Chapter 4

Optimizing the Performance of RNN/LSTM Accelerators

4.1 Introduction

Many applications involve sequential data processing and time-series predictions, e.g., natural language processing, speech recognition, music composition and video activity recognition. As convolution neural networks (CNNs) are specialized for processing image data, recurrent neural networks (RNNs) are specialized in handling sequential data. Processing sequential data requires remembering the contextual information from previous data. Recurrent neural networks (RNNs) are specialized in handling such problems by maintaining an internal state based on previously seen data. RNNs scale well with long sequences and even sequences of variable lengths. They share weights across different time steps. LSTMs [23] are variants of RNNs designed to handle long-range dependencies by storing useful information about previous inputs for a long duration.

LSTM computations involve several large matrix-vector multiplications, and these matrix-vector multiplications are performed for a large number of time steps. The inputs to the network are a time sequence of vectors, and these large matrices hold weights, which are learned during the training process. The size of these matrices can be significant in several MBs and often exceed the size of the accelerator's on-chip memory. These matrices are partitioned into blocks and accessed from off-chip memory repeatedly by the accelerator, which results in a large volume of off-chip memory accesses and energy consumption.

sizes

4.2 Background

LSTM has recurrent connections to capture the long and short-term dependencies. LSTM cells maintain the cell state to store the dependency information derived from the previously seen data and use four gates to modify the cell state and produce the output. Typically the computations of LSTM cell is described by the following equations

$$\begin{aligned} i &= \sigma(W^i \cdot x_t + R^i \cdot h_{t-1} + b^i) \\ f &= \sigma(W^f \cdot x_t + R^f \cdot h_{t-1} + b^f) \\ g &= \tanh(W^g \cdot x_t + R^g \cdot h_{t-1} + b^g) \\ o &= \sigma(W^o \cdot x_t + R^o \cdot h_{t-1} + b^o) \\ c_t &= f \odot c_{t-1} + i \odot g \\ h_t &= o \odot \tanh(c_t) \end{aligned} \tag{4.1}$$

where x_t is the input, h_t is the hidden state and c_t is the cell state at time t. i, f, g, o are the computed gate values. $\odot$ denotes the element wise multiplications. W^j and R^j are the input and hidden state weight matrices, and b^j is the bias vector, respectively, where $j \in \{i, f, o, g\}$. W^j, R^j and b^j are the parameters learned during the training process. Once the network is trained these parameters are used during inferencing. If the dimensions of the input vector x_t is L and hidden state vector h_t is N, the dimensions of W^j, R^j and b^j is $N \times L$, $N \times N$ and N, respectively. N is referred to as the number of hidden states of the LSTM.

Equation 4.1 involves matrix-vector multiplications and element-wise operations. Element-wise operations are vector-vector additions, multiplications, and non-linear functions. The non-linear functions are hyper-tangent ($\tanh$) and sigmoid (σ).

At every time step, Equation 4.1 take vectors x_t as input and compute the cell state (c_t) and hidden state (h_t) using the previous hidden state (h_{t-1}) and cell state (c_{t-1}) vectors. h_t depends on the present input vector x_t and the previous time step cell state (c_{t-1}) and hidden state (h_{t-1}) vectors. The dependency of h_t on h_{t-1} and c_{t-1} prevents the parallel processing of multiple time steps and limits the data reuse.

LSTM accelerators have small on-chip memory. The large weight matrices R and W are stored in off-chip memory. The dependency of h_t and c_t on the previous time step computations makes the reuse of weight matrix R a challenge. Thus the weights are accessed from the off-chip memory at every step, resulting in sizeable off-chip memory accesses and high energy consumption of these accelerators. This work focuses on reducing the off-chip memory accesses of R during the LSTM inference phase.

4.3 Related Work

To address the computational and energy efficiency of LSTMs and in general RNNs, several ASIC [4, 11, 42] and FPGA based accelerators [5, 14, 17, 20, 27] are proposed. The energy efficiency of LSTM accelerators is critical for their widespread usage, and off-chip memory access is the key to improving energy consumption. Most of these works focused on improving energy efficiency by reducing off-chip memory accesses.

Some approaches [14, 27, 37] used on-chip memory to store all the weights. Sizes of weights in recent multi-layer LSTM models can be several MB's, and using large on-chip memory is expensive. These approaches are not scalable and effective only for small LSTM models. The proposed approach is independent of model size and effective for large LSTM models.

Several approaches used the fact that neural networks are error-tolerant and have lots of redundancy. They used the quantization and pruning techniques to compress the models' size. Approaches [14, 41] used 18-bit, Chang et al. [5] used 16-bit, Han et al. [20] used 12-bits precision for storing the inputs and weights, Lee et al. [27] used 8-bit inputs and 6-bits for weights to reduce the model size. The proposed approach is orthogonal to the quantization techniques and can be integrated with different quantization techniques to reduce the memory accesses further.

Han et al. [20] used pruning to compress the model. However, pruning results in irregular network structure, and the sparse matrix require additional computational and storage resources and causes unbalanced load distribution. To overcome this Wang et al. [41] used block-circulant matrices representations to compress the LSTM/RNN model and to eliminate irregularities resulted from compression. Some approaches [20, 32, 33] used load balance aware pruning techniques to overcome the unbalanced load distribution problem.

Quantization and pruning approaches compromise the accuracy of the networks. The other line of works reduced the memory accesses without effecting the accuracy of the NNs output by applying the data-reuse techniques. The matrix-vector multiplication $W^j \cdot x$ in Equation Equation 4.1, where $j \in \{i, f, g, o\}$, is independent of previous state computation. Que et al. [36] proposed a blocking-batching scheme that reuses the weights of W^j matrix by processing a group of input vectors as a batch. The input vectors in the same batch share the same weight matrices (W^j). However, it is difficult to collect the required number of input vectors. As the LSTM cell states (h_t and c_t) computations depend on previous time-step cell states, the benefit of their batching schemes is limited to $W^j \cdot x$. Reusing weights of R across different time steps has not been successful because of the dependency on previous time-step states.

Park et al.([34]) proposed a time-step interleaved weight reuse scheme (TSI-WR) which reuses the weights of R matrix between two adjacent time steps by performing computations in a time-interleaved manner. Their approach logically partitions the R matrix into blocks. A block is accessed from off-chip memory to compute the hidden state vector h_t, and a fraction of it is reused to compute the partial sum of next time step state h_{t+1}. However, their approach does not fully exploit the data reuse, and several weights are accessed repeatedly from the off-chip memory. In addition, the data reuse in the TSI-WR approach depends on the on-chip storage size which benefits accelerators with larger on-chip memory.

Our approach schedules the computations in a way that reuses all the weights of R between two adjacent time steps. The data reuse in our approach is independent of on-chip buffer sizes which even benefits to accelerators with small on-chip memory.

4.4 Split And Combine Computations Approach

In this section, we first describe basic idea of Split And Combine Computations (**SACC**) approach and then its extension to block-wise reuse of data.

4.4.1 Basic Approach

The computation of the h_t can be expressed as shown below

$$h_t[k] = F(S_t[k] + q_t[k]) \tag{4.2}$$

where F is a non-linear function. q_t is computed as $W \cdot x_t + b$ and its computations are independent of previous step cell states. $S_t[k]$ is the sum of N product terms as shown below,

$$S_t[k] = \sum_{n=0}^{N-1} R[k][n] \cdot h_{t-1}[n] \tag{4.3}$$

$S_t[k]$ can be computed as a sum of the following two partial sums $S_t^L[k]$ and $S_t^U[k]$

The intuition behind splitting the computation in this manner is not explained.

$$S_t^L[k] = \sum_{n=0}^{k} R[k][n] \cdot h_{t-1}[n] \tag{4.4}$$

$$S_t^U[k] = \sum_{n=k+1}^{N-1} R[k][n] \cdot h_{t-1}[n] \tag{4.5}$$

The correspondence between the equations, figure 4.1 and the text is not very evident. It may be better to build the overall picture in small steps. For example, you may show portions of figure 4.1 and explanations for each equation separately and then combine everything into the bigger picture.

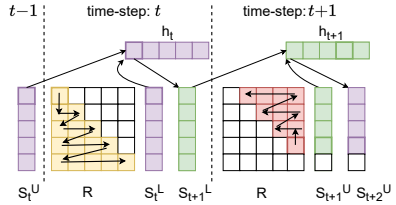


Figure 4.1: Splitting the hidden state vector computations into partial sums

First explain what fig 4.1 is about. What is represented by various arrows? Do not leave these to reader's imagination.

Equation Equation 4.4 uses the lower-diagonal and diagonal elements of R (R^L), and Equation 4.5 uses the upper diagonal elements of R (R^U). As shown in Figure 4.1, R^L and R^U are accessed in consecutive time steps and reused in the partial sum computations of two steps. At time step t, S_t^U and h_{t-1} are the inputs from the previous time step, and R^L is reused to compute the partial sums S_t^L and S_{t+1}^L. Input S_t^U is added to S_t^L to compute h_t, and S_{t+1}^L is passed to $(t+1)^{th}$ step computations. In the same way, at time step $t+1$, R^U is reused to compute S_{t+1}^U and S_{t+2}^U. Elements of R^L are accessed from top to bottom, left to right, while elements of R^U are accessed in the reverse order to satisfy the dependencies. As shown in Figure 4.1, the proposed approach accesses the weight matrix R once, to compute h_t and h_{t+1}. Figure4.2 illustrates the compute-steps (C1 to C9) and weights accessed for computing

See the comments related to fig 4.1.

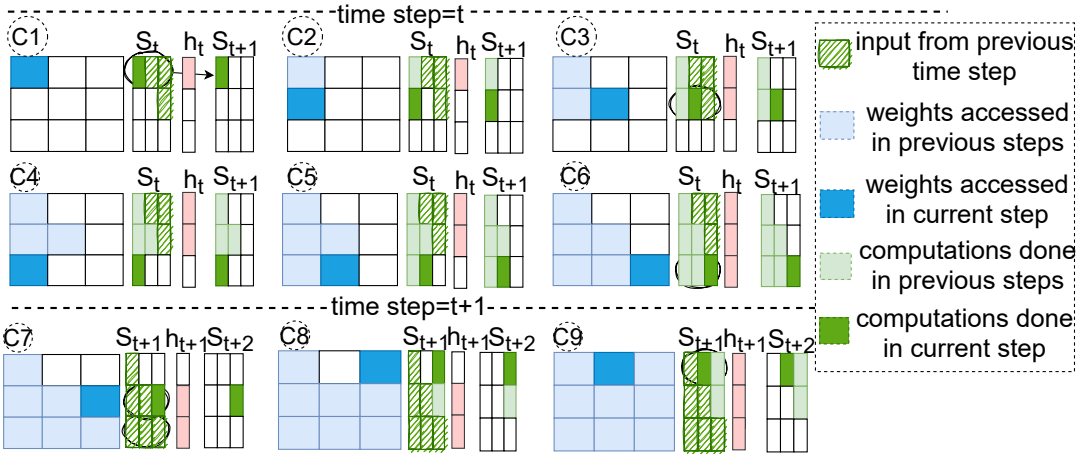


Figure 4.2: Computation of consecutive hidden state vectors h_1, h_2 and h_3 while accessing R matrix from off-chip memory.

the outputs of two time steps for N=3. The shaded rectangular blocks shows the product terms input from the previous time step (t=1). When all the product terms ($R[k][j]\cdot h_{t-1}[j]$) for the partial sum vector element ($S_t[k]$) are computed, then $h_t[k]$ is computed (shown as pink blocks in Figure4.2). The weights of R are reused for the product terms $R[k][j]\cdot h_t[j]$ for computing the partial sum S_{t+1}, using the values of h_t computed in previous or present compute-step. As shown in Figure4.2 matrix R is accessed in 9 compute-steps ($N\times N$) to

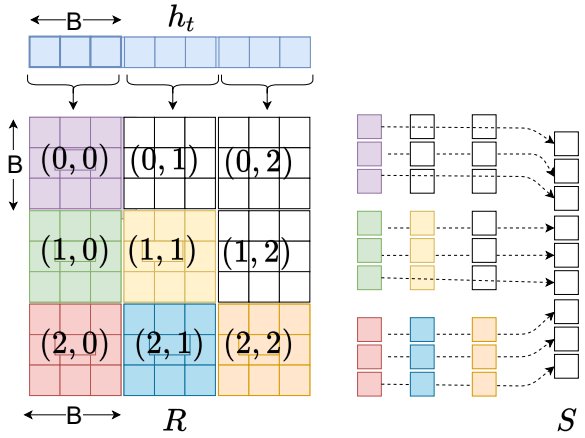


Figure 4.3: Partitions of R in $B \times B$ blocks and partial sum computations.

compute h_t and h_{t+1} and the partial sum S_{t+2} for the next time step $(t+1)$.

4.4.2 Block-wise reuse

such that each block individually

The proposed approach partitions R into square blocks of size $B \times B$, that fits in the accelerator's on-chip memory, as shown in Figure4.3. Each block can be indexed as (r, m), where $0 \leq r, m \leq (\lceil \frac{N}{B} \rceil - 1)$. The proposed approach computes h_t in $\lceil \frac{N}{B} \rceil$ steps and at each step computes a slice of length B. The k^{th} element of r^{th} slice of h_t can be computed as following

How did you get these equations? What is the relationship with the previous equations?

$$h_t[B*r+k] = F\Big(\sum_{m=0}^{\lceil \frac{N}{B} \rceil - 1} S_{(r,m)}[k] + q_t[B*r+k]\Big) \tag{4.6}$$

$$S_{(r,m)}[k] = \sum_{j=0}^{B-1} R[B*r+k][B*m+j] \cdot h_{t-1}[B*m+j] \tag{4.7}$$

,where $0 \leq k \leq B-1$. The summation $\sum_{m=0}^{\lceil \frac{N}{B} \rceil - 1} S_{(r,m)}[k]$ in Equation 4.6 can be expressed as a sum of the following partial sums

$$\sum_{m=0}^{\lceil \frac{N}{B} \rceil - 1} S_{(r,m)}[k] = \sum_{m=0}^{r} S_t^L[k] + \sum_{m=r+1}^{\lceil \frac{N}{B} \rceil - 1} S_t^U[k] \tag{4.8}$$

$S_t^L[k]$ uses the lower-diagonal and diagonal blocks of R (R^L), and $S_t^U[k]$ uses the upper diagonal blocks of $R(R^U)$. The SACC approach reuses blocks of R to compute the partial sums of two consecutive time steps, similar to the approach described in 4.4.1.

Algorithm 2 describes the computations of the SACC approach. The dimensions of W, R, b, and x_t are $4N \times L$, $4N \times N$, $4N \times 1$, and $L \times 1$, respectively. The algorithm stores the vectors

Gradual build-up would be better than abruptly showing a large algorithm followed by a sketchy explanation.

h_t, c_t, and the partial sum vectors (s_{t+1}) in the on-chip memory and accesses the weights from the off-chip memory. It first computes the vector q_t as $W \cdot x + b$, at line 2 and then invokes the procedures UPDIAGREUSE at line 4 or LOWDIAGREUSE at line 7 at alternate time steps. LOWDIAGREUSE accesses blocks of R^L, and UPDIAGREUSE accesses blocks of R^U. The procedures have two nested loops. LOWDIAGREUSE traverses the blocks from top to bottom (at line 11), left to the right (at line 13), while the UPDIAGREUSE traverses the blocks in the opposite order. The inner loop accesses the $(r, m)^{th}$ block of R from the off-chip memory and reuses it to compute the partial sums s^B_{t+1} and s^B_{t+2}. The outer loop iterations compute r^{th} slices of h_{t+1}, c_{t+1}, and s_{t+2}. When all the blocks of the r^{th} row are processed, s^B_{t+1} has the total sum, which is then used to compute the r^{th} slice of the vectors h_{t+1} and c_{t+1} using LSTMEQUATIONS at line 19.

Algorithm 2 SACC algorithm

procedure COMPLSTMCELL(W,R,b,x_{t+1})
  $q_{t+1} \leftarrow$ MXV$(W, x_{t+1}, 4N, L)+b$
  if ($stage$ is even) **then**
    $(h_t, c_t, s_{t+1}) \leftarrow$UPDIAREUSE$(R, q_t, h_t, c_t, s_{t+1})$
    $stage \leftarrow odd$
  else
    $(h_t, c_t, s_{t+1}) \leftarrow$LOWDIAREUSE$(R, q_t, h_t, c_t, s_{t+1})$
    $stage \leftarrow even$
  return (h_t)
procedure LOWDIAGREUSE($R, q_t, h_t, c_t, s_{t+1}$)
  for $r \leftarrow 0$ **to** $(\lceil \frac{N}{B} \rceil - 1)$ **do**
    $(i_s, i_e) \leftarrow (r \cdot B, (r+1) \cdot B - 1), s^B_{t+2} \leftarrow \vec{0}$
    for $m \leftarrow 0$ **to** r **do**
      $\mathbf{R}^B \leftarrow$GETDDRBLKS(R, r, m, B)
      $(h^B_t, c^B_t, q^B_t) \leftarrow$ GETSLICE(h_t, c_t, q_t, m)
      $s^B_{t+1} \leftarrow$ GETSLICES(s_{t+1}, m)
      $s^B_{t+1} \leftarrow s^B_{t+1}+MXV(\mathbf{R}^B, h^B_t, 4B, B)$
      if $m = r$ **then**
        $(h^B_{t+1}, c^B_{t+1}) \leftarrow$LSTMEQNS$(v^B_x, s^B_{t+1}, c^B_t)$
        $h_{t+1}[i_s : i_e] \leftarrow h^B_{t+1}, c_{t+1}[i_s : i_e] \leftarrow c^B_{t+1}$
      $h^B_{t+1} \leftarrow h_{t+1}[m \times B : (m+1) \times B - 1]$
      $s^B_{t+2} \leftarrow s^B_{t+2}+MXV(\mathbf{R}^B, h^B_{t+1}, 4B, B)$
    $s_{t+2} \leftarrow$ UPDATEVECT(s_{t+2}, s^B_{t+2}, r)
  return $(s_{t+2}, h_{t+1}, c_{t+1})$
procedure UPDIAGREUSE($R, q_t, h_t, c_t, s_{t+1}$)
  for $r \leftarrow (\lceil \frac{N}{B} \rceil - 1)$ **downto** 0 **do**
    $(i_s, i_e) \leftarrow (r \cdot B, (r+1) \cdot B - 1), s^B_{t+2} \leftarrow \vec{0}$
    for $m \leftarrow (\lceil \frac{N}{B} \rceil) - 1$ **downto** $r+1$ **do**
      $\mathbf{R}^B \leftarrow$GETDDRBLKS(R, r, m, B)
      $(h^B_t, c^B_t, q^B_t) \leftarrow$ GETSLICE(h_t, c_t, q_t, m)
      $s^B_{t+1} \leftarrow$ GETSLICES(s_{t+1}, m)
      $s^B_{t+1} \leftarrow s^B_{t+1}+MXV(\mathbf{R}^B, h^B_t, 4B, B)$
      $h^B_{t+1} \leftarrow h_{t+1}[i_s : i_e]$
      $s^B_{t+2} \leftarrow s^B_{t+2}+MXV(\mathbf{R}^B, h^B_{t+1}, 4B, B)$;
    $(h^B_{t+1}, c^B_{t+1}) \leftarrow$LSTMEQNS$(v^B_x, s^B_{t+1}, c^B_t)$
    $h_{t+1}[i_s : i_e] \leftarrow h^B_{t+1}, c_{t+1}[i_s : i_e] \leftarrow c^B_{t+1}$
    $s_{t+2} \leftarrow$ UPDATEVECT(s_{t+2}, s^B_{t+2}, r)
  return $(s_{t+2}, h_{t+1}, c_{t+1})$

Both the procedures reuse the blocks of R to reduce the off-chip memory accesses.

Algorithm 3 LSTM Equations

procedure LSTMEQNS(q_{t+1}, s_{t+1}, c_t)
  $(v_i, v_f, v_g, v_o) \leftarrow ExtractVecs(q_t, B)$
  $(s_i, s_f, s_g, s_o) \leftarrow ExtractVecs(s_{t+1}, B)$
  for $n \leftarrow 0$ **to** $B-1$ **do**
    $i[n] \leftarrow \text{SIGMOID}(s_i[n] + v_i[n])$
    $f[n] \leftarrow \text{SIGMOID}(s_f[n] + v_f[n])$
    $g[n] \leftarrow \text{TANH}(s_g[n] + v_g[n])$
    $o[n] \leftarrow \text{SIGMOID}(s_o[n] + v_o[n])$
    $c_{t+1}[n] \leftarrow f[n] \otimes c_t[n] + i[n] \otimes g[n]$
    $h_{t+1}[n] \leftarrow o[n] \otimes \text{TANH}(c)[n]$
    return(h_{t+1}, c_{t+1})

Algorithm 3 implements the LSTM equations using the partial sum vector.

4.5 Experimental Setup And Results

We have implemented LSTM layers using conventional and proposed approaches and synthesized the design using the SDSoC framework, SDx v2018.3. The design can be configured for different input vector lengths, on-chip buffer sizes, and the number of hidden units. We carried out the experiments on Zedboard, and the target frequency is 100MHz. The off-chip memory is DDR3 connected using a 64-bit AXI bus. We have integrated the Xilinx AXI Performance Monitor (APM) IP to log the number of bytes transferred and memory access latencies for DRAM accesses.

4.5.1 Baseline

We have compared our approach with conventional and TSI-WR [34] approaches. We have used the exact implementation for off-chip memory transfer, matrix-vector multiplication, sigmoid, and tanh for both approaches to perform a fair comparison. The on-chip buffer size ($4\times B\times B$) used to store the weight matrices is also kept the same for both the approaches. Table 4.1 shows the FPGA resources utilization reported by Vivado for conventional and proposed approaches. The proposed approach requires additional on-chip memory to store the four partial sum vectors ($4\times N$) and four temporary vectors ($4\times B$). We have also implemented the models to compute the off-chip memory accesses for all three approaches and integrated them with analytical framework (Chapter 2) to compare the off-chip memory accesses of the three approaches.

Table 4.1: FPGA Resource Utilization(%) for B=64

	LUT	FF	Block RAM	DSP
Conv.	66.8	40.61	61.07	82.73
SAC	63.3	37.87	65.36	75.45

Does the number of BRAMs match the theoretical estimate for B=64?

Why these are not integers?

Implementation of TSI-WR?

Why there is a reduction in number of LUTs, FFs and DSPs?

4.5.2 Benchmarks

To demonstrate the efficiency of our approach, we have experimented with LSTM models used in speech recognition (for TIMIT [15]) and character level Language Modelling (LM) [39], which is widely used in natural language processing. The LSTM models are adopted from [4, 20, 32]. Each model has two LSTM layers and the parameters are described in Table 4.2.

Table 4.2: LSTM Models used for experiments

Model	Input Size	#Hidden units	
		Layer 1	Layer 2
Character Level LM [4]	65	128	128
TIMIT-512 [32]	40	512	512
TIMIT-1024 [20]	160	1024	1024

4.5.3 Results

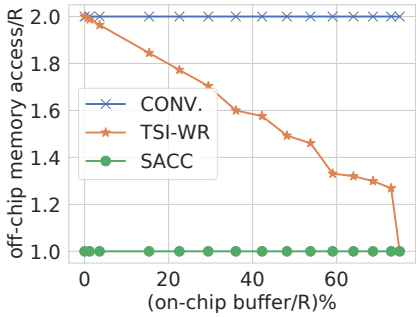
4.5.3.1 Off-Chip Memory Access

Figure 4.4 shows the off-chip memory accesses for two time-steps for conventional, TSI-WR, and SACC approaches. We have experimented for different on-chip memory sizes, when it is lesser than R. Figure 4.4a and Figure 4.4b shows the result computed using Analytical Framework and measured using our hardware implementation on FPGA, respectively.

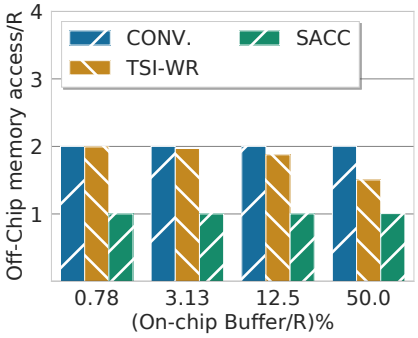
smaller

Give reference to specific equations/ algorithms of chapter 2.

how was measurement done?



(a)



(b)

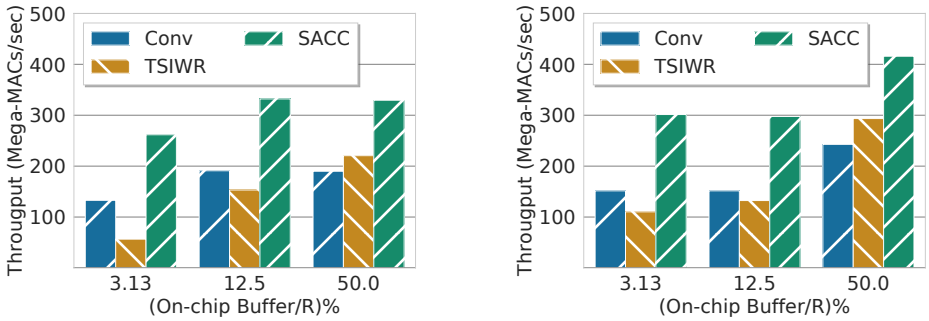
were there multiple implementations with different buffer sizes?

Figure 4.4: Off-chip memory access for matrix-vector multiplication of two consecutive time steps with different on-chip buffer to R matrix size ratio. (a) using analytical frameworks (b) measured on hardware

If the on-chip buffer size is small compared to the weight matrices, tiles of R are accessed from off-chip memory replacing the older tiles in on-chip memory. In conventional approaches, there is no reuse of these tiles for subsequent time step computations, which results in accessing full matrix R every step. For conventional approaches, the off-chip memory accesses remains same, even if on chip mem size is increased as shown by the horizontal line in Figure 4.4a. The TSI-WR approach schedules the tiles in a way to reuse the data from the on-chip memory and reduces off-chip memory access. However, the extent of data reuse in the TSI-WR approach depends on the size of overlap between two consecutive

tiles which is decided by the available on-chip buffer size, as shown in Figure 4.4. When the on-chip buffer to R matrix size is close to 48%, TSI-WR reduces the memory access by ≈25%. The SACC approach splits the computations and perform the scheduling of the tiles that reduces the memory accesses by ≈50%, irrespective of the on-chip buffer size.

4.5.3.2 Throughput Analysis



(a) 64KB on-chip buffer size

(b) 128 KB on-chip buffer size

Figure 4.5: Throughput variation with different on-chip buffer/R ratio

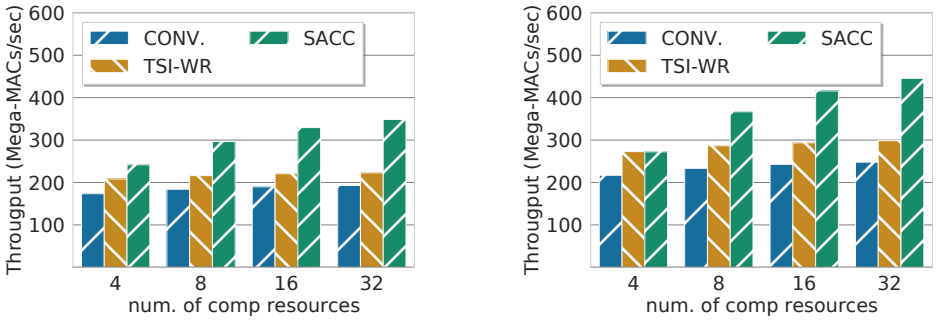
how was throughput measured?

Figure 4.5a and Figure 4.5b compares the throughput for different ratios of on-chip buffer to weight matrix size for 64 and 128 KB on-chip buffer size, respectively. We experimented with different number of hidden units (N) to vary R matrix size. Increasing on-chip buffer to R matrix size ratio results in larger tile sizes and fewer number of iterations, resulting in throughput improvement, for all the three approaches. The TSI-WR approach performs worse than the conventional approach for smaller on-chip buffer/R ratio due to its control logic overhead and insignificant data reuse benefits. However, for a larger on-chip buffer/R size ratio (e.g., 50%), the TSI-WR approach outperforms the conventional approach due to better data-reuse. The proposed approach performs better than the remaining approaches for all the cases due to its significant data-reuse of weight matrix R and minimal control logic overhead. For 50% on-chip buffer to matrix size ratio, the proposed approach improves the throughput by 42% and 33% for 64 KB and 41% and 29% for 128 KB on-chip buffer size, compared to conventional and TSI-WR approaches, respectively.

4.5.3.3 Throughput variation with compute resources

Matrix-Vector multiplication dominates the LSTM accelerators' processing. The performance of LSTM accelerators is limited by the memory bandwidth. For the conventional approach, increasing the number of computing resources does not improve the performance, as shown in

multiple fpga implementations?



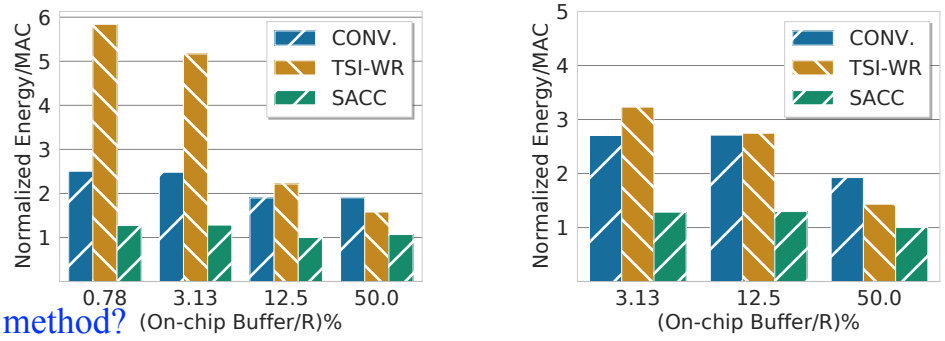
(a) 64 KB on-chip buffer size (b) 128 KB on-chip buffer size

Figure 4.6: Throughput variation of MxV with compute resources for 50% on-chip buffer/R ratio

Figure 4.6. TSI-WR approach improves the performance with increasing compute resources as it reuses the on-chip data and improves the operational intensity (ops/byte). However, the improvement in the TSI-WR approach is noticable only for the large on-chip buffer/R size ratios. In this experiment, we have compared the results when on-chip buffer to matrix size ratio of 50%. The proposed SACC approach, has better operational intensity (ops/byte) and alleviates the memory bandwith issue compared to other two approaches, which results in throughput improvement with increasing the number of parallel resources. The improvement in throughput with increased number of compute resource by the SACC approach is observed for different on-chip buffer sizes as shown in Figure 4.6a and Figure 4.6b.

4.5.3.4 Energy Efficiency Improvement

measurement method?



(a) 64KB on-chip buffer size (b) 128 KB on-chip buffer size

Figure 4.7: Energy improvement with on-chip buffer size/R

Figure 4.7a and Figure 4.7b shows the normalized energy efficiency per MAC operation for different on-chip buffer to R matrix size ratios for 64 KB and 128 KB on-chip buffer sizes,

respectively. All three approaches observe the improvement with the increase in the on-chip buffer size to matrix size ratio due to a reduction in the control logic execution. TSI-WR performs better than the conventional approach only for higher on-chip buffer to R matrix size ratio. Off-chip memory accesses dominates the overall energy consumption. The proposed SACC approach outperforms the other two approaches for all the cases as it reduces the off-chip memory accesses. For 50% on-chip buffer to matrix size ratio, the SACC approach reduces 43% and 32% energy for 64 KB on-chip buffer and 48% and 30% for 128 KB on-chip buffer size compared to conventional and TSI-WR approach, respectively.

4.6 Summary

Long Short-Term Memory (LSTM) networks are widely used in speech recognition and natural language processing (NLP). With enormous growth in number of Edge AI applications, there is a pressing need of efficient execution of these alogrithms on edge devices. These edge devices use customized accelerators to meet the energy and throughput targets. The key to improving the energy efficiency and throughput of DNN accelerators is to reduce the off-chip memory accesses. This work proposes a novel data reuse approach that reduces the off-chip memory accesses of large weight matrices of RNNs/LSTMs by $\approx$50% and improves the throughput significantly. The proposed approach improves the throughput by 55% and 29% and reduces the energy consumption by 52% and 30% for 12.5% and 50% on-chip buffer to matrix size ratio, for 128 KB on-chip buffer size, compared to the state of art TSI-WR approach.