VLSI Architectures for Long Short Term Memory Networks

Krishna Praveen Yalamarthy Saurabh Dhall
(Roll No. 150102077) (Roll No. 150102060)

Bachelor of Technology

Under the guidance of
Prof. Rafi Ahmed

DEPARTMENT OF ELECTRONICS & ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY GUWAHATI
November 2018

Acknowledgments

Sample Acknowledgement

Abstract

Long Short Term Memory Networks, which are a modification of the traditional Recurrent Neural Networks, have gained widespread integration into commercial applications to accomplish tasks in the domains of speech recognition, handwriting recognition and are currently deployed in many embedded devices. These networks are computationally expensive and require considerable memory and storage bandwidth which restricts their further use in resource constrained devices. There has been considerable work on optimizing these networks at an algorithmic level, yet there is little to no work being done to reduce the computational complexity on an architectural level using VLSI-DSP techniques. Intially, state-of-art model compression techniques such as Pruning, Quantization, Knowledge distillation are exlpored to decrease the size of the model itself. The next major component of the thesis concentrates on using Distributed Arithmetic to trade computational complexity with ROM lookups.

Contents

Abstract ii

List of Figures iv

1 Introduction 1

1.1 VLSI Design Flow . . . 1
1.2 Machine Learning . . . 2
1.3 Literature Review . . . 2
1.4 Problem Formulation . . . 2

2 Recurrent Neural Networks 4

2.1 RNN Structure . . . 4
2.2 Long Short Term Memory Networks . . . 5
2.2.1 Multi-Layer LSTM . . . 7
2.2.2 Google LSTM . . . 7
2.3 Profiling an LSTM . . . 7
2.4 TIMIT Dataset and Baseline Model . . . 8

3 Network Compression 9

3.1 Low Displacement Rank Networks . . . 9
3.2 Pruning and Quantization . . . 10
3.2.1 Pruning . . . 10
3.2.2 Quantization . . . 10

4 Distributed Arithmetic 11

4.1 Conventional Distributed Arithmetic . . . 11
4.2 ROM Decomposition . . . 13
4.3 Offset Binary Coding . . . 14
4.4 Applying Distributed Arithmetic to LSTM . . . 16

5 Proposed Algorithm and Architecture 17

5.1 Profiling an LSTM . . . 17
5.2 Matrix-Vector Multiplication . . . 18
5.2.1 Distributed Arithmetic . . . 18

5.2.2 Further Ways to Reduce the Memory 18

6 Future Work **19**

List of Figures

1.1 VLSI Design Flow 1

2.1 Single Layer RNN 4
2.2 Unfolded RNN Cell 5
2.3 Graphical representation of an LSTM cell 6
2.4 Multi-Layer RNN 7
2.5 Profile of LSTM Network 8

5.1 Profile of LSTM Network 17

Chapter 1

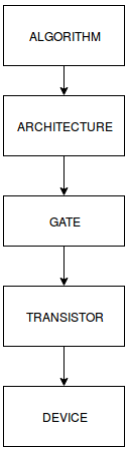
Introduction

1.1 VLSI Design Flow

The development cycle of any digital IC is governed by a few fundmanetal steps or processes that are sequentially followed. The first step begins with designing an algorithm to solve the problem and meet the requirements. Coming up with an algorithm is followed by designing an architecture which can be realized on hardware that is feasible in the desired resource-constrained use case. Different architectures can be proposed for the same algorithm which can be further optimized at the gate level. As shown in the figure 1.1, in order to fully exploit the benefit of having an ASIC design, after the gate level modelling, transistor level optimizations are also important. The last stage of the design cycle ends with modifications at the device level.

A small modification at an earlier level in the design flow can lead to a significant change at a later level. A single multiplier that could be eliminated at the architectural level would lead to saving thousands of transistors indicating the significance of higher level optimizations. Since the effects of an optimization are reflected in the following stages, it is essential to fully explore the various possible models or designs.

Figure 1.1: VLSI Design Flow



1.2 Machine Learning

The field of machine learning is not one that is new. These algorithms have been in existence for a long time although only in recent years they have come into popularity, especially neural networks and deep learning, thanks to the rapid increase in computational power. The GPU has aided in increasing the scale of these models to actually work effectively on real-world tasks. While major strides have been made in improving these algorithms by extensive training and refinement at an algorithmic level, research into architectural changes has been limited.
Due to recent advancements in compute power in embedded/mobile device, there is now a possibility of implementing these computationally expensive models on devices with a tighter power, memory and compute constraint than was previously there on non-mobile devices.
Indicative of the growing interest in embedded implementations of such networks, in the last couple of years there have been some works on FPGA deployment of LSTMs but none of them seem to leveraging the VLSI digital signal processing techniques for optimization.VLSI DSP techniques such as systolic arrays, unfolding, distributed arithmetic, CORDIC, pipelining etc., have been in use for the past 30 years for the fields of communication, filters, image processing lend themselves really well for machine learning algorithms as well. Recent work inspired by these techniques optimize Support Vector Machines and Convolutional Neural Networks.One branch of neural networks that is explicitly suited for tasks that are applicable in mobile devices are Recurrent Neural Networks. These networks are suited for data that is sequential in nature and have been shown to be effective at tasks such as handwriting recognition and generation, speech and speaker recognition, language modelling, speech synthesis to name a few.

1.3 Literature Review

A specialised version of the RNN that is extremely popular for commercial application is the Long Short Term Memory network (LSTM). It is adept at establishing long term dependencies in data . As of 2016, many software giants such as Google, Apple and Microsoft had LSTMs as a fundamental unit in their products. Google incorporated LSTMs for the speech recognition task on the handheld device and also into their smart assistant as well as Google Translate.

The typical computational power required by LSTMs is demanding. The challenge of deploying these models is further increased due to the size of the models themselves which needs to be stored locally to fully execute these tasks.

-original paper -lstm odyssey -compression papers -no proper application of VLSI DSP in this field. can lead to significant improvement to help deploy on devices.

1.4 Problem Formulation

Our goal is to leverage both the best of the available algorithmic optimizations or a combination of those followed by VLSI DSP techniques that are suited to exploit some unique properties of LSTM. We will first go through the preliminaries that establish the algorithm of Long Short Term Memory Networks. We will then systematically go

through the different levels of optimization, from algorithmic to an architectural level, that can be applied to these networks.

Chapter 2

Recurrent Neural Networks

Any supervised deep learning algorithm consists of primarily two phases i.e., training and inference. Training is where the network is operated on a dataset consisting of input features, like a speech segment, and the associated correct output, as in this case the word it represents. Once the model has been trained it has the capability to predict the output given a completely new input to an extent of accuracy that is determined the training scheme, the model, the dataset and various other parameters. We now elaborate on the model of a recurrent neural network, illustrate its operation and training mechanism and then explain in detail the model of a Long Short Term Memory Network (LSTM).

2.1 RNN Structure

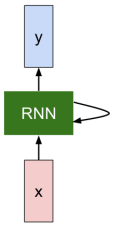
The Recurrent Neural Network consists of a hidden state that is updated on every new input to the network. The updating of the hidden state of the network is influenced by both the previous state of the network and the new input. The output of the network depends on the hidden state of the network. For a single hidden layer the network can be formulated as follows. Let the hidden state vector be represented by $\mathbf{h}_t \in \mathbb{R}^m$ where m is the dimensionality of the hidden state vector at time step t. $\mathbf{x}_t \in \mathbb{R}^n$ is the input vector and the output vector is $\mathbf{y}_t \in \mathbb{R}^k$ at time step t.

$$\mathbf{h}_t = f_W(\mathbf{h}_{t-1}, \mathbf{x}_t) \tag{2.1}$$

$$\mathbf{h}_t = tanh(\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{xh}\mathbf{x}_t) \tag{2.2}$$

$$\mathbf{y}_t = \mathbf{h}_{hy}\mathbf{h}_t \tag{2.3}$$

Figure 2.1: Single Layer RNN



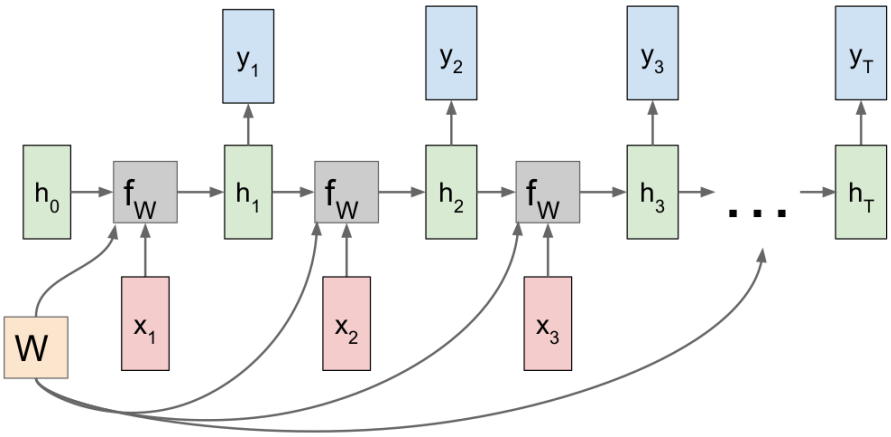
The above equations are applied on every new input. The matrices $\mathbf{W}$ represent the connections between the

input vector $\mathbf{x}$ and the hidden vector $\mathbf{h}$ as well as the connections between the hidden state and the output vector $\mathbf{y}$. The non-linear function $tanh$ compresses each scalar in the hidden vector to a value between -1 and 1. This non-linearity is crucial to the performance of the network.

These connection matrices, more commonly known as weight matrices, are what determine the networks behaviour. The choice of the dimensions of h and the number of hidden layers used determines the capacity of a network to effectively work on the task it is designed for.

A good way to visualize the functioning of the network is to observe its behaviour across multiple time steps and and observe the flow of data between the output, the input and within the cell itself. In 2.2, every vertical column of cells represents the same network but at different time instansts.

Figure 2.2: Unfolded RNN Cell



A salient point to be noted in recurrent neural networks is that the weight matrices are constant throughout every iteration of the algorithm. This causes some restrictions on what the network is capable of learning:

1. The network is poor at learning dependencies between inputs that are separated too far apart as the state of the network tends to slowly forget its past inputs.
2. During training, the repetition of the matrix causes some terms in the calculation of the training algorithm to explode to infinity or to zero.

Due to these limitations, a modification of the network known as Long Short Term Memory Networks has been proposed which we will elaborate below.

2.2 Long Short Term Memory Networks

LSTMs were initially introduced by Hochreiter Schmidhuber in 1997 and through various works was refined into the algorithm used today. An LSTM is a modification of an RNN with additional gates added to it. It consists of three gates that control the flow of information between the input, the hidden state and the output. [ref colahs blog and space] Let there be:

1. Input Connection Weights: $\mathbf{W}_z, \mathbf{W}_s, \mathbf{W}_f, \mathbf{W}_o \in \mathrm{R}^{NXM}$
2. Hidden State Weights: $\mathbf{R}_z, \mathbf{R}_s, \mathbf{R}_f, \mathbf{R}_o \in R^{NXN}$
3. Bias Weights: $\mathbf{b}_z, \mathbf{b}_s, \mathbf{b}_f, \mathbf{b}_o \in R^N$

The governing equations of the LSTM update in a time step are as follows:

$$\mathbf{z}^t = \mathbf{W}_z \mathbf{x}^t + \mathbf{R}_z \mathbf{y}^{t-1} + \mathbf{b}_z \tag{2.4}$$

$$\mathbf{i}^t = \mathbf{W}_i \mathbf{x}^t + \mathbf{R}_i \mathbf{y}^{t-1} + \mathbf{b}_i \tag{2.5}$$

$$\mathbf{i}^t = sigmoid(\mathbf{i}^t) \tag{2.6}$$

$$\mathbf{f}^t = \mathbf{W}_f \mathbf{x}^t + \mathbf{R}_f \mathbf{y}^{t-1} + \mathbf{b}_f \tag{2.7}$$

$$\mathbf{f}^t = sigmoid(\mathbf{f}^t) \tag{2.8}$$

$$\mathbf{c}^t = \mathbf{z}^t \odot \mathbf{i}^t + \mathbf{c}^{t-1} \odot \mathbf{f}^t \tag{2.9}$$

$$\mathbf{o}^t = \mathbf{W}_o \mathbf{x}^t + \mathbf{R}_o \mathbf{y}^{t-1} + \mathbf{b}_o \tag{2.10}$$

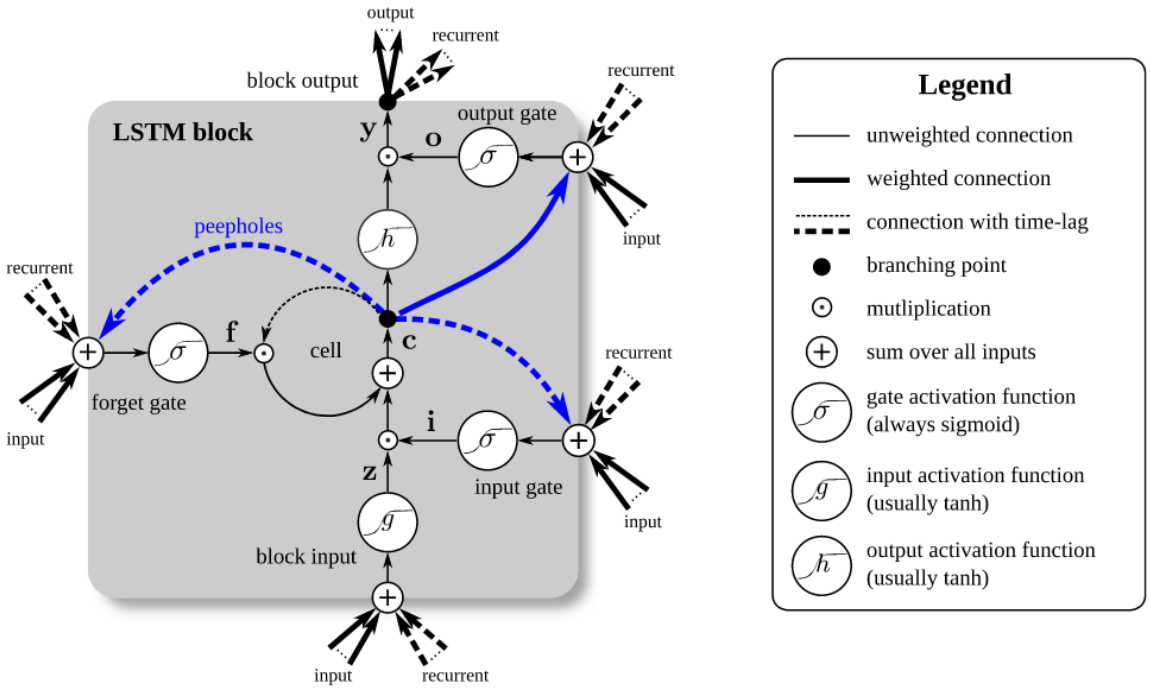
$$\mathbf{o}^t = sigmoid(\mathbf{o}^t) \tag{2.11}$$

$$\mathbf{y}^t = tanh(\mathbf{c}^t) \odot \mathbf{o}^t \tag{2.12}$$

$\odot$ represents point-wise multiplication and sigmoid is a logistic function with the following equation

$$sigmoid(\mathbf{x}) = \frac{1}{1 + exp(-\mathbf{x})}$$

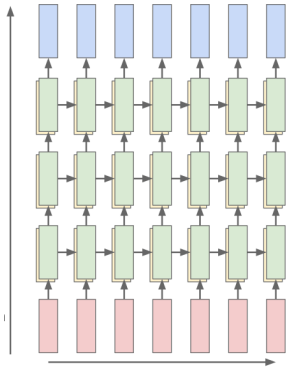
Figure 2.3: Graphical representation of an LSTM cell



Each gate serves a purpose: the forget gate controls how much of the previous state is to be retained in the next update, the input gate dictates the influence the new input will have on the state and the output gate controls how much of the output is influence by the hidden state. Due to the addition of these gates, the network has the ability to $remember$ long term dependencies from the input and can solve complex tasks on sequential data.

2.2.1 Multi-Layer LSTM

Figure 2.4: Multi-Layer RNN



In most state-of-art implementations a single hidden layer isn't used i.e., there isn't just one cell between the input and the output but multiple. This leads to a deep LSTM or a multi-layer LSTM.

2.2.2 Google LSTM

Chapter 3

Network Compression

The size of deep neural networks has been a major hindrance to their implementation on resource constrained hardware devices. The weight matrix sizes consume significant storage and memory bandwidth. Changes on an algorithmic level have been proposed extensively in the last couple of years with regards to reducing the model sizes of deep neural networks. Reducing the number of parameters, not only allows for faster and less distributed training, but for more power-aware inference which is the aim of our work.
Compression of a network, as discussed above, is the reduction of the number of parameters in the model/network. The method of reduction of parameters influences the training method in a unique way ie, the training phase is aware of the compression and must account for the model changes. From recent literature, two broad paradigms have emerged, namely:

1. Low Displacement Rank Networks
2. Pruning and Quantization
3. Knowledge Diffusion

3.1 Low Displacement Rank Networks

This method of compression takes advantage of the structured nature of certain types of matrices to reduce the number of unique parameters from On2 to On. This not only reduces the storage complexity from On2 to On, it also reduces, due to fast matrix multiplication methods, the computational complexity roughly from On2 to Onlogn or Onlogn2. It uses the fact that any matrix can be compressed to a Low Displacement Rank matrix M which after being computed using the appropriate operation, is decompressed back to the desired result. For deep neural networks, an error bound has been proven for such approximations on the performance of a neural network. Examples of structured matrices are: Circulant, Toeplitz, Henkel, Vandermonde, Cauchy. The predictable structure of these matrices reduces the number of parameters and also eases the indexing process due to its predictability, a lack of which affects the first two methods hence increasing the potential optimization that is possible beforehand when designing an ASIC/FPGA implementation. The other added advantage with respect to the pruning method elaborated below is that there is no iterative training process after introducing the compression as the entire training mechanism is aware of the exact structure of the matrix from the beginning. A recent work uses Circulant matrices as suggested to train the network and achieve a significant speedup on their FPGA implementation of LSTMs.

3.2 Pruning and Quantization

3.2.1 Pruning

In a trained neural network, the weight matrix consists of parameter values that vary drastically in magnitude. Similar to the connections amongst neurons, according to the function that the network is trying to imitate, only certain connections are of computational importance. Hence those connections that have a small magnitude can be removed to increase the sparsity of the network. The state-of-art implementation of such a deep neural network pruning scheme is Deep Compression. We will illustrate the technique suggested and adopt the same to reduce the network size and also explore the different avenues of optimization this creates in the Distributed Arithmetic section.
Procedure in training phase: (flow chart insert if possible):

1. Train the network at its full capacity.
2. Prune small magnitude connections by setting a threshold.
3. Retrain the remaining connections for fine-tuning.

3.2.2 Quantization

Further reduction of the model size is possible by quantizing the remaining parameters to K levels. The standard procedure to do this is through K-means clustering. Therefore only logK bits will be required to store the parameters. Literature shows that the two practices of pruning and quantization can be combined together to provide significant drop in the model size. Although the practice of quantization is helpful in reducing the storage bandwidth, it is yet to be used as an way to reduce the implementation of the network.

Chapter 4

Distributed Arithmetic

Distributed Arithmetic is used to design bit-level(bit-serial) architectures for vector-vector multiplications. Distributed Arithmetic replaces the multiplication operations in vector-vector multiplications with ROM lookups. It reorders and mixes the multiplications such that the arithmetic becomes 'distributed' throughout the structure.

The most computationally expensive operation in a LSTM cell is MatMul(matrix-vector multiplication) as observed from the profiling of the algorithm. Matrix-vector multiplication can be converted to vector-vector multiplication by representing the matrix as a column vector of rows or vice versa.

4.1 Conventional Distributed Arithmetic

Consider a simple inner product between two vectors C and X of length N.

$$Y = C.X = \sum_{i=0}^{N-1} c_i x_i \tag{4.1}$$

where c_i's are M-bit constants and x_i's are coded as W-bit 2's complement numbers. The most significant bit in the 2's complement format represents the sign of the number while the rest of the bits have weights in power of two. The value of W-bit x_i is given by :

$$x_i = -x_{i,W-1} + \sum_{j=1}^{W-1} x_{i,W-1-j} 2^{-j} \tag{4.2}$$

Substituting value of x_i in (4.1)

$$\begin{aligned} Y &= \sum_{i=0}^{N-1} c_i(-x_{i,W-1} + \sum_{j=1}^{W-1} x_{i,W-1-j} 2^{-j}) \\ &= -\sum_{i=0}^{N-1} c_i x_{i,W-1} + \sum_{j=1}^{W-1} (\sum_{i=0}^{N-1} c_i x_{i,W-1-j}) 2^{-j} \end{aligned} \tag{4.3}$$

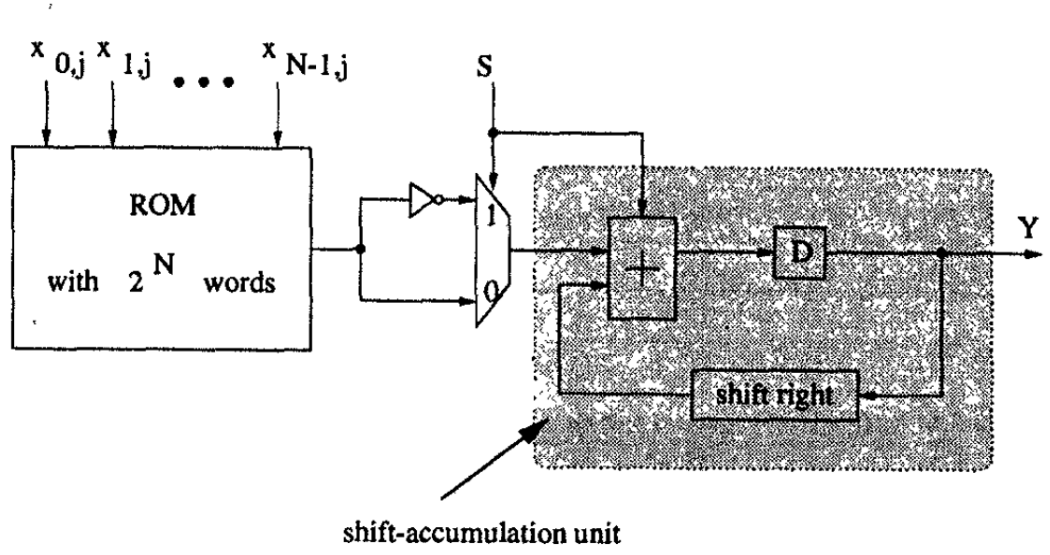
Let C_j's be defined as

$$C_{W-1-j} = \sum_{i=0}^{N-1} c_i x_{i,W-1-j} \quad (j \neq 0),\ C_{W-1} = -\sum_{i=0}^{N-1} c_i x_{i,W-1} \tag{4.4}$$

$$Y = \sum_{j=0}^{W-1} C_{W-1-j} 2^{-j} \tag{4.5}$$

We have redistributed the arithmetic by interchanging the summing order of i and j in (4.3)

The terms C_j(2^N possible values) can be precomputed and stored in a read-only memory(ROM) since C_j's depend on $x_{i,j}$'s. Note that this implies that a ROM of size 2^N has to be used. An input address of N bits $(x_{0,j}, x_{1,j}, x_{2,j}..., x_{N-1,j})$ is used to retrieve the precomputed C_j value. By addressing the ROM and accumulating the outputs, the inner product can be calculated in W clock cycles. So we have reduced the computations from $O(N)$ multiplications to $O(N-1)$ additions and right shift operations. There exists a trade off between computational complexity and memory here which will be exploited again in ROM Decomposition. Figure shows

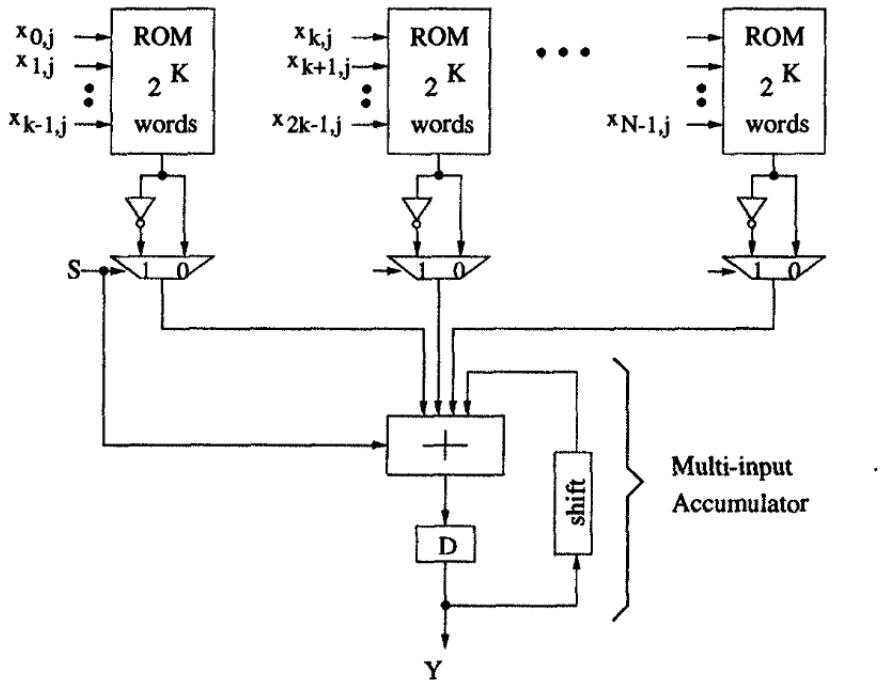


shift-accumulation unit

a typical architecture for computing inner product of two length-N vectors using Distributed Arithmetic. The shift-accumulator is a bit-parallel carry-propagate adder that adds the ROM output to the previous adder output. The control signal S is 1 when $j = W-1$ and 0 otherwise to invert the ROM output in order to calculate C_{W-1}. This inversion is achieved with the help of the MUX and the inverter. The result is accumulated in the adder after W clock cycles($j = 0$ to $j = W-1$).

$x_{0,j}$	$x_{1,j}$	$x_{2,j}$	$x_{3,j}$	ROM
0	0	0	0	0
0	0	0	1	c_3
0	0	1	0	c_2
0	0	1	1	$c_2 + c_3$
0	1	0	0	c_1
0	1	0	1	$c_1 + c_3$
0	1	1	0	$c_1 + c_2$
0	1	1	1	$c_1 + c_2 + c_3$
1	0	0	0	c_0
1	0	0	1	$c_0 + c_3$
1	0	1	0	$c_0 + c_2$
1	0	1	1	$c_0 + c_2 + c_3$
1	1	0	0	$c_0 + c_1$
1	1	0	1	$c_0 + c_1 + c_3$
1	1	1	0	$c_0 + c_1 + c_2$
1	1	1	1	$c_0 + c_1 + c_2 + c_3$

4.2 ROM Decomposition



The ROM size of distributed arithmetic increases exponentially with the vector size $N(2^N)$ which along with the access time can bottleneck the speed of the whole system. The ROM size and access time can be improved by trading off the size with computational complexity of the accumulator. The ROM size of the conventional DA and OBC based DA increases exponentially with the vector size N (2^N in conventional DA and 2^{N-1} in OBC based

DA). When the ROM size is large, the ROM access time can limit the speed of the system. Hence, decreasing the ROM size is a very significant issue.

ROM decomposition provides a way to decrease the ROM size by trading off the ROM size with computational complexity. This can be achieved by dividing the N address bits of the ROM into N/K groups of K bits effectively dividing or decomposing the ROM of size 2^N into N/K ROMs of size 2^K. This reduction in ROM size is at the cost of a multi-input accumulator to add the N/K ROM outputs. The total ROM size is now $(N/K)2^K$ which is a linear function of N as opposed to an exponential increase.

4.3 Offset Binary Coding

Offset Binary Coding(OBC) based DA can reduce the ROM size from 2^N to $2^N - 1$. We write x_i as

$$\begin{aligned} x_i &= \frac{1}{2}[x_i - (-x_i)] \\ &= \frac{1}{2}[-(x_{i,W-1} - \overline{x_{i,W-1}}) + \sum_{j=1}^{W-1}(x_{i,W-1-j} - \overline{x_{i,W-1-j}})2^{-j} - 2^{-(W-1)}] \end{aligned} \tag{4.6}$$

where $-x_i$ is

$$-x_i = -\overline{x_{i,W-1}} + \sum_{j=1}^{W-1} \overline{x_{i,W-1-j}}2^{-j} + 2^{-(W-1)} \tag{4.7}$$

Define $d_{i,j}$ as

$$d_{i,j} = \begin{cases} x_{i,j} - \overline{x_{i,j}}, \ for\ j \neq W-1 \\ -(x_{i,W-1} - \overline{x_{i,W-1}}), \ for\ j = W-1 \end{cases} \tag{4.8}$$

Rewrite (4.6) using (4.8)

$$x_i = \frac{1}{2}[\sum_{j=0}^{W-1} d_{i,W-1-j}2^{-j} - 2^{-(W-1)}] \tag{4.9}$$

From (4.9) and (4.1)

$$\begin{aligned} Y &= \sum_{i=0}^{N-1} c_i[\frac{1}{2}(\sum_{j=0}^{W-1} d_{i,W-1-j}2^{-j} - 2^{-(W-1)})] \\ &= \sum_{j=0}^{W-1}(\sum_{i=0}^{N-1}\frac{1}{2}c_i d_{i,W-1-j})2^{-j} - (\sum_{i=0}^{N-1}\frac{1}{2}c_i)2^{-(W-1)} \end{aligned} \tag{4.10}$$

Define D_j

$$D_j = \sum_{i=0}^{N-1}\frac{1}{2}c_i d_{i,W-1-j}, \ for\ j = 0\ to\ W-1 \tag{4.11}$$

and D_{extra}

$$D_{extra} = \sum_{i=0}^{N-1}\frac{1}{2}c_i \tag{4.12}$$

Substituting D_j's and D_{extra} into (4.10)

$$Y = \sum_{j=0}^{W-1} D_{W-1-j}2^{-j} + D_{extra}2^{-(W-1)} \tag{4.13}$$

$x_{0,j}$	$x_{1,j}$	$x_{2,j}$	$x_{3,j}$	ROM
0	0	0	0	$-(c_0+c_1+c_2+c_3)/2$
0	0	0	1	$-(c_0+c_1+c_2-c_3)/2$
0	0	1	0	$-(c_0+c_1-c_2+c_3)/2$
0	0	1	1	$-(c_0+c_1-c_2-c_3)/2$
0	1	0	0	$-(c_0-c_1+c_2+c_3)/2$
0	1	0	1	$-(c_0-c_1+c_2-c_3)/2$
0	1	1	0	$-(c_0-c_1-c_2+c_3)/2$
0	1	1	1	$-(c_0-c_1-c_2-c_3)/2$
1	0	0	0	$(c_0-c_1-c_2-c_3)/2$
1	0	0	1	$(c_0-c_1-c_2+c_3)/2$
1	0	1	0	$(c_0-c_1+c_2-c_3)/2$
1	0	1	1	$(c_0-c_1+c_2+c_3)/2$
1	1	0	0	$(c_0+c_1-c_2-c_3)/2$
1	1	0	1	$(c_0+c_1-c_2+c_3)/2$
1	1	1	0	$(c_0+c_1+c_2-c_3)/2$
1	1	1	1	$(c_0+c_1+c_2+c_3)/2$

The contents of the ROM have inverse symmetry as observed from the lookup table. This symmetry can be exploited to reduce the number of possible combinations by a factor of 2 reducing the ROM size by half.

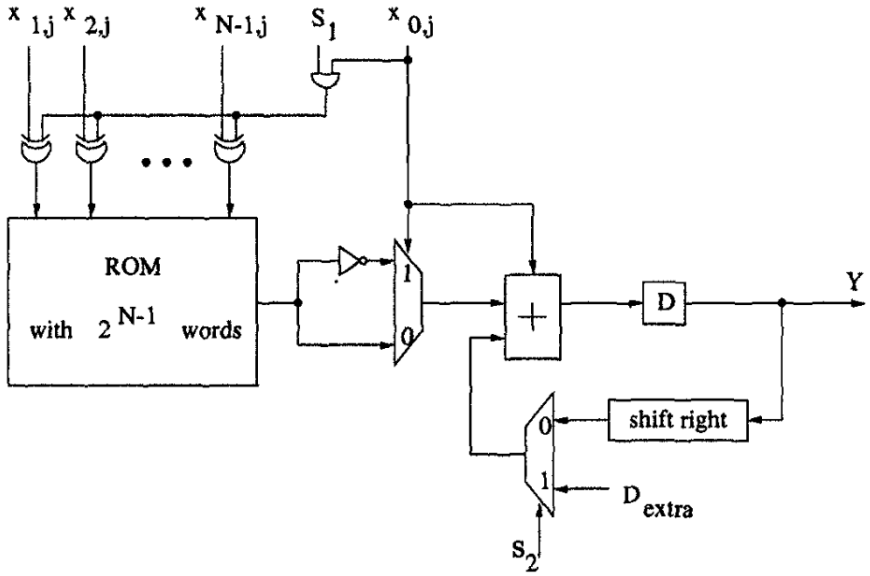


Figure shows a typical architecture for computing inner product of two length-N vectors using OBC based DA. The architecture is mostly the same as conventional DA.

Control signal S_2 is 1 when $j = 0$ and 0 otherwise providing the initial value D_{extra} to the shift-accumulator. The XOR gates are used to select the correct address. The MUX after the ROM inverts the ROM output depending on the sign bit.

4.4 Applying Distributed Arithmetic to LSTM

Refer to LSTM cell equations A simplified version can be written as $f(Wx+b)$ where f is a non-linear activation function. The $Wx+b$ part can be optimized using DA. Write W as

$$W = [w_0, w_1, w_2, w_3..., w_N - 1] \tag{4.14}$$

where w_i's are column vectors of W and x is a column vector. So Wx can be written as

$$Wx = [w_0.x, w_1.x, w_2.x, w_3.x..., w_N - 1.x]^\top \tag{4.15}$$

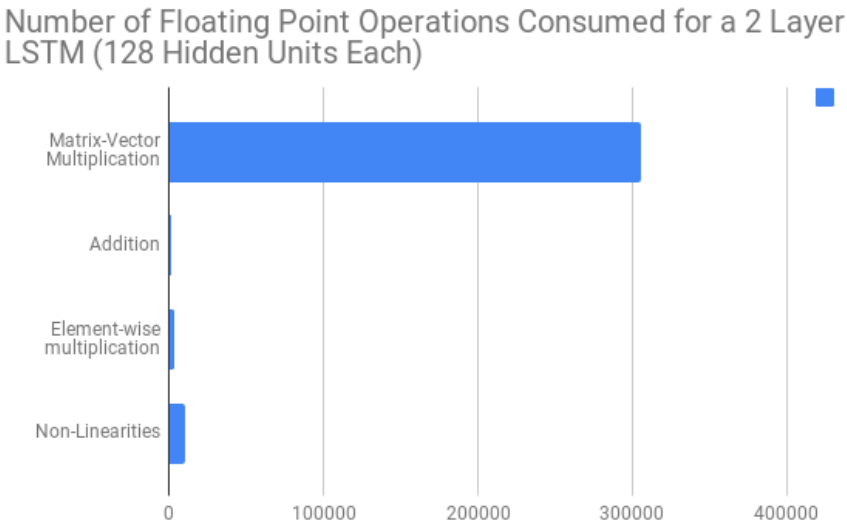
Chapter 5

Proposed Algorithm and Architecture

5.1 Profiling an LSTM

Since the primary objective of our work is to develop an implementation of an LSTM on hardware with minimal power, compute and memory requirement, we will need to first identify what operations in the network's structure occupies the most compute power. The following methodology was adopted to profile the LSTM.
A 2 Layer LSTM with 128 hidden units each was implemented in $Tensorflow$, a computation graph library for deep neural networks and the $Profiler$ tool of the library was used to find the number of floating-point operations(FLOPs) each broad category of operation required. The results are represented graphically below.

Figure 5.1: Profile of LSTM Network



The following conclusions can be drawn from the above graph:

- The most computationally expensive operation is the **matrix-vector multiplication**. Of the matrix and the vector, the vector or the input changes every time instant but the matrix is constant for a particular network regardless of the input. This needs to leveraged when optimizing the network
- The next most computationally expensive operation are the **non-linearities** $sigmoid$ and $tanh$.

- **Element-wise multiplication** and **addition** operations are insignificant in comparison with the other operations.

5.2 Matrix-Vector Multiplication

constant matrix and matmul is the most expensive operation. leads to DA being a possible option

As mentioned above, the matrix-vector multiplication operation is the one that needs to be tackled first. Since the weight matrices are constant for a particular network, distributed arithmetic can be seen as a possible solution to implement the multiplication operation. DA removes the need for a multiplier in the circuit. Therefore, below, DA and its various optimizations are elaborated and applied to task of matrix-vector multiplication. The memory requirement is then analysed and further avenues to reduce this are proposed.

5.2.1 Distributed Arithmetic

Offset Binary Coding

ROM Decomposition

Applying DA to LSTMs

tell the current memory requirement

5.2.2 Network Compression

Pruning

Quantization

Low Displacement Rank Methods

Knowledge Distillation

Logic Based without DA

-Pruning creates 90% sparsity. Is it possible to exploit??? cite the sparse DA paper. -Quantization - need to store less, but indexing is more problematic -Knowledge Distillation has shown to reduce model size 5x(read the paper) -No memory, direct logic gates

Chapter 6

Future Work

A majority of the work that has been done is to come up to speed with the current paradigms in compression of a network and also familiarising with the Digital Signal Processing techniques for optimizing VLSI implementations. The future work can be broadly classified into three phases:

1. **Finalising the architecture of the Network:** The first step is to finalize on the compressed model by experimenting with the various techniques described in Network Compression chapter. This involves trying pruning alone, pruning with quantization and also experimenting with knowledge distillation and sparse matrix decompositions of the weight matrices.

2. **Memory management and DA architecture:** In tandem with step 1, the final optimized structure of DA including ROM decomposition and OBC has to be constructed. Also, the different levels of memory i.e., deciding the on-board SRAM memory size and off-chip DRAM memory size and arrangement have to be decided.

3. **Non-linear activations:** The non-linear functions $sigmoid$ and $tanh$ introduces additional latency in the model. Linearizing these functions piecewise can lead to a speedup of the network.

4. **Pipelining:** This involves two distinct approaches. The first is to pipeline the tasks inside a single LSTM cell. The other avenue of pipelining is to treat the LSTM cell as a black box and pipeline the different the layers of the network.

Bibliography

[1] "Spectrum Policy Task Force Report," *Federal Commun. Commission, Tech. Rep.*, Nov. 2002.
Available at http://hraunfoss.fcc.gov/edocs_publish/attachmatch/DOC-228542A1.pdf

[2] S. Haykin, "Cognitive Radio: Brain Empowered Wireless Communications," *IEEE Journal on Selected Areas in Communication*, Vol. 23, No. 2, pp. 201-220, Feb. 2005.