2018 51st Annual IEEE/ACM International Symposium on Microarchitecture

Towards Memory Friendly Long-Short Term Memory Networks (LSTMs) on Mobile GPUs

1st Xingyao Zhang
ECOMS Lab, ECE Department
University of Houston
Houston, USA
xzhang55@uh.edu

2nd Chenhao Xie
ECOMS Lab, ECE Department
University of Houston
Houston, USA
cxie@uh.edu

3rd Jing Wang
Beijing Adv. Innovation Center for Imaging Tech.
Capital Normal University
Beijing, China
jwang@cnu.edu.cn

4th Weidong Zhang
Beijing Adv. Innovation Center for Imaging Tech.
Capital Normal University
Beijing, China
5591@cnu.edu.cn

5th Xin Fu
ECOMS Lab, ECE Department
University of Houston
Houston, USA
xfu8@central.uh.edu

***Abstract*—Intelligent Personal Assistants (IPAs) with the capability of natural language processing (NLP) are increasingly popular in today's mobile devices. Recurrent neural networks (RNNs), especially one of their forms – Long-Short Term Memory networks (LSTMs), are becoming the core machine learning technique applied in the NLP-based IPAs. With the continuously improved performance of mobile GPUs, local processing has become a promising solution to the large data transmission and privacy issues induced by the cloud-centric computations of IPAs. However, LSTMs exhibit quite inefficient memory access pattern when executed on mobile GPUs due to the redundant data movements and limited off-chip bandwidth. In this study, we aim to explore the memory friendly LSTM on mobile GPUs by hierarchically reducing the off-chip memory accesses. To address the redundant data movements, we propose inter-cell level optimizations that intelligently parallelize the originally sequentially executed LSTM cells (basic units in RNNs, corresponding to neurons in CNNs) to improve the data locality across cells with negligible accuracy loss. To relax the pressure on limited off-chip memory bandwidth, we propose intra-cell level optimizations that dynamically skip the loads and computations of rows in the weight matrices with trivial contribution to the outputs. We also introduce a light-weighted module to the GPUs architecture for the runtime row skipping in weight matrices. Moreover, our techniques are equipped with thresholds which provide a unique tunning space for performance-accuracy trade-offs directly guided by the user preferences. The experimental results show our optimizations achieves substantial improvements on both performance and power with user-imperceptible accuracy loss. And our optimizations exhibit the strong scalability with the increasing input data set. Our user study also shows that our designed system delivers the excellent user experience.**

***Index Terms*—Approximate Computing, GPGPU/GPU, Mobile and Embedded Architectures**

This research is supported by NSF grants CCF-1619243, CCF-1537085(CAREER), CCF-1537062. This research is partially supported by National Natural Science Foundation of China under grants No.61772350, Beijing Noval Z181100006218093.

I. Introduction

Intelligent Personal Assistants (IPAs) with the capability of natural language processing (NLP), such as Apple's Siri [1], Google Assistant [2], and Amazon Alexa [3], have already been heavily used by mobile users, and are expected to become the major workloads in future mobile devices [4]. The core technique to support NLP in IPA applications is artificial neural networks, such as convolution neural networks (CNNs) and recurrent neural networks (RNNs). CNNs have shown the strong ability in achieving high accuracy for image and speech recognition. With the increasing requirements on personalizations, the NLP-based IPAs are expected to understand more complicated requests from humans and perform like a person [5], [6]. These features require capturing the relationship among input samples, which is largely ignored in CNNs. RNNs, especially one of their forms – Long-Short Term Memory networks (LSTMs) [7], show great promise in exploring affiliations among serial samples (details in Section II). Thus, LSTMs are becoming the major technique for NLP in IPAs. Many companies, e.g. Google, Apple, Microsoft, Amazon and Baidu, leverage this advantage of LSTMs in processing the context information for their NLP-based IPA applications, e.g. semantics classification [8], automatic speech recognition [9], [10] and question answering [11].

Traditionally, the computations of NN-based IPA applications are performed in the cloud, which faces large data transmission and privacy issues. With the continuously improved performance of modern mobile devices, localizing these applications becomes promising and has attracted major attentions [12]–[15]. Especially, mobile GPUs have been improved significantly to support parallel computing like neural networks [16]–[21]. They become one of the best candidates to host the computations required by NN-based IPAs such as LSTMs.

978-1-5386-6240-3/18/$31.00 ©2018 IEEE
DOI 10.1109/MICRO.2018.00022

However, we observe that LSTMs exhibit quite inefficient memory access patterns and face two serious memory bottlenecks when executed on mobile GPUs. (1) **Redundant data movements:** as a natural feature of LSTMs, some weight matrices are shared by all the cells (basic units in RNNs, corresponding to neurons in CNNs) in one LSTM layer. And all cells have to be processed *in-order* in each layer due to the context link (i.e., the data dependence) between every two adjacent cells. Given the limited mobile GPUs on-chip storage, this sequential execution causes redundant loads from the off-chip memory for the shared weight matrices across cells. (2) **Limited off-chip bandwidth:** the relatively large working set per LSTM cell also causes severe pressure to the off-chip memory bandwidth. Both these two bottlenecks significantly extend LSTM execution time and cause high power consumption on mobile GPUs. Unfortunately, previous proposed technologies on CNNs cannot effectively address these challenges as LSTMs have completely different computation patterns from CNNs (See detailed comparison between LSTMs and CNNs in Section II-A).

In this study, we aim to explore the memory friendly LSTMs on mobile GPUs, thus, achieving the substantial improvements on both performance and power. We propose the optimizations at both inter-cell and intra-cell levels that address the above two challenges, respectively, to hierarchically reduce the off-chip memory accesses.

At the inter-cell level, we observe an interesting feature of the LSTM layer that the context links between some adjacent cells are weak. We propose to divide the LSTM layer into multiple independent sub-layers at these weak links. Since the weak links are lost between sub-layers, a predicted link is further applied to each sub-layer to recover the accuracy loss. We then parallelize the sub-layers, which enhances the data locality across cells from different sub-layers and substantially reduces the redundant data movements.

At the intra-cell level, unlike CNNs, we observe that some rows in weight matrices have trivial contributions to the cell output. We propose dynamic row skip (DRS) technique to skip the loads and computations of those trivial rows in each LSTM cell with negligible impact on the output accuracy. We introduce a light-weight module to the GPUs architecture. It reorganizes the cooperative thread arrays (CTAs) to dynamically disable the threads which are assigned to process the trivial rows. DRS reduces the number of rows and the size of the weight matrices required by a LSTM cell, hence, effectively relaxing the pressure on the off-chip memory bandwidth.

To our best knowledge, this is the first work to address the memory bottlenecks for LSTMs executing on mobile GPUs. Our proposed techniques gain the substantial performance and power benefits without sacrificing the user-perceptible output accuracy. Moreover, our techniques are equipped with thresholds which provide a unique tunning space for performance-accuracy trade-offs directly guided by the user preferences, leading to the best user experience. The contributions of this paper are as follows:

- We observe that memory is the bottleneck for LSTMs on mobile GPUs. It is mainly caused by the frequent data re-loads across *sequentially* processed LSTM cells, and the large size of the weight matrices per cell.
- We observe the weak context links between some adjacent cells, and leverage this feature to explore the inter-cell level optimizations that intelligently parallelize the processing of the LSTM cells, hence, reducing the data re-loads with user-imperceptible accuracy loss.
- We propose intra-cell level optimizations that dynamically skip the loads and computations of the trivial weight matrix rows with negligible contribution to the outputs. We introduce a light-weighted module to the GPUs architecture for the runtime row skipping in weight matrices.
- The experimental results show that our proposed techniques achieve on average 2.54X (upto 3.24X) performance improvement and 47.23% energy saving on the entire system with only 2% accuracy loss that is generally user imperceptible, comparing with the state-of-the-art LSTM execution on mobile GPUs. And our optimizations exhibit the strong scalability with the increasing input data set. Our user study also shows that our designed system delivers excellent user experiences.

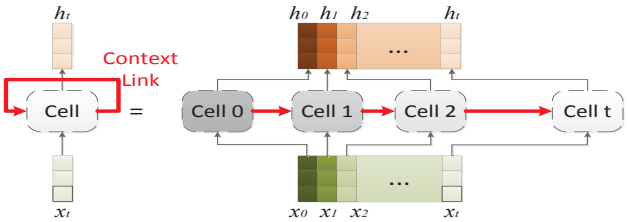
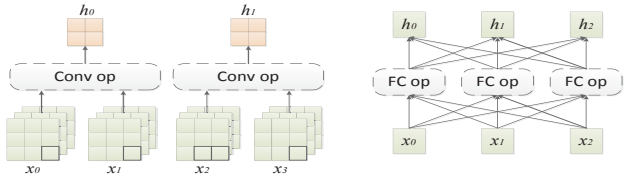


Fig. 1: The schematic of one RNN layer (left) and its unrolled model (right). The cell represents the operations of mapping the inputs to the outputs. In the unrolled model (right), the cell 0 represents cell at timestamp 0 and so on.



(a) Conv layer (b) FC layer

Fig. 2: The schematics of (a) the CNN convolution (Conv) layer, and (b) the CNN fully connected (FC) layer.

II. Background

A. Recurrent Neural Networks

One RNN layer usually contains one cell which integrates the operations of mapping the inputs to the outputs, as shown in Fig.1 (left). The cell produces the outputs **periodically** using not only the activations from the last layer, but also the **historic self-output**, also known as **context link** (highlighted by the red line in Fig.1). This feature helps model the context dependency within the input activations in the sequence modeling tasks (e.g. language modeling tasks). In order to simplify the analysis, the RNN layer can be unrolled into a sequence

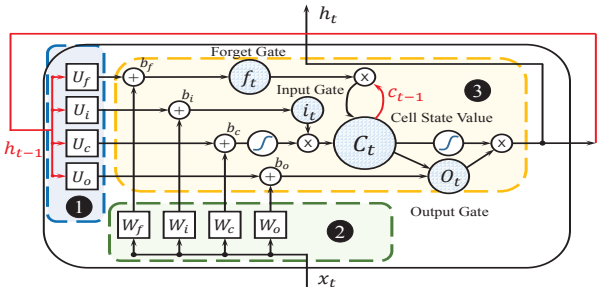


Fig. 3: The LSTM cell schematic.

of cells to represent the cell states at different timestamps, as shown in Fig.1 (right). *To clarify, we focus on the RNN unrolled layer in this study, and a cell in the layer means the unrolled cell at certain timestamp.* Correspondingly, we refer the previous/next cell in the layer as the unrolled cell at the previous/next timestamp.

Even the RNN unrolled layer looks similar to CNN neural network layers, e.g. the convolution (Conv) layer and the fully connected (FC) layer, it has a completely different computation pattern. Fig.2 also demonstrates the schematics of the Conv layer and the FC layer from CNNs for comparison. Firstly, the input formats of these three layers are totally different: the Conv layer processes multiple matrix sets, and the FC layer takes a bunch of single activations as the input while the RNN unrolled layer processes the activation matrix with each cell processing an activation vector. Furthermore, the Conv layer and the FC layer produce the output activations by only using the layer inputs, which are all ready before the layer begins. Thus, the Conv/FC operations of the same layer can be parallelized. However, the operations of each cell in the RNN unrolled layer involve one more dimension besides the layer input and output, which is the context link between the adjacent cells (red line in Fig.1). Therefore, instead of concurrently processing all layer inputs, the RNN layer can only iteratively process partial input activations at each timestamp. In other words, only one vector in an input activation matrix can be processed at a time, and the processing of the following vector should wait until the processing of the previous vector finishes.

B. Long-Short Term Memory Networks (LSTMs)

There are mainly three types of RNNs: Simple RNNs (or vanilla RNNs), Long-Short Term Memory networks (LSTMs) [22] and Gated Recurrent Unit networks (GRUs) [23]. Simple RNNs can hardly connect the useful information between two inputs with the large time interval [24]. LSTMs and GRUs were introduced to address such problem by setting gates inside the RNN cell to filter the information from both the input and the historical self-output, thus only the useful information is well kept through the unrolled cells to enable the long-term "memory". The LSTM cell has more gates than the GRU cell, which increases the computation complexity but ensures a better accuracy. In this paper, we focus on the analysis and optimizations of LSTMs execution on mobile GPUs, the proposed methods can also be applied to GRUs with simple adjustment.

Algorithm 1 The LSTM Execution on Mobile GPUs

```
for each layer in LSTM do
    Kernel Sgemm(W_{f,i,c,o}, x);                          ▷ ❷
    for each cell in layer do
        Kernel Sgemv(U_{f,i,c,o}, h_{t-1});                ▷ ❶
        Kernel lstm_ew(f_t, i_t, c_{t-1}, c_t, o_t, h_t);  ▷ ❸
    end for
end for
```

Fig.3 shows the zoom-in view of one LSTM cell located at the tth timestamp. There are three gates in one LSTM cell: Input Gate i_t, Forget Gate f_t and Output Gate o_t. They help modify the cell state, which "stores" context information over the arbitrary time interval. The LSTM cell takes three inputs: the layer input x_t, the historic self-output h_{t-1}, and the cell state value of the previous cell c_{t-1}; they are all in the form of vector. The cell also has two outputs: the cell state value of the current cell c_t and the output activations h_t; both them are in the form of vector. The following equations represent all the computations within one cell in LSTMs:

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f) \tag{1}$$

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i) \tag{2}$$

$$c_t = f_t \cdot c_{t-1} + i_t \cdot \tanh(W_c x_t + U_c h_{t-1} + b_c) \tag{3}$$

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o) \tag{4}$$

$$h_t = o_t \cdot \tanh(c_t) \tag{5}$$

Eq.1 generates the forget gate f_t which will be applied to the cell state value of the previous cell c_{t-1}. Eq.2 produces the input gate i_t which will be merged into the cell state value of the current cell c_t. Eq.3 updates the old cell state c_{t-1} to a new cell state c_t. Eq.4 and Eq.5 output h_t based on the cell state c_t with the output gate o_t filtering the output information.

C. LSTM Execution on Mobile GPUs

In modern GPUs with strong backend libraries, e.g. cuDNN [25], the above computations within one LSTM cell are divided into three parts, as shown in Fig.3 ❶❷❸.

In ❶, since all the $(U \times h_{t-1})$ functions from Eq.1,2,3, and 4 share the same input h_{t-1}, they are integrated into one matrix-vector multiplication kernel $\boldsymbol{Sgemv(U_{f,i,c,o}, h_{t-1})}$ with weight matrices (U_f, U_i, U_c, U_o) concatenated into an united weight matrix $U_{f,i,c,o}$.

Similarly, in ❷, all the $(W \times x_t)$ computations are combined into an united matrix-vector multiplication kernel $\boldsymbol{Sgemv(W_{f,i,c,o}, x_t)}$. In large-scale GPUs (e.g. Tesla M40 [26]), cells from different layers can be executed in parallel as long as they have no data dependence, e.g. the cell at the jth layer and the $t+1$th timestamp can be parallelized with the cell at $(j+1)$th layer at the tth timestamp. However, such layer level parallelism requires a large-size memory to hold the weight matrices of multiple layers, and can be hardly implemented on mobile GPUs (e.g. Tegra X1 [27]) with limited on-chip storage. As a result, the LSTM layers are processed sequentially on mobile GPUs and the whole layer's

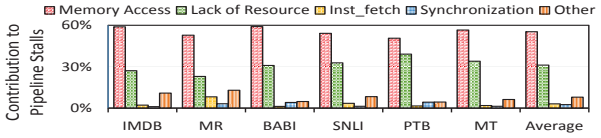


Fig. 4: The contribution of each major factor to the pipeline stall cycles when executing $Sgemv()$.

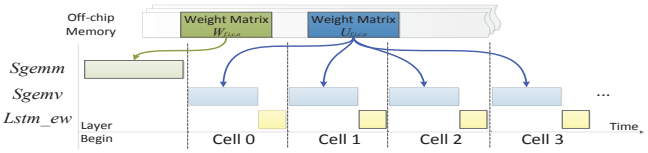


Fig. 5: The sketch map of the kernel execution for the LSTM layer

inputs $x_1 - x_n$ are ready at the beginning of each LSTM layer execution. To gain the matrix multiplication efficiency, the originally independent matrix-vector multiplications per cell shown in ❷ are then transformed to one matrix-matrix multiplication kernel $\boldsymbol{Sgemm(W_{f,i,c,o}, x)}$ per layer.

Finally, the remaining operations of the cell in ❸ are included into one kernel $\boldsymbol{lstm\_element\_wise(f, i, c, o)}$ which consists of adding and activation functions for each individual elements. Algorithm.1 summarizes the state-of-the-art LSTM execution on the mobile GPUs with the strong backend library (e.g. cuDNN).

III. The Memory Bottleneck

Although several optimizations have been made by GPU backend libraries, the LSTM execution on mobile GPUs is still inefficient. In this study, we implement the state-of-the-art LSTM execution on a typical mobile GPU, the Jetson-TX1 board, and observe that kernel $Sgemv$ dominates the overall LSTM execution time (over 90%). We further investigate the GPU pipeline stalls during the $Sgemv$ execution. Several factors can cause the pipeline stall, such as the off-chip memory access, the barrier synchronization and so on. Fig.4 plots the contribution of each major factor to the overall pipeline stall cycles when executing $Sgemv$ kernels (benchmark details are presented in Section.VI-A). As it shows, the off-chip memory access is the major contributor. Besides, previous works [28], [29] find that the off-chip memory accesses are also very expensive for mobile GPUs from the power perspective. In this section, we describe two major memory challenges at both inter-LSTM-cell and intra-LSTM-cell levels that lead to the performance and power bottleneck for the efficient LSTM execution on the mobile GPUs.

A. The Inter-Cell Level Memory Bottleneck: Redundant Data Movements

As illustrated in Algorithm.1❶, the $Sgemv$ kernel is launched per cell when executing one LSTM layer. The united weight matrix $U_{f,i,c,o}$ is then repeatedly requested by the $Sgemv$ kernels across cells at different timestamps in the layer. Unfortunately, the matrix $U_{f,i,c,o}$ exhibits quite poor data locality in the GPUs on-chip storage, leading to the redundant data movements and intensive off-chip memory accesses, as described in Fig.5. This is mainly caused by the unique LSTM execution pattern: as shown in Fig.3❶, the $Sgemv$ kernel at current cell takes h_{t-1} as the input which is data dependent on the previous cell in the same layer. This prevents the $Sgemv$ kernels across cells from being integrated into one matrix-matrix multiplication kernel, which only needs one-time load for the weight matrix and being processed once per layer. As a result, each $Sgemv$ kernel accesses the weight matrix separately. Even worse, the limited on-chip storage fails to hold such large-size weight matrix, causing the frequent loads and evictions for the useful data. We also observe that the size of the actually loaded data is upto 100X larger than the original data size, which indicates the quite in-efficient data re-loads. Moreover, the redundant data movements become severer as the number of cells increases in the layer since adding one cell requires additional loads for the united weight matrix.

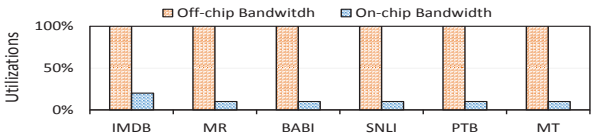


Fig. 6: Utilization of on-chip and off-chip memory when executing $Sgemv()$.

To efficiently minimize the redundant data loads and improve the data locality across cells, we propose the inter-cell level optimization scheme called LSTM layer reorganization. It divides one LSTM layer into multiple parallel sub-layers, cells from different sub-layers become independent and are further combined to enable the reuse on the weight matrix $U_{f,i,c,o}$. More details are described in Section.IV.

B. The Intra-Cell Level Memory Bottleneck: Limited Off-Chip Bandwidth

The $Sgemv$ kernel inside the cell requires to load the united weight matrix ($U_{f,i,c,o}$) with numerous elements. However, the limited off-chip memory bandwidth of mobile GPUs fails to fulfill such high demands. Fig.6 plots both off-chip and on-chip bandwidth utilization during the $Sgemv$ kernel execution. As it shows, the off-chip bandwidth is almost fully utilized, while the on-chip bandwidth is lightly consumed.

To release the off-chip bandwidth limitation, we propose to effectively shrink the input data size for the $Sgemv$ kernel. We explore the dynamic row skip scheme by leveraging the unique computation features of the LSTM cell to dynamically skip the data loads for rows in the united weight matrix with trivial contribution to the final outputs. More details are presented in Section.V.

IV. Inter-Cell Level Optimizations

In this section, we focus on the **inter-cell level** optimizations to enhance the data locality across cells in one LSTM layer.

A. The Irrelevance Between Two LSTM Cells

The sigmoid function (σ) and hyperbolic tangent function ($tanh$) are used as the activation functions for the LSTM cell computations [7]. The sigmoid function takes the input within

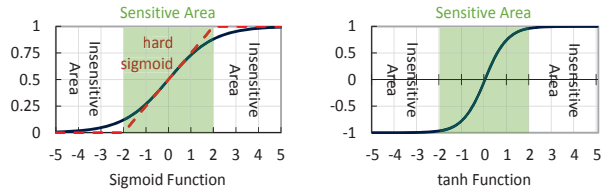


Fig. 7: Sigmoid activation function and tanh activation function

the range of $[-\infty, +\infty]$ and its output is within the range of $[0, 1]$, as shown in Fig.7(a). Interestingly, when the input is in the range of $[-2, 2]$, its output is nearly linear to the input, we refer it as sensitive area, as shown in Fig.7(a); on the other hand, its output is insensitive to the input within the range of $[-\infty, -2]$ and $[2, +\infty]$, we refer it as insensitive area. This is also the case for the $tanh$ function as shown in Fig.7(b). In some neural network frameworks, the sigmoid function is modeled by the hard sigmoid function (shown in Fig.7(a)) to accelerate the computations [30]. The boundaries to partition the sensitive and insensitive areas fit both sigmoid and fast sigmoid functions.

According to Eq.5, the range of the previous cell's output h_{t-1} is $[-1, 1]$ because o_{t-1} is the output of sigmoid function (shown in Eq.4) with the range of $[0, 1]$ and the output of $tanh(c_{t-1})$ is within the range of $[-1, 1]$. As shown in Eq.1, h_{t-1} is also the input data for the current cell which will be multiplied to the matrix U_f, and the range of the multiplication outputs can be derived once U_f is known. Moreover, the range of the sigmoid function's input in Eq.1 can further be derived once $(W_{f,i,c,o} \times x_t)$ is finished at the beginning of the layer processing. And when the range of the input is $[2, +\infty]$, one would easily tell that the output (i.e. f_t in Eq.1) is always close to 1 based on the feature of sigmoid function discussed above. In other words, the output f_t is irrelevant to the h_{t-1} values in this case. The similar derivation can be applied to Eq.2,3,4, and their outputs i_t, c_t, and o_t are irrelevant to the h_{t-1} values as well. To summarize, when $U_{f,i,c,o}$, $(W_{f,i,c,o} \times x_t)$, and $b_{f,i,c,o}$ are known, given that h_{t-1} is within the range of $[-1, 1]$, the range of $(W_{f,i,c,o}x_t + U_{f,i,c,o}h_{t-1} + b_{f,i,c,o})$ can be derived, and if it falls in insensitive area, the previous cell's output h_{t-1} can be considered as irrelevant to f_t, i_t, c_t, and o_t, thus, having no impact to the current cell's computation. In other words, there is no context link between these two cells.

B. LSTM Layer Division

Based on the above observation that the context links between every two cells are not uniform throughout the LSTM layer, we propose the LSTM layer division scheme which breaks the context link between cells with no or quite weak link so that a LSTM layer is divided into multiple independent sub-layers, as shown in Fig.8(a1). This opens the door for sub-layer parallelization and reducing the data reloads which will be explored in Section IV-C. Though it is weak, the context link is lost between sub-layers which may affect the final output accuracy, a predicted context link (Fig.8(a2)) is further applied to the first cell of each sub-layer (except the first sub-layer) to recover the accuracy.

Algorithm 2 Relevance Value Acquisition

Input: Hidden Layer Size $\boldsymbol{Dim}$; Weight Matrices $\boldsymbol{U_f, U_i, U_c}$ and $\boldsymbol{U_o}$; Output Vectors $\boldsymbol{X'_f, X'_i, X'_c}$ and $\boldsymbol{X'_o}$ from the matrix multiplications (i.e., $W_f x_t$, $W_i x_t$, $W_c x_t$, $W_o x_t$); Offset Vectors $\boldsymbol{b_f, b_i, b_c}$ and $\boldsymbol{b_o}$
Output: Relevant Value $\boldsymbol{S}$

1: $S \leftarrow 0$; ▷ initial the relevance value
2: $D_{f,i,c,o} \leftarrow sum(abs(U_{f,i,c,o}))$;
3: **for** $j \in [0, Dim - 1]$ **do**
4: $\quad S^j_f \leftarrow min(4, max(X'^{j}_f + b^j_f + D^j_f + 2, 0))$;
5: $\quad S^j_{i,c,o} \leftarrow min\{\left[2 + min(2, abs(X'^{j}_{i,c,o} + b^j_{i,c,o}))\right], \left[min(2, 2 + D^j_{i,c,o} - max(2,\ abs(X'^{j}_{i,c,o} + b^j_{i,c,o})))\right]\}$;
6: $\quad S^j \leftarrow S^j_o \times (S^j_f + S^j_i \times S^j_c)$;
7: $\quad S \leftarrow S + S^j$;
8: **end for**
9: **return** S;

Breakpoints Search: Theoretically, breaking the context link between two irrelevant cells has no impact on the output accuracy. However, in most cases, it is hard to find two consecutive cells in the layer that are completely irrelevant. Breaking the weak links becomes the main target for the LSTM layer division scheme, and quantitatively justifying the relevance between two cells is the first step towards finding the weak context links for the breakpoints. In this study, we introduce the relevance value S to describes the impact of precedent cell's output on current cell. A smaller S implies a weaker link between the cells and "0" means totally irrelevant.

Algorithm.2 calculates S for the link between the precedent and current cells. In line 2, the range $[-D, D]$ is computed for each element in the output vector of matrix multiplication $(U_{f,i,c,o} \times h_{t-1})$ in current cell. In line 4-5, the computed ranges, values from the offset vector $b_{f,i,c,o}$, and the values from the output vector of matrix multiplications $(W_{f,i,c,o} \times x_t)$ in current cell are used to calculate the range of input values for the activation function. The range is then compared with the sensitive area to measure the overlapping value between them. Since there are multiple activation functions in each LSTM cell, in line 6, the range overlapping values for all activation functions are combined to calculate S for one input element. Finally in line 7, Ss for all elements in the input vector are summed up to derive the overall S for current cell.

At the beginning of each LSTM layer, the relevance value Ss for each two cells are computed since our algorithm does not take any timestamp-based value. Then each S value will be compared with a relevance threshold α_{inter} to determine the weak context links for current LSTM layer: if S is lower than the threshold, the two cells are considered as weakly linked which will be selected as the breakpoint.

Accuracy Recovery: We use a pre-determined vector to predict all the context links lost at the breakpoints (one context link is one vector). Although the predicted vector is not quite accurate when applying to all breakpoints, it can well recover the application output accuracy since the weak context links

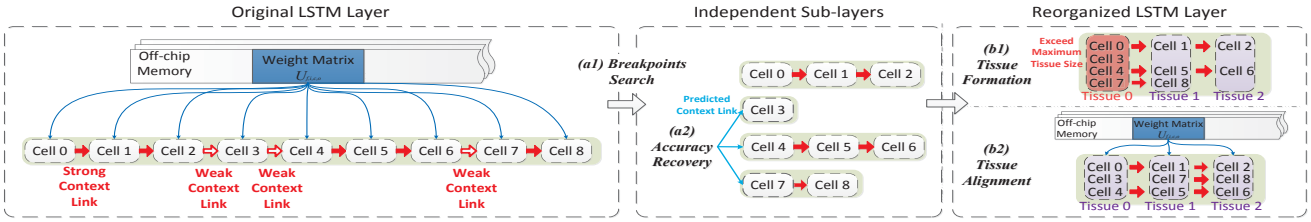


Fig. 8: The overview of the inter-cell level optimization.

have relatively small impact on the application output and are insensitive to a small prediction error.

We predict the weak context links by analyzing the distribution of a large set of context links which are collected through executing LSTMs offline with large training datasets. Note that we study the distribution of all context links since the weak context links share quite similar distribution pattern with strong context links. It is unnecessary to particularly focus on the weak context links which vary with the relevance threshold. Since the context link is in the form of vector, the value distribution for each element will be collected, and the expectation of each element in the weak context link can be achieved by the following equation:

$$\overline{h_j} = \sum_{i=0}^{n} h_j(i) \times \rho_{ij} \tag{6}$$

where $\overline{h_j}$ is the expectation for jth element in the context link, and ρ_{ij} represents the possibility for the distribution of the jth element in the context link. The expectations of all the elements compose a vector which is the predicted context link at the breakpoints.

C. LSTM Layer Reorganization

Tissue Formation: Given the independent LSTM sub-layers, we parallelize them via fusing cells from the sub-layers into tissues. One cell will be selected per sub-layer, and the selected cells together form a tissue. For example, in Fig.8, the LSTM layer is divided into four sub-layers, cells 0, 3, 4, and 7 from them, respectively, are combined into one tissue; and the next cells from these sub-layers which are only cells 1, 5, and 8 from the first three sub-layers as the second sub-layer only contains cell 3, are combined into another tissue. As Fig.8 shows, the LSTM layer is transformed into a sequence of tissues. The cells inside each tissue will be executed concurrently. Note that the data dependency across cells in each sub-layer still maintains which is treated as the data dependency across tissues.

In this study, we define the number of the cells per tissue as the tissue size. Ideally, when there are more sub-layers and more cells are fused into a tissue, there will be fewer tissues in the layer and thus, fewer re-loads for the weight matrices and performance are improved as well. However, we observe that keeping increasing the tissue size would even hurt the performance. Fig.9 demonstrates the normalized performance of one LSTM layer as the tissue size increases when executing the investigated benchmarks (The baseline case introduced in Section VI-A.). As it shown, the performance first increases with the increasing tissue size, and then drops when the tissue

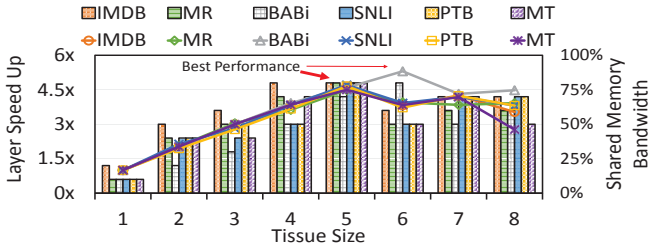


Fig. 9: The normalized performance of one LSTM layer and shared memory bandwidth utilization as the tissue size increases.

size exceeds a certain number (e.g. 6 for BABI benchmark, 5 for the others in Fig.9). We define this number as the maximum tissue size (MTS).

The performance drop is caused by the limited on-chip bandwidth (i.e. shared memory bandwidth) of the mobile GPUs. Fig.9 also plots the utilization of the shared memory bandwidth. As it shows, the bandwidth utilization increases with the increasing tissue size, and it approaches to 100% at the MTS. Further increasing the tissue size would cause the kernel re-configuration at the compilation time to ensure that the on-chip bandwidth utilization is below 100%. The re-configuration reduces the on-chip bandwidth requirements per thread but increases the thread amount in the kernel. As a result, the execution time per tissue significantly increases which could not be well compensated by the saved time on the reduced matrix re-loads, leading to the overall performance droop. Note that the MTS is determined by the GPU configurations, a framework is needed to dynamically implement the LSTM layer reorganization scheme for various LSTM layer configurations on different mobile GPUs.

Tissue Alignment: Since the tissue formation mechanism simply combines multiple cells into tissues but ignores the MTS, it may generate both fat and thin tissues. Fat tissues have more cells than MTS (e.g. Tissue 0 in Fig8(b1) as MTS is 3 in this example) leading to the over-utilized share-memory bandwidth, while thin tissues have quite few cells (e.g. Tissue 2 in Fig8(b1)) and are unable to effectively reuse the weight matrix. Both will affect the performance boost. To maximize the performance, we further explore the tissue alignment mechanism to well balance the tissue size by moving cells from the fat tissues to thin tissues, e.g. moving cell7 and 8 from Tissue0 and Tissue1 to Tissue1 and Tissue2, respectively, in Fig8(b1). Note that tissue alignment does not further break any context link and ensures every tissue size is below or equal to the MTS.

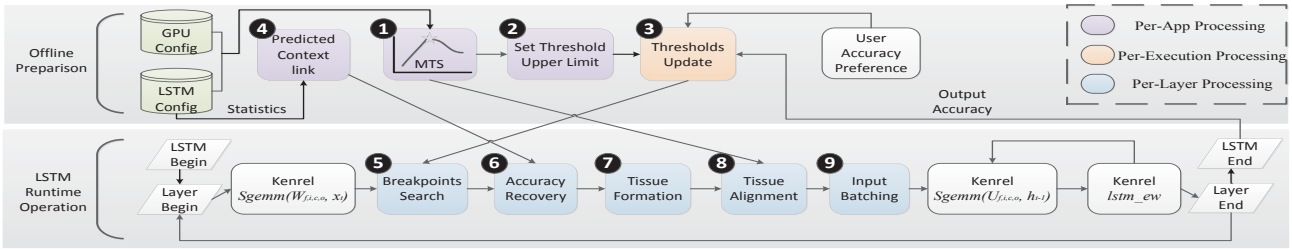


Fig. 10: The implementation of the inter-cell level optimizations.

D. The Implementation

Offline Operations: We first execute LSTMs on the target GPUs platform with various tissue sizes to determine the MTS (Fig.10❶). Ideally, when the tissue size is MTS for every tissue in the layer, the number of tissues for this layer is minimized, leading to the maximal performance. Therefore, the minimal number of tissues can be conducted by:

$$N_{min} = \left\lceil \frac{N_{origin}}{MTS} \right\rceil \tag{7}$$

where N_{origin} is the original number of LSTM cells in the layer. Next, we execute LSTMs equipped with our optimizations to obtain a value for the relevance threshold α_{inter} which leads to N_{min} number of tissues. This value is set as the upper limit for α_{inter} (Fig.10❷). Then, α_{inter} is initialized to its upper limit aiming to the best performance. Since the accuracy loss may be considerable when the system gains the best performance, α_{inter} will be adjusted per each execution of the application given the accuracy difference between the user preferred accuracy and the application output accuracy, thus, leading to the optimal performance-accuracy trade-offs from the user perspective (Fig.10❸). Furthermore, the predicted context link is produced based on the LSTM configurations (Fig.10❹). Note that the operations ❶❷❹ are determined by the GPU and LSTM configurations and only processed once per application.

Runtime Operations: At the LSTM runtime, after performing the per-layer multiplication kernel $Sgemm(W_{f,i,c,o}, x)$, the breakpoints search (Fig.10❺) and the accuracy recovery (Fig.10❻) are triggered to break the layer into a set of sub-layers, which will be further transformed into a set of tissues with balanced tissue size (Fig.10❼❽). In each tissue, the cells are concurrently processed by batching their input vectors h_t into a united input matrix H_t, and the input state vectors c_t are batched into one matrix C_t as well (Fig.10❾). Correspondingly, the originally per-cell matrix-vector multiplication $Sgemv(U_{f,i,c,o}, h_t)$ kernels are now combined into one per-tissue matrix-matrix multiplication $Sgemm(U_{f,i,c,p}, H_t)$ kernel. The weight matrix $U_{f,i,c,o}$ is effectively re-used by all the cells within the tissue, and its loading frequency reduces from one per cell to one per tissue.

V. Intra-Cell Level Optimizations

As illustrated in Section.III, besides the redundant data movements across cells, the off-chip memory bandwidth is the other major performance limitation for each LSTM cell. Since weight matrix $U_{f,i,c,o}$ is the largest input data for one cell, it is important to shrink it, hence addressing the bottleneck inside the LSTM cell. In this section, we focus on the **intra-cell level** optimization and effectively reducing the data loads per cell. Generally, a common mechanism to shrink the weight matrix size is targeting at each weight element and erasing the near-zero ones [31]. Noticing that weights in LSTM are processed in the row order, and especially, the elements from different rows are totally irrelevant. We leverage this unique feature to propose the row-level weight compression technique, called dynamic row skip, which compacts weights in the matrix at the row level without affecting the output accuracy.

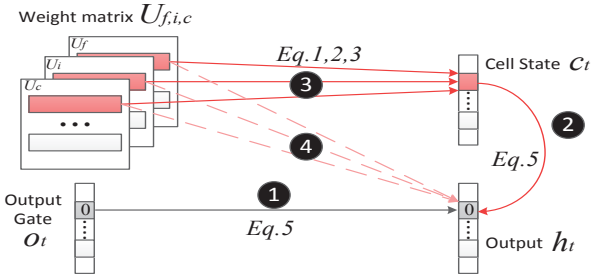


Fig. 11: Irrelevant rows in weight matrix $U_{f,i,c}$ that have trivial impact on cell output h_t.

A. Dynamic Row Skip

As an interesting observation, we find that some rows in the weight matrix $U_{f,i,c}$ have trivial contribution to the cell output vector h_t. This is because h_t is strongly affected by the output gate o_t. As shown in Eq.5, if one element in o_t is near zero, the corresponding output element in h_t will become near-zero (Fig.11❶) no matter what value the corresponding element in c_t is (Fig.11❷). Furthermore, since c_t is calculated based on the weight matrices U_f, U_i, U_c as shown in Eq.1,2,3 (Fig.11❸), the corresponding rows in these matrices can be treated as irrelevant to the final output element in h_t (Fig.11❹).

We propose the Dynamic Row Skip (DRS) scheme to dynamically skip those irrelevant rows in the weight matrices U_f, U_i, U_c whose computations have trivial contribution to the cell output vector h_t. By doing this, the skipped rows will not be loaded and their computations are ignored as well, leading to both performance and energy optimizations. Note that the DRS will also affect the state vector c_t, some of its elements corresponding to the skipped rows will be approximated to zero. However, this impact is observed to be quite limited to the overall accuracy since the next cell will use the forget gate

Algorithm 3 LSTM Computation Flow with DRS

1: **for** *each layer in LSTM* **do**
2: $Kernel\ \boldsymbol{Sgemm}(W_{f,i,c,o}, x)$;
3: **for** *each cell in layer* **do**
4: $Kernel\ \boldsymbol{Sgemv}(U_o,\ h_{t-1})$;
5: $Kernel\ \boldsymbol{lstm\_ew}(o_t)$;
6: $Kernel\ \boldsymbol{DRS}(o_t,\ \alpha_{intra},\ R)$
7: $Kernel\ \boldsymbol{Sgemv}(U_{f,i,c},\ h_{t-1},\ R)$;
8: $Kernel\ \boldsymbol{lstm\_ew}(f_t, i_t, c_{t-1}, c_t, h_t)$;
9: **end for**
10: **end for**

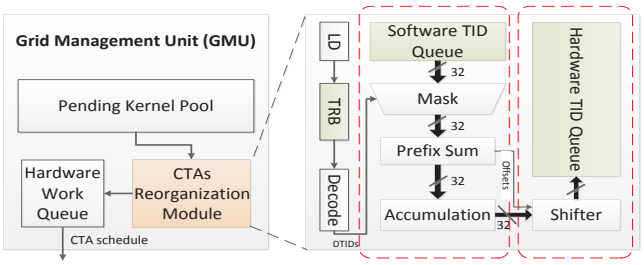


Fig. 12: The architecture of CTAs reorganization module.

f_{t+1} to filter the state vector from previous cell as shown in Eq.3. Unlike the traditional weight pruning methods performed offline [31], DRS is conducted at runtime for each LSTM cell as it requires the latent information, and the rows to be skipped vary across different LSTM cells.

B. The Implementation

In DRS, the rows to be skipped is determined by the latent vector o_t. In other words, only when the near-zero elements in the o_t are available, the corresponding rows in weight matrices U_f, U_i and U_c can be identified and skipped in the cell execution. This requires the modification of the computation flow for the LSTM cell to generate o_t before processing the weight matrices U_f, U_i, and U_c. We split the $Sgemv(U_{f,i,c,o}, h_{t-1})$ kernel into two kernels: one multiplies weight matrix U_o with the input vector h_{t-1}, and the other multiplies the weight matrix $U_{f,i,c}$ with h_{t-1}.

Algorithm 3 illustrates the reorganized computation flow by the DRS. In each cell, $Sgemv(U_o, h_{t-1})$ kernel will be launched first (line 4) followed by the $lstm\_ew(o_t)$ kernel to compute the latent vector o_t (line 5). Then, each element in o_t is compared with a near-zero threshold (α_{intra}) in our $DRS(o_t, \alpha_{intra}, R)$ kernel to obtain the trivial rows whose ID are saved as the list R. (line 6). Next, $Sgemv(U_{f,i,c}, h_t, R)$ kernel will be launched to perform matrix multiplication $U_{h,i,c} \times h_t$ with the trivial rows in $U_{f,i,c}$ disabled, which are indicated by R (line 7). Finally, the remaining computations in the cell are finished by $lstm\_ew(f_t, i_t, c_{t-1}, c_t, h_t)$ (line 8). Note that the near-zero threshold is also adjusted based on the user preferred accuracy requirement to deliver the user satisfied performance-accuracy trade-offs.

Hardware Design: Even our DRS can be implemented by the pure software method, i.e. assigning different operations for trivial and non-trivial rows, it causes branch divergence during the GPU execution and decreases the warp efficiency

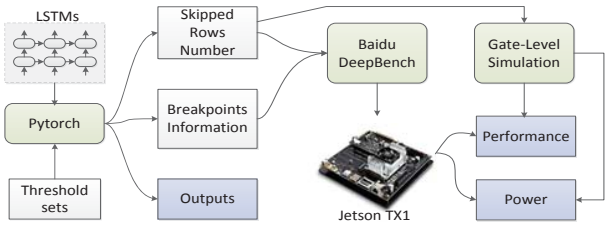


Fig. 13: The evaluation diagram for our optimizations.

TABLE I: Platform Specifications

Hardware	Specification
System	Tegra X1 SoC
CPU	Cortex-A57 +Cortex-A53
Memory	4GB LPDDR4, 25.6GB/s
GPU	Maxwell, 256 Core, 998MHz

and the system performance. We propose a hardware design to support the DRS, which introduces CTAs reorganization module (CRM) to the grid management unit (GMU), as illustrated in Fig.12. The CRM is able to identify the threads assigned to process the trivial rows and re-organize the CTAs to skip them. After a kernel being launched, its information (e.g. kernel name and argument number) can be acquired through the initialization. And a kernel with additional argument (i.e., R) implies containing trivial rows. It will then be assigned to CRM for CTAs re-organization.

Once a kernel enters into the CRM, a load module (LD) first loads and saves the trivial rows IDs to the trivial rows buffer (TRB), and then the disabled thread IDs (DTIDs) can be decoded based on the trivial rows IDs and the grid configurations. Note that there are two kinds of thread IDs during the GPU kernel execution, one is the software thread ID (STID) within a kernel launch, and the other is the hardware thread ID (HTID) that indicates the hardware thread slot assigned to the software thread. Since some threads will be disabled during the kernel execution, there will be an offset between the STID and HTID for each thread. Next, each thread's STID in the kernel is filtered by the DTID and sent to the prefix sum to determine the offset, which is then used to sort and shift the STID to acquire the correct HTID. This HTID acquisition process is conducted at the unit of 32 threads, which is the warp size that is usually divisible by the CTA size. It is further partitioned into two stages shown by the dash boxes in Fig.12 for the pipelined process in the CRM. Finally, the re-organized CTAs will be sent to hardware work queue and wait to be issued.

VI. Evaluation

A. Experimental Setup

In this work, we leverage a software-hardware cooperated method to evaluate our optimizations for the LSTM execution. From the software side, we employ PyTorch [32] which is a popular open-source machine learning framework that supports the dynamic computation graphs; we also use Baidu DeepBench [33], a tool to benchmark the operations of deep learning on the hardware platform. From the hardware side, considering that the current GPU architecture simulators (e.g. GPGPU-Sim [34]) cannot support the latest backend libraries

TABLE II: The state-of-the-art NLP applications investigated in our study

Name	Abbr.	Hidden_Size	Layers	Length
IMDB [37]	SC	512	3	80
MR [38]	SC	256	1	22
BABI [11]	QA	256	3	86
SNLI [39]	ET	300	2	100
PTB [40]	LM	650	3	200
MT [41]	MT	500	4	50

for machine learning (e.g. cuDNN [25] and cuBLAS [35]), we employ the Jetson Tegra-X1 develop kit [36], a representative mobile GPU development board with the configurations listed in Table I.

Fig.13 illustrates the evaluation diagram. Both our inter- and intra-level optimizations require data approximation that affects the output accuracy, and computation flow change that affects the performance and power. The data approximation can be implemented on PyTorch to obtain the accuracy results. The computation flow changes can hardly be implemented on PyTorch as the latest GPU machine learning backend libraries (i.e. cuDNN) are released as pre-compiled binaries. We thus use PyTorch, DeepBench, and real GPU cooperated method to evaluate our techniques on performance and power. We first use PyTorch to produce the breakpoints information for the inter-level optimizations and the number of trivial rows for the intra-level optimizations. These informations will then be sent to the DeepBench to simulate the LSTM execution with our optimizations on the Jetson Tegra-X1 board. We obtain both performance and energy results from the board, note the obtained energy result describes the energy consumption of the overall system including CPU, GPU, etc. To consider the performance and power overheads caused by our hardware design, we model it via the gate-level simulation and include the overheads into our results.

Benchmarks: We employ 6 state-of-the-art NLP Apps listed in Table II as the LSTM benchmarks. Each App has the unique LSTM configurations, where the $Hidden\_Size$ indicate the weight matrix size and the $length$ indicates the number of cells per LSTM layer. IMDB [37] and MR [38] perform sentiment classification (SC) that predict the positive or negative attitude of texts. BABI [11] performs question answering (QA) for automatic text understanding and reasoning. SNLI [39] is a collection of 570k human-written English sentence pairs manually labeled for balanced classification with the labels entailment (ET). PTB [40] is used for word-level language modeling (LM). And MT [41] performs the English to French translation (MT).

In general, 2% accuracy loss is imperceptible to the end users. We first fix the user preferred accuracy requirement as 98% when evaluating the performance and energy improvements gained by our techniques. We also conduct the user study by tunning the accuracy requirement per each individual user.

B. The Effectiveness of the Overall System

Fig.14 plots the performance speed-up and the energy savings obtained by our inter-cell level optimizations, intra-cell level optimizations, and the overall system with the combined optimizations. The results are normalized to the baseline case that executing the state-of-the-art LSTMs on mobile GPUs.

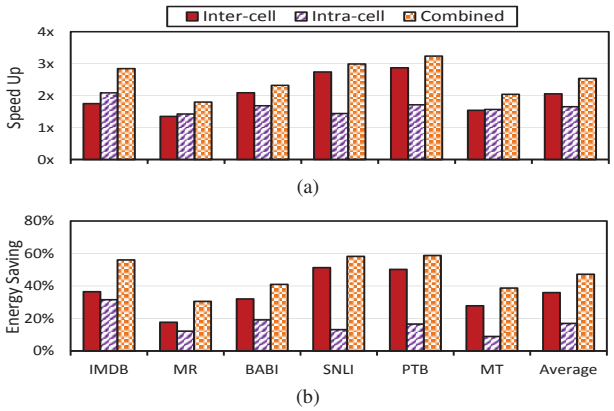


Fig. 14: The (a) speedup and (b) energy saving achieved by our system when applying inter-cell level optimizations, intra-cell level optimizations and the overall system with the combined optimizations.

1) Inter-cell Level Optimizations: On average, the inter-cell level optimizations achieve $2.05X$ speed up and 35.94% energy saving compared with the baseline case. We observe that our techniques show even stronger capability in improving the performance and energy consumptions when the length (i.e. the number of LSTM cells) of the LSTM layer increases. For example, PTB with the longest layer length among all investigated benchmarks achieves the highest performance and energy enhancements. This implies that our techniques well scales with the longer LSTM layer.

We further investigate the effectiveness of our inter-cell optimizations on each LSTM layer, shown in Fig.15. As it shows, our techniques perform better for the earlier (e.g., layer 1) than the latter layers (e.g., layer 3). This is because the context information is closer to the original text inputs, and the context links are more distinct for the earlier layers. They can be divided into more sub-layers for higher performance and energy gains.

2) Intra-cell Level Optimizations: Fig.14 also shows that on average, our intra-cell level optimizations achieve $1.65X$ speed up and 16.93% energy saving compared with the baseline case. We observe that our techniques gain higher performance and energy improvements with the larger weight matrices. For example, PTB with the largest weight matrices among all investigated benchmarks achieves the highest performance and energy saving. In one sentence, our techniques exhibit the strong scalability with the increasing input data set, which is the trend of the NLP-based IPA applications.

We further compare our intra-cell level techniques with the popular weight matrix compression scheme (zero-pruning [31]) and pure software-based DRS, shown in Fig.16. As it shows, the zero-pruning scheme reduces 37% data movements with only 7% power saving and even degrades the performance by 35%, comparing with the baseline case. This is because the zero-pruning scheme prunes the near-zero elements in the weight matrices without considering the possible branch divergences when executing the LSTMs on GPUs. Excitingly,

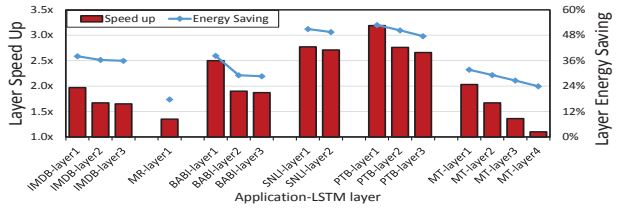


Fig. 15: Per-layer speed up and energy saving when applying the inter-cell level optimizations.

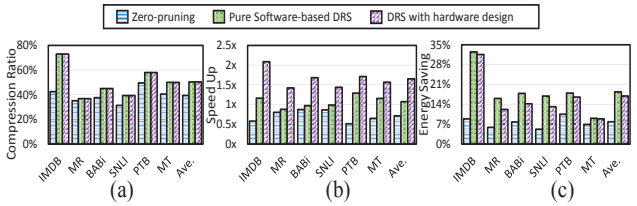


Fig. 16: (a) Weight matrix compression ratio, (b) speed up and (c) energy saving when applying different weight compression schemes

our DRS scheme achieves better weight compression ratio (i.e., on average 50.35%) and better energy saving (i.e., 16.92%) than the zero-pruning scheme. The pure software-based DRS still induces the branch divergence and can only achieve small performance gain $1.07X$ on acerage. With the hardware design to enable the CTAs re-organization, our intra-cell level optimizations maintain the high warp efficiency and achieve additional 57.78% speed up than the pure software method.

3) Putting It All Together: As shown in Fig.14, on average, our system with the combined intra- and inter-level optimizations outperforms the baseline case by $2.54X$ (upto $3.24X$) in performance and 47.23% (upto 58.82%) in energy saving. Note that the improvements gained by the overall system are not the sum of the improvements obtained by each technique as there are some overlaps on the data movements reduction between the two level techniques.

C. Performance-Accuracy Trade-offs

To explore the design space for the performance and accuracy trade-offs, we conduct the sensitivity analysis by tunning the two thresholds applied in our techniques, i.e., α_{inter} and α_{intra}. Note that the energy saving is proportional to the performance boost and we mainly analyze the performance-accuracy trade-offs. For each threshold, we explore 11 values increasing from '0' (representing the baseline case without any accuracy loss) to its maximal value (representing the most aggressive case with the maximal performance boost). We then obtain 11 threshold sets with each set containing a pair of values for α_{inter} and α_{intra}, respectively. Threshold set 0 (10) has the lowest (highest) threshold values. Fig.19 demonstrates the normalized speedup and accuracy when different sets of thresholds are applied for the investigated applications. We denote AO (accuracy oriented) as the threshold set corresponding to the optimizations with user-imperceptible accuracy loss (i.e., 2%). As the figure shows, a higher threshold value leads to a better performance gain, and we denote BPA (best performance-accuracy) as the threshold set leading to the highest Speedup $\times$ Accuracy.

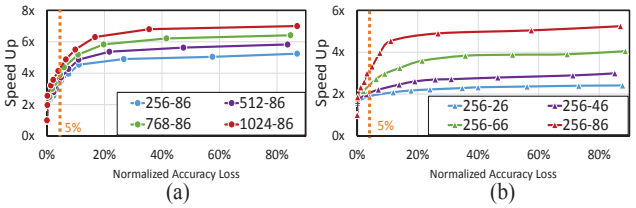


Fig. 17: The performance-accuracy trade-offs of LSTMs for BABI with (a) different hidden unit sizes; (b) different input lengths. Each line represents a configuration of (hidden unit size - input length) pair.

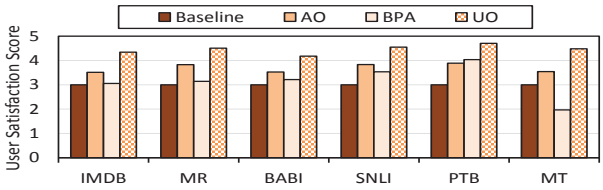


Fig. 18: The user satisfaction score on different schemes

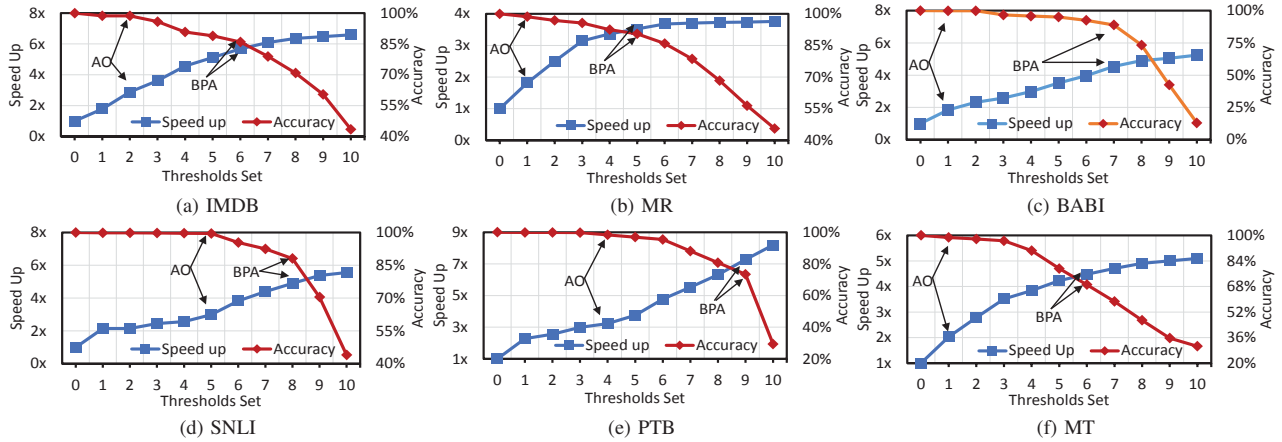
D. Impact of Model Capacity

Model capacity defines the size and input format of the LSTMs, which affects computation scale. To evaluate the impact of model capacity on our techniques, we conduct a sensitivity analysis on performance-accuracy trade-offs of LSTMs given different model capacity (e.g., hidden unit size and input length) when applying our techniques. Fig.18 shows the trade-off results for one representative benchmark BABI due to the space limit. As it shows, given the same accuracy requirement, our techniques achieve higher speedup when the hidden unit size or the input length increases. On the other hand, when the accuracy loss is small (e.g. $<5\%$), the speedup achieved by our technique varies slightly across different hidden unit size and input length. In other words, model capacity has trivial impact on our technique since NLP tasks usually have high accuracy requirement.

E. User Study

Since our techniques require both software and hardware simulations, it is impossible to test on a real product. To evaluate our system impact on the user experience, we build a replay program that provides users the pre-produced outputs (thus, output accuracy) with a response delay (thus, performance) according to the selected thresholds. We compare four schemes: the baseline case, the scheme applying the AO threshold set, the scheme applying the BPA threshold set, and finally, our UO (user oriented) scheme that dynamically adjusts the thresholds by further taking each individual user's preferences as the user input.

We randomly recruit 30 participants on a college campus. We let them experience multiple replays for several NLP applications, and rate the satisfaction score (i.e., 1 being unsatisfied and 5 being most satisfied) based on the output and the response delay. Each participant will be asked to rate 100 replays for each application with the scheme changed every



(a) IMDB (b) MR (c) BABI

(d) SNLI (e) PTB (f) MT

Fig. 19: Performance-Accuracy trade-offs under different sets of thresholds across the different applications.

25 replays. The order of the schemes is random. Fig.18 shows the averaged user satisfaction scores on different schemes. As it shows, AO always achieves better user satisfaction score against the baseline case because the application response time is reduced and the users can not notice the accuracy loss. However, BPA does not achieve good user satisfaction score as most users are not willing to trade much accuracy loss for aggressive performance improvements. Finally, in general, our UO scheme achieves the best users satisfaction score among all the four schemes since it takes the user preferences into the consideration to dynamically tune the threshold for the excellent user experience.

F. Overhead Analysis

The inter-cell level optimizations introduce some light-weight computations, causing only 2.23% performance and 1.65% power overheads on average. The intra-cell level optimizations modify the LSTM computation flow and introduce some extra computations at the software side, which cause 3.39% performance and 3.21% power overheads on average. At the hardware side, the CTAs reorganization module is mainly composed of simple logic gates which only causes 1.47% performance and <1% power overheads based on our gate-level simulations.

VII. Related Works

There have been multiple studies on CNNs optimizations. Some of them target at CNNs optimizations on mobile GPUs [16], [18]. while others design the ASIC accelerators for high performance neural netwroks [42], [43]. Also several studies have been well conducted on the weight compression for CNNs via erasing trivial elements [19], [20], [44]–[46]. And the execution-efficiency-aware weight matrices compression for CNNs are well studied by [13], [47]–[49]. For example, [13] proposes DeftNN to compress the CNNs weight matrix by eliminating columns, [47] explores the node pruning for CNNs. Since the execution patterns of LSTMs are far different from CNNs, these works are not applicable to the LSTMs.

There are also some works on RNNs computation optimizations. [50], [51] propose the scheme to eliminate memory bandwidth pressure of uploading recurrent weights on-chip. However, these optimizations can hardly be implemented on mobile device as the limited on-chip storage of mobile GPUs can not eliminate the redundant data accesses. Besides, [52], [53] explore the accelerator design for high performance RNNs execution. Our work focus on mobile GPUs which are more flexible to process various applications with different LSTMs configurations.

Several studies have exploited optimizations on computation flow [54]–[57]. These works leverage the computation characteristics of their applications to explore the parallelism. However, none of these works can be directly applied to the layer processing of LSTMs. Our work is the first work to explore the parallelism inside each LSTM layer via analyzing the unique mathematical characteristics of LSTM cell computations.

VIII. Conclusion

NLP-based IPAs become increasingly popular in mobile devices. Their core machine learning technique is RNNs, especially LSTMs. With the continuously improved performance of mobile GPUs, local processing has become a promising solution to the large data transmission and privacy issues induced by the cloud-centric computations of IPAs. However, LSTMs exhibit quite inefficient memory access pattern when executed on mobile GPUs due to the redundant data movements and limited off-chip bandwidth. In this study, we propose two level optimizations to hierarchically explore the memory friendly LSTM on mobile GPUs, thus, achieving the substantial improvements on both performance and power. At the inter-cell level, we propose LSTM layer division and reorganization techniques to greatly improve the data locality across cells. At the intra-cell level, we propose dynamic row skip (DRS) techniques to conduct dynamic row-level weight matrix compression. Based on our experiment results, our proposed techniques achieve on average $2.54X$ (upto $3.24X$) performance improvement and 47.23% energy saving on the entire system with only 2% accuracy loss that is generally user imperceptible, comparing with the state-of-the-art LSTM execution on mobile GPUs. And our optimizations have the strong scalability in dealing with the increasing size of input data. Our user study also shows that our designed system delivers excellent user experiences.

REFERENCES

[1] "Apple Siri." https://www.apple.com/ios/siri/.

[2] "Google Assistant." https://assistant.google.com/.

[3] "Amazon Alexa." https://developer.amazon.com/alexa.

[4] M. Nagappan and E. Shihab, "Future trends in software engineering research for mobile apps," in Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on, vol. 5, pp. 21–32, IEEE, 2016.

[5] P. Cohen, A. Cheyer, E. Horvitz, R. El Kaliouby, and S. Whittaker, "On the future of personal assistants," in Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems, pp. 1032–1037, ACM, 2016.

[6] D. Gustafson and C. Danson, "Personality-based intelligent personal assistant system and methods," July 12 2016. US Patent 9,390,706.

[7] F. A. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: Continual prediction with lstm," 1999.

[8] "Google Cloud Natural Language API." https://cloud.google.com/natural-language/.

[9] A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates, et al., "Deep speech: Scaling up end-to-end speech recognition," arXiv preprint arXiv:1412.5567, 2014.

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

[11] J. Weston, A. Bordes, S. Chopra, A. M. Rush, B. van Merriënboer, A. Joulin, and T. Mikolov, "Towards ai-complete question answering: A set of prerequisite toy tasks," arXiv preprint arXiv:1502.05698, 2015.

[12] M. Song, K. Zhong, J. Zhang, Y. Hu, D. Liu, W. Zhang, J. Wang, and T. Li, "In-situ ai: Towards autonomous and incremental deep learning for iot systems," in High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on, pp. 92–103, IEEE, 2018.

[13] P. Hill, A. Jain, M. Hill, B. Zamirai, C.-H. Hsu, M. A. Laurenzano, S. Mahlke, L. Tang, and J. Mars, "Deftnn: addressing bottlenecks for dnn execution on gpus via synapse vector elimination and near-compute data fission," in Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 786–799, ACM, 2017.

[14] "Apple Core ML." https://developer.apple.com/documentation/coreml.

[15] "Google launch Tensorflow Lite." https://techcrunch.com/2017/05/17/googles-tensorflow-lite-brings-machine-learning-to-android-devices/.

[16] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," in Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 615–629, ACM, 2017.

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

[18] M. Song, Y. Hu, H. Chen, and T. Li, "Towards pervasive and user satisfactory cnn across gpu microarchitectures," in High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on, pp. 1–12, IEEE, 2017.

[19] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in Proceedings of the 43rd International Symposium on Computer Architecture, pp. 267–278, IEEE Press, 2016.

[20] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network," in Proceedings of the 43rd International Symposium on Computer Architecture, pp. 243–254, IEEE Press, 2016.

[21] E. Nurvitadhi, J. Sim, D. Sheffield, A. Mishra, S. Krishnan, and D. Marr, "Accelerating recurrent neural networks in analytics servers: comparison of fpga, cpu, gpu, and asic," in Field Programmable Logic and Applications (FPL), 2016 26th International Conference on, pp. 1–4, IEEE, 2016.

[22] S. Hochreiter and J. Schmidhuber, "Lstm can solve hard long time lag problems," in Advances in neural information processing systems, pp. 473–479, 1997.

[23] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," arXiv preprint arXiv:1406.1078, 2014.

[24] S. Hochreiter, "Untersuchungen zu dynamischen neuronalen netzen," Diploma, Technische Universität München, vol. 91, p. 1, 1991.

[25] "cuDNN 5.1." https://developer.nvidia.com/cudnn/.

[26] "Tesla M40 Product Brief." http://images.nvidia.com/content/tesla/pdf/tesla-m40-product-brief.pdf/.

[27] "Tegra X1 Product Brief." http://www.nvidia.com/object/tegra-x1-processor.html/.

[28] S. Tarkoma, M. Siekkinen, E. Lagerspetz, and Y. Xiao, Smartphone energy consumption: modeling and optimization. Cambridge University Press, 2014.

[29] J. M. Arnau Montañés, "Energy-efficient mobile gpu systems," 2015.

[30] "Theano Neural Network Optimizations." http://deeplearning.net/software/theano/library/tensor/nnet/nnet.html/.

[31] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," arXiv preprint arXiv:1510.00149, 2015.

[32] A. Paszke, S. Gross, S. Chintala, and G. Chanan, "Pytorch: Tensors and dynamic neural networks in python with strong gpu acceleration," 2017.

[33] "Baidu DeepBench." https://svail.github.io/DeepBench/.

[34] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on, pp. 163–174, IEEE, 2009.

[35] "cuBLAS." http://docs.nvidia.com/cuda/cublas/index.html.

[36] "NVidia Jetson-TX1 development board." http://www.nvidia.com/object/embedded-systems-dev-kits-modules.html/.

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

[38] B. Pang and L. Lee, "Seeing stars: Exploiting class relationships for sentiment categorization with respect to rating scales," in Proceedings of the ACL, 2005.

[39] S. R. Bowman, G. Angeli, C. Potts, and C. D. Manning, "A large annotated corpus for learning natural language inference," arXiv preprint arXiv:1508.05326, 2015.

[40] M. P. Marcus, M. A. Marcinkiewicz, and B. Santorini, "Building a large annotated corpus of english: The penn treebank," Computational linguistics, vol. 19, no. 2, pp. 313–330, 1993.

[41] "Tatoeba." https://tatoeba.org.

[42] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in Proceedings of the 44th Annual International Symposium on Computer Architecture, pp. 27–40, ACM, 2017.

[43] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," IEEE Journal of Solid-State Circuits, vol. 52, no. 1, pp. 127–138, 2017.

[44] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: ineffectual-neuron-free deep neural network computing," in Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on, pp. 1–13, IEEE, 2016.

[45] M. Rhu, M. O'Connor, N. Chatterjee, J. Pool, Y. Kwon, and S. W. Keckler, "Compressing dma engine: Leveraging activation sparsity for training deep neural networks," in High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on, pp. 78–91, IEEE, 2018.

[46] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size," arXiv preprint arXiv:1602.07360, 2016.

[47] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing dnn pruning to the underlying hardware parallelism," in Proceedings of the 44th Annual International Symposium on Computer Architecture, pp. 548–560, ACM, 2017.

[48] S. Narang, E. Elsen, G. Diamos, and S. Sengupta, "Exploring sparsity in recurrent neural networks," arXiv preprint arXiv:1704.05119, 2017.

[49] S. Narang, E. Undersander, and G. Diamos, "Block-sparse recurrent neural networks," arXiv preprint arXiv:1711.02782, 2017.

[50] G. Diamos, S. Sengupta, B. Catanzaro, M. Chrzanowski, A. Coates, E. Elsen, J. Engel, A. Hannun, and S. Satheesh, "Persistent rnns: Stashing recurrent weights on-chip," in International Conference on Machine Learning, pp. 2024–2033, 2016.

[51] F. Zhu, J. Pool, M. Andersch, J. Appleyard, and F. Xie, "Sparse persistent rnns: Squeezing large recurrent networks on-chip," arXiv preprint arXiv:1804.10223, 2018.
[52] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, et al., "Ese: Efficient speech recognition engine with sparse lstm on fpga.," in FPGA, pp. 75–84, 2017.
[53] C. Gao, D. Neil, E. Ceolini, S.-C. Liu, and T. Delbruck, "Deltarnn: A power-efficient recurrent neural network accelerator," in Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 21–30, ACM, 2018.
[54] T. Mytkowicz, M. Musuvathi, and W. Schulte, "Data-parallel finite-state machines," in ACM SIGARCH Computer Architecture News, vol. 42, pp. 529–542, ACM, 2014.
[55] S. Maleki, M. Musuvathi, and T. Mytkowicz, "Parallelizing dynamic programming through rank convergence," ACM SIGPLAN Notices, vol. 49, no. 8, pp. 219–232, 2014.
[56] A. Subramaniyan and R. Das, "Parallel automata processor," in Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on, pp. 600–612, IEEE, 2017.
[57] M. Looks, M. Herreshoff, D. Hutchins, and P. Norvig, "Deep learning with dynamic computation graphs," arXiv preprint arXiv:1702.02181, 2017.