VLSI Architectures for Long Short Term Memory Networks

Krishna Praveen Yalamarthy
(Roll No. 150102077)

Saurabh Dhall
(Roll No. 150102060)

Bachelor's Thesis Project: Phase 1

Under the guidance of
Prof. Rafi Ahmed

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

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. These networks are computationally expensive and require considerable memory and storage requirements which restricts their further deployment 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 at an architectural level using VLSI-DSP techniques. This thesis aims to develop a design of an LSTM on hardware that leverages the recent algorithmic advancements along with the full stack of optimizations that can be made in a traditional VLSI design flow to achieve an efficient implementation of LSTMs.

Contents

List of Figures

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 **??**, 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.

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 advent of the GPU has enabled increasing the size 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 devices, there is now a possibility of implementing these computationally expensive models on devices with a tighter power, memory and compute constraint that was previously there only 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.[2] 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

[3] and Convolutional Neural Networks [4]. One branch of neural networks that is explicitly suited for tasks that are applicable in mobile devices are Recurrent Neural Networks. This class of 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.[5]

1.3 Literature Review and Problem Formulation

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.[6]

LSTMs were initially introduced in 1997 [7] and have since gone through several modifications more details of which can be found in [5]. The typical computational power required by LSTMs is demanding. The challenge of deploying these models is further increased due to the tasks these networks are designed to target. They are adept at understanding sequential data and hence good hardware implementations must support the latency constraint and throughput required.

At an algorithmic level, there are many ways to reduce the size of the model and hence its complexity.[8] [9] [10] [11] These techniques are discussed in detail in section 3.3. With respect to architectural optimizations there has been some work done to optimize these networks for FPGA implementations [2] [12], yet architectural enhancements to take advantage of the algorithmic level modifications using VLSI-DSP techniques has been almost non-existent.

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 by 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 [1]

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.

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

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

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

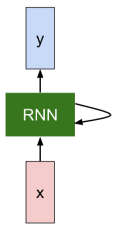


Figure 2.1: Single Layer RNN

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 2.1, every vertical column of cells represents the same network but at different time instansts.

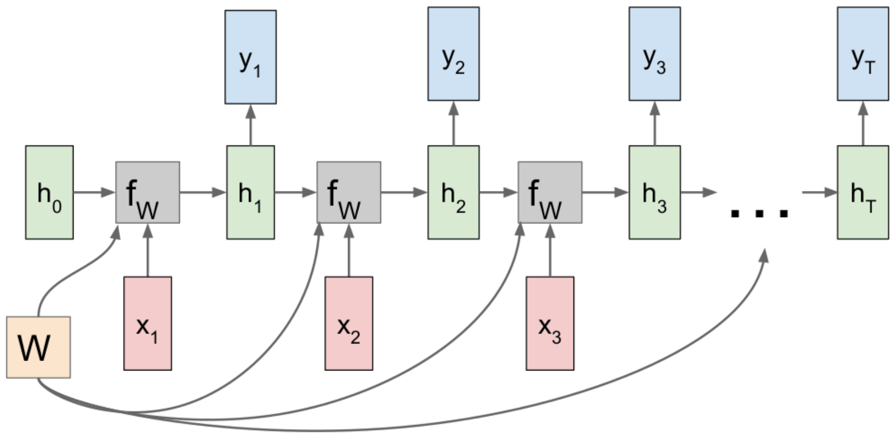


Figure 2.2: Unfolded RNN Cell

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 and Schmidhuber in 1997 [7] 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. [13]
Let there be:

1. Input Connection Weights: $\mathbf{W}_z, \mathbf{W}_s, \mathbf{W}_f, \mathbf{W}_o \in R^{NXM}$
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

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

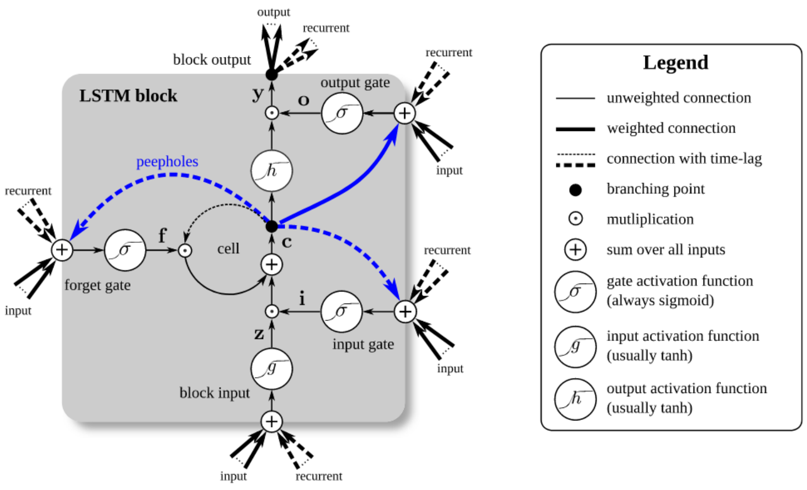


Figure 2.3: Graphical representation of an LSTM cell

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. [5]

2.2.1 Multi-Layer LSTM

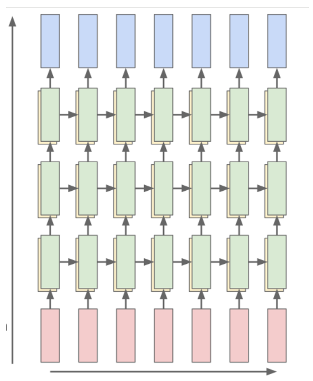


Figure 2.4: Multi-Layer RNN

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

Chapter 3

Proposed Algorithm and Architecture

3.1 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.

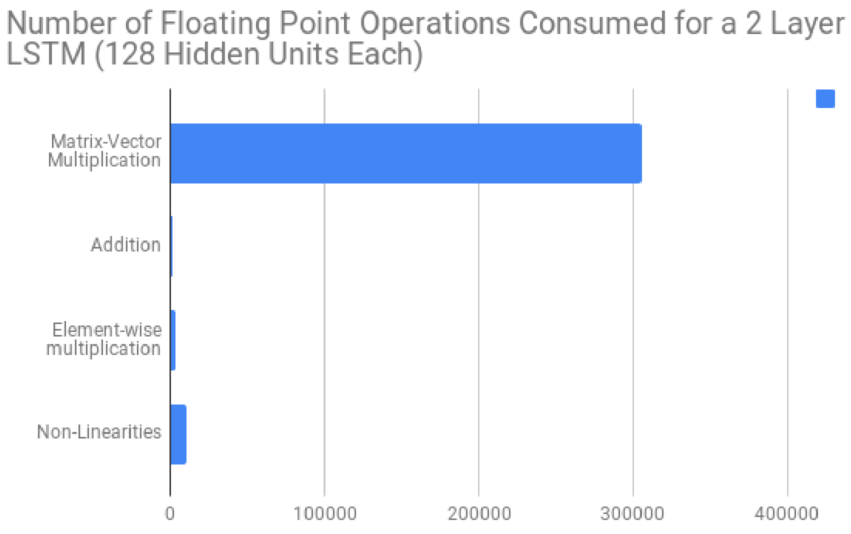


Figure 3.1: Profile of LSTM Network

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.

3.2 Matrix-Vector Multiplication

The most computationally expensive operation in a LSTM cell is MatMul(matrix-vector multiplication) as observed from the profiling of the algorithm so it is the operation that needs to be tackled first. Since the weight matrices are constant for a particular network, Distributed Arithmetic(DA) can be seen as a possible solution to implement the multiplication operation. Matrix-vector multiplication can be converted to vector-vector multiplication by representing the matrix as a column vector of rows or vice-versa.

Distributed Arithmetic is used to design bit-level(bit-serial) architectures for vector-vector multiplications. [14] It replaces the multiplication operations in vector-vector multiplications with ROM look-ups. It reorders and mixes the multiplications such that the arithmetic becomes 'distributed' throughout the structure. Therefore, below, DA and its various optimizations are elaborated and applied to task of matrix-vector multiplication. The memory requirement is then analyzed and further avenues to reduce this are proposed.

3.2.1 Conventional Distributed Arithmetic

Consider a simple inner product between two vectors $\mathbf{C}$ and $\mathbf{X}$ of length N.

$$Y = \mathbf{C}.\mathbf{X} = \sum_{i=0}^{N-1} c_i x_i \tag{3.1}$$

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 :

$$x_i = -x_{i,W-1} + \sum_{j=1}^{W-1} x_{i,W-1-j} 2^{-j} \tag{3.2}$$

Substituting value of x_i in (3.1)

$$\begin{aligned} Y &= \sum_{i=0}^{N-1} c_i(-x_{i,W-1} + \sum_{j=1}^{W-1} x_{i,W-1-j} 2^{-j}) \\ &= -\sum_{i=0}^{N-1} c_i x_{i,W-1} + \sum_{j=1}^{W-1} (\sum_{i=0}^{N-1} c_i x_{i,W-1-j}) 2^{-j} \end{aligned} \tag{3.3}$$

Let C_j's be defined as

$$C_{W-1-j} = \sum_{i=0}^{N-1} c_i x_{i,W-1-j} \quad (j \neq 0),\ C_{W-1} = -\sum_{i=0}^{N-1} c_i x_{i,W-1} \tag{3.4}$$

$$Y = \sum_{j=0}^{W-1} C_{W-1-j} 2^{-j} \tag{3.5}$$

We have redistributed the arithmetic by interchanging the summing order of i and j in (3.3)

Table 3.1: ROM contents for vector length $N = 4$

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

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.

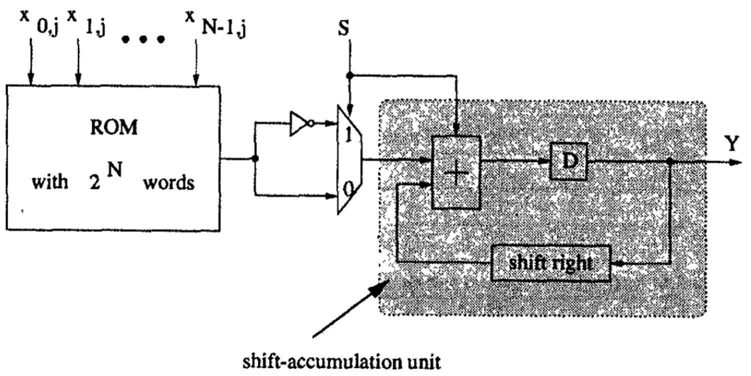


Figure 3.2: Architecture for computing inner product between two length N vectors using Distributed Arithmetic

Figure 3.2 shows 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 $j = W - 1$).

3.2.2 Offset Binary Coding based Distributed Arithmetic

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

$$\begin{aligned} x_i &= \frac{1}{2}[x_i - (-x_i)] \\ &= \frac{1}{2}[-(x_{i,W-1} - \overline{x_{i,W-1}}) + \sum_{j=1}^{W-1}(x_{i,W-1-j} - \overline{x_{i,W-1-j}})2^{-j} - 2^{-(W-1)}] \end{aligned} \tag{3.6}$$

where $-x_i$ is

$$-x_i = -\overline{x_{i,W-1}} + \sum_{j=1}^{W-1} \overline{x_{i,W-1-j}}2^{-j} + 2^{-(W-1)} \tag{3.7}$$

Define $d_{i,j}$ as

$$d_{i,j} = \begin{cases} x_{i,j} - \overline{x_{i,j}}, \; for\; j \neq W-1 \\ -(x_{i,W-1} - \overline{x_{i,W-1}}), \; for\; j = W-1 \end{cases} \tag{3.8}$$

Rewrite (3.6) using (3.8)

$$x_i = \frac{1}{2}[\sum_{j=0}^{W-1} d_{i,W-1-j}2^{-j} - 2^{-(W-1)}] \tag{3.9}$$

From (3.9) and (3.1)

$$\begin{aligned} Y &= \sum_{i=0}^{N-1} c_i[\frac{1}{2}(\sum_{j=0}^{W-1} d_{i,W-1-j}2^{-j} - 2^{-(W-1)})] \\ &= \sum_{j=0}^{W-1}(\sum_{i=0}^{N-1}\frac{1}{2}c_i d_{i,W-1-j})2^{-j} - (\sum_{i=0}^{N-1}\frac{1}{2}c_i)2^{-(W-1)} \end{aligned} \tag{3.10}$$

Define D_j

$$D_j = \sum_{i=0}^{N-1}\frac{1}{2}c_i d_{i,W-1-j}, \; for\; j = 0\; to\; W-1 \tag{3.11}$$

and D_{extra}

$$D_{extra} = \sum_{i=0}^{N-1}\frac{1}{2}c_i \tag{3.12}$$

Substituting D_j's and D_{extra} into (3.10)

$$Y = \sum_{j=0}^{W-1} D_{W-1-j}2^{-j} + D_{extra}2^{-(W-1)} \tag{3.13}$$

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 3.3 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.

Table 3.2: ROM contents for vector length $N = 4$

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

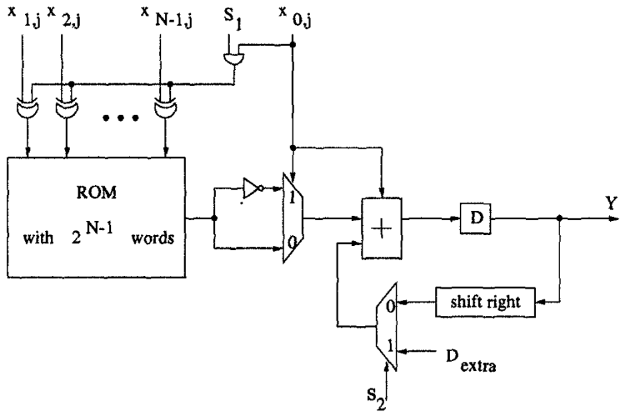


Figure 3.3: Architecture for computing inner product of two length-N vectors using OBC based DA

3.2.3 ROM Decomposition for Distributed Arithmetic

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(refer to figure 3.4). 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.

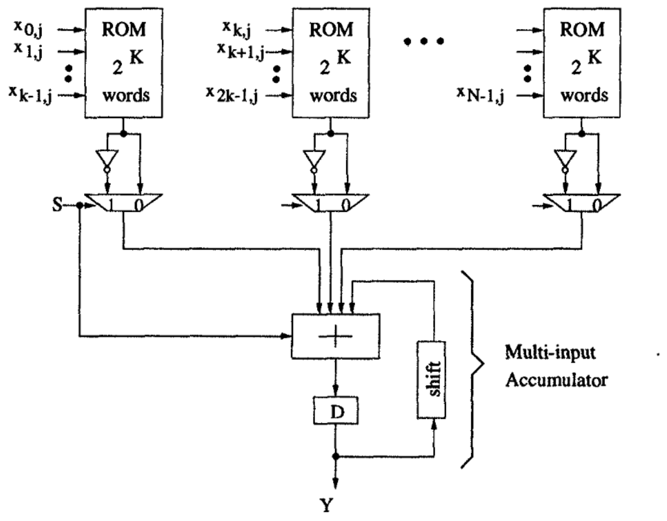


Figure 3.4: Architecture for computing inner product of two length-N vectors using DA with ROM decomposition

3.2.4 Applying Distributed Arithmetic to LSTM

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

$$W = [w_0, w_1, w_2, w_3..., w_{N-1}]^\top \tag{3.14}$$

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

$$Wx = [w_0.x, w_1.x, w_2.x, w_3.x..., w_{N-1}.x]^\top \tag{3.15}$$

So a matrix-vector product can be rewritten as an inner product between vectors. We can apply DA to each of these vector multiplications. Applying DA with ROM decomposition, the required ROM size is $(N/K)2^K$ for a single vector product but since we are applying DA to all the vector products, the ROM size would be $N * (N/K)2^K = (N^2/K)2^K$.

A modest network has $N = 128$ and using ROMs of size $K = 8$, using $(N^2/K)2^K$, the ROM size turns out to be 512 MB assuming a word size of 8 bits. This can be further halved by using OBC which brings it down to 256 MB.

This can be further decreased by exploiting some unique properties of the weight matrices which we will explore in the next section.

3.3 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 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 i.e., 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. Pruning and Quantization
2. Low Displacement Rank Networks
3. Knowledge Diffusion

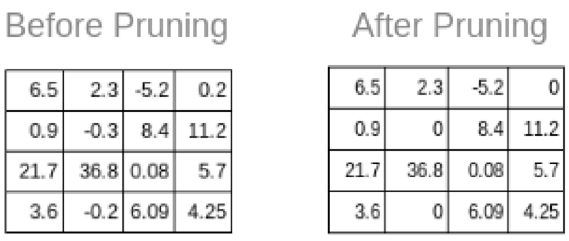
3.3.1 Pruning and Quantization

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[8].

The following is the procedure to prune during the training phase:

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.



Before Pruning

6.5	2.3	-5.2	0.2
0.9	-0.3	8.4	11.2
21.7	36.8	0.08	5.7
3.6	-0.2	6.09	4.25

After Pruning

6.5	2.3	-5.2	0
0.9	0	8.4	11.2
21.7	36.8	0.08	5.7
3.6	0	6.09	4.25

Figure 3.5: A $4X4$ matrix pruned with a threshold magnitude of 0.25

[8] shows that up to 80-90 percent of the parameters have a minimal contribution to the performance of a network. Therefore introducing sparsity in the network up to 90 percent can theoretically reduce the required storage size to about 25.6 MB in the previous calculation for the network of 128 hidden units. It is still yet to be explored how to leverage this large sparsity into the DA implementation. [15] uses an innovative way to exploit factorization of certain sparse matrices to reduce the memory requirement of DA.

Quantization

Further reduction of the model size is possible by quantizing the remaining parameters to K levels [8]. The standard procedure to do this is through K-means clustering. Therefore only $logK$ bits will be required to store the parameters. Literature [8] shows that the two practices of pruning and quantization can be combined together to provide significant drop in the model size.

3.3.2 Low Displacement Rank Methods

This method of compression takes advantage of the structured nature of certain types of matrices to reduce the number of unique parameters from $O(n^2)$ to $O(n)$. This not only reduces the storage complexity from $O(n^2)$ to $O(n)$, it also reduces, due to fast matrix multiplication methods, the computational complexity roughly from $O(n^2)$ to $O(nlogn)$ or $O(nlogn^2)$. 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.[9]
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 method. 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 achieves a significant speedup on their FPGA implementation of LSTMs [2].

3.3.3 Knowledge Distillation

It is often the case that while training a network to reach optimal efficiency requires large models, most of the fully trained network's connections are not of significant importance as mentioned in the Pruning section. [16] suggests to instead of pruning, training a smaller model to imitate the behaviour of the trained large model. The largeer network is often called the $teacher$ and the smaller network the $student$. [10] utilizes both quantization and distillation to reduce the model size many-fold.

Chapter 4

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] Cs231n: Recurrent neural networks. `http://cs231n.stanford.edu/slides/2017/`.

[2] Shuo Wang, Zhe Li, Caiwen Ding, Bo Yuan, Qinru Qiu, Yanzhi Wang, and Yun Liang. C-lstm: Enabling efficient lstm using structured compression techniques on fpgas. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 11–20. ACM, 2018.

[3] Qian Wang, Peng Li, and Yongtae Kim. A parallel digital vlsi architecture for integrated support vector machine training and classification. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 23 (8):1471–1484, 2015.

[4] Jichen Wang, Jun Lin, and Zhongfeng Wang. Efficient hardware architectures for deep convolutional neural network. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 65(6):1941–1953, 2018.

[5] Klaus Greff, Rupesh K Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber. Lstm: A search space odyssey. *IEEE transactions on neural networks and learning systems*, 28(10):2222–2232, 2017.

[6] Google ai blog. `https://ai.googleblog.com/2016/09/a-neural-network-for-machine.html`, .

[7] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. 1999.

[8] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. *CoRR*, abs/1510.00149, 2015. URL `http://arxiv.org/abs/1510.00149`.

[9] Liang Zhao, Siyu Liao, Yanzhi Wang, Zhe Li, Jian Tang, Victor Pan, and Bo Yuan. Theoretical properties for neural networks with weight matrices of low displacement rank. *arXiv preprint arXiv:1703.00144*, 2017.

[10] Antonio Polino, Razvan Pascanu, and Dan Alistarh. Model compression via distillation and quantization. *arXiv preprint arXiv:1802.05668*, 2018.

[11] Haşim Sak, Andrew Senior, and Françoise Beaufays. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In *Fifteenth annual conference of the international speech communication association*, 2014.

[12] 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 *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 75–84. ACM, 2017.

[13] Understanding lstm networks. `colah.github.io/posts/2015-08-Understanding-LSTMs/`,.

[14] Keshab K Parhi. *VLSI digital signal processing systems: design and implementation*. John Wiley & Sons, 2007.

[15] Shrutisagar Chandrasekaran and Abbes Amira. Novel sparse obc based distributed arithmetic architecture for matrix transforms. In *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*, pages 3207–3210. IEEE, 2007.

[16] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.