Enabling Efficient Neural Network Computation Via Hardware And Software Co-Design

by
Xingyao Zhang

A dissertation submitted to the Department of Electrical and Computer Engineering,

Cullen College of Engineering

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in Electrical Engineering

Chair of Committee: Xin Fu

Committee Member: Jinghong Chen

Committee Member: Miao Pan

Committee Member: David Jackson

Committee Member: Xuqing Wu

University of Houston
August 2020

ABSTRACT

In recent years, the neural networks have achieved great successes in the many area, e.g. automotive driving, medical and Intelligent Personal Assistants (IPAs). Among the neural network models, Long-Short Term Memory network (LSTM) and Capsule Network (CapsNet) are popular but exhibit low efficient when processed on the hardware device.

In this dissertation, I introduce two hardware and software co-design approaches to efficiently execute the inference stage of the LSTM and the CapsNet. In the first work, we observe that LSTMs exhibit quite inefficient memory access pattern when executed on mobile GPUs due to the redundant data movements and limited off-chip bandwidth. To address the redundancy, we propose inter-cell level optimizations 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.

In the second work, CapsNet execution is observed low efficiency due to the execution features of their routing procedure, including massive unshareable intermediate variables and intensive synchronizations. we propose the software-hardware co-designed optimizations, *SH-CapsNet*, which includes the software-level optimizations named *S-CapsNet* and a hybrid computing architecture design named *PIM-CapsNet* . In software-level, S-CapsNet reduces the computation and memory accesses by exploiting the computational redundancy and data similarity of the routing procedure. In hardware-level, the PIM-CapsNet leverages the processing-in-memory capability of today's 3D stacked memory to conduct the off-chip in-memory acceleration solution for the routing procedure, while pipelining with the GPU's on-chip computing capability for accelerating CNN types of layers in CapsNet.

TABLE OF CONTENTS

ABSTRACT iii

LIST OF FIGURES viii

1 Introduction 1

2 Background 4
2.1 Long-Short Term Memory Networks (LSTMs) . . . 4
2.1.1 Recurrent Neural Networks . . . 4
2.1.2 LSTM Computation . . . 5
2.1.3 LSTM Execution on Mobile GPUs . . . 7
2.2 Capsule Network (CapsNet) . . . 8
2.2.1 CapsNet Structure . . . 9
2.2.2 Essential Mechanism For Avoiding Feature Loss: Routing Procedure (RP) . . . 10

3 Towards Memory Friendly Long-Short Term Memory Networks (LSTMs) on Mobile GPUs 13
3.1 The Memory Bottleneck . . . 13
3.1.1 The Inter-Cell Level Memory Bottleneck: Redundant Data Movements 14
3.1.2 The Intra-Cell Level Memory Bottleneck: Limited Off-Chip Bandwidth 15
3.2 Inter-Cell Level Optimizations . . . 15
3.2.1 The Irrelevance Between Two LSTM Cells . . . 16
3.2.2 LSTM Layer Division . . . 17
3.2.3 LSTM Layer Reorganization . . . 19
3.2.4 The Implementation . . . 22
3.3 Intra-Cell Level Optimizations . . . 23
3.3.1 Dynamic Row Skip . . . 23
3.3.2 The Implementation . . . 24
3.4 Evaluation . . . 26
3.4.1 Experimental Setup . . . 26
3.4.2 The Effectiveness of the Overall System . . . 29
3.4.3 Performance-Accuracy Trade-offs . . . 31
3.4.4 Impact of Model Capacity . . . 33
3.4.5 User Study . . . 33
3.4.6 Overhead Analysis . . . 34

4 Enabling Highly Efficient Capsule Networks Processing Through A PIM-Based Architecture Design 35
4.1 Characterization and Analysis . . . 35
4.1.1 Overall Evaluation for CapsNet Inference . . . 35
4.1.2 Root Causes for Inefficient RP Execution . . . 37
4.2 S-CapsNet: Software Optimizations . . . 40
4.2.1 Key Observations . . . 40

4.2.2 RP Execution Compression 44
4.3 Architectural-Level Opportunity: Processing-in-Memory + Pipelining . . . 47
4.4 PIM-CapsNet: Architecture-level Optimization 49
4.4.1 Inter-Vault Level Design 49
4.4.2 Intra-Vault Level Design 55
4.4.3 Contention Reduction in CapsNet Design 59
4.5 Evaluations 62
4.5.1 Experimental Setup 62
4.5.2 Effectiveness of SH-CapsNet 64
4.5.3 Compression Schemes Comparison 67
4.5.4 Effectiveness of Intra-Vault and Inter-Vault Level Designs 68
4.5.5 Sensitivity to PE Frequency Scaling 69
4.5.6 Overhead Analysis 70

5 Future Works: Improving Computation Efficiency for LSTM Training 72
5.1 Reducing LSTM Backpropagation Workloads 72
5.2 Insignificant cell skipping within LSTM BP layer 74
5.3 Reducing insignificant computations within LSTM BP cell 75

6 Related Works 77

7 Conclusions 79

BIBLIOGRAPHY 81

LIST OF FIGURES

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. . 4
2 The schematics of (a) the CNN convolution (Conv) layer, and (b) the CNN fully connected (FC) layer. 5
3 The LSTM cell schematic. 6
4 The comparison of neural networks in identifying lung cancer cells, where CapsNet outperforms the traditional CNN on detection accuracy. The heat maps indicate the detected features. 8
5 The computation diagram of CapsNet for MNIST. 9
6 Dynamic Routing Procedure (RP) in CapsNet: describing computation flow and multiple ways for parallel processing. 9
7 The contribution of each major factor to the pipeline stall cycles when executing $Sgemv()$. 13
8 The sketch map of the kernel execution for the LSTM layer. 14
9 Utilization of on-chip and off-chip memory when executing $Sgemv()$. . . . 15
10 Sigmoid activation function and tanh activation function. 16
11 The overview of the inter-cell level optimization. 17
12 The normalized performance of one LSTM layer and shared memory bandwidth utilization as the tissue size increases. 20
13 The implementation of the inter-cell level optimizations. 20
14 Irrelevant rows in weight matrix $U_{f,i,c}$ that have trivial impact on cell output h_t. 24
15 The architecture of CTAs reorganization module. 26
16 The evaluation diagram for our optimizations. 27
17 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. 29
18 Per-layer speed up and energy saving when applying the inter-cell level optimizations. 30
19 (a) Weight matrix compression ratio, (b) speed up and (c) energy saving when applying different weight compression schemes. 30
20 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. 32
21 Performance-Accuracy trade-offs under different sets of thresholds across the different applications. 32
22 The user satisfaction score on different schemes. 33
23 The overall execution time breakdown of CapsNets on GPU across different layers. Red line represents the actual inference time. 36
24 The breakdown for pipeline stall cycles during RP execution on Tesla P100 Pascal GPU. 37

25 (a) Ratio of intermediate variables' size to on-chip storage of different GPUs; (b) the impact of on-chip storage sizes of state-of-the-art GPUs on RP's execution. A: 1.73MB (K40m), B: 5.31MB (Tesla P100), C: 9.75MB (RTX2080Ti), D: 16MB (Tesla V100). 38
26 The impact of memory bandwidth on the overall RP performance. GDDR5:288GB/s (K40m), GDDR5X: 484GB/s (GTX 1080Ti), GDDR6: 616GB/s (RTX 2080Ti), HBM2: 897GB/s (Tesla V100). 39
27 The pattern of the capsule formation from feature map neurons (left part). Such pattern causes the information similarity between capsules (right part). Note that the similarity could decrease with capsule-capsule distance increase. 41
28 The average similarity across the coefficients for different datasets. D1/D2/D3 represent the distance of the corresponding input capsules. 42
29 The heat map displays the average coefficients similarity for the different capsules with its adjacent capsules. Representative results are generated from (a) Caps-MN1 and (b) Caps-CF1. 42
30 The means (x-axis) and variations (y-axis) of the similarity between the same coefficients from adjacent iterations. 43
31 The Diagram of RP compression scheme: (a) simple compression, (b) Importance-aware compression with different EssCoe/SimCoe region identifications. Each block represents the coefficients corresponding to certain input capsule. The blank coefficients are those to be compressed, the green coefficients are SimCoe, while blue ones are EssCoe. 45
32 Left: HMC Organization. Right: HMC Block Diagram. The Red dashed box marks the vault structure. 48
33 The overview of PIM-CapsNet Design. 49
34 The execution diagram for the RP procedure with B-dimension distribution. The workloads in green blocks can be split across vaults, but workloads in purple blocks cannot be distributed via B-dimension. 54
35 Intra-vault level Architecture Design. 56
36 (a) An example of transferring the exponent representation A (i.e., $\lfloor y \rfloor + b$) and fraction representation B (i.e., $2^{y-\lfloor y \rfloor} - 1$) to the exponential function's result C in FP32 format. (b) Combining the exponent representation A and the fraction representation (i.e., D that transferred from B), and applying a unified bit shifting to obtain the exponential function's result C. 57
37 (a)The default address mapping of 8GB in HMC Gen3; (b) Our address mapping. 59
38 The Evaluation Infrastructure. 62
39 The (a) speedups and (b) normalized energy consumption of different designs on the RP execution. 65
40 The (a) speedups and (b) normalized energy consumption of different design on entire CapsNet execution. 66
41 The accuracy loss (bar) and speedup (line) when implementing different compression schemes for the RP execution on the baseline GPU. The Simple-Dn represents the simple compression scheme with the reuse-distance of n, and the IAC is the importance-aware compression scheme. 68

42 The breakdown of the factor to (a) normalized performance and (b) normalized energy consumption when performing RP execution on different PIM designs. 70
43 The speedup (heat map) achieved by different workloads distribution dimensions (X-axis) under different HMC execution frequency. Red means better improvement. 71
44 LSTM BP workload reduction with different LSTM fine-tuning cell patterns. 72
45 An example of insignificant computation identification for both DIC ($1 \sim 4$) and SIC (5) during the execution of $W \odot \delta gates = \delta x$ and $\delta gates \otimes x = \nabla W$ from BP of LSTM training. The colored elements are removable, and the dashed boxes indicate the insignificant computations located via insignificant data. 75

1 Introduction

Recently, machine learning has bloomed into rapid growth and been widely applied in many areas, including medical [1, 2], security [3], social media [4], engineering [5] and etc. These applications adopt different neural network models. There are three major models: 1. convolutional neural network (CNN) are widely used to support general image identifications. 2. 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 natural language processing (NLP) based IPAs. 3. Capsule network (CapsNet) exhibits extraordinary ability for precise human life related recognition tasks.

While many works propose the CNN optimizations, LSTM and CapsNet remain low efficient. 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 2.1.1).

On the other hand, Because CapsNets execution exhibits a high percentage of matrix operations, state-of-the-art GPUs have become primary platforms for accelerating CapsNets by leveraging their massive on-chip parallelism and deeply optimized software library [6, 7].

However, processing efficiency of CapsNets on GPUs often cannot achieve the desired level for fast real-time inference. To investigate the root causes of this inefficiency, we conduct a comprehensive performance characterization on CapsNets' execution behaviors on modern GPUs, and observe that the computation between two consecutive capsule layers, called *routing procedure* (Section 2.2.2), presents the major bottleneck. Through runtime profiling, we further identify that the inefficient execution of the routing procedure originates from (i) tremendous data access to off-chip memory due to the massive unshareable intermediate variables, and (ii) intensive synchronizations to avoid the potential write-after-read and write-after-write hazards on the limited on-chip storage. These challenges are induced by the unique features of the routing procedure execution, and cannot be addressed well via common NN optimization techniques [8–19] as well as software-level on-chip memory management (e.g., register manipulation or shared memory multiplexing).

To address the inefficient execution of the LSTM and CapsNet, I propose two hardware and software co-design approaches via modifying the execution flow and enabling processing-in-memory technique, respectively. To summarize, I make the following contributions:

- 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 also 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. Besides, 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.

- We conduct a comprehensive characterization study on CapsNet inference on modern GPUs and identify its root causes for execution inefficiency. Based on the interesting insights from the characterization and further algorithm analysis, we propose the software-hardware co-designed optimizations, SH-CapsNet, which includes the software-level optimizations named *S-CapsNet* and a hybrid computing architecture design named *PIM-CapsNet* . In software-level, S-CapsNet reduces the computation and memory accesses by exploiting the computational redundancy and data similarity of the routing procedure. In hardware-level, the PIM-CapsNet leverages the processing-in-memory capability of today's 3D stacked memory to conduct the off-chip in-memory acceleration solution for the routing procedure, while pipelining with the GPU's on-chip computing capability for accelerating CNN types of layers in CapsNet. Evaluation results demonstrate that either our software or hardware optimizations can significantly improve the CapsNet execution efficiency. Together, our co-design can achieve greatly improvement on both performance (3.41x) and energy savings (68.72%) for CapsNet inference, with negligible accuracy loss.

2 Background

2.1 Long-Short Term Memory Networks (LSTMs)

2.1.1 Recurrent Neural Networks

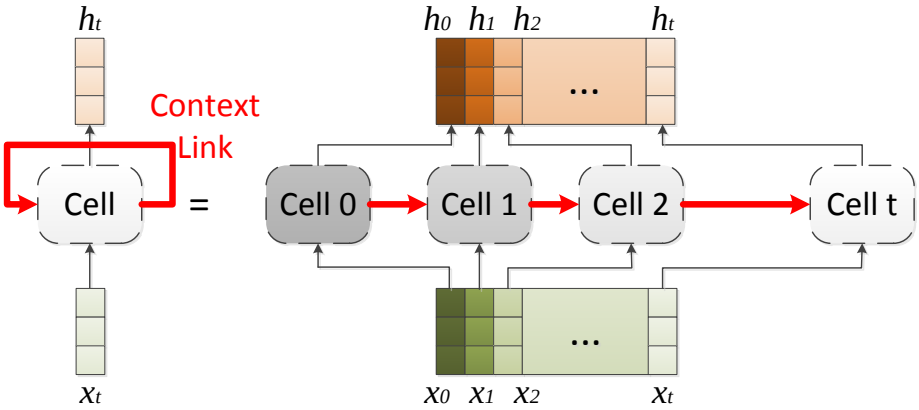
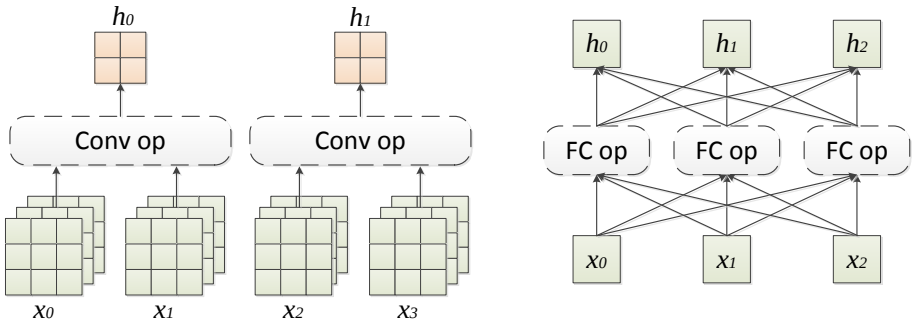


Figure 1: The schematic of one RNN layer (left) and its unrolled model (right). The cell represents the operations of mapping the inputs to the outputs. In the unrolled model (right), the cell 0 represents cell at timestamp 0 and so on.

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 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



(a) Conv layer (b) FC layer

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

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.

2.1.2 LSTM Computation

There are mainly three types of RNNs: Simple RNNs (or vanilla RNNs), Long-Short Term Memory networks (LSTMs) [20] and Gated Recurrent Unit networks (GRUs) [21]. Simple RNNs can hardly connect the useful information between two inputs with the large

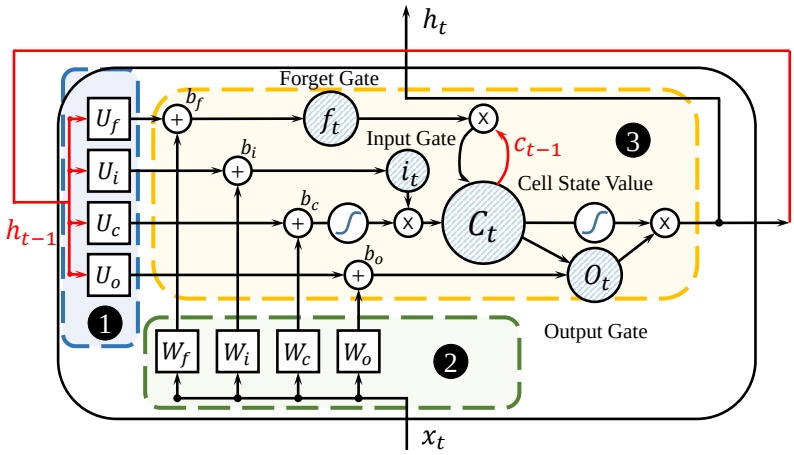


Figure 3: The LSTM cell schematic.

time interval [22]. 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.

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:

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

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

Algorithm 1 The LSTM Execution on Mobile GPUs

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

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

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

$$\text{and} \quad h_t = o_t \cdot \tanh(c_t). \tag{5}$$

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.

2.1.3 LSTM Execution on Mobile GPUs

In modern GPUs with strong backend libraries, e.g. cuDNN [23], the above computations within one LSTM cell are divided into three parts, as shown in Fig. 3 1 2 3.

In 1, 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 2, 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 [24]), 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 [25]) with limited on-chip storage. As a result, the LSTM layers are processed sequentially on mobile GPUs and the whole layer's 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 2 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 3 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).

2.2 Capsule Network (CapsNet)

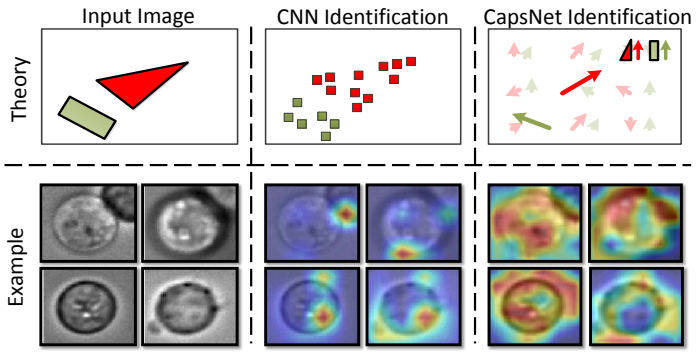


Figure 4: The comparison of neural networks in identifying lung cancer cells, where CapsNet outperforms the traditional CNN on detection accuracy. The heat maps indicate the detected features.

As shown in Fig. 5 [26], CapsNet inherits the convolutional (Conv) and fully connected (FC) layers from the standard CNNs, but introduces new layers (i.e., Caps layers) to realize the concept of "capsule" for better information representation. A capsule is a group of neurons (Fig. 4) whose activity vector represents instantiation parameters of a specific type entity (e.g., the location and pose of an object). It introduces equivariance which makes

standard CNNs understand rotation and proportional change (e.g., in Fig. 4). CapsNet significantly lifts the limitation of happenstance translational invariance of pooling operations applied in the traditional CNNs, thus being considered to be superior in image segmentation and object detection [27, 28].

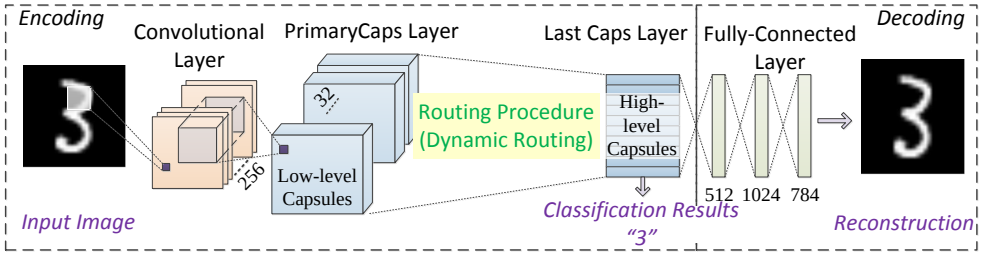


Figure 5: The computation diagram of CapsNet for MNIST.

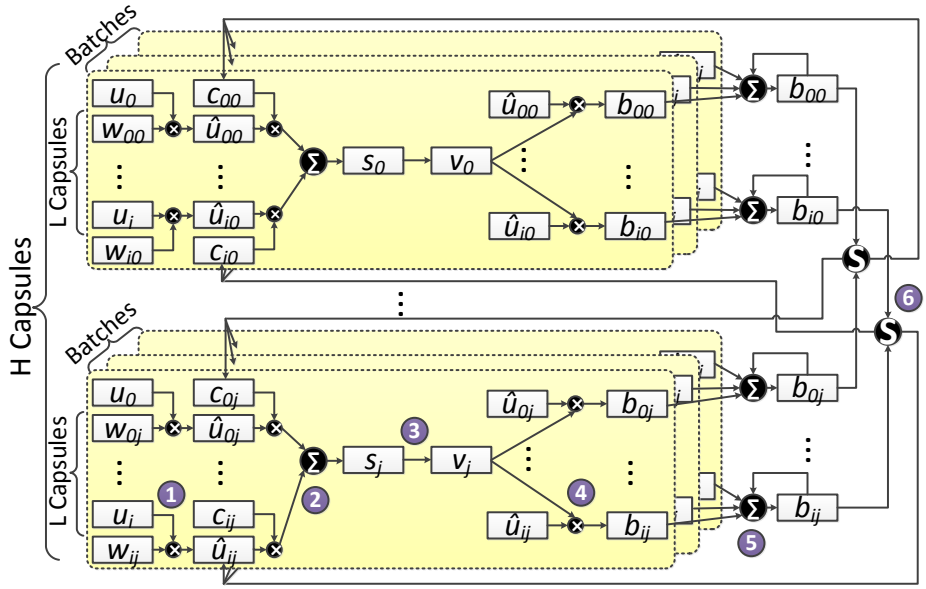


Figure 6: Dynamic Routing Procedure (RP) in CapsNet: describing computation flow and multiple ways for parallel processing.

2.2.1 CapsNet Structure

Fig. 5 takes CapsNet-MNIST [28] as an example to illustrate a basic CapsNet structure. It contains two computation stages: encoding and decoding. The encoding stage is composed of Conv layers and Caps layers. The convolutional operations are first performed

on the data mapping from neurons of the Conv layer to the capsules of the first Caps layer, which is defined as PrimeCaps layer. It is often followed by at least one other Caps layer. The data mapping between the capsules of the adjacent Caps layers is performed via the routing procedure (RP). Finally, the last Caps layer produces the classification information towards categorization, with each capsule representing one category. Following the encoding stage, the decoding function includes multiple FC layers which attach to the last Caps layer for image reconstruction (i.e., improving model accuracy in training or plotting abstract features in inference) [28].

Algorithm 2 Dynamic Routing Procedure

Input: L capsules u, weight matrix W
Output: H capsules v

for all L capsule i & all H capsule j from all input set k:
    $\hat{u}^k_{j|i} \leftarrow u^k_i \times W_{ij}$ ▷ Eq. 6
for all L capsule i & all H capsule j:
    $b_{ij} \leftarrow 0$ ▷ Initialize Routing Coeffcients
for Routing iterations **do**
    for all L capsule i:
        $c_{ij} \leftarrow softmax(b_{ij})$ ▷ Eq. 10
    for all H capsule j from all input set k:
        $s^k_j \leftarrow \sum_i \hat{u}^k_{j|i} \times c_{ij}$ ▷ Eq. 7
    for all H capsule j from all input set k:
        $v^k_j \leftarrow squash(s^k_j)$ ▷ Eq. 8
    for all L capsule i & H capsule j:
        $b_{ij} = \sum_k v^k_j \hat{u}^k_{j|i} + b_{ij}$ ▷ Eq. 9
end for
Return v

2.2.2 Essential Mechanism For Avoiding Feature Loss: Routing Procedure (RP)

The routing procedure (RP) is introduced to route the information from low-level capsules (or L capsules) to high-level capsules (or H capsules) without feature loss. There have been several routing algorithms used in routing procedure such as Dynamic Routing [28] and Expectation-Maximization routing [29]. In this work, we use the popular Dynamic

Routing [28] as an example to explain the RP execution.

Fig. 6 and Algorithm.2 demonstrate the computation flow and possible dimensions for parallelization. Given the kth batched input set, in order to generate jth H capsule, ith L capsule in this input set (u_i^k) first multiplies with the corresponding weight (W_{ij}) to generate the prediction vector ($\hat{u}_{j|i}$) (Fig. 6 1):

$$\hat{u}_{j|i}^k = u_i^k \times W_{ij}. \tag{6}$$

Then, these prediction vectors will multiply with their corresponding routing coefficients (c_{ij}) with results aggregated across all L capsules (Fig. 6 2:

$$s_j^k = \sum_i \hat{u}_{j|i}^k \times c_{ij}. \tag{7}$$

The non-linear "squashing" function is then implemented on the aggregated results of s_j^k to produce the jth H capsule v_j (Fig. 6 3):

$$v_j^k = \frac{||s_j^k||^2}{1 + ||s_j^k||^2} \frac{s_j^k}{||s_j^k||}. \tag{8}$$

Note that the v_j^k can not be considered as the final value of jth H capsule unless the features of L capsules have been correctly inherited. The information difference between an L and H capsule can be quantified by the agreement measurement via the scalar production of the prediction vector $\hat{u}_{j|i}^k$ and the H capsule v_j^k (Fig. 6 4), where the "0" output means the information is precisely inherited. In the case of a large divergence, the agreements will be accumulated into an intermediate variable b_{ij} (Fig. 6 5), which will be used to update the routing coefficients via the "softmax" function (Fig. 6 6):

$$b_{ij} = \sum_k v_j^k \hat{u}_{j|i}^k + b_{ij} \tag{9}$$

$$\text{and} \quad c_{ij} = \frac{\exp(b_{ij})}{\sum_k \exp(b_{ik})}. \tag{10}$$

The updated routing coefficients will then be integrated in Eq.(7) to start another iteration in this routing procedure (Fig. 6 2).

The number of iterations is determined by the convergence of routing coefficients and set by programmers. Several recent studies indicate that the number of iterations increases for tasks with large datasets and categories [30, 31]. Once all the iterations complete, the features of L capsules should have already been routed to the H capsules and ready to proceed to the following layers.

Summary. Generally, the routing algorithms (e.g., Dynamic Routing, Expectation-Maximization Routing) share the similar execution pattern and exhibit several core features in CapsNet routing procedure: (1) The execution of the RP exhibits strong data dependency and needs to be sequentially processed. (2) The procedure adopts all-to-all computation, which routes all the L capsules to the H capsules and forms aggregation in all possible dimensions. (3) The procedure produces a large amount of intermediate variables. (4) The routing procedure is iteratively processed to generate the dynamic coefficients to pass the feature information. We will discuss these features in relation to performance characterization of CapsNet in Section 4.1, and our optimizations on Dynamic Routing in the following Sections can be easily applied to other routing algorithms with simple adjustment.

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

3.1 The Memory Bottleneck

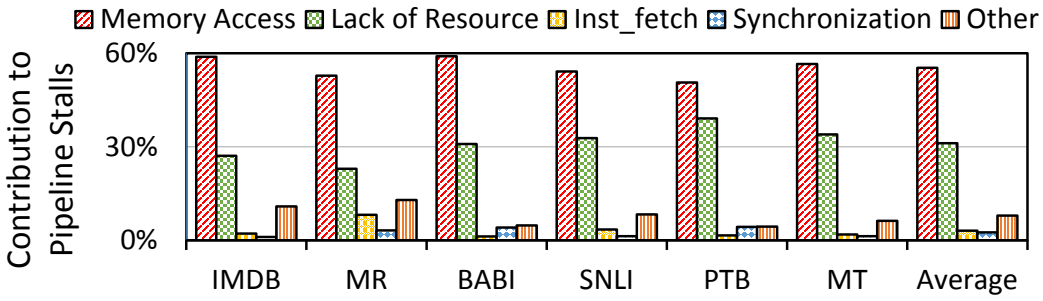


Figure 7: The contribution of each major factor to the pipeline stall cycles when executing *Sgemv*().

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. 7 plots the contribution of each major factor to the overall pipeline stall cycles when executing *Sgemv* kernels (benchmark details are presented in Section 3.4.1). As it shows, the off-chip memory access is the major contributor. Besides, previous works [32,33] 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.

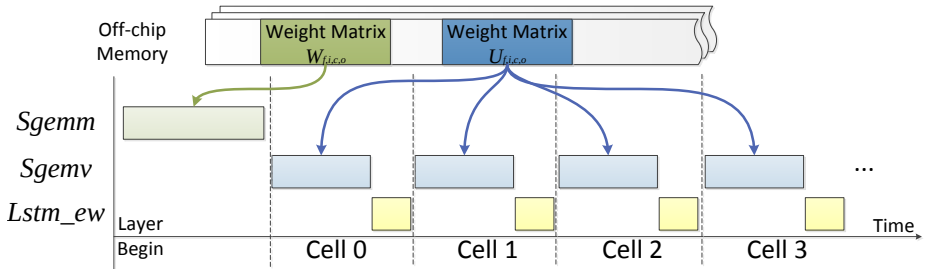


Figure 8: The sketch map of the kernel execution for the LSTM layer.

3.1.1 The Inter-Cell Level Memory Bottleneck: Redundant Data Movements

As illustrated in Algorithm.1-1, 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. 8. This is mainly caused by the unique LSTM execution pattern: as shown in Fig. 3-1, 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.

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 3.2.

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

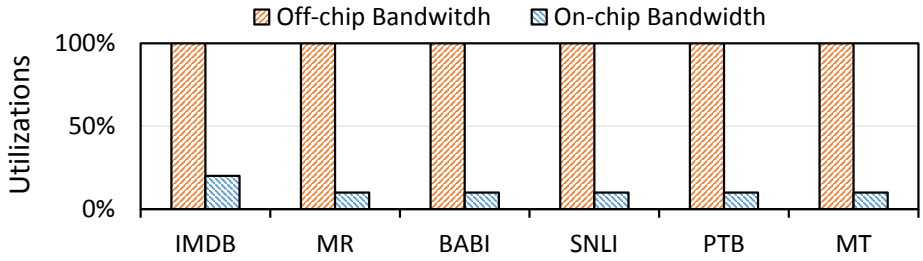


Figure 9: Utilization of on-chip and off-chip memory when executing $Sgemv()$.

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. 9 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 3.3.

3.2 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.

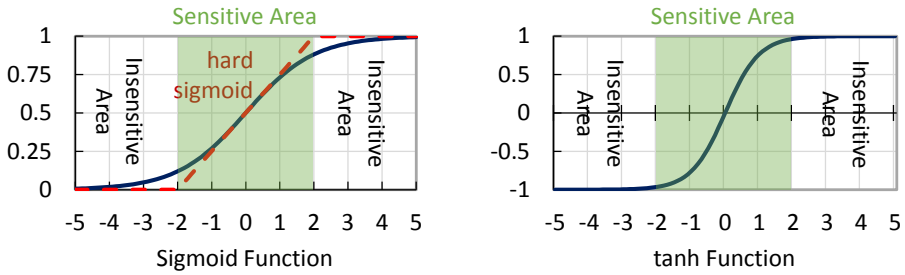


Figure 10: Sigmoid activation function and tanh activation function.

3.2.1 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 [34]. The sigmoid function takes the input within the range of $[-\infty, +\infty]$ and its output is within the range of $[0, 1]$, as shown in Fig. 10(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. 10(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. 10(b). In some neural network frameworks, the sigmoid function is modeled by the hard sigmoid function (shown in Fig. 10(a)) to accelerate the computations [35]. 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

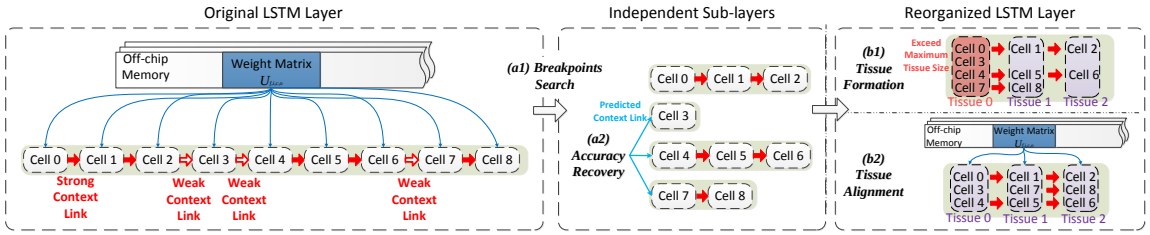


Figure 11: The overview of the inter-cell level optimization.

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.

3.2.2 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. 11(a1). This opens the door for sub-layer parallelization and reducing the data reloads which will be explored in Section 3.2.3. Though it is weak, the context link is lost between sub-layers which may affect the final output accuracy, a predicted context link (Fig. 11(a2)) is further applied to the first cell of each sub-layer (except the first sub-layer) to recover the accuracy.

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

Algorithm 3 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: $\quad S_f^j \leftarrow min(4, max(X'^j_f + b_f^j + D_f^j + 2, 0))$;

5: $\quad S_{i,c,o}^j \leftarrow min\{\left[2 + min(2, abs(X'^j_{i,c,o} + b_{i,c,o}^j))\right], \left[min(2, 2 + D_{i,c,o}^j - max(2,\ abs(X'^j_{i,c,o} + b_{i,c,o}^j)))\right]\}$;

6: $\quad S^j \leftarrow S_o^j \times (S_f^j + S_i^j \times S_c^j)$;

7: $\quad S \leftarrow S + S^j$;

8: **end for**

9: **return** S;

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.3 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 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:

$$\overline{h_j} = \sum_{i=0}^{n} h_j(i) \times \rho_{ij}. \tag{11}$$

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.

3.2.3 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. 11, the LSTM layer is divided into four sub-layers, cells 0, 3, 4, and 7 from them, respectively, are combined into one tissue;

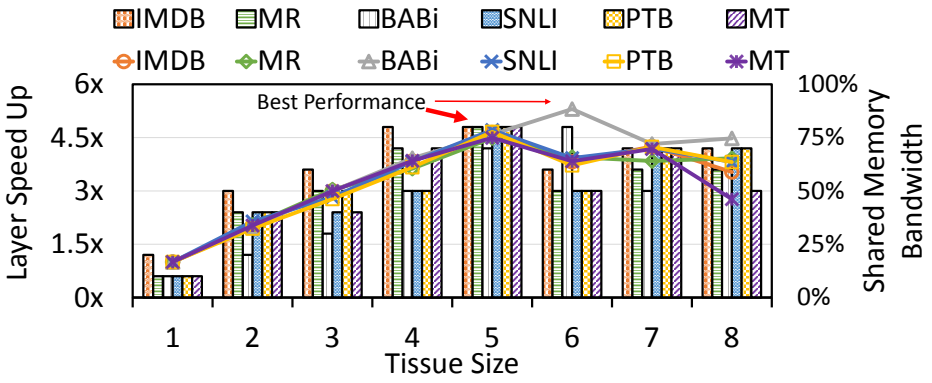


Figure 12: The normalized performance of one LSTM layer and shared memory bandwidth utilization as the tissue size increases.

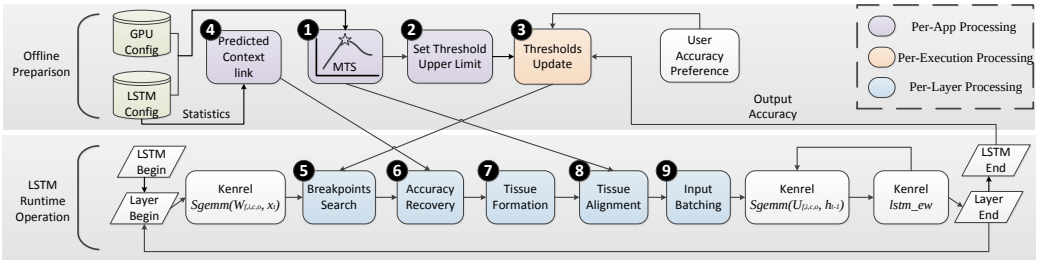


Figure 13: The implementation of the inter-cell level optimizations.

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. 11 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. 12 demonstrates the normalized performance of one LSTM layer as the tissue size increases when executing the investigated benchmarks (The baseline case introduced in Section 3.4.1.). As it shown, the performance first increases with the increasing tissue size, and then drops when the tissue size exceeds a certain number (e.g. 6

for BABI benchmark, 5 for the others in Fig. 12). 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. 12 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 Fig11(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 Fig11(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 Fig11(b1). Note that tissue alignment does not further break any context link and ensures every tissue size is below or equal to the MTS.

3.2.4 The Implementation

Offline Operations: We first execute LSTMs on the target GPUs platform with various tissue sizes to determine the MTS (Fig. 13-1). 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:

$$N_{min} = \left\lceil \frac{N_{origin}}{MTS} \right\rceil. \tag{12}$$

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. 13-2). 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. 13-3). Furthermore, the predicted context link is produced based on the LSTM configurations (Fig. 13-4). Note that the operations 1/2/4 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. 13-5) and the accuracy recovery (Fig. 13-6) 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. 13-7/8). 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. 13-9). 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.

3.3 Intra-Cell Level Optimizations

As illustrated in Section 3.1, 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 [36]. 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.

3.3.1 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. 14-1) no matter what value the corresponding element in c_t is (Fig. 14-2). 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. 14-3), the corresponding rows in these matrices can be treated as irrelevant to the final output element in h_t (Fig. 14-4).

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

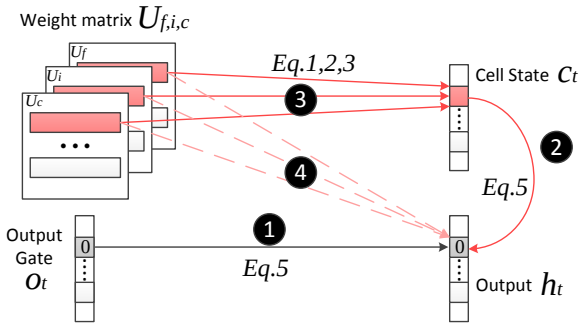


Figure 14: Irrelevant rows in weight matrix $U_{f,i,c}$ that have trivial impact on cell output h_t.

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 f_{t+1} to filter the state vector from previous cell as shown in Eq. 3. Unlike the traditional weight pruning methods performed offline [36], 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.

3.3.2 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 4 illustrates the reorganized computation flow by the DRS. In each cell,

Algorithm 4 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**

$Sgemv(U_o, h_{t-1})$ 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 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. 15. 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.

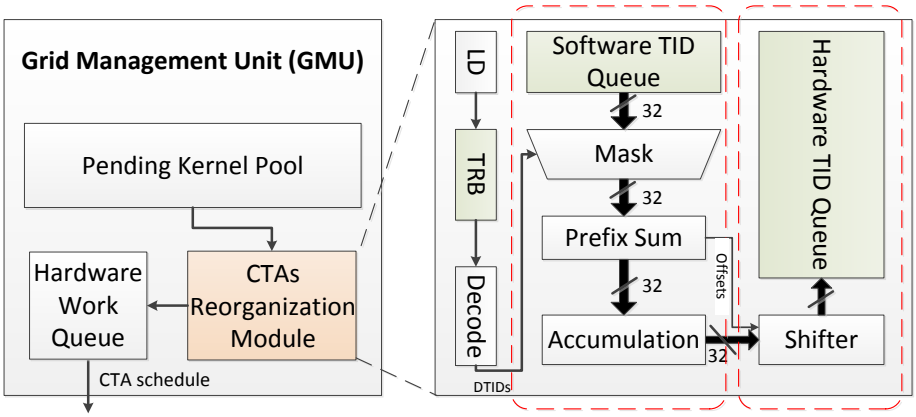


Figure 15: The architecture of CTAs reorganization module.

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. 15 for the pipelined process in the CRM. Finally, the re-organized CTAs will be sent to hardware work queue and wait to be issued.

3.4 Evaluation

3.4.1 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 [37] which

Table 1: Platform Specifications

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

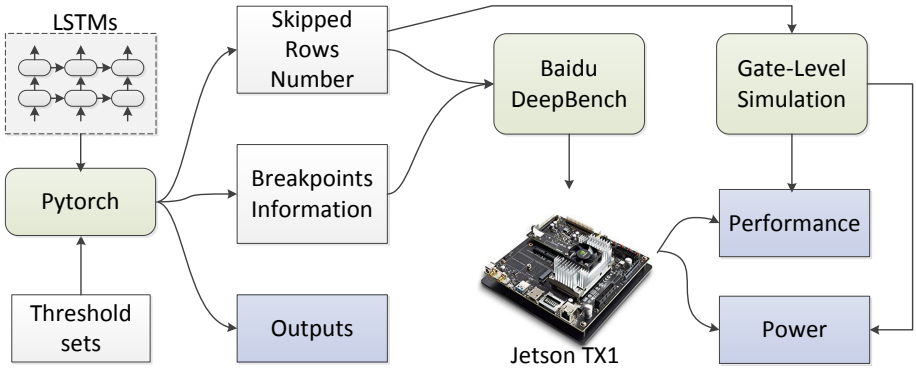


Figure 16: The evaluation diagram for our optimizations.

is a popular open-source machine learning framework that supports the dynamic computation graphs; we also use Baidu DeepBench [38], 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 [39]) cannot support the latest backend libraries for machine learning (e.g. cuDNN [23] and cuBLAS [40]), we employ the Jetson Tegra-X1 develop kit [41], a representative mobile GPU development board with the configurations listed in Table 1.

Fig. 16 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

Table 2: The state-of-the-art NLP applications investigated in our study

Name	**Abbr.**	**Hidden_Size**	**Layers**	**Length**
IMDB [42]	SC	512	3	80
MR [43]	SC	256	1	22
BABI [44]	QA	256	3	86
SNLI [45]	ET	300	2	100
PTB [46]	LM	650	3	200
MT [47]	MT	500	4	50

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 2 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 [42] and MR [43] perform sentiment classification (SC) that predict the positive or negative attitude of texts. BABI [44] performs question answering (QA) for automatic text understanding and reasoning. SNLI [45] is a collection of 570k human-written English sentence pairs manually labeled for balanced classification with the labels entailment (ET). PTB [46] is used for word-level language modeling (LM). And MT [47] 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.

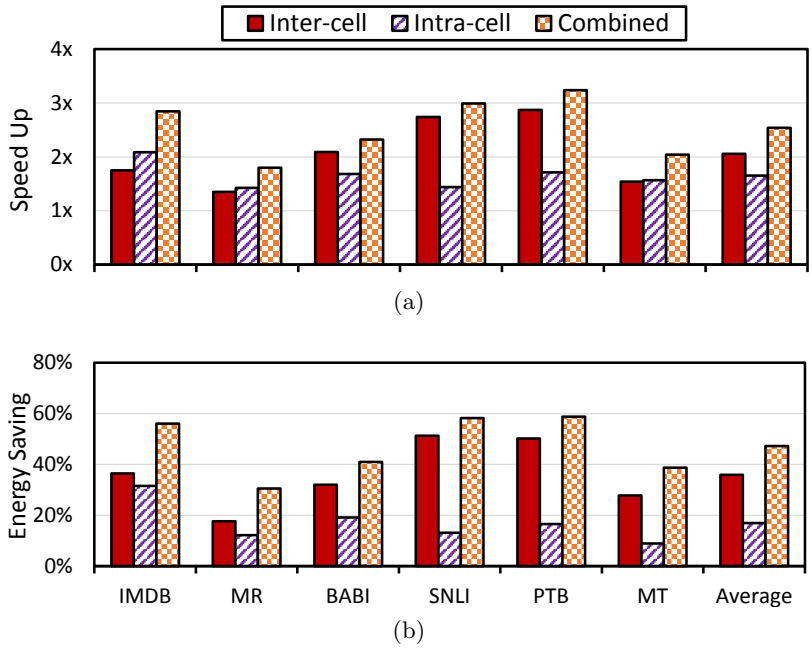


Figure 17: The (a) speedup and (b) energy saving achieved by our system when applying inter-cell level optimizations, intra-cell level optimizations and the overall system with the combined optimizations.

3.4.2 The Effectiveness of the Overall System

Fig. 17 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.

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.

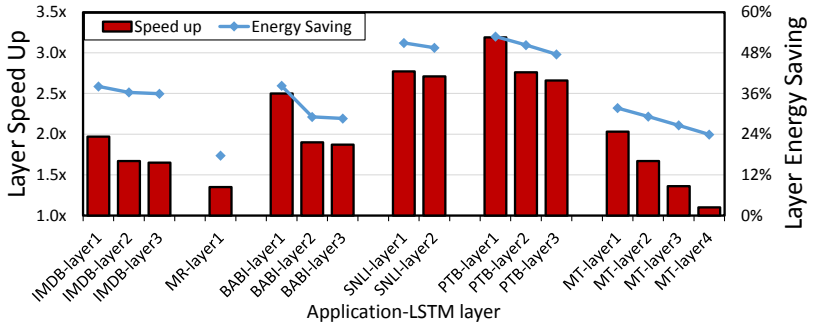


Figure 18: Per-layer speed up and energy saving when applying the inter-cell level optimizations.

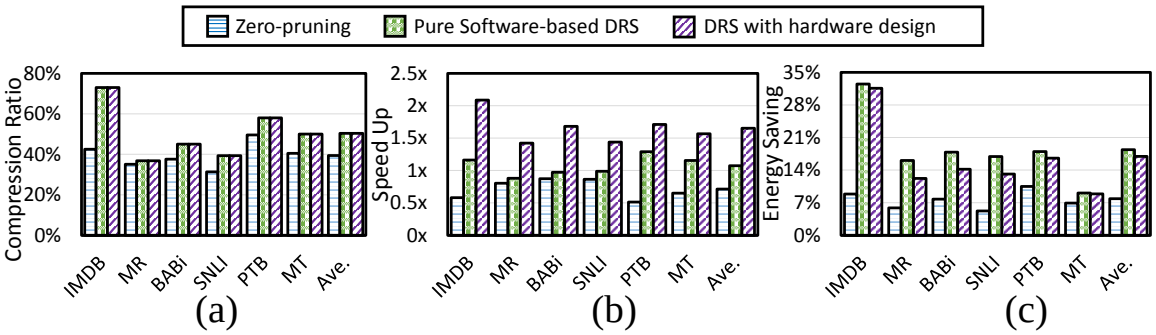


Figure 19: (a) Weight matrix compression ratio, (b) speed up and (c) energy saving when applying different weight compression schemes.

We further investigate the effectiveness of our inter-cell optimizations on each LSTM layer, shown in Fig. 18. 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.

Intra-cell Level Optimizations: Fig. 17 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 [36]) and pure software-based DRS, shown in Fig. 19. 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, 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.

Putting It All Together: As shown in Fig. 17, 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.

3.4.3 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

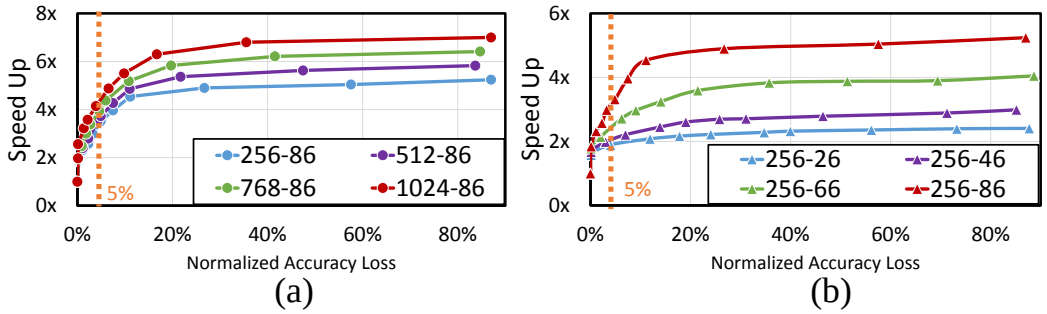


Figure 20: The performance-accuracy trade-offs of LSTMs for BABI with (a) different hidden unit sizes; (b) different input lengths. Each line represents a configuration of (hidden unit size - input length) pair.

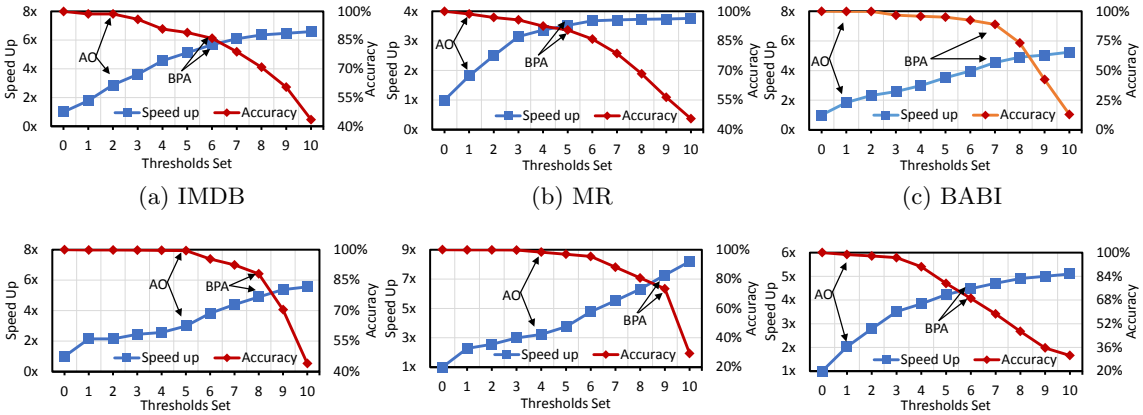


Figure 21: Performance-Accuracy trade-offs under different sets of thresholds across the different applications.

α_{inter} and α_{intra}, respectively. Threshold set 0 (10) has the lowest (highest) threshold values. Fig. 21 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.

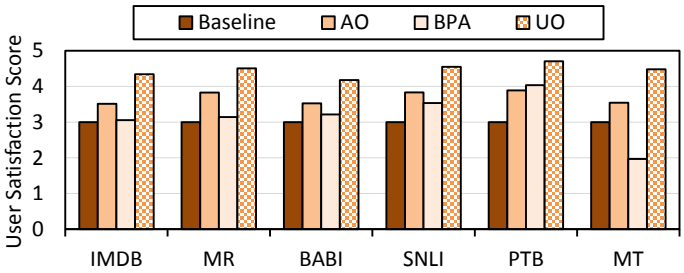


Figure 22: The user satisfaction score on different schemes.

3.4.4 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.

3.4.5 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 25 replays. The order of the schemes is random. Fig. 22 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.

3.4.6 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.

4 Enabling Highly Efficient Capsule Networks Processing Through A PIM-Based Architecture Design

4.1 Characterization and Analysis

While the CapsNet starts gaining popularity from both academia and industry, the performance characterization of CapsNet on modern high-performance platforms is largely neglected. Given that GPU has become the major platform for executing CapsNet due to high computation capability and deep optimization on matrix operations (which CapsNet has a large amount of), we adopt some of the state-of-the-art NVIDIA GPU platforms to conduct a comprehensive characterization towards the execution behaviors of various CapsNets listed in Table 3. We use 4 different datasets and corresponding 12 CapsNets with CapsNet-MNIST like structure (Section 2.2.1), and different configurations on batch size (BS in Table 3), L capsules, H capsules and routing iteration number. These CapsNets' inference are processed via PyTorch framework [48] with the latest deep learning library (i.e., CuDNN [49]), which already enables the state-of-the-art CNN optimizations [50].

4.1.1 Overall Evaluation for CapsNet Inference

Fig. 23 demonstrates the performance breakdown of each layer to the overall CapsNet execution. It shows that, across different CapsNet configurations, the routing procedure (RP) accounts for an average of 74.62% of the entire inference time, becoming the major performance bottleneck. we further conduct detailed analysis on the performance results of CapsNets on GPU and make the following observations:

Observation 1: ineffectiveness of batched execution. One common strategy to accelerate CNNs is to conduct batched execution for improving hardware utilization, especially when input dataset is large. However, it cannot improve the RP's performance during inference. As Fig. 23 illustrates, with the increasing of batch size (i.e., Caps-MN1

Table 3: CapsNet Benchmark Configurations

Net-work	Dataset	Configuration			
		BS	L Caps	H Caps	Iter
Caps-MN1	MNIST [51]	100	1152	10	3
Caps-MN2		200	1152	10	3
Caps-MN3		300	1152	10	3
Caps-CF1	CIFAR10 [52]	100	2304	11	3
Caps-CF2		100	3456	11	3
Caps-CF3		100	4608	11	3
Caps-EN1	EMNIST_Letter [53]	100	1152	26	3
Caps-EN2	EMNIST_Balanced [53]	100	1152	47	3
Caps-EN3	EMNIST_By_Class [53]	100	1152	62	3
Caps-SV1	SVHN [54]	100	576	10	3
Caps-SV2		100	576	10	6
Caps-SV3		100	576	10	9

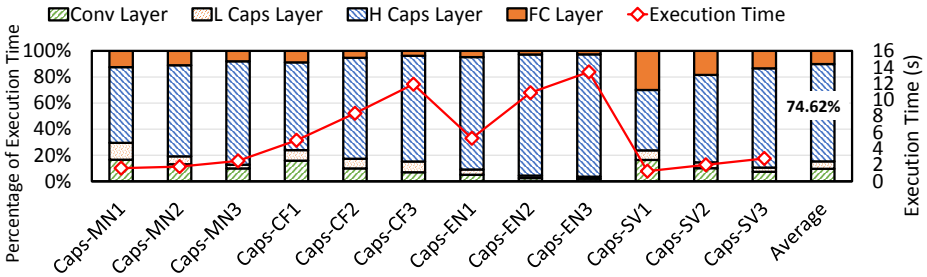


Figure 23: The overall execution time breakdown of CapsNets on GPU across different layers. Red line represents the actual inference time.

→ Caps-MN3),the overall CapsNet inference time increases; meanwhile, the RP proportion also expands with batch size.

Observation 2: sensitivity to network scaling. Table 3 shows the network size (formed by a combination of L capsules, H capsules and routing iterations) for each individual case, e.g., Caps-SV1 being the smallest. The red curve in Fig. 23 also demonstrates that the overall inference time and RP's percentage generally increases when scaling up the network size (e.g., comparing Caps-MN1, Caps-CF1, Caps-EN1 and Caps-SV1). This implies that RP's execution time is sensitive to network size as well.

To summarize, using the highly optimized deep learning library, the RP execution time

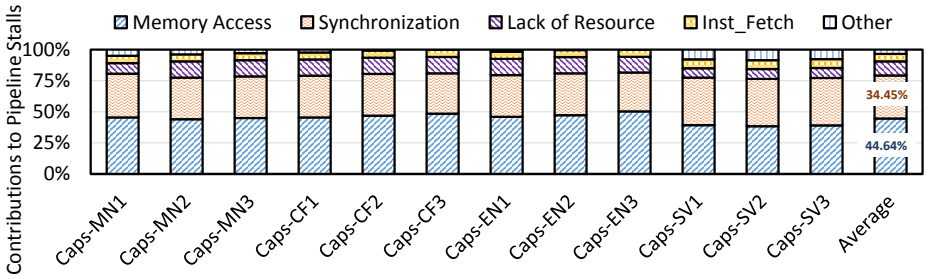


Figure 24: The breakdown for pipeline stall cycles during RP execution on Tesla P100 Pascal GPU.

on GPU can not be effectively reduced through the general CNN optimization techniques such as batch execution. Moreover, it exhibits a certain level of sensitivity to network scaling. Both of these factors make the RP execution a dominating performance bottleneck for CapsNet inference, especially with a growing size and complexity of future CapsNet structures [30, 31].

4.1.2 Root Causes for Inefficient RP Execution

To understand the root causes of RP's inefficiency on GPU, we use NVprofiler [55] to collect runtime GPU stats for comprehensive analysis. We observe two root causes for poor RP execution efficiency on GPU-like architectures:

(1) Significant Off-Chip Memory Accesses: We profile the utilization of several major GPU function units during RP execution on a NVIDIA Tesla P100 GPU. We observe that the arithmetic logic unit (ALU) is lightly utilized (i.e., on average only 38.6% across the investigated benchmarks) while the load/store unit (LDST) is heavily stressed with an average utilization of 85.9%. This implies that CapsNets' RP phase fails to effectively leverage the GPU strong computation capability and is severely limited by the intensive off-chip memory access. We further investigate the factors that may contribute to GPU pipeline stalls during RP, including the off-chip memory access, barrier synchronization, lack of resource, etc. Fig. 24 profiles the individual contribution of each major factor to the overall pipeline stall cycles. As can be seen, the majority of the pipeline stalls are induced

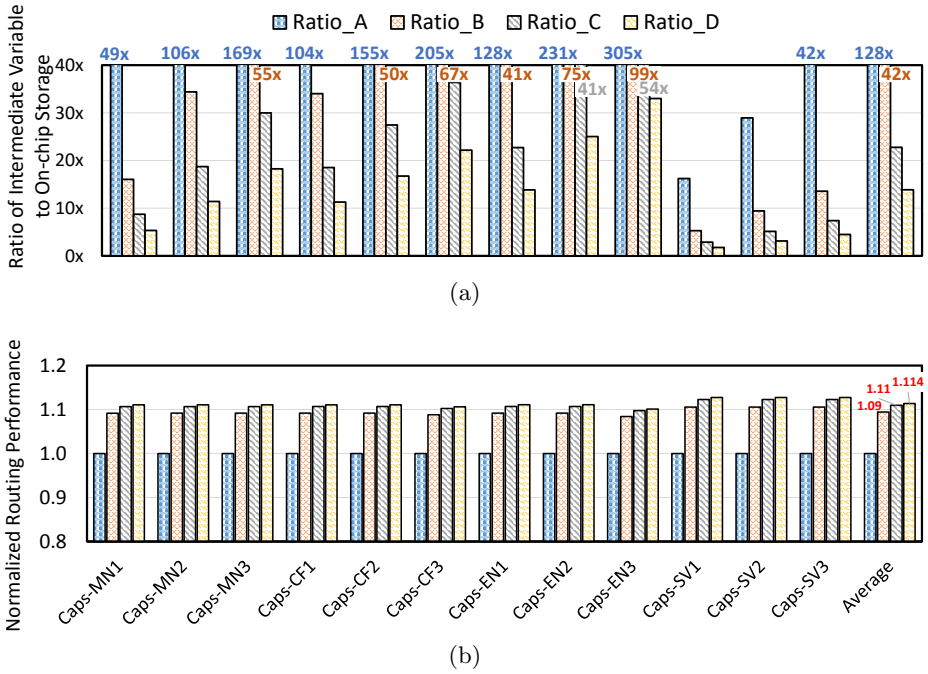


Figure 25: (a) Ratio of intermediate variables' size to on-chip storage of different GPUs; (b) the impact of on-chip storage sizes of state-of-the-art GPUs on RP's execution. A: 1.73MB (K40m), B: 5.31MB (Tesla P100), C: 9.75MB (RTX2080Ti), D: 16MB (Tesla V100).

by the memory access (i.e., on average 44.64%). This further confirms that RP performance is significantly limited by the off-chip memory access. This is caused by a combination of massive intermediate variables from RP execution and limited on-chip storage. Fig. 25(a) illustrates the ratio of RP's intermediate variables' size to on-chip memory sizes of different generations of NVIDIA GPUs. As can be seen, the size of RP's intermediate variables far exceeds the GPU on-chip storage. Given the iterative computation pattern of RP, these variables (e.g., $\hat{u}, s_j, v_j, b_{ij}, c_{ij}$) need to be periodically loaded into GPU cores from the off-chip memory due to the limited on-chip storage. Moreover, the intermediate variables are not sharable among different input batches, which also explains the ineffectiveness of batched execution as we observed in Section 4.1.1. Due to the large data volume and lack of temporal value similarity from these variables, software-level schemes such as register manipulation and shared memory multiplexing are also not very effective.

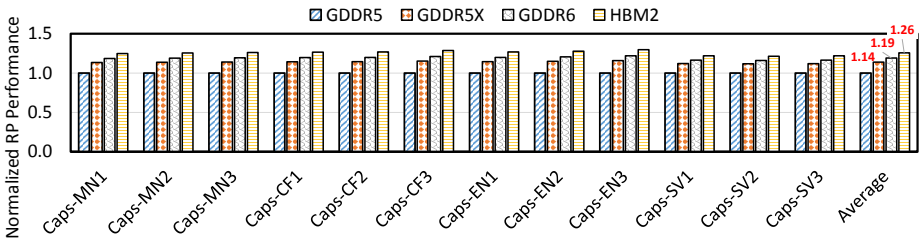


Figure 26: The impact of memory bandwidth on the overall RP performance. GDDR5:288GB/s (K40m), GDDR5X: 484GB/s (GTX 1080Ti), GDDR6: 616GB/s (RTX 2080Ti), HBM2: 897GB/s (Tesla V100).

(2) Intensive Synchronization: Fig. 24 also indicates that the frequent barrier synchronization is the second major contributor (i.e., on average 34.45%) to the pipeline stalls. These synchronization overheads are induced by the *__syncthread()* calls, which coordinate shared memory accesses for all threads in one thread block. There are two major factors causing the frequent synchronization during the RP execution: (i) the RP execution contains numerous aggregation operations (e.g., Eq. 7), inducing massive data communication between the threads through shared memory; (ii) the size of the intermediate variables far exceeds the shared memory size, leading to frequent data loading from the global memory. Thus, *__syncthread()* calls occur frequently to avoid the potential write-after-write and write-after-read hazards.

To address these issues above, we attempt to apply two naive solutions: scaling up the on-chip and off-chip memory capacity. Fig. 25(b) shows the impact of on-chip memory sizes of different generations of NVIDIA GPUs on RP execution. We can observe that increasing on-chip memory size can help alleviate the challenges above but not very effective, e.g., only up to an average of 14% performance improvement for the 16MB V100. This is because the nonsharable intermediate variables' size from RP still far exceeds the current GPUs' on-chip storage, shown in Fig. 25(a). Similarly, Fig. 26 shows that only increasing memory bandwidth from 288GB/s GDDR5 to 897 GB/s HBM slightly improves the overall RP's performance by an average of 26%. This indicates that higher off-chip memory bandwidth can only solve a small part of the problem but itself does not reduce the high intensity

of the off-chip memory accesses. Therefore, to significantly improve CapsNet's inference performance, we need a customized solution to address the root causes for RP's inefficient execution.

4.2 S-CapsNet: Software Optimizations

In this section, we first explore the opportunities of software-level optimizations to reduce the computation intensity and the large unsharable intermediate variables generated by RP, thus improving RP's overall execution efficiency through the perspectives of software design.

4.2.1 Key Observations

According to the analysis in Section 4.1, the tremendous intermediate variables generated by the iterative algorithm results in poor data locality under the limited GPU on-chip storage. The repetitive accesses to these intermediate variables significantly reduces RP's processing efficiency due to the saturated memory bandwidth as well as the large synchronization overhead, leading to the extremely long execution time.

To a great extent, such inefficiency is first and foremost caused by the large volume of the intermediate variables during the iterative process, which is very difficult to be solely addressed by system-level optimizations. The first fundamental step towards solving this efficiency issue is to explore data volume and memory access reduction mechanisms that do not hurt the overall accuracy. To achieve this goal, we investigate RP's intrinsic pattern similarity or redundancy at both data and program level. More specifically, we make two key observations from the capsule-capsule information mapping properties: *capsule-wise mapping similarity* (data similarity) and *loop-wise mapping consistency* (program-level redundancy).

Capsule-Wise Mapping Similarity: In the RP, the input capsules are designed to transfer their information to the output capsules, where the information bridging and filtering are accomplished via multiplying input capsules with their corresponding coefficients,

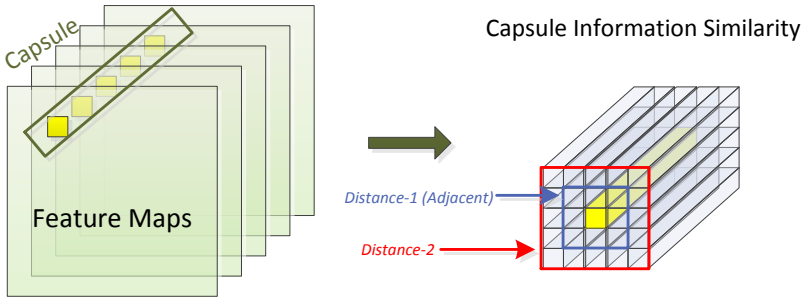


Figure 27: The pattern of the capsule formation from feature map neurons (left part). Such pattern causes the information similarity between capsules (right part). Note that the similarity could decrease with capsule-capsule distance increase.

i.e., c_{ij} in Eq. 10. And the purpose of the iterative execution in RP is to update these coefficients to reflect the capsule-capsule mapping relations. First, we profile the amount of similarity in the capsule-wise mappings by calculating the cosine-similarity, as described by:

$$\text{CosSim}(C_A, C_B) = \frac{C_A \cdot C_B}{\|C_A\| \times \|C_B\|}, \tag{13}$$

between the coefficient vectors of different input-capsules.

From the profiling, we make the observation that data similarities exist among the coefficients of the nearby capsules. This is the result of adopting Conv layers in CapsNet for image feature extraction, making nearby neurons (coming from the nearby pixel blocks) in the feature maps to contain similar information. As shown in Fig. 27, one capsule is formed using the same positioned neurons from several features maps, which ensures the consistency of the information likelihood with its surround capsule values. Thus, it leads to similar mapping behaviors towards the output capsules.

Fig. 28 demonstrates the average similarity across the coefficients for different datasets, under various distances between the input-capsules and their surrounding capsules. The adjacent capsules with distance of 1 appear to have the highest similarity in the information mappings, i.e., on average 99.34% across different testing inputs. This capsule-wise mapping similarity originates from the value likeness of the feature map neurons. For instance, for

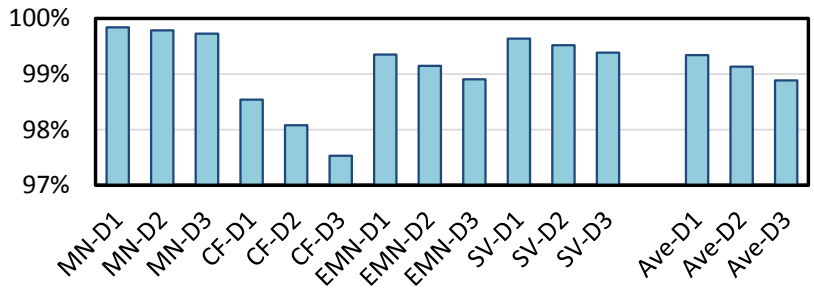
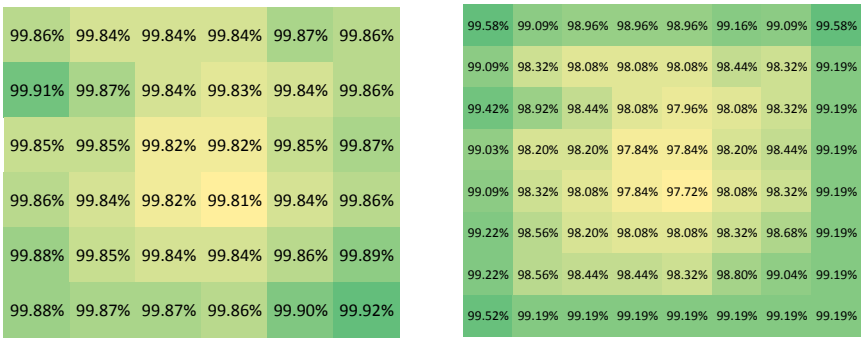


Figure 28: The average similarity across the coefficients for different datasets. D1/D2/D3 represent the distance of the corresponding input capsules.



(a) (b)

Figure 29: The heat map displays the average coefficients similarity for the different capsules with its adjacent capsules. Representative results are generated from (a) Caps-MN1 and (b) Caps-CF1.

the image processing tasks, nearby pixels tend to be alike. Moreover, we also notice that the mapping similarity decreases with the growing distance between capsules due to the large information representations gaps between far-way neurons.

In addition to explore the average impact of the capsule-capsule distances on the mapping similarity, we also observe that such impact is not unified across all the capsules. Fig. 29 illustrates this observation using Caps-MN1/Caps-CF1 as examples. The heat maps represent the average coefficients' similarity for the capsules and their adjacent capsules. A darker color represents higher similarity: more surround capsules can share the same mapping pattern without significantly damaging accuracy; while a lighter color represents

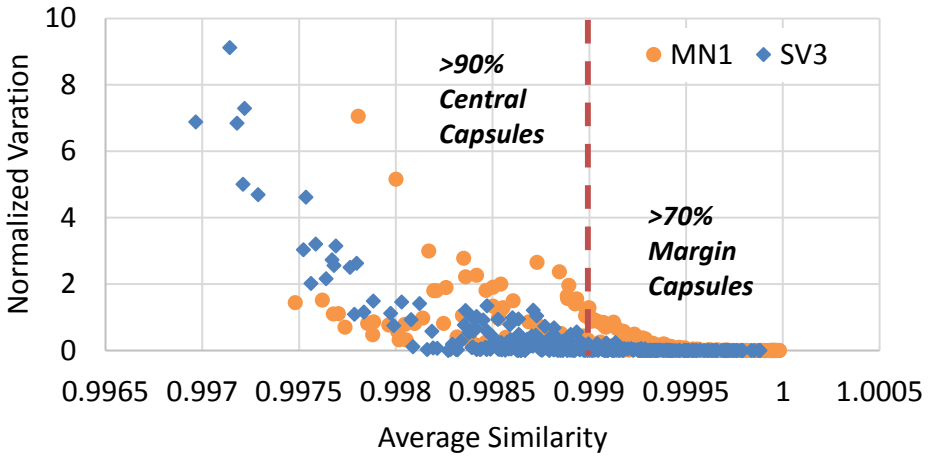


Figure 30: The means (x-axis) and variations (y-axis) of the similarity between the same coefficients from adjacent iterations.

capsules with more different mapping patterns than their surrounding capsules. Furthermore, the figure shows that for both datasets the light-colored areas reside at the center of the heat maps, indicating that the central capsules have more unique mapping patterns. This is because the corresponding central neurons are generated from the central pixels of an input image, which typically contain more critical information than the marginal area. On the contrary, darker areas contain little or even no information due to the usage of ReLU functions that produces numerous "0" neurons, leading to similar information mapping patterns.

Loop-wise Updating Consistency: The coefficients are initialized at runtime and require several RP iterations to finish updating. Such iterative process not only ensures the correctness of input-output capsule mapping, but also may induce significant data movements between on-chip units and off-chip DRAM. As discussed previously, the iterative process generates a tremendous mount of intermediate variables, resulting in bad data locality and saturating the DRAM bandwidth.

However, not all the capsules require the entire number of iterations to finish the coefficient updating. We make a key observation that some coefficients updating becomes

trivial during the RP iterations; in other words, different capsules require different iteration numbers to reach the accurate coefficients update, and only a portion of capsules require the whole iteration procedure. Fig. 30 illustrates the mean and variation values of the similarities between the same coefficients from the adjacent iterations, which indicates the average coefficients changes corresponding to all the input capsules throughout the RP iterations. According to the variation magnitude, these coefficients can be classified into two categories. One category contains the relatively stable coefficients that require less iterations (right side of the right line); the other category contains coefficients that actually benefit from the entire RP iterations (left side of the red line). Specifically, we identify that most of the capsules belonging to the first category are from the marginal areas. These input capsules contain less information and their coefficients require less iterations to be determined. On the other hand, the input capsules from the second category embedded with essential information and must maintain the entire iteration processing to ensure the correctness of the information mapping.

4.2.2 RP Execution Compression

Based on the previous two key observations, there exist computation and memory access redundancy for the RP procedure. Therefore, at software level, we propose S-CapsNet to conduct RP execution compression using the capsule-wise mapping similarity and loop-wise updating Consistency.

Simple Compression: According to the first observation (the close capsules share the similar coefficients value), we propose to modify the RP algorithm that apply only a few input capsules to calculate the coefficients, and reuse these coefficients for the surrounding capsules to calculate the output capsules. This compression pattern is shown in Fig. 31(a). We locate and group every 3x3 capsules to calculate only the center coefficients with the iteration skipped for the surrounding capsules. This compression method is convenient to implement, and it can significantly reduce the computation and the intermediate variables

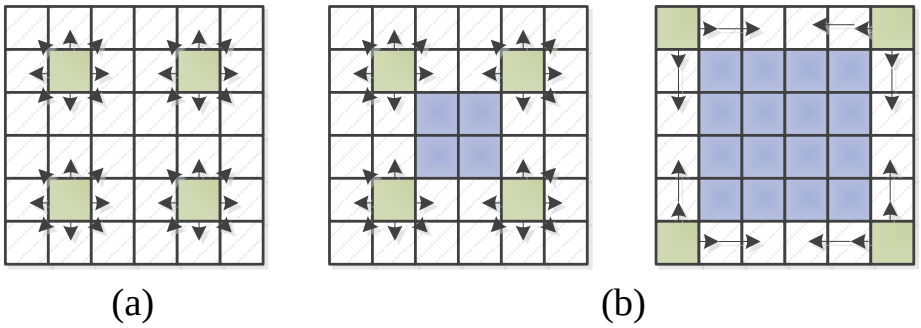


Figure 31: The Diagram of RP compression scheme: (a) simple compression, (b) Importance-aware compression with different EssCoe/SimCoe region identifications. Each block represents the coefficients corresponding to certain input capsule. The blank coefficients are those to be compressed, the green coefficients are SimCoe, while blue ones are EssCoe.

involved in the RP iterations. Note that this method only allows a capsule coefficient to be reused by the adjacent capsules (i.e., distance equals to 1) to maintain the accuracy. Although these coefficients have trivial differences for the remote capsules (e.g., less than 2% in Fig 29), they could eventually induce significant accuracy loss for the output capsules. Therefore, this simple compression method is ineffective to address accuracy-sensitive applications.

Importance-Aware Compression: In order to maintain high accuracy while reducing the computation intensity and memory access, we propose to apply different schemes to different area capsules.

As demonstrated in Fig. 29, compared with the marginal area coefficients, the coefficients of central area have lower similarity to the surrounding coefficients, and they tend to experience big value changes throughout the RP iterations. This makes it very risky to aggressively skip computation for the central area coefficients. Thus, we define the central area coefficients as essential coefficients, or *EssCoe*, because they are critical to the information mapping between input and output capsules. We also define the marginal area coefficients as similar coefficients, or *SimCoe*, due to their highly similar values with the surrounding coefficients. The solution to maintain the overall high accuracy is to maintain

the EssCoe computation while reducing the SimCoe computation; and how to identify these two regions becomes the key challenge.

We further explore the EssCoe/SimCoe regions with different datasets and found that the EssCoe/SimCoe regions are highly related to the ratio of the underlying object to the entire picture as well as the object complexity. For example, in Fig. 31(b)-left figure, since the objects in MNIST occupy a small area and are relatively simple, high accuracy can be achieved with only the central area set as EssCoe. However, for datasets like CIFAR10 [1] shown in Fig. 31(b)-right figure, its objects occupy nearly the whole image and contain quite complex information, resulting in a larger EssCoe region compared to the MNIST dataset.

Given the analysis above, the method to identify the EssCoe/SimCoe regions is explained as follows. The identification can be conducted before runtime. For an image dataset, the average object size can be calculated and the object complexity can be evaluated using the previous method for image complexity measurement [56]. Based on the Conv layer configuration before the capsule layer, we propose to determine the initial EssCoe area as the area of the corresponding feature neurons from the object pixels. And for those datasets with less object complexity, their EssCoe area can be furthered shrunk.

From Fig. 30, we can observe that the SimCoe values trend to be stable after the first iteration. Therefore, in addition to compress the number of the SimCoe involved in the RP procedure, we propose to further reduce its iterative computation. The deductible iteration number is determined by the size of the SimCoe region. For a smaller SimCoe, less information is preserved, leading to faster convergence for the SimCoe value. On the contrary, a larger SimCoe contains more information and also higher risk to skip iterative computation.

In this section, we have discussed the software-level optimization opportunities. We

[1] Note that the actual feature map of CIFAR10 before capsule layer should be 8×8. We use 6×6 here as an example to simplify the discussion.

make several key observations to explore the data similarity and redundancy of the computation and memory access in RP procedure. These software-level optimizations can provide moderate improvement on GPU execution efficiency while maintaining a high accuracy. However, after value trimming and compression, the intermediate variable size generated by RP's iterative process is still much larger than GPU's on-chip storage, i.e., at least 4.64×. The RP execution remains bounded by the hardware limitations, which motivates us to conduct deeper architectural-level exploration.

4.3 Architectural-Level Opportunity: Processing-in-Memory + Pipelining

As discussed previously, we aim to address CapsNets' significant off chip-memory access caused by massive unshareable intermediate variables and intensive synchronization due to numerous aggregation operations in RP. Meanwhile, we also want to utilize the excellent core computing capability provided by modern GPUs for deep learning's matrix operations. Thus, we propose a hybrid computing engine named *"PIM-CapsNet"*, shown in Fig. 33. It utilizes GPU's native on-chip units as the host for fast processing layers such as Conv and FC, while pipelining with an off-chip in-memory acceleration that effectively tackles RP's inefficient execution. For example, since multiple input sets are generally batched together to be concurrently processed in RP to avoid the local optimal solution of the routing coefficients [57], host processors can start processing Conv/FC operations from the different batches of the input sets while waiting for RP's results from in-memory processing on the current batch, forming an execution pipeline. Furthermore, the in-memory accelerators of our PIM-CapsNet design can hierarchically improve RP's execution efficiency by minimizing data movement and maximizing parallel processing. Note that our proposed design is a general optimization solution that is applicable to different routing algorithms used in RP.

To build our in-memory acceleration capability for RP, we resort to one of the emerging

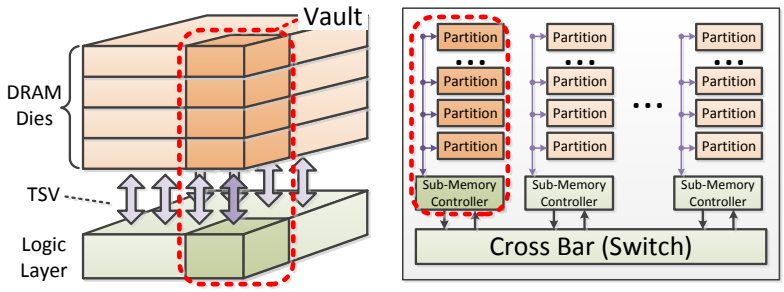


Figure 32: Left: HMC Organization. Right: HMC Block Diagram. The Red dashed box marks the vault structure.

3D stacked technologies, i.e., hybrid memory cube (HMC) [58], which has become a promising PIM platform [59–63]. The major reason to replace current GPU's off-chip memory (e.g., GDDRX or HBMs) with HMC in our design is their lack of logic layer for in-memory integration. As illustrated in Fig. 32, HMC stacks several DRAM dies on the CMOS logic layer as a cube (specification 2.1 [64]). The memory cube connects with the host processor (i.e., GPU cores in our case) via the fully-duplex links which can provide up to 320GB/s of external memory bandwidth. Additionally, the logic layer is split into 32 sub-memory controllers with each communicates its local DRAM banks through Through-Silicon Vias (TSVs) which together provide an internal memory bandwidth of 512GB/s [58, 61]. The sub-memory controller and its local DRAM partitions (each partition contains multiple DRAM banks) form the *vault* architecture, as highlighted in the red dashed box. The logic layer receives system commands and routes memory access to different vaults. A crossbar is integrated in the logic layer to support the communication between SerDes links and vaults. Note that relatively simple computation logic can be integrated onto HMC's logic layer which can directly access data from vaults via memory controllers and benefit from large internal memory bandwidth. This layer is very suitable for integrating in-memory accelerators for RP execution. Next, we will discuss the detailed PIM-CapsNet design.

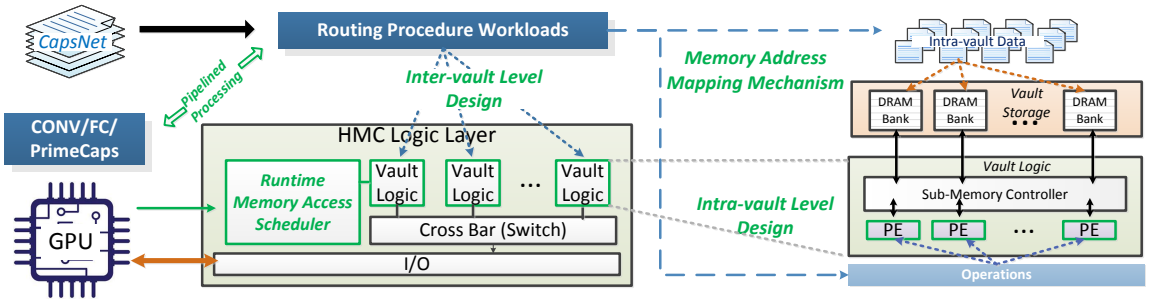


Figure 33: The overview of PIM-CapsNet Design.

4.4 PIM-CapsNet: Architecture-level Optimization

There are two **key design objectives** in PIM-CapsNet: (i) maintaining workload balance with minimal data communication at *inter-vault level*(Section.4.4.1 and Section.4.4.3), and (ii) maximizing parallel processing at *intra-vault level* but within architecture design constraints (Section.4.4.2 and Section.4.4.3).

4.4.1 Inter-Vault Level Design

As an interesting feature, the operations of RP equations (Section 2.2.2) are highly parallelizable. For example, in Eq.(6), the vector-matrix multiplications for all the low- to high-level (L-H) capsule pairs are independent. We define such independent operations on L capsules as parallelism in the L-dimension, while the independent operations on H capsules are defined as parallelism in the H-dimension. Additionally, if operations corresponding to different batches are independent, they are defined as parallelism in the B-dimension. Thus, we make the following key observations:

Observation I: *Operations of each equation in the RP can be partitioned into multiple independent sub-operations on at least one of the three dimensions (L, H, or B), suggesting highly parallelizable feature.*

Table 4 further demonstrates which possible dimensions these five equations of the dynamic routing procedure can be parallelized through. Based on it, we also make the second key observation:

Table 4: Possible Parallelizable Dimensions

	Batch (B-dimension)	**Low-level Caps (L-dimension)**	**High-level Caps (H-dimension)**
Eq. 6	**x**	**x**	**x**
Eq. 7	**x**		**x**
Eq. 8	**x**		**x**
Eq. 9		**x**	**x**
Eq. 10		**x**	

Observation II: *All the RP equations cannot be concurrently executed through the same dimension.*

Observation I indicates that the inter-vault data communication can be reduced by parallelizing the independent sub-operations of each equation on *one chosen dimension* and then distributing them across HMC vaults. By doing so, the major communication across vaults is only required by the aggregation operations at that dimension (aggregations required by the other two dimensions will be performed locally within vaults), which is relatively low. Observation II, however, emphasizes that the inter-vault communication can not be fully eliminated for RP as none of the three dimensions can support the parallelization for all the RP equations. When distributing the entire RP workloads on a certain dimension, some related data have to be reloaded to another designated vault for aggregation operations.

Parallelization and Workload Distribution: Since distributing the workloads on different dimensions leads to different communication overheads and execution patterns, it is important to apply an intelligent workload distributor to achieve the optimal design for power and performance. To minimize the inter-vault communication, *our distributor only distributes the RP workload on a single chosen dimension.* Fig. 34 illustrates an example of the RP execution flow when distributing on the B-dimension. As it shows, the RP operations of Eq. 6,7,8 (1/2/3) exhibit parallelism along the B-dimension. Additionally, the multiplication operations of Eq. 9 (4) (i.e., $v_j^k \times \hat{u}_{j|i}^k$) can also be parallelized along the B-dimension. These parallelable workloads can be divided into snippets (i.e. green blocks) and distributed across the HMC vaults. Note that typical CapsNet workloads will generate

Table 5: Parameters for Modeling Inter-Vault Data Movement

Symbol	Description
I	DR iteration number
N_B	Scale for B-dimension i.e. batch size
N_L	Scale for L-dimension i.e. number of low-level capsules
N_H	Scale for H-dimension i.e. number of high-level capsules
N_{LC}	Scale for actual intra-iteration L-dimension
N_{vault}	Number of Vault
C_L	Dimension of low-level capsule i.e. number of scaler per low-level capsule
C_H	Dimension of high-level capsule i.e. number of scaler per high-level capsule
$SIZE_x$	Data size of a variable or packet head&tail
R	Compression ratio using algorithm-level optimization

way more snippets than the number of vaults in the HMC (e.g., up to 32 vaults in HMC Gen3).

Due to the aggregation requirements on all the dimensions during RP execution, there are always some workloads that cannot be processed in parallel on the selected dimension, leading to workload imbalance. For instance, the remaining operations of the RP procedure in Fig. 34 (i.e., the purple blocks), including partial Eq. 9 (5) and Eq. 10 (6) cannot be divided into snippets due to the lack of parallelism on the B-dimension. Additionally, these workloads usually require data to be aggregated in one place, making it hard to utilize all the HMC computation resources. Furthermore, the data size requested by the aggregation operations can be large, which may cause massive inter-vault communication and even the crossbar stalls. To reduce the inter-vault communication and increase hardware utilization, we propose to pre-aggregate the partial aggregation operations inside each vault for the corresponding allocated snippets. For example, when the RP workloads are distributed on B-dimension, the pre-aggregation can be performed for Eq.(9) to combine the b_{ij}^k from the snippets assigned to a specific vault before performing global inter-vault aggregation.

Guiding Distribution via an Execution Score: To achieve the optimal power/thermal and performance results, we propose a metric called execution score S to guide workload distribution, which quantitatively estimates the RP execution efficiency under a given workload distribution. S considers workload assignment to each vault, inter-vault communication overheads, device-dependent factors and inter-vault memory bandwidth. S is modeled as $S = 1/(\alpha E + \beta M)$.

where E represents the largest workloads distributed to a single vault (since even distribution across vaults is typically not feasible), which can be quantified based on the amount of allocated operations. M represents the amount of inter-vault data movement. Both E and M are affected by the distribution strategy and the model configuration. Finally, α and β represent the device-dependent coefficients, determined by HMC frequency and inter-vault memory bandwidth, respectively.

The calculation of S is independent of the online parameters, thus, the distribution strategy can be determined off-line before the actual inference. Then, the in-vault operations according to this selected distribution dimension will be generated by compiler and the corresponding workloads will be assigned into each vault via a hardware scheduler at runtime.

Note that our software optimization also works for the PIM processing, hence, we need to firstly produce the correctly dimension information after the compression to conduct the most efficient workload distribution. Our S-CapsNet will reduce the L-dimension during the processing of the RP iterations, i.e. N_L, which could significantly change the amount of the computation and the data movement involved in the iterations. But the computation outside the iterations remain the same, leading to two different dimension information. In order to precisely predict the best parallelism dimension, we additionally mark the actual L-capsule number inside the RP iterations as N_{LC}, which can be calculated using following equation:

$$N_{LC} = N_L/(R + 1\text{e}{-8}). \tag{14}$$

Where the N_L is the original L-dimenison scale, R represents the compression ratio (e.g. "1" means no compression) and $1e^{-8}$ is adopted to the keep the computation safe. Note that the iteration number (I) is another parameter that could be potentially impacted by our S-CapsNet. However, since the E represents the largest in-vault workloads, the iteration number for that vault can be considered as unchanged, because entire iterations are required for those EssCoe as discussed in Section.4.2.2.

With the parameter updated, we now demonstrate how to model S via estimating E and M on the three distribution dimensions.

Distribution on B-dimension: As Fig. 34 illustrates, the largest workload assigned to a single vault (E) consist of the workload snippets including 1/2/3/4 and the partial operations 5/6. With our optimizations, the single vault can get at most $\frac{\lceil \log_2(N_{vault}) \rceil}{N_{vault}}$ of the unparallelizable operations, where N_{vault} represents number of the HMC vaults. Using parameters shown in Table 5, E can be modeled as follows:

$$\begin{aligned} E_B = \lceil \frac{N_B}{N_{vault}} \rceil \times N_L \times N_H \times C_H \times (2C_L - 1) + I \times [\lceil \frac{N_B}{N_{vault}} \rceil \times \\ N_H \times C_H \times (2N_{LC} - 1) + \lceil \frac{N_B}{N_{vault}} \rceil \times N_H \times (3C_H + 19) + \\ \lceil \frac{N_B}{N_{vault}} \rceil \times N_{LC} \times N_H \times (2C_H - 1) + \frac{\lceil \log_2(N_{vault}) \rceil}{N_{vault}} + 4 \times C_H]. \end{aligned} \tag{15}$$

Since $N_L \gg 1$, the above equation can be simplified as:

$$E_B = \lceil \frac{N_B}{N_{vault}} \rceil \times N_L \times N_H \times [(\frac{4I}{R + 1\text{e}{-8}} - 1)C_H + 2C_L C_H - \frac{I}{R + 1\text{e}{-8}}]. \tag{16}$$

The inter-vault data communication consists of sending pre-aggregated b_{ij} from all the vaults to a single vault and scattering c_{ij} across all the vaults. The data transmission is in the form of packets with the head and tail size represented as $SIZE_{pkt}$. Therefore, the amount of data movements M can be represented as:

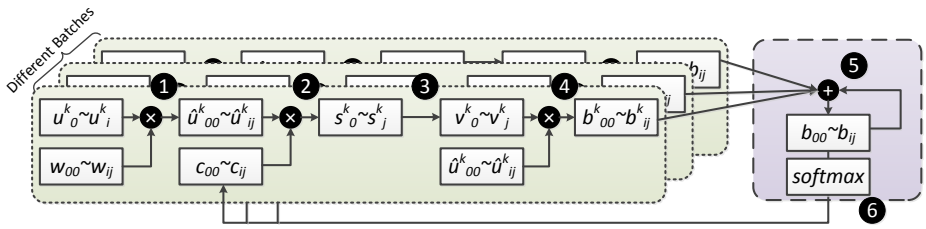


Figure 34: The execution diagram for the RP procedure with B-dimension distribution. The workloads in green blocks can be split across vaults, but workloads in purple blocks cannot be distributed via B-dimension.

$$\begin{aligned} M_B = I \times [&(N_{vault} - 1) \times N_{LC} \times N_H \times (SIZE_{b_{ij}} + SIZE_{pkt}) \\ &+(N_{vault} - 1) \times N_{LC} \times N_H \times (SIZE_{c_{ij}} + SIZE_{pkt})]. \end{aligned} \tag{17}$$

Distribution on L-dimension: As Table 4 illustrates, the RP operations of Eq. 6,9,10 can be divided into workload snippets on the L-dimension. Besides, partial operations from Eq. 7 (i.e., $\hat{u}^k_{j|i} \times c_{ij}$) also exhibit parallelism on the L-dimension. Thus, E can be represented as:

$$E_L = N_B \times \lceil \frac{N_L}{N_{vault}} \rceil \times N_H \times [\frac{2I}{R + 1\text{e}{-8}}(2C_H - 1) + C_H(2C_L - 1)]. \tag{18}$$

The inter-vault communication contains data all-reducing for of s_j and broadcasting v^k_j:

$$\begin{aligned} M_L = I \times [&N_B \times (N_{valut} - 1) \times N_H \times (SIZE_{s^k_j} + SIZE_{pkt}) \\ &+N_B \times (N_{vault} - 1) \times N_H \times (SIZE_{v^k_j} + SIZE_{pkt})]. \end{aligned} \tag{19}$$

Distribution on H-dimension: As Table 4 presents, only Eq. 10 cannot be parallelized on this dimension. Hence, E can be represented as:

$$E_H = N_B \times N_L \times \lceil \frac{N_H}{N_{vault}} \rceil \times C_H \times [2C_L - 1 + \frac{2I}{R + 1\text{e}{-8}}]. \tag{20}$$

The inter-vault communication contains data all-reducing for of b_{ij} and broadcasting c_{ij}:

$$M_H = I \times [(N_{vault} - 1) \times N_{LC} \times (SIZE_{b_{ij}} + SIZE_{pkt}) + N_{LC} \times (SIZE_{cij} + SIZE_{pkt})]. \quad (21)$$

4.4.2 Intra-Vault Level Design

In this section, we propose the intra-vault level design that effectively processes the sub-operations of each equation that are allocated to a vault. We target the design for IEEE-754 single precision (FP32) format, which provides sufficient precision range for CapsNet workloads [28]. Our design can also fit other data formats with minor modifications.

Intra-Vault Level Workload Distribution: In a basic HMC design, the logical layer of each vault contains a sub-memory controller to handle the memory access. In order to conduct RP specific computation, we introduce 16 processing elements (PEs) into each vault. This design overhead has been tested to satisfy both area and thermal constraints for HMC [63] (see detailed overhead analysis in Section 4.5.6). These PEs (integrated onto the logic layer) are connected to the sub-memory controller in the vault, shown in Fig. 35(left). Note that the number of parallel sub-operations on certain dimension is generally the orders of magnitude higher than the number of vaults in HMC. In other words, many parallel sub-operations will be allocated to the same vault. Hence they can be easily distributed on the same dimension and concurrently processed via the PEs without introducing additional communication overheads. There may exist some extreme cases that the number of parallel sub-operations allocated to the vault is smaller than the number of PEs, leading to low PE utilization. Since most equations in RP can be parallelized on more than one dimension, the workloads can then be distributed along a different dimension which can produce enough parallelism to fully utilize the PE resources.

Customized PE Design: There have been some studies [62, 63, 65] that integrate adders and multipliers onto the HMC logic layer to perform multiply-accumulation (MAC)

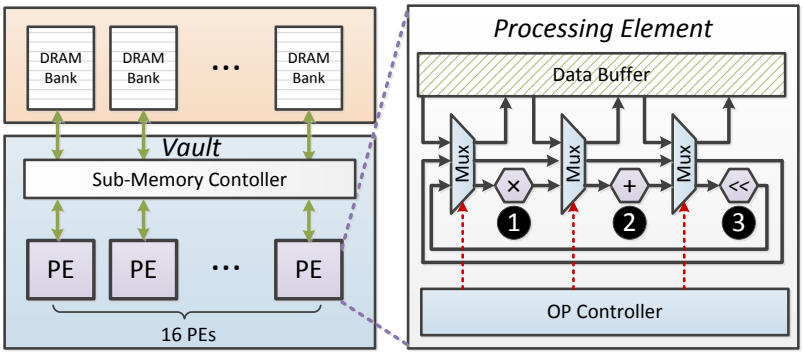


Figure 35: Intra-vault level Architecture Design.

which is an essential operation for deep learning (e.g., CNN). However, CapsNet's RP execution involves other operations beyond MAC. As discussed in Section 2.2.2, among the five RP equations, Eq. 6, Eq. 7 and Eq. 9 can be translated into MAC operations. But the other operations including Eq. 8 and Eq. 10 involve more complex functions such as division (Eq. 8), inverse square-root (Eq. 8) and exponential functions (Eq. 10) that require complicated logic design, resulting in large performance and power/thermal overheads [66, 67].

Operation Approximation: To gain high performance while maintaining low hardware design complexity, we propose approximation to simplify these operations with negligible accuracy loss. For simplifying division and inverse square-root functions of the FP32, we apply bit shifting [68], which is widely adopted in graphics processing domain [69–71]. For exponential function, we approximate the operation as follows:

The original exponential function can be transformed in the form of power function with the base as 2 [72]:

$$e^x = 2^{\log_2(e)\times x} = 2^y = 2^{\lfloor y \rfloor}(1 + 2^{y-\lfloor y \rfloor} - 1). \tag{22}$$

Where $\lfloor y \rfloor$ is the integer part of y, and $y - \lfloor y \rfloor$ is the decimal part.

Fig. 36(a) illustrates an example of representation transfer. First, the ep of both A and B will be subtracted from the bias b to get their real exponents (i.e., $ep-b$), as shown in Fig. 36(a)1& 3. Then, the most significant $ep - b + 1$ bits of A's significand (i.e., $1 + fraction$)

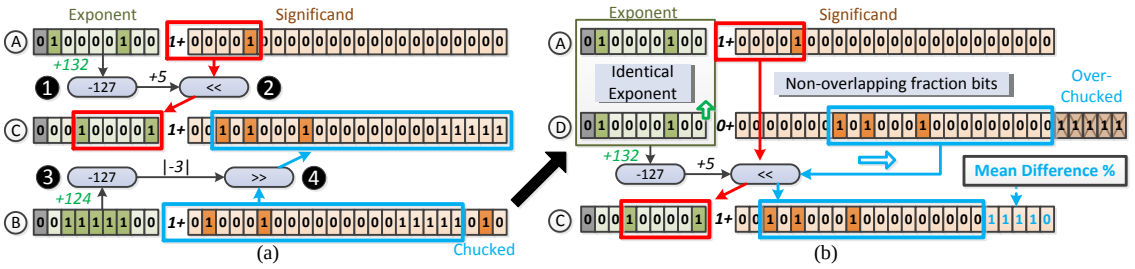


Figure 36: (a) An example of transferring the exponent representation A (i.e., $\lfloor y \rfloor + b$) and fraction representation B (i.e., $2^{y-\lfloor y \rfloor} - 1$) to the exponential function's result C in FP32 format. (b) Combining the exponent representation A and the fraction representation (i.e., D that transferred from B), and applying a unified bit shifting to obtain the exponential function's result C.

will be chunked and filled into the least significant $ep - b + 1$ bits of C's exponent field, with the remaining exponent bits in C filled by zeros, as shown in Fig. 36(a)2. We conduct s similar operation to transfer B to the C's fraction. Since B is a fraction value, its exponent is a non-positive number. B's significand will be logical shift right by $|ep - b|$ bits and then its most significant 23 bits are filled into C's fraction field, as shown in Fig. 36(a)4.

The above two transfers can be considered as bit shifting on the significand (i.e., $1 + fraction$) in FP32 format with the distance and direction determined by the real exponent (i.e., $ep - b$). As illustrated in Fig. 36, theB's exponent can increase to match the exponent of A with its significand bits logically shifting right. By doing this, the faction representation can be described as D in Fig. 36(b). Given the matched real exponent value (i.e., $ep - b$), the two representations, i.e., A and D in Fig. 36(b), now share the identical bit shifting operations. Additionally, A and D correspond to ExpResult's (i.e.,exponential function's result) integer and fraction, respectively. There is no overlapping between their fraction bits. Thus, these two representations can be combined (i.e., A OR D) followed by a unified bit shifting operation on the significand. Note that the exponent matching procedure (B→D) could over chuck several least significant bits which would originally be mapped into the ExpResult.

Since the exponent matching and combination of two FP32 numbers can be simply

considered as a FP32 addition, we can treat the ExpResult computation as an addition of the exponent and fraction representations (i.e., $\lfloor y \rfloor + b + 2^{y-\lfloor y \rfloor} - 1$) followed by the bit shifting operations. Note that the power of 2 function causes high complexity in both execution and logic design, we propose to simplify it as follows:

The above polynomial can be expanded as $(y + 2^{y-\lfloor y \rfloor} - (y - \lfloor y \rfloor) + b - 1)$; then, the average value Avg of $(2^{y-\lfloor y \rfloor} - (y - \lfloor y \rfloor))$ can be achieved via integrating the polynomial over $(y - \lfloor y \rfloor) \in [0, 1)$, which is fixed and can be obtained offline. With y represented by x, the exponential function can be described as follows:

$$ExpResult \simeq BS(\log_2(e) \times x + Avg + b - 1). \tag{23}$$

Where the BS is bit shifting operations with information from $ep - b$; and $log_2(e)$ is a constant that is computed offline.

Accuracy Recovery: Under the worst case scenari, othere might be several lowest significand bits chucked when mapping from D to C. It may cause some accuracy loss. To minimize the bit chucking impact, we analyze 10,000 exponential executions to collect the value differences between the approximated and original results. During the approximation execution, the accuracy loss will be recovered via enlarging the results by the mean percentage of the value difference. Note that our accuracy recovery scheme only introduces one additional multiplication operation during the inference, which guarantees the high performance and the low design complexity compared to other exponential approximation methodology, e.g., applying look-up tables [67]. Section 4.5.6 shows the detailed accuracy analysis.

Final PE Structure: According to the discussion above, the special functions can be simplified as a combination of addition, multiplication, and bit shifting operations. Thus, our intra-vault PE employs adders, multipliers, and bit-shifters to construct these special functions, as shown in Fig. 35. Specifically, our PE enables the flow configuration via the

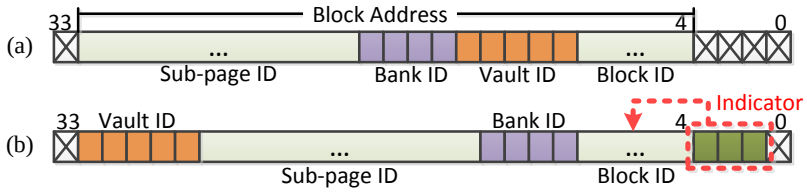


Figure 37: (a)The default address mapping of 8GB in HMC Gen3; (b) Our address mapping.

multiplexer (MUX) to support different types of operations. For example, PE execution flow 1-2 is for MAC operations; 3-2-1-2-1 is for inverse square-root operations; and 1-2-2-3 is for exponential function.

4.4.3 Contention Reduction in CapsNet Design

In this section, we discuss strategies to combat the memory-level contention in our PIM-CapsNet design. **Memory Address Mapping Mechanism:** In the default HMC design, memory access granularity is 16 bytes which is defined as a block, and the sequential interleaving mapping mechanism is applied to benefit from better bandwidth utilization. Moreover, MAX block is introduced to define the maximum number of blocks that one bank can provide at a time. Its size can be set to 32B, 64B, 128B, or 256B [58]. In this study, we redefine MAX block as a subpage in order to differentiate it from block, and one subpage is composed of multiple blocks. Fig.37(a) illustrates the default address mapping from HMC Gen3 specifications [58]. The lowest 4 bits and the highest 1 bit are ignored, and the remaining bits describe the block address. From its lower to higher bits, a block address is composed of several fields: the block ID in the sub-page (the number of bits is determined by the sub-page size), the 5-bit vault ID for 32 vaults, the 4-bit Bank ID for 16 banks per vault, and the sub-page ID in the bank. As can be seen, sub-pages with consecutive addresses will be first spread sequentially to different vaults and then different DRAM banks.

Note that our inter-vault level design requires consecutive blocks allocated into one vault to avoid high inter-vault communication. This can be easily addressed by moving up the

vault ID field to the highest field of the block address (as shown in Fig.37(b)) so that vault ID remains unchanged during the intra-vault memory address mapping. However, at the intra-vault level, PEs process their workloads in parallel and concurrently generate data requests, which may result in serious bank conflicts and vault request stalls (VRS).

Interestingly, we observe that most of the concurrent PE requests assigned to the same bank actually visit different blocks. Based on this, we propose a new memory addressing scheme to distribute these blocks to different banks, in order to significantly alleviate bank conflicts and decrease the VRS. However, simply distributing blocks to different banks could further increase the VRS as one PE may request multiple consecutive blocks at a time. Because in this case these blocks will reside in multiple banks, it leads to multiple accesses to these banks, resulting in higher bank conflicts. To ensure the consecutive blocks required by one PE are stored in the same bank, our scheme will dynamically determine the sub-page size according to the size of the requested data. As shown in Fig.37(b), we leverage bit 1 $\sim$ bit 3 in the lowest 4 ignored bits as the indicator to determine the sub-page size for the data requests, where range "000" $\sim$ "100" represents the sub-page size from 16B $\sim$ 256B. Given that the data requests from PEs and host GPU need to be allocated into different banks, the indicator bits are dynamically assigned by the page table during the virtual-physical address translation according to the storage requested by each variable involved in each execution thread.

Identifying Memory Access Priority: During CapsNet execution, resource contention from concurrent data requesting to the same vault from both host GPU and PEs could occur. Although the Conv/FC layers exhibit much better data locality than the RP, they may still need to periodically request data from the HMC. By default, the priority of a request is determined by the arrival time. But this can cause stalls if both sides are requesting to access the same bank in a vault, which may occur more frequently after applying our new address mapping.

To address this issue, we propose a runtime memory access scheduler (RMAS) to dynamically identify the priority of memory requests from both the host GPU and vault PEs. We first observe that, with our address mapping mechanism, consecutive data are likely to be distributed across the banks within a vault instead of being scattered over all the vaults. Thus, it is highly likely that each consecutive data request from the host will only access a single or few vaults at a time instead of all the vaults. This provides opportunities for some vaults to serve the GPU memory requests in rotation without affecting the overall HMC execution.

To quantify the impact of vault access priority, the runtime scheduler (RMAS) first collects the information of the issued operations by HMC and the number of PE requests (Q) from the vault request queues in HMC. It also collects the information of the issued operations from the host GPU about which and how many of vaults the operations are requesting from the HMC (defined as n_{max}). The collected information above from RMAS is associated with the HMC performance overhead if granting priority to access from either side. Thus, the performance impact of serving the two types of access requests from HMC and GPU can be quantified via the following overhead functions:

$$\kappa = \gamma_v \times n_h \times \overline{Q} + \gamma_h \times \frac{n_{max}}{n_h}. \tag{24}$$

Where κ represents the quantified performance impact; γ_v and γ_h represent the impact factors that are determined by the issued operations' type from HMC and host GPU, e.g., memory intensive operations corresponds to a large γ as their performance is more sensitive to the memory bandwidth than the computation intensive operations; $\overline{Q}$ is the average number of PEs' requests in the request queues from the targeted vaults; n_h is the number of vaults that are granted with access priority from the host GPU, which is in the range of $[0, n_{max}]$. If n_h is "0", all the target vaults will first process current HMC PEs requests before processing the GPU requests; while if n_h is n_{max}, all the target vaults will grant

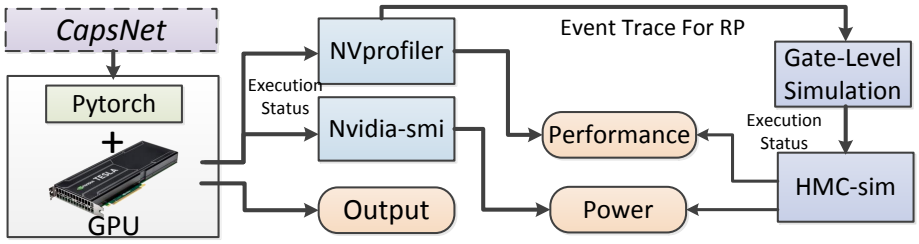


Figure 38: The Evaluation Infrastructure.

priority to the GPU requests. To achieve the minimal impact (i.e. $\min(\kappa)$), n_h should be equal to $\left|\sqrt{\frac{n_{max}\times\gamma_h}{Q\times\gamma_v}}\right|$, where $n_h \in [0, n_{max}]$. The RMAS will then send the control signals to n_h vaults that will give host GPU higher priority to access. Note that a vault with smaller Q has higher priority to be chosen by our RMAS to further reduce the impact on execution.

Given the S-CapsNet could skip the RP computation and data movement, some vaults could originally contain less workload compared with the other vaults. we propose to identify them and store more GPU related data in these less workloads vaults via enabling the RMAS with a two-level vault priority mechanism. The RMAS will first choose a broad range of low Q value vaults to be the potential GPU data provider and destination. Then, it will further locate those vaults with less workloads. Such information is generated along with the offline workload distribution, and saved into the buffer inside the RMAS. With the RMAS, the parallel workload unbalancing caused by the compression could be neutralized, thus, improving efficiency of both GPU-HMC pipeline.

4.5 Evaluations

4.5.1 Experimental Setup

Our evaluation infrastructure to evaluate the algorithm and architural optimization is shown in Fig. 38. We first employ Pytorch [48], a popular open-source machine learning framework that supports dynamic computation graphs, to perform the CapsNet on our host GPU. We then implement our software optimizations via modifying pytorch program

Table 6: Platform Specifications

Host Processor (GPU)	
Shading Unit	3584 @ 1190MHz
On-chip Storage	L1Cache/Shared: 24KB x 56
	L2 Cache: 4MB
Default Memory	HBM, 8GB, 320GB/s
HMC	
Capacity	8 GB, 32 Vault, 16 Banks/Vault
Bandwidth	Extl:320 GB/s, Intl: 512GB/s
No. of PEs per Vault	16
Frequency	312.5MHz

corresponding to the RP procedure. For the evaluation of our architectural level optimization, We then conduct a physical-simulator cooperated platform which is able to obtain the detailed performance and energy results using the execution status of CapsNet provided by Pytorch. From the physical side, we adopt the Nvidia Tesla-P100 [73] as our host processor to evaluate performance and energy consumption of the Conv/PrimeCaps/FC layers of CapsNet. The detailed execution time and power information for these layers are captured by using NVprofiler [55] and Nvidia-smi [74]. From the simulator side, we collect the event trace from host with the NVprofiler and pass it to a modified HMC-sim 3.0 [75] to simulate the computing and memory accesses of the HMC. Considering that HMC-sim 3.0 cannot provide precise execution cycles and power information for the logic layer design, we conduct a gate-level simulation on Cadence [76] to measure the execution latency and power consumption for our logic design (PE). We then integrate the gate level results and our PIM design in HMC-sim to obtain the performance and energy consumption of RP execution. Finally, since the execution of CapsNet is pipelined on the host processor and HMC, we combine the results from both sides via overlapping the execution time and accumulating the energy consumption. The detailed platform configurations are shown in Table 6. In this work, we select 12 different CapsNets corresponding to several datasets with different model configurations as our benchmarks which are introduced in Section 4.1 and shown in Table 3.

To evaluate the effectiveness of our software-hardware optimization, we compare with the following design scenarios: (1) Baseline: the state-of-the-art GPU accelerated CapsNet execution with the HBM memory (320GB/s). (2) GPU-ICP: the GPU accelerated CapsNet execution with ideal cache replacement policy. (3) S-CapsNet: our software-level optimization. (4) PIM-CapsNet: our combined inter-vault level and intra-vault level design for RP acceleration with the new memory addressing scheme and RMAS scheme. (5) PIM-Intra: our PIM-CapsNet without inter-vault level design, and the memory addressing scheme does not optimize the inter-vault data distribution. (6) PIM-Inter: our PIM-CapsNet without intra-vault level design, and the memory addressing scheme does not support the intra-vault bank conflict optimization. (7) RMAS-PIM and (8) RMAS-GPU: Our PIM-CapsNet with the naive memory access scheduling, which always grants HMC PEs higher priority than GPU for RMAS-PIM, and always grants GPU higher priority than HMC PEs for RMAS-GPU. (9) All-in-PIM: the HMC accelerated CapsNet execution, including compressed RP and other layers' execution. (10) SH-CapsNet: our combined software-hardware co-design for efficient CapsNet Execution.

4.5.2 Effectiveness of SH-CapsNet

Performance and Energy for RP execution: We first evaluate the effectiveness of our software and hardware optimizations on RP execution. Fig. 39 illustrates the normalized performance and energy consumption of our software-hardware co-design compared with both the GPU-based design (i.e., The Baseline and GPU-ICP) and the HMC-based design (i.e. PIM-CapsNet) for the RP execution. From the figure, we first observe that the GPU-ICP only outperforms Baseline by 1.14% on performance and 0.77% on energy during RP execution. This is because RP requires a large number of intermediate variables which exceed the on-chip storage. As a result, the cache policy improvements can barely reduce the off-chip memory accesses. Then, our GPU-based S-CapsNet can achieve on-average 1.91× (upto 2.76×) speedup and 43.53% energy consumption saving compared

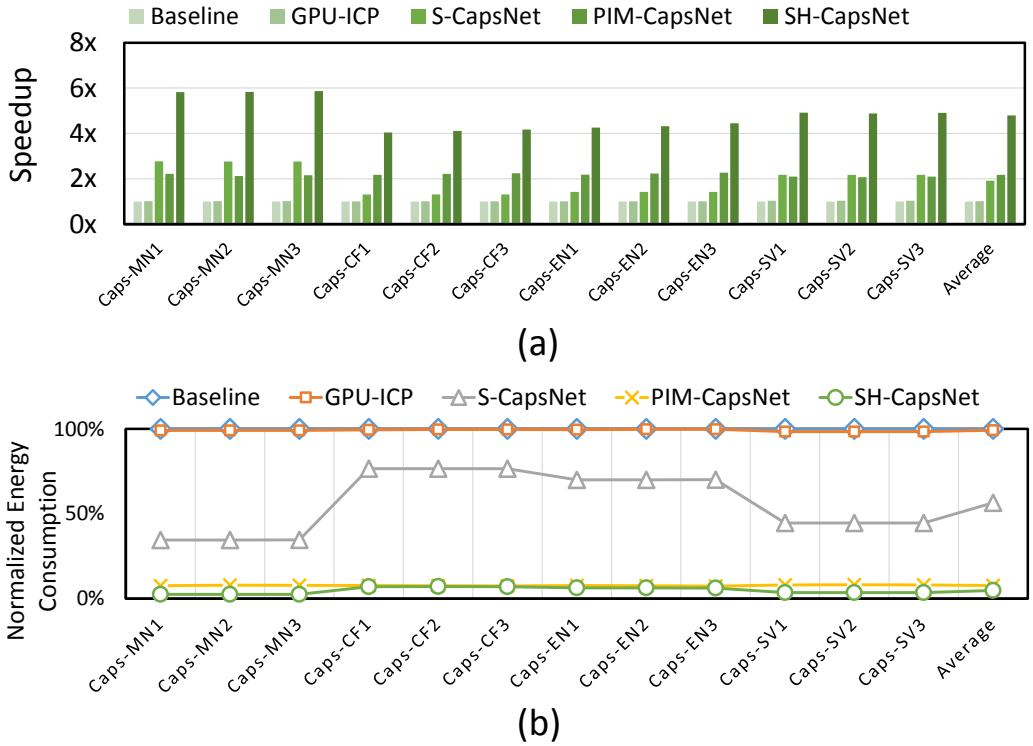


Figure 39: The (a) speedups and (b) normalized energy consumption of different designs on the RP execution.

with the baseline case. That is because S-CapsNet deducts both the computation and the data movements involved in the RP processing.

On the other hand, with RP procedure offloaded into the HMC, our PIM-CapsNet outperforms Baseline by 117% on performance by addressing the large number of memory access as well as the intensive synchronizations. Additionally, from Fig. 39(b), we observe PIM-CapsNet saves 92.18% on energy consumption comparing to Baseline. This is because the entire working power of our PIM design is much lower then host and PIM-CapsNet is able to reduce a huge number of data movements between host and HMC. Moreover, we observe that PIM-CapsNet can achieve better performance and energy saving for RP execution in larger size CapsNet, e.g. 2.27× speedup and 92.52% energy saving of accelerating Caps-EN3 compared with 2.09× speedup and 91.90% energy saving of accelerating Caps-SV1. This implies that PIM-CapsNet exhibits the scalability in optimizing the RP execution with the

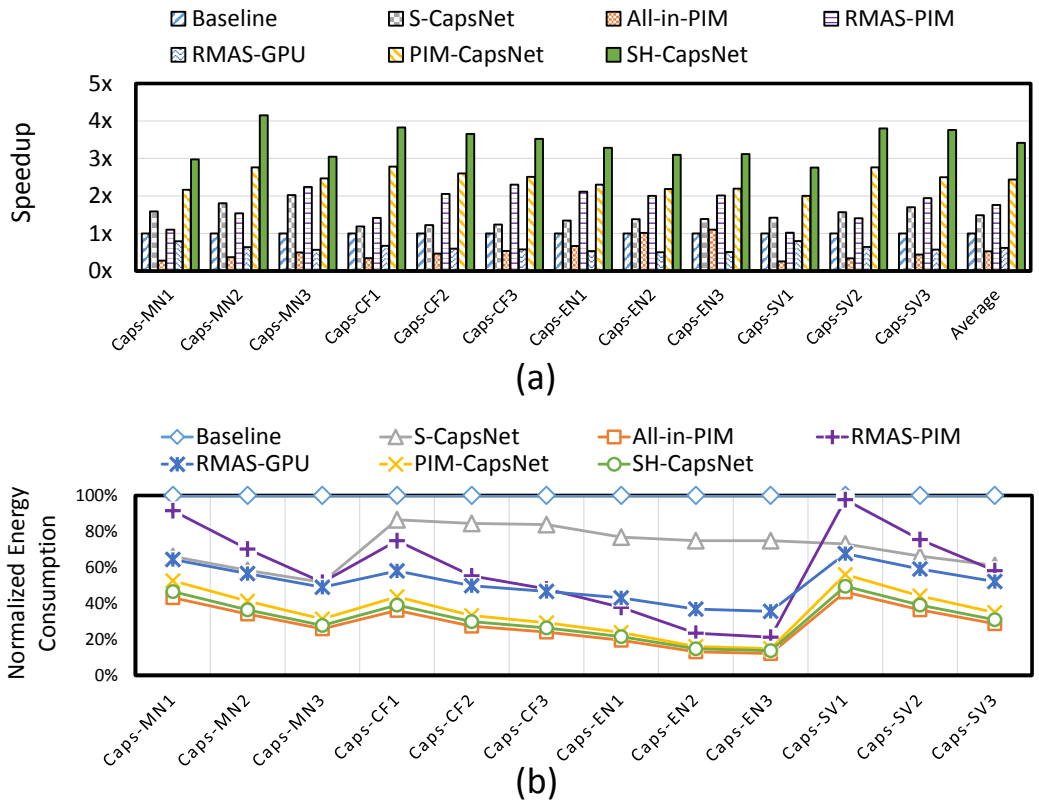


Figure 40: The (a) speedups and (b) normalized energy consumption of different design on entire CapsNet execution.

ever-increasing network size.

Finally, our SH-CapsNet takes both advantages of the computation reduction and the efficient low power PIM design, that achieves significantly on average 4.79× speedup and saves 95.05% of the energy compared with the baseline case.

Performance and Energy for Entire CapsNet: Fig. 40 shows the normalized speedup and energy of different design schemes during entire CapsNet execution. To understanding the effectiveness of the pipeline execution after enabling processing-in-memory. We compare the speedup and energy under different architecture design schemes: the GPU-based designs (i.e., Baseline and S-CapsNet); HMC-based design (All-in-PIM); and GPU-HMC hybrid design (i.e. RMAS-PIM, RMAS-GPU, PIM-CapsNet and SH-CapsNet). From the figure, we observe that the S-CapsNet can achieve on average 1.48x performance

improvement and 28.51% energy saving over the baseline case due to the software-level optimization on the RP execution. Compared to the GPU-based processing, although the All-in-PIM design achieves better execution efficency (i.e. performance /energy consumption) by reducing the energy consumption by 71.11%, it causes the 47.56% performance drop. This is because we mainly focuses on the RP procedure optimization at minimal design cost in HMC, and such strategy can hardly achieve the best performance for Conv/FC layers.

For the GPU-HMC hybrid designs, We observe that these naive schedulers (RMAS-PIM and RAMS-GPU) can hardly achieve the best performance and energy saving compared with either the PIM-CapsNet or the SH-CapsNet due to the stalls caused by unbalanced GPU-HMC memory accesses. We also observe that, with RP compression, the SH-CapsNet outperforms the PIM-CapsNet by $1.41x$ on the performance and 3.82% on the energy saving. This is because the compression not only reduces the RP workloads, but also enhances our RMAS scheduling on the memory access balancing between the GPU and the HMC. In summary, our software and hardware co-design outperforms baseline on both performance (i.e., on average 3.76×) and energy saving (i.e., 68.73%). According to above discussion, our software optimization can even enhance the hardware optimizations to enable efficient pipelined execution, leading to even higher performance improvement compared with the acceleration for RP execution only.

4.5.3 Compression Schemes Comparison

In order to evaluate the sensitive of our proposes under different compression schemes, we compare the accuracy loss and the speedup of RP processing between our simple compression schemes (i.e., Simple-D1, Simple-D2 and Simple-D3) and our importance-aware compression scheme (IAC). As shown in Fig. 41, the simple compression schemes are able to significantly improve the performance over the baseline by trading the accuracy of CapsNet. For example, on average, simple-D1 achieves 3.93× performance improvement over

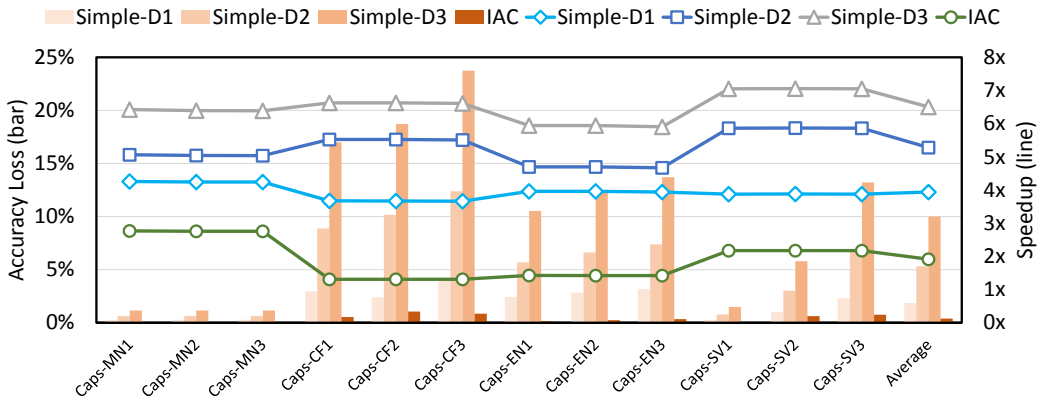


Figure 41: The accuracy loss (bar) and speedup (line) when implementing different compression schemes for the RP execution on the baseline GPU. The Simple-Dn represents the simple compression scheme with the reuse-distance of n, and the IAC is the importance-aware compression scheme.

the baseline with 5.72% accuracy loss. the performance improvement and accuracy loss even increase With the growth of the reuse distance (from D1 to D3). This is because the simple compression, on the one hand, reduce more computations that could accumulate more value difference. On the other hand, the simple compression fails to reserve the important coefficients that results in imprecise output capsule value. We also notice that the accuracy loss of the simple compression for Caps-MN and Caps-SV1 is relatively small, indicating that it can still be apply to the simple tasks with small iterations requirements, e.g. Caps-MN, where the coefficients are highly similar.

Finally, We observe that the IAC can always maintain a high accuracy (only 0.3% accuracy loss on average). Meanwhile, the IAC is capable to achieve on average 1.91× speedup (upto 2.76×). Given the CapsNet is design to achieve the higher accuracy, the IAC can be widely applied to achieve better performance without critical accuracy penalty.

4.5.4 Effectiveness of Intra-Vault and Inter-Vault Level Designs

To better understanding the effectiveness of our intra-vault and inter-vault level designs, we compare PIM-CapsNet with other PIM design scenarios for the RP execution only. Fig. 42(a) illustrates the evaluation results of normalized performance with the

breakdown factors for different PIM designs. From the figure, we have several observations. First, even though PIM-Intra achieves 1.22× speedup over Baseline, the inter-vault communication overheads contribute on averages 45.24% to the overall performance. This is because the PIM-intra design induces massive concurrent data transfer between the centralized computation units in the logic layer and the DRAM banks in vaults, leading to high crossbar utilization and serious stalls. Second, PIM-Inter decreases the performance by 4.73% compared with the Baseline. Compared with PIM-Intra, the inter-vault communication overheads have been significantly reduced in PIM-Inter, but the vault request stalls (VRS) grow which contribute on average 57.91% to the execution time due to the serious bank conflicts within the vault. Finally, PIM-CapsNet improves the performance about 127.83%/76.62% on average comparing to PIM-Inter/PIM-Intra by reducing both inter-vault communications and VRS. From the energy perspective, as Fig. 42(b) shows, these PIM designs achieve high energy saving compared with Baseline by executing RP on energy-efficient PIM design and our PIM-CapsNet on average outperforms the PIM-Inter/PIM-Intra by 4.81%/4.52% respectively on energy saving.

4.5.5 Sensitivity to PE Frequency Scaling

We conduct the sensitivity analysis on inter-vault level workload distributions in our PIM-CapsNet when different frequency is adopted in the PEs, e.g., 312.5MHz, 625MHz and 937.5MHz. Note the frequency scaling will be controlled under a certain range without violating the HMC thermal constraint. Fig. 43 shows the speedup achieved by selecting different distribution dimensions (i.e., B-dimension, L-dimension, and H-dimension) under the above three different frequencies. The darker color indicates the higher speedups (e.g., the red color indicates the substantial speedups while the yellow color means trivial speedups).

It is obvious that PIM-CapsNet can achieve better improvement with higher execution frequency. We also notice that the selection of the distribution dimension changes with frequency scaling. For example, for Caps-SV3, the L-dimension distribution can achieve the

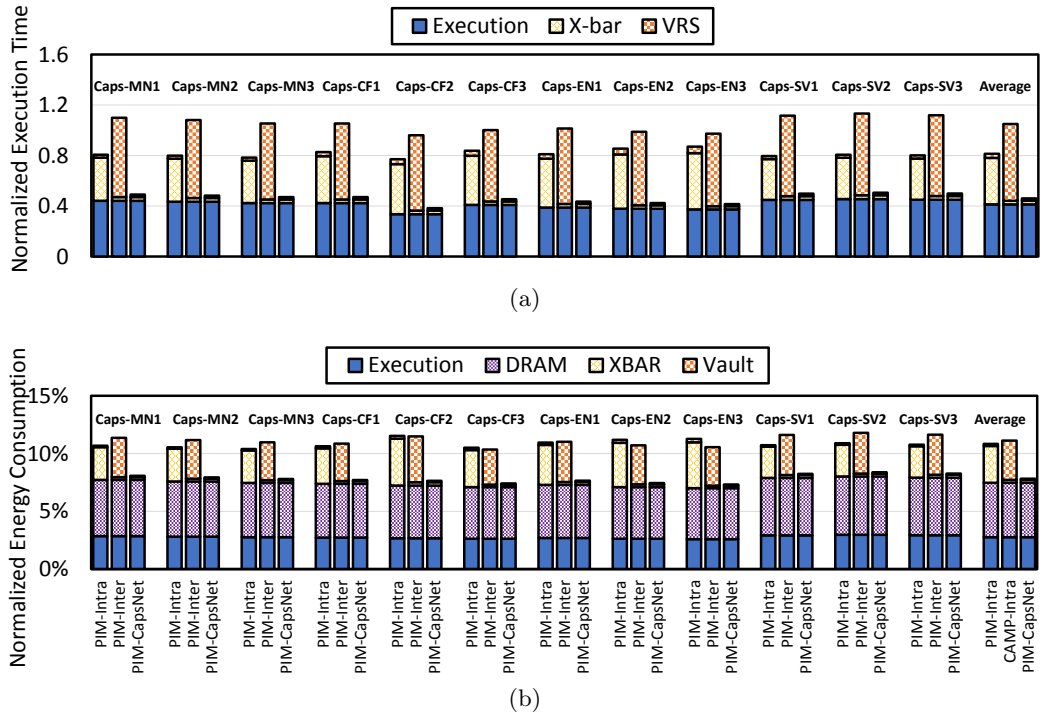


Figure 42: The breakdown of the factor to (a) normalized performance and (b) normalized energy consumption when performing RP execution on different PIM designs.

best performance under 312.5MHz execution frequency; but the H-dimension distribution achieves the best performance when frequency increases to 937.5MHz. This indicates that the dimension selection is affected by not only the network configurations, but also the hardware configurations, e.g. processing frequency.

4.5.6 Overhead Analysis

Accuracy Analysis: Table.7 illustrate the absolute accuracy after implementing either our software optimization (S-CapsNet) or hardware optimization (PIM-CapsNet) or both (SH-CapsNet). As it shows, our software optimization only induce on average 0.38% accuracy loss, while our approximation in PIM-CapsNet with accuracy recovery will only cause 0.04% accuracy difference. Finally, our SH-CapsNet achieves the same accuracy as S-CapsNet, which indicates that the tiny difference of value caused by our PIM-CapsNet

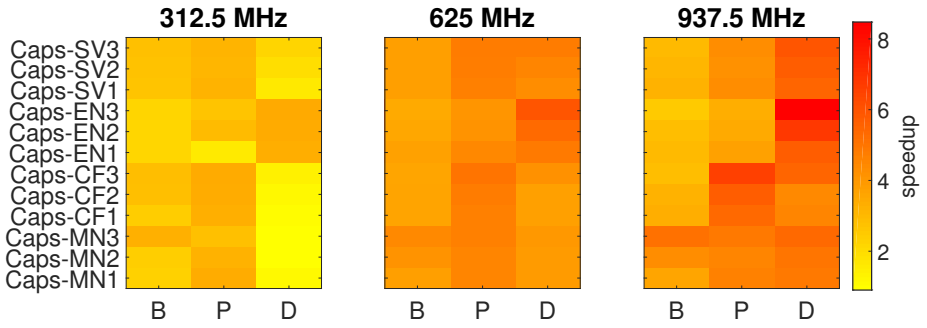


Figure 43: The speedup (heat map) achieved by different workloads distribution dimensions (X-axis) under different HMC execution frequency. Red means better improvement.

Table 7: Accuracy Validations

	Caps-MN1	Caps-MN2	Caps-MN3	Caps-CF1	Caps-CF2	Caps-CF3
Origin	99.75%	99.75%	99.75%	89.40%	90.03%	90.43%
S-CapsNet	99.74%	99.74%	99.74%	88.86%	88.99%	89.58%
PIM-CapsNet	99.75%	99.75%	99.75%	89.37%	90.02%	90.39%
SH-CapsNet	99.74%	99.74%	99.74%	88.86%	88.99%	89.58%
	Caps-EN1	Caps-EN2	Caps-EN3	Caps-SV1	Caps-SV2	Caps-SV3
Origin	88.74%	85.01%	82.36%	96.70%	95.90%	95.90%
S-CapsNet	88.60%	84.77%	82.02%	96.69%	95.29%	95.15%
PIM-CapsNet	88.69%	84.96%	82.34%	96.42%	95.90%	95.90%
SH-CapsNet	88.60%	84.77%	82.02%	96.69%	95.29%	95.15%

approximation can be neutralized by our S-CapsNet scheme.

Area Analysis: Our PIM-CapsNet introduces 16 PEs and operation controller into each HMC vault architecture, with one RMAS module located in the logic layer. Based on our gate-level simulations, our logic design for 32 vaults and the RMAS together incurs $3.11mm^2$ area overheads under the $24nm$ process technology, which only occupy 0.32% HMC logic surface area.

Thermal Analysis: Note that our logic design raises the total power of the HMC, which could cause thermal issues for the 3D stacked memory architecture. We observe the average power overhead of our logic design is $2.24W$, which is below the maximum power overhead (i.e., $10W$ thermal design power (TDP) [77]) the HMC can tolerate.

5 Future Works: Improving Computation Efficiency for LSTM Training

Note that LSTMs have completely different computation patterns from CNNs, moreover, within the LSTM training, the forward (FW) and backpropagation (BP) procedures have totally different dataflows and computations. In the future works, we will leverage the unique features of LSTM networks to explore further algorithm and architecture support for fast and low-power LSTM BP procedure, and enable the real-time training for LSTM to satisfy the relaxed request on real-time learning speed.

5.1 Reducing LSTM Backpropagation Workloads

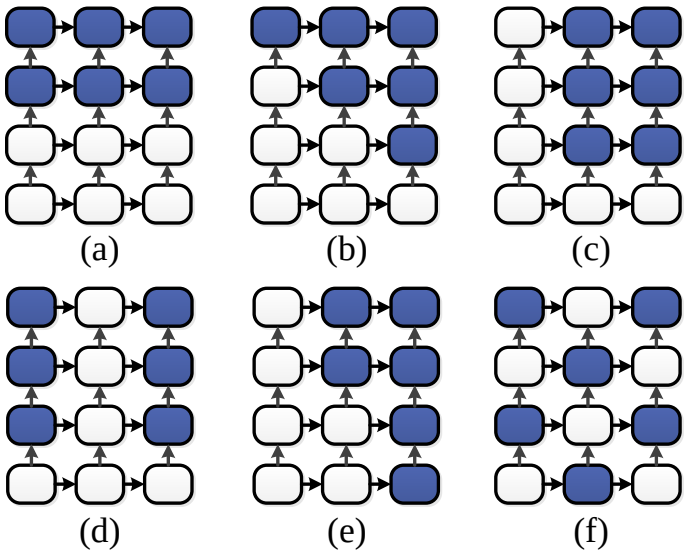


Figure 44: LSTM BP workload reduction with different LSTM fine-tuning cell patterns.

With the proceeding of training iterations, partial weight updating and the corresponding computations becomes trivial. In order to improve the training efficiency while maintaining the accuracy, we propose to leverage the knowledge of fine-tuning from the transfer learning (TL). TL can transfer the model knowledge domain for the specific tasks via measuring the similarity between the training model and target model, followed by the different

model tuning strategy according to the measurement. If the target model is highly overlapped with the trained datasets, light training will be conducted for last few layers; whereas, for the case target model exhibit very different knowledge domain compared with the trained model, more layers will be involved in the TL procedure. Although TL can enhance the user experience on the LSTM model accuracy, its localized executions on mobile devices will induce many challenges. (i) TL pattern exploration for LSTM: Currently, the transfer learning for LSTM share the similar strategy with CNN, which conduct the training for the entire unrolled cells in the last few layers. And when there exists the huge difference between the knowledge domain of local dataset and the general dataset, the executions of the TL exhibit high complexity, as TL involves large number of layers. However, we observe the essential difference between the BP execution of the CNN layer and the LSTM layer is that the weight gradients of LSTM are sequential accumulated by the cells. Note that the last few layers are more sensitive to the knowledge domain, and our previous work (chapter.3) indicates that unrolled cells exhibit different impact on their next timestamp cell, we conjecture only partial cells are sensitive to the knowledge domain changes. And we propose to explore the sensitive cell pattern for LSTM transfer learning. We will evaluate the different sensitive cell patterns with different applications. For sensitive cell pattern in one cell, two patterns are considered: (a) cells gains the sensitivity with the growth of the timestamp; (b) sensitive cells evenly distributed through the timestamps. On the other hand, there are also two sensitive cell patterns across the sensitive layers: (a) sensitive layers contains same amount of sensitive cell; (b) sensitive cells number changes with sensitivity of the layers. (ii) Architecture design for efficient TL execution: The transfer learning will bring addition executions for mobile device, causing performance and energy overheads. Therefore, it is essential to develop an architecture for high efficient executions. Note that the purpose of FW and BP are different, it is essential to conduct the specific design for both executions of Inf and BP procedure. And usually applications generate different requirements on the response time, power budget and output accuracy, hence, we plan to equip our FW and

BP accelerator design with controller to enable the different scheme policies of sensitive cell selection, approximation computing, DVFS (dynamic voltage & frequency scaling) for achieving those requirements.

5.2 Insignificant cell skipping within LSTM BP layer

The sequence length of the unrolled layer largely affects the LSTM BP performance and energy due to the sequential execution of BP cells in the layer. The deduction in the sequence length could potentially benefit the performance at the cost of accuracy drop. We notice that the cell error varies across cells in the same LSTM layer which produces different weight gradient, thus, we propose to analyze the importance of different BP cells and skipping the insignificant cells that has trivial impact on gradients computation. (i) Insignificant cell identification and execution reorganization: We plan to explore the relationship between the output error and the weight gradients, and locate the insignificant cells via analyzing the output errors at the beginning of the LSTM BP layer execution. When skipping the insignificant cell, the execution dependence connected to this cell can be removed so that cells from the same layer can be concurrently processed to enhance the computation parallelism. (ii) Skipping all insignificant cells may affect the error values for the previous layer, and may cause the reduction of the model accuracy and/or convergence speed. We thus propose the accuracy-aware selective cell skipping. We will explore the importance of the errors pass between the LSTM layers using the cell inputs and gates values of the previous layer obtained during the FW stage. If the passing error makes little contribution to the cell error of the cell in previous layer, the cell that passes this kind of error can be skipped. (iii) We would like to further compensate the weight gradients caused by the insignificant cell skipping. One possible approach is to estimate weight gradients by exploring the relationship between the output error and the weight gradients.

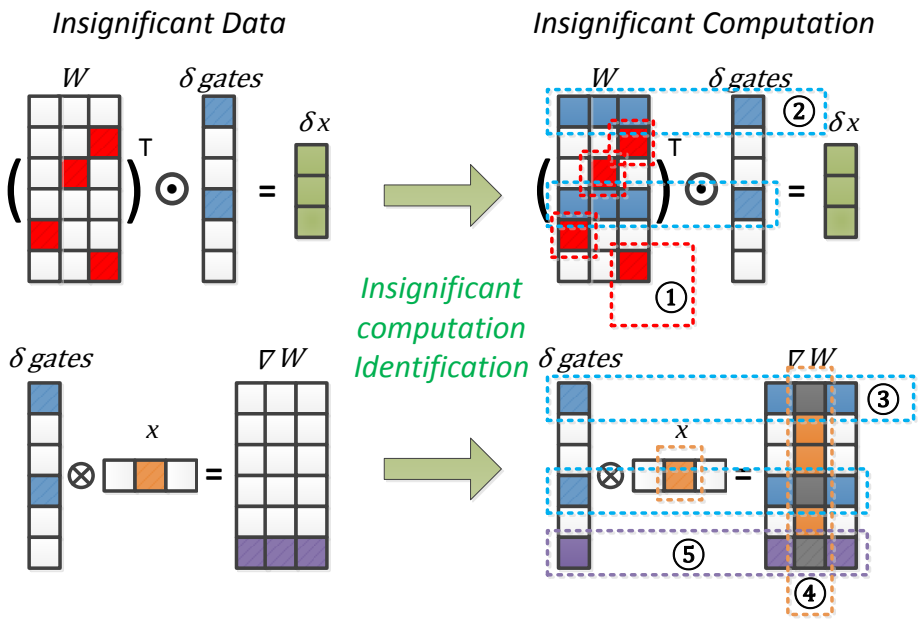


Figure 45: An example of insignificant computation identification for both DIC ($1 \sim 4$) and SIC (5) during the execution of $W \odot \delta gates = \delta x$ and $\delta gates \otimes x = \nabla W$ from BP of LSTM training. The colored elements are removable, and the dashed boxes indicate the insignificant computations located via insignificant data.

5.3 Reducing insignificant computations within LSTM BP cell

Due to the tremendous difference between DNN and LSTM on network type and scale, there are few insignificant weights in the LSTM to enable previous pruning technique [78]. In the future work, we aim to explore the LSTM-specific insignificant computation during the BP procedure. We propose to investigate all data required by the major executions within each LSTM BP cell, i.e. layer input weights W, context weights U, gate loss $\delta gates$, context h, layer input x, layer input weight updates ∇W, context weight updates ∇U, comprehensively figure out the insignificant computation at algorithm level and furthermore, maximize data reduction with a customized architecture design. There are several research challenges: (i) How to locate the insignificant computations? The insignificant computation can be classified into dynamic and static insignificant computation (i.e., DIC and SIC). The DICs vary across cells and training iterations, they are determined by the low-magnitude data of cell inputs (i.e., $W, U, \delta gates, h, x$) which generate the near-zero outputs for the cell

execution. We will locate DICs via the runtime identification of the near-zero input data at element-level, as shown in Fig. 45($1 \sim 4$). On the other hand, the SICs are relatively stable throughout the entire training procedure, they are determined by the trivial data in cell outputs (i.e.,$\nabla W, \nabla U$) which have less impact on the weight updating throughout the training stage. We observe that if the rows in $\nabla W/nablaU$ conjugate with the corresponding cell inputs, the weight updating of these rows can be considered as trivial, as shown in Fig.45(5). We plan to use this to identify the SIC computations at the beginning several training iterations via locating the insignificant $\nabla W, \nabla U$ data at row-level. (ii) How to explore efficient pruning scheme achieving the optimal trade-offs between performance/energy and accuracy? Although DIC can be considered as prunable in the LSTM training procedure, inefficient pruning (e.g., pruning caused irregular parallelism and over-pruning) may affect the model accuracy, the convergence speed of the LSTM training, and also the performance/energy gains. In addition, since the SIC identification requires the additional executions on locating insignificant data at row level, the identification speed needs to be considered. We would like to develop an intelligent pruning scheme that takes the features of LSTM BP training into the consideration to dynamically control the prune pattern for DIC and identification speed for SIC. (iii) Our pruning scheme makes the computations more irregular and sparser, we plan to further explore a customized architecture design to efficiently support the pruning in LSTM BP. Specifically, we aim to achieve the dynamic workload adjustment for different data sparsity across cells which intelligently matches the BP cell workloads from different execution batches with the execution capability of the hardware device.

6 Related Works

CNNs Optimizations: There have been multiple studies on CNNs optimizations. Some of them target at CNNs optimizations on mobile GPUs [10,79]. while others design the ASIC accelerators for high performance neural netwroks [80, 81]. Also several studies have been well conducted on the weight compression for CNNs via erasing trivial elements [82–86]. And the execution-efficiency-aware weight matrices compression for CNNs are well studied by [87–90]. For example, [87] proposes DeftNN to compress the CNNs weight matrix by eliminating columns, [88] 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.

RNNs Optimizations: There are also some works on RNNs computation optimizations. [91, 92] 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, [93, 94] 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.

Computation Flow Optimizations: Several studies have exploited optimizations on computation flow [95–98]. 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.

PIM-Based NN Acceleration: There have been multiple studies focus on exploring PIM-based neural network accelerators [99–104]. For example, [100] employs the ReRAM-based PIM to conduct efficient neural network execution. However, the massive intermediate variables involved in the RP could induce both performance and energy overheads

in the ReRAM design for frequent value updating. Besides, [62, 63, 105] propose the CNN accelerator design using 3D stack PIM technique. Since the execution pattern of the RP procedure are far different from CNN layers, previous logic layer designs of 3D stacked memory exhibit low efficiency on the RP execution. To our best knowledge, this is the first work that leverages HMC to explore a customized architectural design for efficient CapsNet acceleration.

Workload Distributions: Several studies have explored the workload distribution for efficient neural network execution [106–110]. For example, [108] proposes to distribute the parallel workloads of convolutional layer among the computation units within a single device; and [110] splits the convolutional execution and distribute the workloads into multiple devices. Their methods have achieved significant CNN accelerations via greatly improving the resource utilization. Since the execution of the RP procedure is much more complicated than the convolutional layer and involves strong execution dependence, these studies are not applicable to the CapsNet acceleration.

Exponential Approximation: There are also some works on approximation for exponential function [66,67,111,112]. For example, [67] leverages the lookup table to conduct the fast exponential execution, which causes substantial performance and area overheads. [112] accelerates exponential function via software optimizations, which are hard to be implemented via the simple logic design. In this work, we conduct the efficient acceleration for exponential function (Section 4.4.2) with low design complexity and low power overhead.

7 Conclusions

In recent years, the LSTM and CapsNet has outperformed the CNN on the natural language processing and image processing tasks and becomes increasing popular in many areas. However, LSTMs exhibit quite inefficient memory access pattern when executed on mobile GPUs due to the redundant data movements and limited off-chip bandwidth. And processing efficiency of CapsNets on GPUs often cannot achieve the desired level for fast real-time inference. To address these challenge, I propose two hardware and software co-design approaches via modifying the execution flow and enabling processing-in-memory technique, respectively.

In the first work, 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.

In the second work, we propose the software-level optimizations named *S-CapsNet* along with a hybrid computing architecture design named *PIM-CapsNet*. The S-CapsNet reduces the computation and data movements leveraging the data-similarity and the redundancy of the computation and memory access of the routing procedure. And our PIM-CapsNet leverages the processing-in-memory capability of today's 3D stacked memory to conduct the off-chip in-memory acceleration solution for the routing procedure, while pipelining

with the GPU's on-chip computing capability for accelerating CNN types of layers in CapsNet.Evaluation results demonstrate that either our software or hardware optimizations can significantly improve the CapsNet execution efficiency. Together, our co-design can achieve greatly improvement on both performance (3.41x) and energy savings (68.72%) for CapsNet inference, with negligible accuracy loss.

Bibliography

[1] Igor Kononenko. Machine learning for medical diagnosis: history, state of the art and perspective. *Artificial Intelligence in medicine*, 23(1):89–109, 2001.

[2] Bradley J Erickson, Panagiotis Korfiatis, Zeynettin Akkus, and Timothy L Kline. Machine learning for medical imaging. *Radiographics*, 37(2):505–515, 2017.

[3] Anna L Buczak and Erhan Guven. A survey of data mining and machine learning methods for cyber security intrusion detection. *IEEE Communications Surveys & Tutorials*, 18(2):1153–1176, 2016.

[4] Justin Grimmer. We are all social scientists now: How big data, machine learning, and causal inference work together. *PS: Political Science & Politics*, 48(1):80–83, 2015.

[5] Jens Kober, J Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013.

[6] Daniel Anthony Lopez. *Evolving GPU-Accelerated Capsule Networks*. PhD thesis, 2018.

[7] Ayomide Yusuf and Shadi Alawneh. A Survey of GPU Implementations for Hyperspectral Image Classification in Remote Sensing. *Canadian Journal of Remote Sensing*, pages 1–19, 2018.

[8] Arthur Stoutchinin, Francesco Conti, and Luca Benini. Optimally Scheduling CNN Convolutions for Efficient Memory Access. *arXiv preprint arXiv:1902.01492*, 2019.

[9] Chao Li, Yi Yang, Min Feng, Srimat Chakradhar, and Huiyang Zhou. Optimizing memory efficiency for deep convolutional neural networks on GPUs. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 633–644. IEEE, 2016.

[10] Mingcong Song, Yang Hu, Huixiang Chen, and Tao Li. Towards Pervasive and User Satisfactory CNN across GPU Microarchitectures. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*, pages 1–12. IEEE, 2017.

[11] Zhaoyun Chen, Lei Luo, Wei Quan, Yang Shi, Jie Yu, Mei Wen, and Chunyuan Zhang. Multiple CNN-based Tasks Scheduling across Shared GPU Platform in Research and Development Scenarios. In *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 578–585. IEEE, 2018.

[12] Xingyao Zhang, Chenhao Xie, Jing Wang, Weidong Zhang, and Xin Fu. Towards Memory Friendly Long-Short Term Memory Networks (LSTMs) on Mobile GPUs. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 162–174. IEEE, 2018.

[13] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 609–622. IEEE Computer Society, 2014.

[14] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. ShiDianNao: Shifting vision processing closer to the sensor. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 92–104. ACM, 2015.

[15] Xuda Zhou, Zidong Du, Qi Guo, Shaoli Liu, Chengsi Liu, Chao Wang, Xuehai Zhou, Ling Li, Tianshi Chen, and Yunji Chen. Cambricon-S: Addressing Irregularity in Sparse Neural Networks through A Cooperative Software/Hardware Approach. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 15–28. IEEE, 2018.

[16] Mostafa Mahmoud, Kevin Siu, and Andreas Moshovos. Diffy: a Déjà vu-Free Differential Deep Neural Network Accelerator. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 134–147. IEEE, 2018.

[17] Youngeun Kwon and Minsoo Rhu. Beyond the Memory Wall: A Case for Memory-Centric HPC System for Deep Learning. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 148–161. IEEE, 2018.

[18] Youjie Li, Jongse Park, Mohammad Alian, Yifan Yuan, Zheng Qu, Peitian Pan, Ren Wang, Alexander Schwing, Hadi Esmaeilzadeh, and Nam Sung Kim. A Network-Centric Hardware/Algorithm Co-Design to Accelerate Distributed Training of Deep Neural Networks. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 175–188. IEEE, 2018.

[19] C Deng, S Liao, Y Xie, KK Parhi, X Qian, and B Yuan. Permdnn: Efficient compressed deep neural network architecture with permuted diagonal matrices. In *MICRO*, 2018.

[20] Sepp Hochreiter and Jürgen Schmidhuber. LSTM can solve hard long time lag problems. In *Advances in neural information processing systems*, pages 473–479, 1997.

[21] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *arXiv preprint arXiv:1406.1078*, 2014.

[22] Sepp Hochreiter. Untersuchungen zu dynamischen neuronalen Netzen. *Diploma, Technische Universität München*, 91:1, 1991.

[23] cuDNN 5.1. `https://developer.nvidia.com/cudnn/`.

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

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

[26] Aryan Mobiny and Hien Van Nguyen. Fast capsNet for lung cancer screening. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 741–749. Springer, 2018.

[27] Geoffrey E Hinton, Alex Krizhevsky, and Sida D Wang. Transforming auto-encoders. In *International Conference on Artificial Neural Networks*, pages 44–51. Springer, 2011.

[28] Sara Sabour, Nicholas Frosst, and Geoffrey E Hinton. Dynamic routing between capsules. In *Advances in neural information processing systems*, pages 3856–3866, 2017.

[29] Geoffrey E Hinton, Sara Sabour, and Nicholas Frosst. Matrix capsules with EM routing. 2018.

[30] Edgar Xi, Selina Bing, and Yang Jin. Capsule network performance on complex data. *arXiv preprint arXiv:1712.03480*, 2017.

[31] Sai Samarth R Phaye, Apoorva Sikka, Abhinav Dhall, and Deepti Bathula. Dense and diverse capsule networks: Making the capsules learn better. *arXiv preprint arXiv:1805.04001*, 2018.

[32] Sasu Tarkoma, Matti Siekkinen, Eemil Lagerspetz, and Yu Xiao. *Smartphone energy consumption: modeling and optimization*. Cambridge University Press, 2014.

[33] Jose Maria Arnau Montañés. Energy-efficient mobile GPU systems. 2015.

[34] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to Forget: Continual Prediction with LSTM. 1999.

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

[36] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[37] Adam Paszke, Sam Gross, Soumith Chintala, and Gregory Chanan. PyTorch: Tensors and Dynamic neural networks in Python with strong GPU acceleration, 2017.

[38] Baidu DeepBench. `https://svail.github.io/DeepBench/`.

[39] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 163–174. IEEE, 2009.

[40] cuBLAS. `http://docs.nvidia.com/cuda/cublas/index.html`.

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

[42] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. Learning Word Vectors for Sentiment Analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142–150, Portland, Oregon, USA, June 2011. Association for Computational Linguistics.

[43] Bo Pang and Lillian Lee. Seeing stars: Exploiting class relationships for sentiment categorization with respect to rating scales. In *Proceedings of the ACL*, 2005.

[44] Jason Weston, Antoine Bordes, Sumit Chopra, Alexander M Rush, Bart van Merriënboer, Armand Joulin, and Tomas Mikolov. Towards AI-Complete Question Answering: A Set of Prerequisite Toy Tasks. *arXiv preprint arXiv:1502.05698*, 2015.

[45] Samuel R Bowman, Gabor Angeli, Christopher Potts, and Christopher D Manning. A large annotated corpus for learning natural language inference. *arXiv preprint arXiv:1508.05326*, 2015.

[46] Mitchell P Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a large annotated corpus of English: The Penn Treebank. *Computational linguistics*, 19(2):313–330, 1993.

[47] Tatoeba. `https://tatoeba.org`.

[48] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[49] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[50] Optimizations To Accelerate Deep Learning Training on NVIDIA GPUs. `https://devblogs.nvidia.com/new-optimizations-accelerate-deep-learning-training-gpu/`.

[51] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[52] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

[53] Gregory Cohen, Saeed Afshar, Jonathan Tapson, and André van Schaik. EMNIST: an extension of MNIST to handwritten letters. *arXiv preprint arXiv:1702.05373*, 2017.

[54] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. Reading digits in natural images with unsupervised feature learning. 2011.

[55] Nvidia Profiler. `https://docs.nvidia.com/cuda/profiler-users-guide/index.html`.

[56] I Mario, M Chacon, D Alma, and S Corral. Image Complexity Measure: A Human Criterion Free Approach. In *NAFIPS 2005-2005 Annual Meeting of the North American Fuzzy Information Processing Society*, pages 241–246. IEEE, 2005.

[57] Rinat Mukhometzianov and Juan Carrillo. CapsNet comparative performance evaluation for image classification. *arXiv preprint arXiv:1805.11195*, 2018.

[58] Joe Jeddeloh and Brent Keeth. Hybrid memory cube new DRAM architecture increases density and performance. In *2012 symposium on VLSI technology (VLSIT)*, pages 87–88. IEEE, 2012.

[59] Salessawi Ferede Yitbarek, Tao Yang, Reetuparna Das, and Todd Austin. Exploring specialized near-memory processing for data intensive operations. In *2016 Design,*

Automation & Test in Europe Conference & Exhibition (DATE), pages 1449–1452. IEEE, 2016.

[60] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. PIM-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 336–348. IEEE, 2015.

[61] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. *ACM SIGARCH Computer Architecture News*, 43(3):105–117, 2016.

[62] Jiawen Liu, Hengyu Zhao, Matheus A Ogleari, Dong Li, and Jishen Zhao. Processing-in-memory for energy-efficient neural network training: A heterogeneous approach. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 655–668. IEEE, 2018.

[63] Duckhwan Kim, Jaeha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. Neurocube: A programmable digital neuromorphic architecture with high-density 3D memory. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 380–392. IEEE, 2016.

[64] HMC Specification 2.1. `http://hybridmemorycube.org/files/SiteDownloads/HMC-30G-VSR_HMCC_Specification_Rev2.1_20151105.pdf`.

[65] Dong-Ik Jeon, Kyeong-Bin Park, and Ki-Seok Chung. HMC-MAC: Processing-in Memory Architecture for Multiply-Accumulate Operations with Hybrid Memory

Cube. *IEEE Computer Architecture Letters*, 17(1):5–8, 2018.

[66] Davide De Caro, Nicola Petra, and Antonio GM Strollo. High-performance special function unit for programmable 3-D graphics processors. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 56(9):1968–1978, 2009.

[67] Johannes Partzsch, Sebastian Höppner, Matthias Eberlein, Rene Schüffny, Christian Mayr, David R Lester, and Steve Furber. A fixed point exponential function accelerator for a neuromorphic many-core system. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4. IEEE, 2017.

[68] Chris Lomont. Fast inverse square root. *Tech-315 nical Report*, 32, 2003.

[69] Matthew Robertson. *A brief history of invsqrt.* Department of Computer Science & Applied Statistics, 2012.

[70] Andreas Zoglauer, Steven E Boggs, Michelle Galloway, Mark Amman, Paul N Luke, and R Marc Kippen. Design, implementation, and optimization of MEGAlib's image reconstruction tool Mimrec. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 652(1):568–571, 2011.

[71] Lars Middendorf and Ch Haubelt. A programmable graphics processor based on partial stream rewriting. In *Computer Graphics Forum*, volume 32, pages 325–334. Wiley Online Library, 2013.

[72] William Kahan. IEEE standard 754 for binary floating-point arithmetic. *Lecture Notes on the Status of IEEE*, 754(94720-1776):11, 1996.

[73] Nvidia Tesla P100 Whitepaper. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf.

[74] NVIDIA-smi. https://developer.nvidia.com/nvidia-system-management-interface.

[75] John D Leidel and Yong Chen. Hmc-sim: A simulation framework for hybrid memory cube devices. *Parallel Processing Letters*, 24(04):1442002, 2014.

[76] Gate-Level Simulation Methodology - Cadence. https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/system-design-verification/gate-level-simulation-wp.pdf.

[77] Dongping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L Greathouse, Lifan Xu, and Michael Ignatowski. TOP-PIM: throughput-oriented programmable processing in memory. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 85–98. ACM, 2014.

[78] Jiaqi Zhang, Xiangru Chen, Mingcong Song, and Tao Li. Eager pruning: algorithm and architecture support for fast training of deep neural networks. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 292–303. ACM, 2019.

[79] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia 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, pages 615–629. ACM, 2017.

[80] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 27–40. ACM, 2017.

[81] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, 2017.

[82] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 243–254. IEEE Press, 2016.

[83] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Cnvlutin: ineffectual-neuron-free deep neural network computing. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 1–13. IEEE, 2016.

[84] Minsoo Rhu, Mike O'Connor, Niladrish Chatterjee, Jeff Pool, Youngeun Kwon, and Stephen W Keckler. Compressing DMA Engine: Leveraging Activation Sparsity for Training Deep Neural Networks. In *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*, pages 78–91. IEEE, 2018.

[85] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and¡ 0.5 MB model size. *arXiv preprint arXiv:1602.07360*, 2016.

[86] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. Minerva: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 267–278. IEEE Press, 2016.

[87] Parker Hill, Animesh Jain, Mason Hill, Babak Zamirai, Chang-Hong Hsu, Michael A Laurenzano, Scott Mahlke, Lingjia Tang, and Jason 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*, pages 786–799. ACM, 2017.

[88] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. Scalpel: Customizing DNN pruning to the underlying hardware parallelism. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 548–560. ACM, 2017.

[89] Sharan Narang, Erich Elsen, Gregory Diamos, and Shubho Sengupta. Exploring sparsity in recurrent neural networks. *arXiv preprint arXiv:1704.05119*, 2017.

[90] Sharan Narang, Eric Undersander, and Gregory Diamos. Block-Sparse Recurrent Neural Networks. *arXiv preprint arXiv:1711.02782*, 2017.

[91] Greg Diamos, Shubho Sengupta, Bryan Catanzaro, Mike Chrzanowski, Adam Coates, Erich Elsen, Jesse Engel, Awni Hannun, and Sanjeev Satheesh. Persistent rnns: Stashing recurrent weights on-chip. In *International Conference on Machine Learning*, pages 2024–2033, 2016.

[92] Feiwen Zhu, Jeff Pool, Michael Andersch, Jeremy Appleyard, and Fung Xie. Sparse Persistent RNNs: Squeezing Large Recurrent Networks On-Chip. *arXiv preprint arXiv:1804.10223*, 2018.

[93] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, et al. ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA. In *FPGA*, pages 75–84, 2017.

[94] Chang Gao, Daniel Neil, Enea Ceolini, Shih-Chii Liu, and Tobi Delbruck. DeltaRNN: A Power-efficient Recurrent Neural Network Accelerator. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 21–30. ACM, 2018.

[95] Todd Mytkowicz, Madanlal Musuvathi, and Wolfram Schulte. Data-parallel finite-state machines. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 529–542. ACM, 2014.

[96] Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. Parallelizing dynamic programming through rank convergence. *ACM SIGPLAN Notices*, 49(8):219–232, 2014.

[97] Arun Subramaniyan and Reetuparna Das. Parallel automata processor. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*, pages 600–612. IEEE, 2017.

[98] Moshe Looks, Marcello Herreshoff, DeLesley Hutchins, and Peter Norvig. Deep learning with dynamic computation graphs. *arXiv preprint arXiv:1702.02181*, 2017.

[99] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. In *ACM SIGARCH Computer Architecture News*, volume 44, pages 27–39. IEEE Press, 2016.

[100] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar. ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. *ACM SIGARCH Computer Architecture News*, 44(3):14–26, 2016.

[101] Fan Chen, Linghao Song, and Yiran Chen. ReGAN: A pipelined ReRAM-based accelerator for generative adversarial networks. In *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 178–183. IEEE, 2018.

[102] Haiyu Mao, Mingcong Song, Tao Li, Yuting Dai, and Jiwu Shu. LerGAN: A Zero-Free, Low Data Movement and PIM-Based GAN Architecture. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 669–681. IEEE, 2018.

[103] Biresh Kumar Joardar, Bing Li, Janardhan Rao Doppa, Hai Li, Partha Pratim Pande, and Krishnendu Chakrabarty. REGENT: A Heterogeneous ReRAM/GPU-based Architecture Enabled by NoC for Training CNNs. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 522–527. IEEE, 2019.

[104] Hajar Falahati, Pejman Lotfi-Kamran, Mohammad Sadrosadati, and Hamid Sarbazi-Azad. ORIGAMI: A Heterogeneous Split Architecture for In-Memory Acceleration of Learning. *arXiv preprint arXiv:1812.11473*, 2018.

[105] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. Tetris: Scalable and efficient neural network acceleration with 3d memory. In *ACM SIGARCH Computer Architecture News*, volume 45, pages 751–764. ACM, 2017.

[106] Yongming Shen, Michael Ferdman, and Peter Milder. Maximizing CNN accelerator efficiency through resource partitioning. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 535–547. IEEE, 2017.

[107] Jianxin Guo, Shouyi Yin, Peng Ouyang, Leibo Liu, and Shaojun Wei. Bit-Width Based Resource Partitioning for CNN Acceleration on FPGA. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 31–31. IEEE, 2017.

[108] Siqi Wang, Gayathri Ananthanarayanan, Yifan Zeng, Neeraj Goel, Anuj Pathania, and Tulika Mitra. High-Throughput CNN Inference on Embedded ARM big. LITTLE Multi-Core Processors. *arXiv preprint arXiv:1903.05898*, 2019.

[109] Chi Lo, Yu-Yi Su, Chun-Yi Lee, and Shih-Chieh Chang. A Dynamic Deep Neural Network Design for Efficient Workload Allocation in Edge Computing. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 273–280. IEEE, 2017.

[110] Swarnava Dey, Arijit Mukherjee, Arpan Pal, and P Balamuralidhar. Partitioning of CNN Models for Execution on Fog Devices. In *Proceedings of the 1st ACM International Workshop on Smart Cities and Fog Computing*, pages 19–24. ACM, 2018.

[111] A Cristiano I Malossi, Yves Ineichen, Costas Bekas, and Alessandro Curioni. Fast Exponential Computation on SIMD Architectures. *Proc. of HIPEAC-WAPCO, Amsterdam NL*, 2015.

[112] Federico Perini and Rolf D Reitz. Fast approximations of exponential and logarithm functions combined with efficient storage/retrieval for combustion kinetics calculations. *Combustion and Flame*, 194:37–51, 2018.