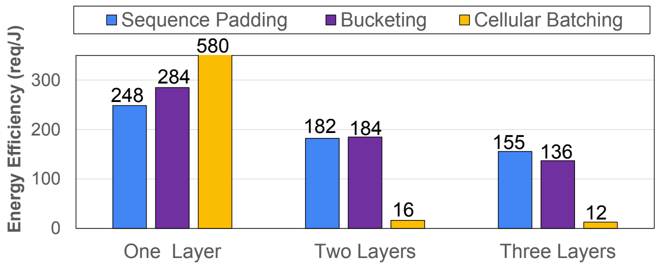

Fig. 8. Energy Efficiency of sequence padding and cellular batching for the GNMT [34] model on E-PUR. The system load is 100 request per second and a batch size of 64.

3.2 Low Hardware Resource Utilization

There are two leading causes of low hardware utilization of RNNs systems. One is due to padded sequences, whereas the other is due to a limited amount of requests.

Figure 7 shows a breakdown for the percentage of hardware utilization for useful and wasteful computations on a system under a moderate workload (i.e., 1000 requests per second). As can be seen, when employing sequence padding around 26% of the hardware utilization is due to wasteful computations. On the contrary, when bucketing is employed, only 5.2% of the hardware is utilized for wasteful calculations.

Some wasteful computations are performed by lanes that complete the evaluation of requests assigned to them early. For these cases, we could reduce the number of wasteful computations by sending available requests to those lanes since there are no data dependencies among requests. Note that, typically, when a batch is being evaluated in the hardware, to increase weight reuse, the same model parameters (i.e., weights) are used by each processing lane. Therefore, time-steps from an available request can only be sent to a lane when its weights are the same as the weights being used to compute the current set of batched requests. We provide more details about this issue in Section 5.

On the other hand, when the number of requests waiting to be processed is not enough to fill the available lanes, batches of small sizes are created. For this reason, some lanes will perform padded computations. Besides and more importantly, weights are swapped more frequently (i.e., for each new batch and layer evaluated), hence decreasing weight reuse and dramatically increasing energy consumption. For instance, for the model evaluated in Figure 7, when the workload is low (i.e., number of requests is smaller than the batch size), waiting for more new requests to arrive to create batches with a more considerable amount of requests, decreases energy consumption per request processed by 2.3x on average.

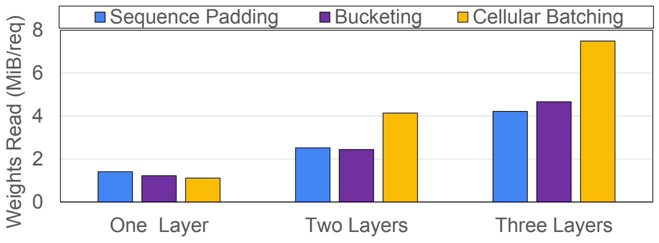

Fig. 9. Number of weight reads per request for sequence padding and cellular batching.

3.3 Poor Weight Locality

For RNN inference, one of the primary sources of energy consumption is the memory accesses to fetch the weights [29], accounting for up to 80% of overall energy consumption. Therefore, hardware accelerators for RNN include local on-chip memories to increase weight reuse. Not surprisingly, when batching RNN sequences, energy consumption due to memory access keeps being the dominant factor.

For Deep RNNs, on each batch evaluation, the weights must be fetched for each layer. Note that while evaluating an RNN layer, due to data dependencies, all the time-steps of a group of sequences in a batch are usually computed before proceeding with deeper layers. Hence, depending on the number of time-steps of the batched sequences, the overhead of accessing the weights will be more or less severe. This is particularly problematic when evaluating cellular batching on accelerators because input sequences are split into multiple batches. Moreover, batches, where the sequences on them have a small number of time-steps (e.g., 5), are created. Therefore, for RNN models with more than one layer, weights are fetched multiple times for each sequence on average.

Figure 8 shows the average energy consumed per request for each of the three previous batching strategies, evaluated on E-PUR [29]. As it can be seen, when evaluating an RNN model with only one layer, cellular batching is highly efficient since it avoids wasteful computations. Also, because it decreases waiting time, static energy consumption is improved. Note that for one layer model, weights are only loaded into on-chip memory only once. Despite cellular batching being highly efficient for a one-layer model, it is very inefficient for deeper models. As shown in Figure 8, when the model has more than one layer, the energy efficiency of padding and bucketing is 11x compared to cellular batching. The main issue with cellular batching for deep RNN models is that weights are swapped too frequently from on-chip memory. More specifically, batches with a small number of requests are evaluated, which decreases weight reuse. As illustrated in Figure 9, in the case of cellular batching, increasing the number of layers increases the number of bytes read per request rapidly. However, for bucketing and sequence padding, the increment in weight reads per request is less severe. Note that, for the three-layer model, sequence padding has better energy efficiency. The reason is that because of the*bucket size* constraint, batches with a small number of requests are created for sequence padding.

One approach to increase weight reuse is to use large batch sizes so that more requests are batched together. However, increasing the batch size will also increment the number of lanes needed to evaluate them. Another approach is to create batches in which several requests are concatenated before being sent to a processing lane, where they are evaluated sequentially. The main drawback of this approach is that it increases the number of time-steps evaluated sequentially per processing lane for a given batch, which increases the average latency. However, significant improvements in energy efficiency can be achieved (as shown later in Section 7).

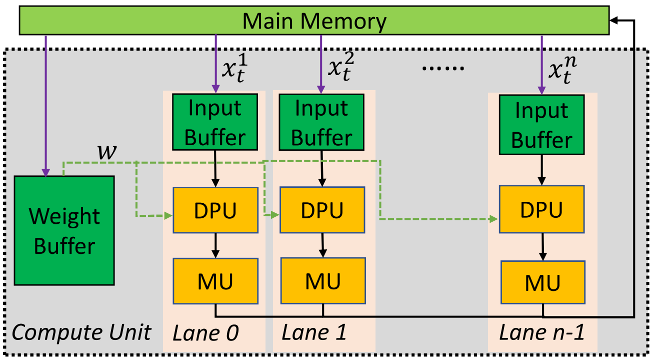

Fig. 10. Architecture of a Compute Unit (CU) in E-PUR. In order to support batching, several processing lanes are included.

To improve energy efficiency in RNN batching, we develop E-Batch, a novel batching scheme that allows RNN sequences to join a batch while it is being executed dynamically. Also, our proposal strives to create batches with a low amount of padding by employing a multi-way greedy partition algorithm. Furthermore, to increase weight reuse, E-Batch tries to evaluate many time-steps in each lane. Finally, to improve resource utilization, E-Batch implements a timeout to batch more requests during intervals when the number of requests arriving at the system is low.

We present the details of E-Batch in the following sections of the paper. We implemented our proposal on top of E-PUR and TPU, two state-of-the-art RNN accelerators. First, a brief description of the overall hardware baseline is presented. Next, we detail the extensions added to the baseline to support batching. Then, E-Batch is described. Finally, we discuss the runtime and hardware support required by our proposal.

4 RNN ACCELERATOR

The proposed RNN batching scheme, described in Section 5, can be implemented on top of any RNN accelerator. We first implement it on top of E-PUR, an energy-efficient accelerator for RNNs [29]. E-PUR consists of four computational units dedicated to evaluating the four gates in an LSTM cell. Moreover, it includes several on-chip memories to store the weights and intermediate results. In this work, we extend E-PUR to support batching. The following subsections provide further details on E-PUR's architecture and describe the required modifications.

To illustrate the general applicability of the proposed batching technique, we also implement it on top of a TPU-like architecture and provide experimental results in Section 7.3.

4.1 Architecture Overview

E-PUR consists of four Compute Units (CUs) whose overall structure is shown in Figure 10. The main components of each CU are a dot product unit (DPU) and a Multi-functional Unit (MU). The DPU is tailored to the computations of the matrix-vector multiplications between the input sequences (x_t and h_{t-1}) and the weights. On the other hand, the MU is used to compute activation functions (i.e., sigmoid and hyperbolic tanh) and scalar operations. Note that the computations in E-PUR are performed using either 8 or 16 bits.

In E-PUR, all gates are evaluated in parallel. 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 are added to the previously accumulated partial dot product. This process is repeated until all K sub-vectors are computed and added together. Once the output value n_k is computed, the DPU sends it to the MU, where bias and scalar calculations are performed. Finally, the MU computes the activation function and stores the result in the on-chip memory for intermediate results. Finally, these steps are repeated until all the elements of h_t are evaluated.

4.2 Supporting Batching

The base architecture of E-PUR does not support batching. Hence, two main modifications are done to support the evaluation of multiple input sequences in a batch. First, E-PUR includes an on-chip memory to store intermediate results, which saves accesses to main memory and, therefore, improves energy efficiency [29].

Intermediate results are generated after computing the output vector (h_t) and are used as input to the following time-step of execution and as input to the next layer. However, including an on-chip memory to store the intermediate results for all the time-steps during batching is unfeasible since the memory requirements will be huge (hundreds of megabytes). Note that the output for each time-step must be kept in memory until it is consumed by the next layer and, hence, intermediate results quickly become large when increasing batch size. Therefore, a vast amount of intermediate results are generated for large batch sizes (e.g. $time\_steps * batch\_size * model\_neurons$). Consequently, we decide to store intermediate results among RNN layers in the main memory. Note that intermediate results are only loaded once from the main memory when a new layer will be evaluated. Moreover, the output of the previous time-steps is kept on-chip since it is consumed when computing the next time-step.

Despite storing intermediate results in main memory, RNN batching improves weight locality largely since weights are fetched once and reused to compute multiple input sequences in parallel, reducing main memory accesses for fetching weights. This trade-off is highly lucrative since the memory footprint for the weights is typically much larger than the size of the intermediate results. For instance, on E-PUR, when sequence padding is employed with a batch size of 64, energy consumption and performance are improved by 3.15x and 36x on average, respectively.

The second extension to the baseline architecture adds extra DPUs and MUs on each CU, as shown in Figure 10. On each CU, we include N DPUs and MUs such that we have the necessary hardware needed to evaluate N input sequences in parallel (for a batch of size N). Hence, a lane in E-PUR consists of a DPU and a MU. Each lane includes its private on-chip input buffer, whereas the weight buffer is shared among all the lanes.

After these modifications, the evaluation of a batch is performed as follows. First, once a batch of sequences is created by the runtime (discussed in Section 5.7), it is sent to E-PUR. In the accelerator, each input sequence in the batch is distributed to a lane. Then, on each lane, the evaluation of a sequence is done, as explained in Section 4.1. Note that since all the lanes share the weight buffer, weights are broadcast to each of them where they are multiplied by their corresponding input sequence. Finally, for each lane, once the output h_t of an input sequence is computed, the result is stored in the main memory.

5 E-BATCH

In Section 3, we described the primary sources of inefficiencies in RNN batching. We propose E-Batch, a novel RNN batching scheme that tries to optimize both latency and energy consumption to mitigate these issues. First, we present an overview of the execution flow of E-Batch. Then, we

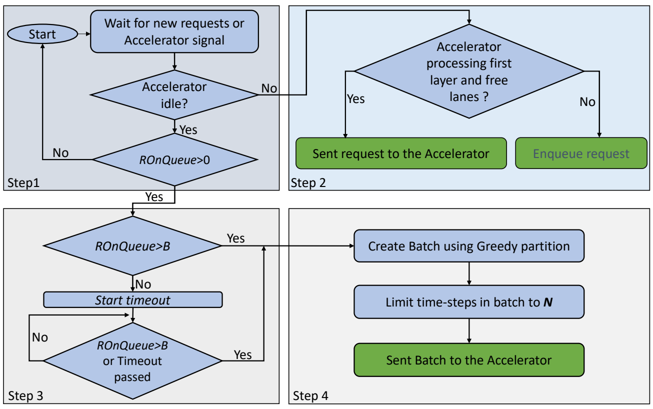
Start
Wait for new requests or Accelerator signal
Accelerator idle?
No
Yes
No
ROnQueue>0
Step1
Yes
Accelerator processing first layer and free lanes ?
Yes
No
Sent request to the Accelerator
Enqueue request
Step 2
ROnQueue>B
Yes
No
Start timeout
No
ROnQueue>B or Timeout passed
Yes
Step 3
Create Batch using Greedy partition
Limit time-steps in batch to N
Sent Batch to the Accelerator
Step 4

Fig. 11. E-Batch batch creation flow. *ROnQueue* is the number of requests currently on queue waiting to be evaluated. *B* is the batch size (i.e., number of processing lanes)

describe the main optimizations included in E-Batch. Finally, the hardware and software support needed to run E-Batch is detailed.

5.1 Overview

In E-Batch, the creation of a batch is performed on the CPU by a runtime system, whereas the RNN computations are done in an RNN accelerator (E-PUR, TPU, ...). Figure 11 shows an overview of E-batch execution flow. Typically, the runtime is waiting for a new request or a notification from the accelerator. In this case, step one is performed. For step one, when a new request arrives at the system, the runtime will check if the accelerator is idle or busy. If the accelerator is busy, the runtime advances to step 2. Otherwise, it verifies if there are some requests on queue and proceeds to step 3.

In step 2, the accelerator tries to add time-steps from the new request to the current batch. However, this is only possible when the current batch is being evaluated for the first layer (as explained in 5.3), and the accelerator has available lanes. In this case, some of the time-steps from the new request are included with the current batch.

When a new request arrives at the system and the accelerator is idle, the runtime moves to step 3, where it tries to create a new batch. The runtime proceeds to step 4 when the number of requests on the queue exceeds the number of processing lanes. Otherwise, a time-out is started to wait for more requests. This is done to create batches with many time-steps, thus, increasing weight reuse, as described in 5.2.

In step 4, the runtime will make a new batch using all the available requests. For this step, a greedy partition algorithm is employed to distribute all the time-steps of all the queued requests among the available processing lanes. After this, the runtime will limit the number of time-steps per processing lane to a value *N*. This is done to decrease the latency of processing a batch. By changing the value of *N*, we can trade latency for energy efficiency, as explained in 5.4. Finally, once a batch is created, it is sent to the accelerator, where it is evaluated for all the network layers.

5.2 Increasing Weight Reuse

In contrast to other solutions, in our scheme batches are created considering all the available requests. Since our main target is energy efficiency, E-Batch increases weight reuse by using all the available requests. When the number of requests being batched is larger than the number of available lanes, multiple requests are assigned to a given lane and are executed sequentially. In this case, the number of time-steps for a given lane is defined as the sum of the time-steps of all the requests assigned to that lane.

Since the number of time-steps for each request tends to be different, the number of time-steps on each lane also differs. Hence, to decrease the number of padded computations (see Section 3.1), our scheme tries to minimize the difference in time-steps between each lane by distributing the available requests among lanes in a way that their number of time-steps are as similar as possible. Note that this problem is an instance of the Multi-Way Number partitioning problem, which is known to be NP-Complete [17]. We employ a greedy heuristic to assign each request to a given lane. For example, given the requests with the following number of time-steps (4,5,6,8,7) and a system with two lanes, we first sort all the available requests in descending order by their number of time-steps: (8,7,6,5,4). Next, we process the requests sequentially in descending order, assigning each request to the lane with the minimum amount of time-steps (e.g., 8 to the first lane, 7 to second, 6 to the second, etc on). In the example above, the first lane receives requests with time-steps (8,5,4), and the second lane gets requests with time-steps (7,6).

Greedy partitioning does not always yield an optimal solution [17]. As a result, some sequences may need to be padded. In this case, we use hardware assistance to either clock gate the lanes that will perform the padded sequences' evaluations or assign new requests to these lanes. To this end, when a lane finishes the evaluation of all the time-steps assigned to it, E-PUR queries the runtime for a new request and, assuming that a request is available, the runtime sends it to E-PUR where its evaluation starts immediately to avoid padding. On the contrary, if there is no request available, the lane is clock gated. Due to data dependencies, time-steps must be evaluated sequentially and cannot be distributed to multiple lanes.

5.3 Decreasing Padding

In order to reduce the impact of padding and improve throughput, E-Batch allows new requests to join a batch, but only while the first RNN layer is being processed. If a new request arrives at the server while the accelerator is processing the first (i.e., input) layer of an RNN, E-batch's runtime system tries to add the request to the current batch to reduce padded time-steps. Once the computation of the first layer for a given batch is finished, the batch is locked, and it cannot be modified for the subsequent RNN layers. The rationale behind this decision is to guarantee that all the requests in a batch belong to the same RNN layer and are processed simultaneously to maximize weight reuse, which results in significant energy savings, as shown in Section 7.

5.4 Trading Latency for Energy-Efficiency

In E-Batch, the number of time-steps processed by a given lane is limited to a threshold (i.e., N). By using a threshold, we can trade latency for energy consumption. Batching with a small number of time-steps per lane decreases latency but incurs in a large number of weight swaps, which decreases weight locality. On the contrary, batching with many time-steps per lane increases weight reuse, which significantly reduces energy consumption at the expense of an increase in latency..

When a batch is being evaluated for the first layer of the RNN, E-Batch proceeds to compute the next layer after N time-steps have been evaluated per lane. If the amount of time-steps in a lane is larger than N, the remaining time-steps for those requests assigned to that lane are evaluated in a future batch. In other words, the evaluation of some requests is split among different batches to allow requests to progress through the subsequent layers in the RNN and reduce latency.

When a new batch needs to be started, and the number of requests available is smaller than the batch size, the runtime waits for T milli-seconds to increase hardware utilization and weight reuse (i.e., more requests are batched together) at the expense of some penalty in latency. Furthermore,

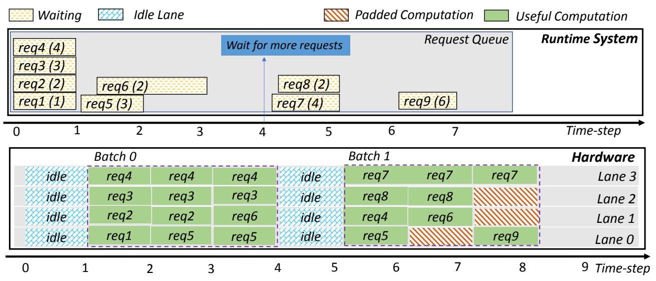

Fig. 12. At time-steps 2 and 3, *lane 0* and *1* become idle and the evaluation of *requests 5 and 6* starts. The maximum amount of time-steps per lane (N) is set to 3.

if a batch is being executed and a new request arrives while the first layer of the RNN is being processed, the runtime checks for any idle lane, and if there is any, the new request is sent to the accelerator where it is assigned to that lane. Note that, to evaluate layers after the first one is computed, the previous layer's result is needed. Therefore, even though some lanes could be idle, if a new request arrives when a batch is being evaluated for any subsequent layer, it cannot be assigned to any of those idle lanes. Also, only requests for the same RNN model are batched together.

5.5 Execution Flow

Figure 12 illustrates the execution flow of E-Batch. Like the previous examples, a batch size of 4 is used, and an LSTM model with one layer. Furthermore, a value of 3 time-steps is used for N, whereas T is set to two time-steps. In the beginning, *requests 1-4* are in the waiting queue. Since the number of requests in the queue is four, a new batch (*batch 0*) is created by the runtime, assigning each request in the batch to a given lane using greedy partitioning. Afterwards, *batch 0* is sent to the accelerator in time-step 1. At time-step 2, the execution of *request 1* is completed. Hence, the accelerator sends a signal to the runtime to indicate that *lane 0* is idle. Thus, *request 5* is assigned to *lane 0* since it is the oldest request in the waiting queue. Similarly, *request 6* is assigned to *lane 1* after *request 2* is finished. At time-step 4, the execution of *batch 0* is completed since N is 3. At this point, *requests 1-2* are completed, and the RNN output is sent back to the client. However, for *requests 4-6*, only 3, 2, and 1 time-steps have been evaluated. Therefore, the remaining time-steps will be computed in a future batch. After the evaluation of *batch 0*, only *requests 4-6* are in the queue. Hence, since the number of requests on the queue is smaller than the number of lanes, the runtime waits for T time-steps or enough requests to fill all the lanes.

As seen in Figure 12, at time-step 5 *request 4-8* are in queue, hence *batch 1* is created by the runtime. In this case, since the number of requests in the queue is greater than the number of lanes, *requests 4 and 6* are assigned to lane 1. Finally, *batch 1* is sent to the accelerator, where its evaluation is performed in manner similar to *batch 0.*

Figure 13 shows the behavior of our scheme for RNN models with more than one layer. Similar to the previous example, for time-step 1, *requests 1-4* are batched and sent to the accelerator. Furthermore, at time-step 2, *request 1* is completed and *request 5* is assigned to *lane 0.* Since N is 3, after three time-steps have been evaluated, we proceed with the evaluation of the next layer. Note that *requests 6-7* are available at time-step four and that lane 1 is idle after time-step 6. However, as mentioned before, new requests cannot join a batch after the first layer's evaluation is completed. Hence, *requests 6-7* wait until the evaluation of *batch 0* finishes for the second layer. After *batch 0*

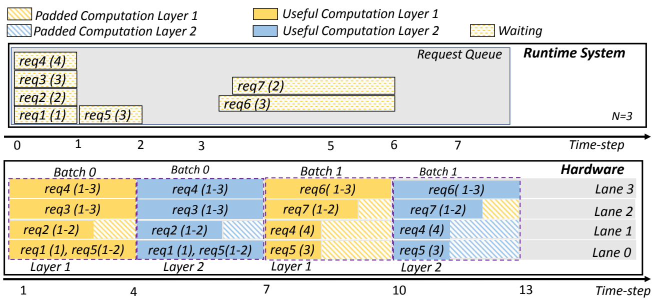

Fig. 13. Evaluation of an RNN with two LSTM layers using E-Batch. Batch 0 is evaluated for the first and second layer, before a new batch is created. *req 5* arrives before finishing the evaluation of the first layer and, hence, it joins Batch 0 immediately. On the contrary, *req 6-7* arrives after the evaluation of the first layer, so its evaluation is deferred until a new batch is created.

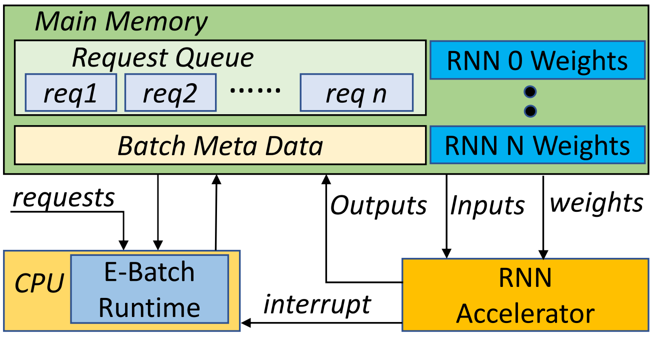

Fig. 14. Overview of E-Batch System Architecture.

is completed, the output results for *requests 1-2* are sent to the user, whereas a new batch is created using the remaining time-steps of *request 4-5* and the new requests *request 6-7*.

The overall architecture of E-Batch is shown in Figure 14. It is composed of a runtime system and an RNN accelerator. The runtime is in charge of creating and managing batches of requests, whereas evaluations are performed in the accelerator. In the following sub-sections, we describe in more detail these components.

5.6 Hardware support

In order to support E-Batch in E-PUR, the following modifications are done. First, we include an interrupt, which is used to signal the runtime when a lane becomes idle. Furthermore, we include a small buffer (i.e., *the request buffer*) to keep track of the lane where each request is being evaluated. Also, in *the request buffer*, we store the number of time-steps that have to be processed for each request. Finally, we include a register to store the parameter N (i.e., the maximum number of time-steps). Once a batch is received, the evaluation of the first layer works as follows. For each request in the batch, an entry in the *request buffer* is created and initialized with its meta-information. Then, when a time-step is evaluated, the number of time-steps processed for the active requests are updated in the *request buffer*. If all the time-steps for a given request are completed, we proceed with the next request in the lane. However, if there are no more requests in the lane, a signal is sent to runtime to indicate that a lane is idle so that a new request can join the current batch

Table 1. RNN Networks used for the experiments.

Network	App Domain	Cell Type	Layers	Cell Size	Dataset
DeepSpeech2 [3]	Speech Recognition	GRU	5	800	LibriSpeech
MNMT [4]	Machine Translation	LSTM	8	1024	WMT'15 En $\rightarrow$ Ge

being executed. After N time-steps have been evaluated, we continue with the computation of the subsequent layers. Finally, once the evaluation of a batch is completed, the number of time-steps evaluated for each request is sent back to the runtime.

5.7 Runtime support

The runtime is in charge of the management and creation of batches. It includes a queue where new requests arriving at the system are stored. Furthermore, for each request, the runtime tracks the number of time-steps that have been evaluated. Also, it knows whether the accelerator is processing or idle.

When the accelerator is idle, the system will create a new batch using all the available requests and employing the Greedy partitioning algorithm described in 5.1. Then, once a batch is created, for each batched request, the lane assigned to it and the number of time-steps that need to be evaluated are sent to the accelerator. Once the evaluation of a batch is completed, the number of time-steps evaluated are updated for all its requests. Moreover, requests that are completed (i.e., all its time-steps have been executed) are returned to the user. On the other hand, new requests and requests with time-steps pending for evaluation are batched together. If the available requests are not sufficient to create a complete batch, the runtime will wait until the batch is completed or T milli-seconds have passed.

5.8 Supporting E-Batch On a TPU-Like Architecture

TPU is a state-of-the-art accelerator for neural networks [15]. It is composed of a systolic array of processing elements (PEs) and on-chip memory for weights and activations. In order to evaluate LSTM models on TPU like architectures, an output stationary dataflow is employed [25, 27]. In this regard, for an RNN model with N neurons, neuron n_k will be mapped to all the PEs in column k of the array. Regarding the weights, they are mapped to the columns of the systolic array, whereas the input sequences are mapped to rows of the array (one input sequence per row). Thus, to evaluate n_k, its weights are streamed down through column k, whereas the elements of each input sequence are streamed through the rows. Note that the maximum batch size supported is equal to the number of rows in the array and that each of them will correspond to a *processing lane.* To support E-Batch in a TPU-like architecture, as done in E-PUR, we add an interrupt to signal the end of a sequence to the runtime and a *request buffer* as described in 5.6.

6 EVALUATION METHODOLOGY

To evaluate our proposal, we use the state-of-the-art LSTM networks shown in Table 1. They consist of RNNs for popular applications such as machine translation [34] and speech recognition [3]. In the case of DeepSpeech2, it has five layers, and the size of each cell is 800. Regarding the number of time-steps on its input sequence, it ranges from 10 to 1800. On the other hand, MNMT has eight layers and a cell with 1024 units. For this network, the number of time-steps ranges from 20 to 100. Note that our system's performance and energy consumption only depend on the characteristics of the input sequence.

Table 2. Hardware configuration.

E-PUR	
Parameter	**Value**
Technology	28 nm
Frequency	500 MHz
Weight Buffer	2 MiB per CU
Input Buffer	128 KiB per CU
DPU Width	64 operations
TPU-like accelerator	
Frequency	700 MHz
SRAM Buffer	24 MiB
Systolic Array PEs	128x128

To estimate performance and energy, we use a cycle-level simulator of E-PUR and TPU, using the configuration parameters in Table 2. For E-PUR, we used an in-house implementation of its architecture, described in [29]. On the other hand, we use SCALE-Sim [27] simulator to model a TPU-like architecture. These simulators model the execution time and dynamic accesses for each of the computational units on E-PUR and TPU. Also, the memory accesses to the on-chip and the main memory are modeled. For a given network, its configuration and the number of time-steps are fed to the simulator. Then, the simulator generates a trace that includes the timing and dynamic accesses for each computational unit. For a TPU-like architecture, we set the number of filters as the number of neurons in the model and the input features as the number of weights per neuron. The width and height of the filters are set to one, whereas the width of the input features is set to the batch size.

To estimate energy consumption, we modeled the different components of E-PUR and TPU in Verilog. We synthesized them using Synopsis Design Compiler, and to determine the static and dynamic energy consumption of on-chip memories, we employed CACTI [22]. Note that both timing and energy consumption are estimated by combining the synthesis, cacti, and the traces generated by the cycle simulator. As mentioned in Section 5.2, when a processing lane is not being used, it is clock gated. In this regard, we disable the clock for idle processing lanes. We follow a methodology similar to [2, 18, 32], and the clock signal of idle lanes is disabled at the beginning of the evaluation of a batch. Finally, for main memory, we used the MICRON power model [21] and modeled a 4GB DDR4 DRAM. For the modeled memory, the peak bandwidth is 25.6 GB/s, and the data rate is 3200 MT/s. Regarding the scheduling policy, we assumed FR-FCFS (i.e., First-ready, first-come-first-service). Also, the power model includes key timings parameters such as maximum and minimum clock cycle rate, minimum and average refresh-to-refresh cycle time, and the minimum activate-to-activate timing. The values of these parameters are detailed in the datasheet for the DDR4 memory [20], and we omit them for the sake of brevity. Our final evaluations include both the timing and energy consumption of transferring the weights and intermediate results to the main memory.

Regarding the runtime, we implemented a system that employs the timing estimation and status of the accelerator' simulator (E-PUR or TPU) to batch new requests. For our experiments, we simulate several hours of execution. To this end, the train/test datasets are not sufficient since they would be processed in a short period. Aiming to generate more requests to be processed by our system, we analyzed the distribution of the number of time-steps for each input sequence in the train and test set of the RNN models in the benchmarks. In this regard, we employ the following

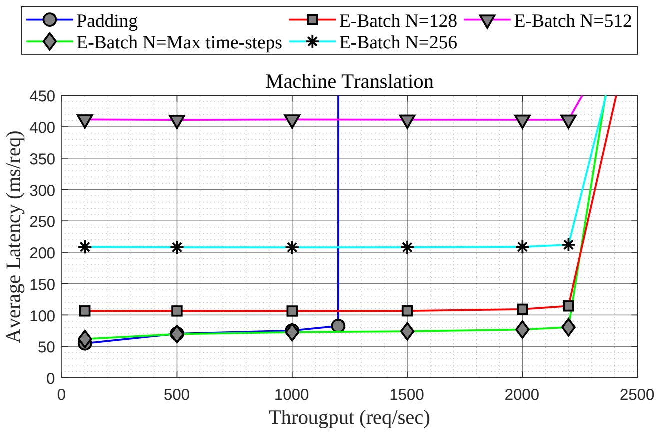

Fig. 15. Average Latency vs Throughput for Machine Translation [4] using a batch size of 64 on E-PUR.

methodology to generate new requests where the number of time-steps follows a distribution similar to the datasets. First, we created a histogram for the time-step distribution of each dataset. From this, we obtained a normal distribution for each dataset. Then, we replicate this distribution using its mean and standard deviation. In this case, to generate a new request, we randomly chose a bin from the histogram previously created. Then, from the selected bin, we randomly choose a value for the number of time-steps and assign it to the new request. With this methodology, we ensure that for each new request, the number of time-steps comes from a distribution that mimics the distribution in the original datasets (i.e., they have the same mean and standard deviation). Note that for a given request, its execution and energy consumption only depend on the RNN parameters and the number of time-steps.

To reproduce a real environment, we simulate the arrival time of each user's request. To this end, we use a Poisson distribution [33]. First, to increase or decrease the number of requests per second arriving at the system (i.e., system load), we change the parameter *lamda*, which is the main parameter of the Poisson Distribution and represents the average number of requests per second. Then, for a given *lamda*, we generate new requests at arrival times that the Poisson Distribution determines. Note that, for each of our experiments, *lamda* is kept constant.

7 EXPERIMENTAL RESULTS

This section presents the evaluation of our proposal on an E-PUR-like accelerator and a TPU-like architecture. Regarding the batch size, we tested several batch sizes and found that the results are similar. Thus, we chose 64 as it delivered the best trade-off between performance and area for E-PUR, whereas for a TPU-like architecture, we use 128. The rest of this section is organized as follows. First, we discuss the evaluation of bucketing and cellular batching on E-PUR. Second, we discuss the performance and energy consumption of E-Batch for E-PUR. Finally, we discuss the results of E-Batch for a TPU-like architecture.

7.1 Sequence bucketing and cellular batching on an E-PUR-like Accelerator

Regarding Bucketing, our experiments show that sequence padding is, on average, 7x faster than bucketing and delivers 1.6x higher requests per joule for the machine translation model in Table 1. As discussed in Section 3.2, bucketing with a small *bucket width* tends to create batches with a low number of requests and, thus, weights are swapped more frequently, resulting in larger energy

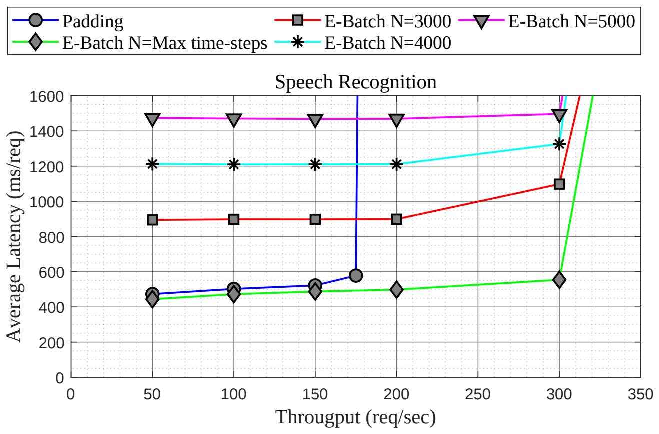

Fig. 16. Average Latency vs Throughput for Speech Recognition [3] using a batch size of 64 on E-PUR.

consumption than sequence padding, which manages to create batches with a larger number of requests. Moreover, with bucketing, requests have to wait longer before they are sent to the accelerator, thus increasing their latency. Note that when the number of requests per second arrives at the system is low, both bucketing and padding achieve similar results since they behave similarly.

Regarding cellular batching, it achieves a maximum throughput of 100 requests per second for the GNMT model, shown in Table 1. Moreover, it consumes 20x more energy per request than sequence padding. Also, on average, the latency is 900ms. As mentioned in Section 3.3, cellular batching incurs a large amount of weight swapping when evaluating RNN models with more than one layer. As previously shown in Figures 8 and 9, as the number of layers increases, the number of bytes read per request increases, and thus, energy efficiency decreases.

7.2 E-Batch on an E-PUR-like Accelerator

Figures 15 and 16 show the average latency per request for Machine Translation and Speech Recognition RNNs, respectively, on an E-PUR-like accelerator. Both figures include the baseline and our E-batch scheme under different loads and using 64 lanes (maximum batch size). For E-Batch, we use different values of the threshold N (i.e., the maximum number of time-steps in a lane) to evaluate the trade-off between energy and latency. In these plots and others, $N = Max\ time - steps$ refers to the E-Batch configuration where the maximum number of time-steps in a lane is set to the number of time-steps of the longest sequence in the batch when it is created. The Speech Recognition network has a larger average latency since its input sequences are typically larger than the input sequences of the Machine Translation model; therefore, they result in more considerable processing and memory transfer times.

As can be seen for both networks, when N is *N=Max time-steps*, the latency of using sequence padding and E-Batch are similar because, in this case, the longest sequence in a given batch is the same for both schemes. However, since E-Batch can add new requests to a batch while it is being processed, the queuing time decreases, henceforth reducing the average latency. Note that for low loads (i.e., 100 requests per second in Figure 15), the average latency is sightly higher for E-Batch. The reason is that E-batch waits for some time to increase the number of requests that are batched together, thus improving weight locality and energy efficiency.

Regarding the maximum throughput, as it can be seen in Figures 15 and 16 E-Batch achieves 1.83x and 1.77x improvement over padding for Machine Translation and Speech Recognition, respectively.

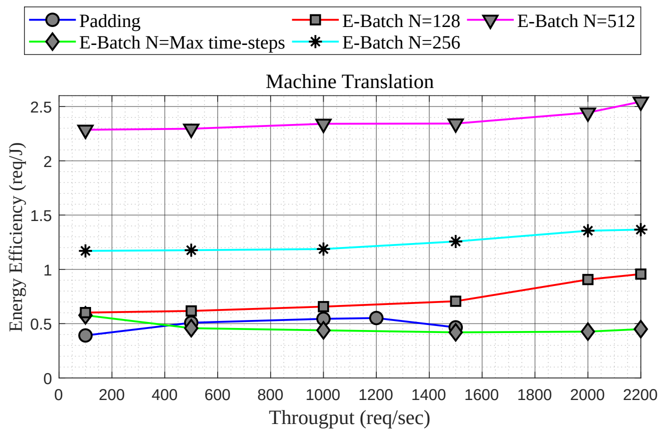

Fig. 17. Average number of Requests per Joule vs Throughput for Machine Translation [4] using a batch size of 64 on E-PUR.

This improvement in throughput comes from allowing new requests to start execution while the first layer of an RNN is being evaluated. Also, because of the variability in the number of time-steps, when the evaluation of small requests is completed, a new request from the waiting queue can join the current batch being executed. Therefore, hardware utilization is increased, improving system throughput. The maximum throughput obtained by E-Batch for different values of N in Figures 15 and 16 are similar, slightly increasing when N becomes very large.

Figures 17 and 18 show the average number of requests per joule for Machine Translation and Speech Recognition, respectively. As can be seen, when N is *N=Max time-steps*, sequence padding tends to have a slightly better energy efficiency than E-Batch for low loads. The reason is that when the number of requests in a batch is small, as new requests arrive at the system, they are added to the batch being processed. However, since N is *N=Max time-steps* once the time-steps for the longest and oldest request in the batch are computed, the system proceeds to evaluate deeper layers. As a result, the computations of some requests are divided among several batches, and they require more memory accesses to fetch the weights. On the contrary, as the load increases, batches with a larger amount of time-steps are created, and thus, weight reuse increases. Regarding the energy breakdown, around 53% of the system's energy consumption is due to dynamic energy, whereas 47% is due to static energy.

As shown in Figures 17 and 18, increasing the value of N will dramatically increase the energy efficiency of E-Batch for both networks. The reason for this is that since the system waits until N time-steps are evaluated in the first layer, a large amount of time-steps are batched together, and, as a result, weight reuse is increased.

7.3 E-Batch on a TPU-like Accelerator

Figure 19 shows the average latency per request for sequence padding and E-Batch running on top of a TPU-like accelerator. The number of lanes used is 128 since this is the number of rows of the systolic array in the TPU. As can be seen, when N is *N=Max time-steps*, the average latency of sequence padding and E-Batch are similar since, in this case, the largest number of time-steps for a given batch is the same in both schemes. However, since E-Batch allows new requests to join a batch while it is being computed for the first layer, the maximum throughput of E-Batch is 2.1x higher.

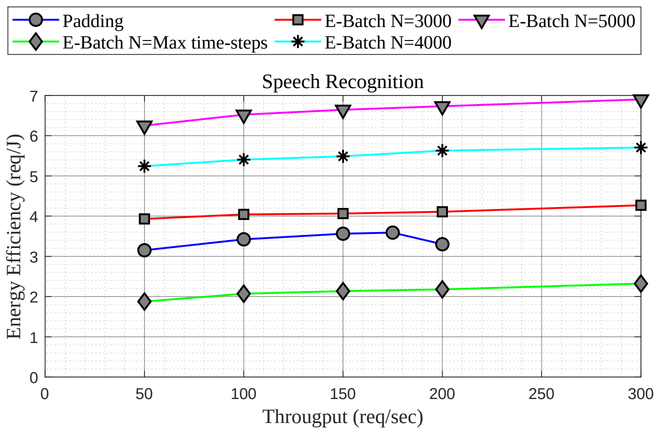

Fig. 18. Average number of Requests per Joule vs Throughput for Speech Recognition [4] using a batch size of 64 on E-PUR.

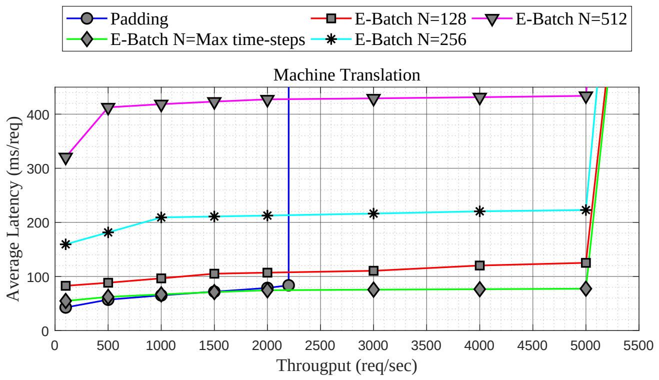

Fig. 19. Average Latency vs Throughput for Machine Translation [4] using a batch size of 128 for TPU.

Figure 20 shows the number of requests per joule for the machine translation network evaluated on a TPU-like architecture. Similar to E-PUR, when the load is small and N is *N=Max time-steps*, many batches with a small number of time-steps are created. As a result, the number of memory accesses increases. However, as the number of requests per second increases, a more significant number of requests are batched together, thus increasing weight reuse. Similarly, as N increases, a larger amount of time-steps are batched together, increasing weight reuse. For instance, when N is 128, and the load is 2000 req/sec, E-batch achieves an energy efficiency improvement of 1.3x compared to the baseline, whereas when N is 256 or 512, energy efficiency is improved by 1.46x and 1.6x, respectively. However, note that when improving energy efficiency, the average latency is also increased.

8 RELATED WORK

Recurrent Neural Networks (RNNs) have achieved tremendous success in sequence to sequence problems. Not surprisingly, several hardware accelerators [8, 10, 15, 35] and software libraries [6, 14, 36] tailored to improve energy efficiency and performance of RNNs have been proposed recently. In

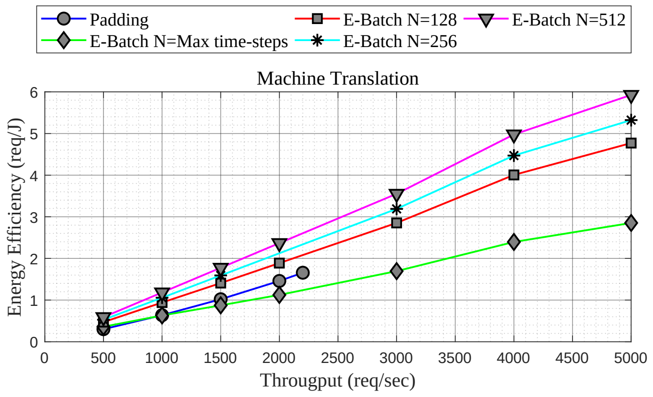

Fig. 20. Average number of Requests per Joule vs Throughput for Machine Translation [4] using a batch size of 128 for TPU.

addition to hardware acceleration, several techniques such as pruning [12, 25], model compression [11], and computations reuse[26, 30] have been employed. Our optimizations are orthogonal to those techniques.

Regarding software libraries such as [6, 14, 24], they are mainly tailored to GPUs and CPUs [36], whereas our proposal targets specialized accelerators. In order to handle sequences of differences sizes, previous proposals employ sequence padding or bucketing. In CNTK [28], wasted computations are avoided by trying to batch small sequences when they can fit in the padded space. However, this is not always possible since small sequences may not be available. Conversely, in E-Batch, a sequence can be split among different batches so that only the amount of time-steps required to fill the padded space is used.

Batching RNN sequences are supported in hardware accelerators such as BrainWave and LSTM-Sharp. BrainWave [8] is highly optimized for batches of size one, and inputs are processed sequentially, by computing one single input at a time. In this accelerator, sequence padding is not required. However, employing batch sizes larger than two is unfeasible since inputs are processed sequentially. LSTM-Sharp [35] focuses on increasing resource utilization. It addresses the issue of extra padded computations that occur when performing matrix-vector multiplications, and the number of multipliers per tile are not multiples of the input vector size. This padding is different to padding several sequences to make their sizes homogeneous. Sequence padding is also needed for LSTM-Sharp, to support batching. E-batch is orthogonal to this work and can be implemented on top of this accelerator. TPU [15] supports large batch sizes employing sequence padding.

9 CONCLUSIONS

In this paper, we presented E-Batch, a batching system for recurrent neural networks that increases throughput while improving energy efficiency. E-Batch consists of a runtime and minor extensions to the hardware accelerator. In E-Batch, a large number of input sequences are batched together to decrease memory accesses. Furthermore, throughput is increased by allowing new requests to join other requests while their execution is ongoing. We evaluated E-Batch on top of E-PUR and TPU, two state-of-art hardware accelerators for RNNs. Our experimental results show that in E-PUR, E-Batch improves throughput by 1.8x and energy efficiency by 3.6x, whereas for TPU, throughput is enhanced by 2.1x and energy efficiency by 1.6x.

ACKNOWLEDGMENTS

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 (MCIN/AEI) under grant PID2020-113172RB-I00, and the ICREA Academia program.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 265–283.

[2] Kanak Agarwal, Harmander Deogun, Dennis Sylvester, and Kevin Nowka. 2006. Power gating with multiple sleep modes. In *7th International Symposium on Quality Electronic Design (ISQED'06)*. IEEE, 5–pp.

[3] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, et al. 2016. Deep speech 2: End-to-end speech recognition in english and mandarin. In *International conference on machine learning*. PMLR, 173–182.

[4] Denny Britz, Anna Goldie, Minh-Thang Luong, and Quoc V. Le. 2017. Massive Exploration of Neural Machine Translation Architectures. *CoRR* abs/1703.03906 (2017). arXiv:1703.03906 http://arxiv.org/abs/1703.03906

[5] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *CoRR* abs/1512.01274 (2015). arXiv:1512.01274 http://arxiv.org/abs/1512.01274

[6] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. *CoRR* abs/1410.0759 (2014). arXiv:1410.0759 http://arxiv.org/abs/1410.0759

[7] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *CoRR* abs/1406.1078 (2014). arXiv:1406.1078 http://arxiv.org/abs/1406.1078

[8] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. 2018. A Configurable Cloud-scale DNN Processor for Real-time AI. In *Proceedings of the 45th Annual International Symposium on Computer Architecture* (Los Angeles, California) *(ISCA '18)*. IEEE Press, Piscataway, NJ, USA, 1–14. https://doi.org/10.1109/ISCA.2018.00012

[9] Pin Gao, Lingfan Yu, Yongwei Wu, and Jinyang Li. 2018. Low Latency RNN Inference with Cellular Batching. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. ACM, New York, NY, USA, Article 31, 15 pages. https://doi.org/10.1145/3190508.3190541

[10] Yijin Guan, Zhihang Yuan, Guangyu Sun, and Jason Cong. 2017. FPGA-based accelerator for long short-term memory recurrent neural networks. In *Design Automation Conference (ASP-DAC), 2017 22nd Asia and South Pacific*. IEEE, 629–634.

[11] U. Gupta, B. Reagen, L. Pentecost, M. Donato, T. Tambe, A. M. Rush, G. Wei, and D. Brooks. 2019. MASR: A Modular Accelerator for Sparse RNNs. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 1–14. https://doi.org/10.1109/PACT.2019.00009

[12] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, Huazhong Yang, and William (Bill) J. Dally. 2017. ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '17)*. ACM, New York, NY, USA, 75–84. https://doi.org/10.1145/3020078.3021745

[13] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[14] Connor Holmes, Daniel Mawhirter, Yuxiong He, Feng Yan, and Bo Wu. 2019. GRNN: Low-Latency and Scalable RNN Inference on GPUs. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. ACM, New York, NY, USA, Article 41, 16 pages. https://doi.org/10.1145/3302424.3303949

[15] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson,

Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/3079856.3080246

[16] Viacheslav Khomenko, Oleg Shyshkov, Olga Radyvonenko, and Kostiantyn Bokhan. 2016. Accelerating recurrent neural network training using sequence bucketing and multi-gpu data parallelization. In *2016 IEEE First International Conference on Data Stream Mining & Processing (DSMP)*. IEEE, 100–103.

[17] Richard E. Korf. 2009. Multi-way Number Partitioning. In *Proceedings of the 21st International Jont Conference on Artifical Intelligence* (Pasadena, California, USA) *(IJCAI'09)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 538–543. http://dl.acm.org/citation.cfm?id=1661445.1661531

[18] Hai Li, Swarup Bhunia, Yiran Chen, TN Vijaykumar, and Kaushik Roy. 2003. Deterministic clock gating for microprocessor power reduction. In *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.* IEEE, 113–122.

[19] Yajie Miao, Mohammad Gowayyed, and Florian Metze. 2015. 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, 167–174.

[20] Micron Inc. [n. d.]. DDR4 SDRAM. https://www.micron.com/-/media/client/global/documents/products/data-sheet/dram/ddr4/8gb_ddr4_sdram.pdf. Accessed on 15 October 2021.

[21] Micron Inc. [n. d.]. TN-53-01: LPDDR4 System Power Calculator. https://www.micron.com/support/tools-and-utilities/power-calc. Accessed on 15 September 2021.

[22] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. 2009. CACTI 6.0: A tool to model large caches. *HP Laboratories* (2009), 22–31.

[23] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. In *NIPS-W*.

[24] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic Differentiation in PyTorch. In *NIPS Autodiff Workshop*.

[25] M. Riera, J. Arnau, and A. González. 2019. CGPA: Coarse-Grained Pruning of Activations for Energy-Efficient RNN Inference. *IEEE Micro* 39, 5 (Sep. 2019), 36–45. https://doi.org/10.1109/MM.2019.2929742

[26] Marc Riera, Jose-Maria Arnau, and Antonio González. 2018. Computation reuse in DNNs by exploiting input similarity. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*. IEEE Press, 57–68.

[27] Ananda Samajdar, Jan Moritz Joseph, Yuhao Zhu, Paul Whatmough, Matthew Mattina, and Tushar Krishna. 2020. A systematic methodology for characterizing scalability of dnn accelerators using scale-sim. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 58–68.

[28] Frank Seide and Amit Agarwal. 2016. CNTK: Microsoft's Open-Source Deep-Learning Toolkit. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California, USA) *(KDD '16)*. ACM, New York, NY, USA, 2135–2135. https://doi.org/10.1145/2939672.2945397

[29] Franyell Silfa, Gem Dot, Jose-Maria Arnau, and Antonio González. 2018. E-PUR: An Energy-efficient Processing Unit for Recurrent Neural Networks. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques* (Limassol, Cyprus) *(PACT '18)*. ACM, New York, NY, USA, Article 18, 12 pages. https://doi.org/10.1145/3243176.3243184

[30] Franyell Silfa, Gem Dot, Jose-Maria Arnau, and Antonio González. 2019. Neuron-Level Fuzzy Memoization in RNNs. In *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. ACM, New York, NY, USA, 782–793. https://doi.org/10.1145/3352460.3358309

[31] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. 2016. Show and tell: Lessons learned from the 2015 mscoco image captioning challenge. *IEEE transactions on pattern analysis and machine intelligence* 39, 4 (2016), 652–663.

[32] Po-Han Wang, Chia-Lin Yang, Yen-Ming Chen, and Yu-Jung Cheng. 2011. Power gating strategies on GPUs. *ACM Transactions on Architecture and Code Optimization (TACO)* 8, 3 (2011), 1–25.

[33] Wayne L Winston and Jeffrey B Goldberg. 2004. *Operations research: applications and algorithms*. Vol. 3. Thomson Brooks/Cole Belmont.

[34] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. 2016. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *CoRR* abs/1609.08144 (2016). arXiv:1609.08144 http://arxiv.org/abs/1609.08144

[35] Reza Yazdani, Olatunji Ruwase, Minjia Zhang, Yuxiong He, José-María Arnau, and Antonio González. 2019. LSTM-Sharp: An Adaptable, Energy-Efficient Hardware Accelerator for Long Short-Term Memory. *CoRR* abs/1911.01258 (2019). arXiv:1911.01258 http://arxiv.org/abs/1911.01258

[36] Minjia Zhang, Samyam Rajbhandari, Wenhan Wang, and Yuxiong He. 2018. DeepCPU: Serving RNN-based Deep Learning Models 10x Faster. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, Berkeley, CA, USA, 951–965. http://dl.acm.org/citation.cfm?id=3277355.3277446