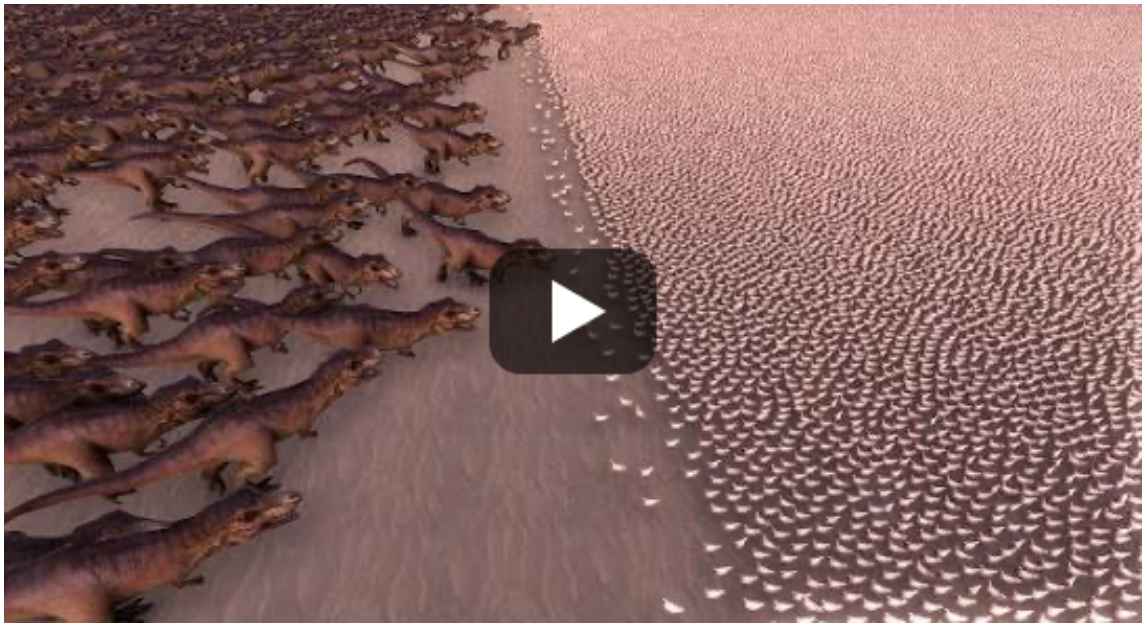
INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY
HYDERABAD

Introduction to Parallel and Scientific Computing

Optimizing Performance of Recurrent Neural Networks on GPUs: An implementation using OpenMP



Students:
Shubodh and Manika

Professor:
Dr. Pawan Kumar

Academic year 2019-2020

Indice

1

Introduction

You must have observed the image on the cover of this document. It is: '1000 T-rex or 80,000 chicken?'. A person who is unware of the concept of 'multithreading', 'GPU', 'cores', etc may say that 1000 T-rexs will win.

As students of the Introduction to 'parallel' and scientific computing, we will say that 80,000 chicken will win.

Don't believe us? Watch the same simulation video on youtube, but we would like you to go to the next page of our document for a deeper understanding :)

2

Our motivation to do this paper

After much research, we chose 'Optimizing Performance of Recurrent Neural Networks on GPUs' to be the topic for our coursework project. To be honest, we didn't have much motivation about the topic before hand. Hence, we started reading about the project through our allocated research paper, wikipedia, Ian Goodfellow, and what not.

Why? We will ask- Why not?

How cool would it be to implement a neural network that is used in text prediction, image captioning, ... you name the application, RNN(or should we say, extended RNN) are used there!!

Tip we learnt: Think about RNN when you hear the word: 'memorization'

3

System Configuration

We have used Indus system for our project implementation.

It has 2 sockets with 12 cores per socket. Hence, 24 physical cores.
We haWith virtual threading, we can have 24*2 = 48 cores virtual cores.

```
[manika@indus Final]$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                48
On-line CPU(s) list:   0-47
Thread(s) per core:    2
Core(s) per socket:    12
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 85
Model name:            Intel(R) Xeon(R) Silver 4116 CPU @ 2.10GHz
Stepping:              4
CPU MHz:               2100.000
BogoMIPS:              4200.00
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              1024K
L3 cache:              16896K
```

4

Recurrent Neural Network

Definition: A class of artificial neural networks with 'feedback connections'. The connection between the 'nodes' in an RNN form a directed graph along a time series.

Formally, we can say an RNN is: $input + previousHidden- > Hidden- > output$ for initial layer, or $input + previousHidden- > Hidden- > output$ for subsequent layers

Notations:
W, V, U: weights learnt during training
x, s, o: input, hidden and output states, respectively

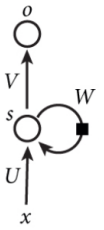


Figura 1: Computational Graph of RNN

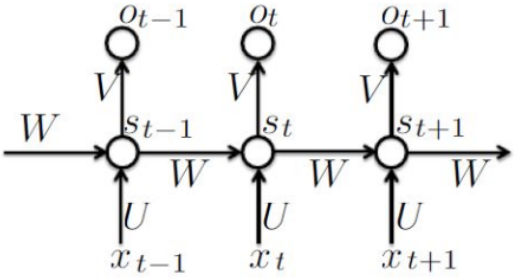


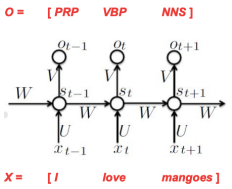
Figura 2: Unfolded computational Graph of RNN

5

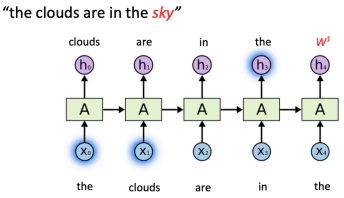
Uses of Recurrent Neural Network

These are a few application where RNN(or extended forms of RNN) are used. . . To remember, the applications are not limited till here, there are a lot of applications using RNN as sentiment classification, machine translation, video classification on frame level, etc. . .

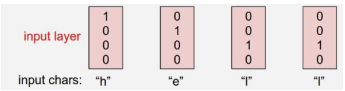
- Speech tagging: Given a sentence, tag each word 'x_{t+i}' to it's corresponsing grammar class(example vbp for verb, nns for noun, prp for preposition, etc)



- Language Modelling - Word Level: Given an incomplete sentence, generate the corresponding word to complete the sentence



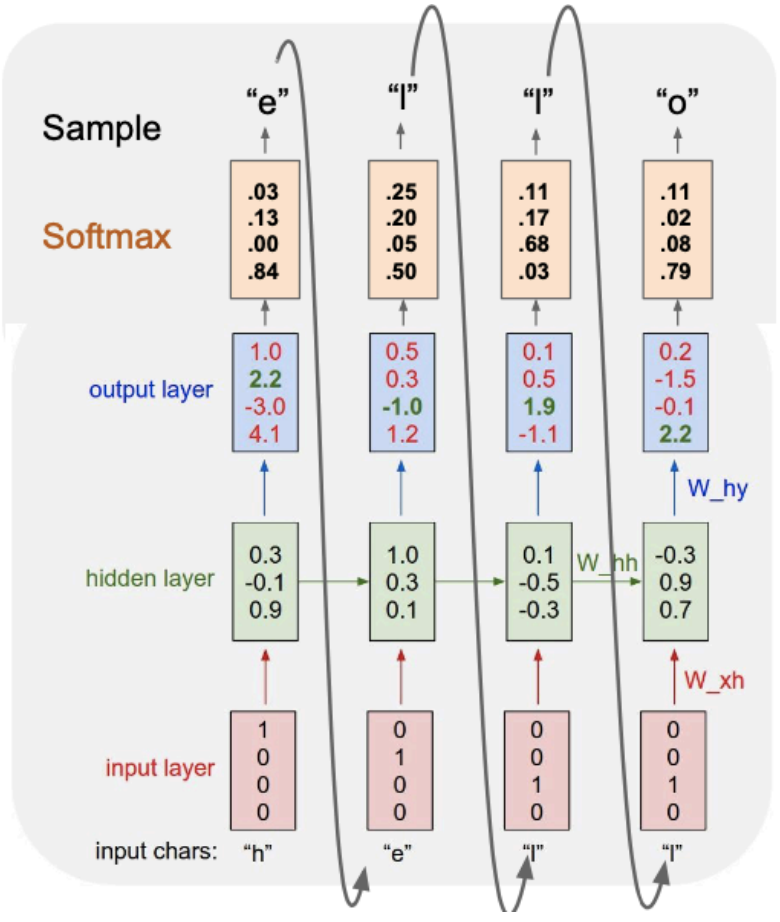
- Language Modelling - Character Level: Given a set of characters, fill the next missing character to form a word. For example, using the given vocabulary, tell what will come in place of the question mark: hell?



6

Explanation of an application implemented via RNN

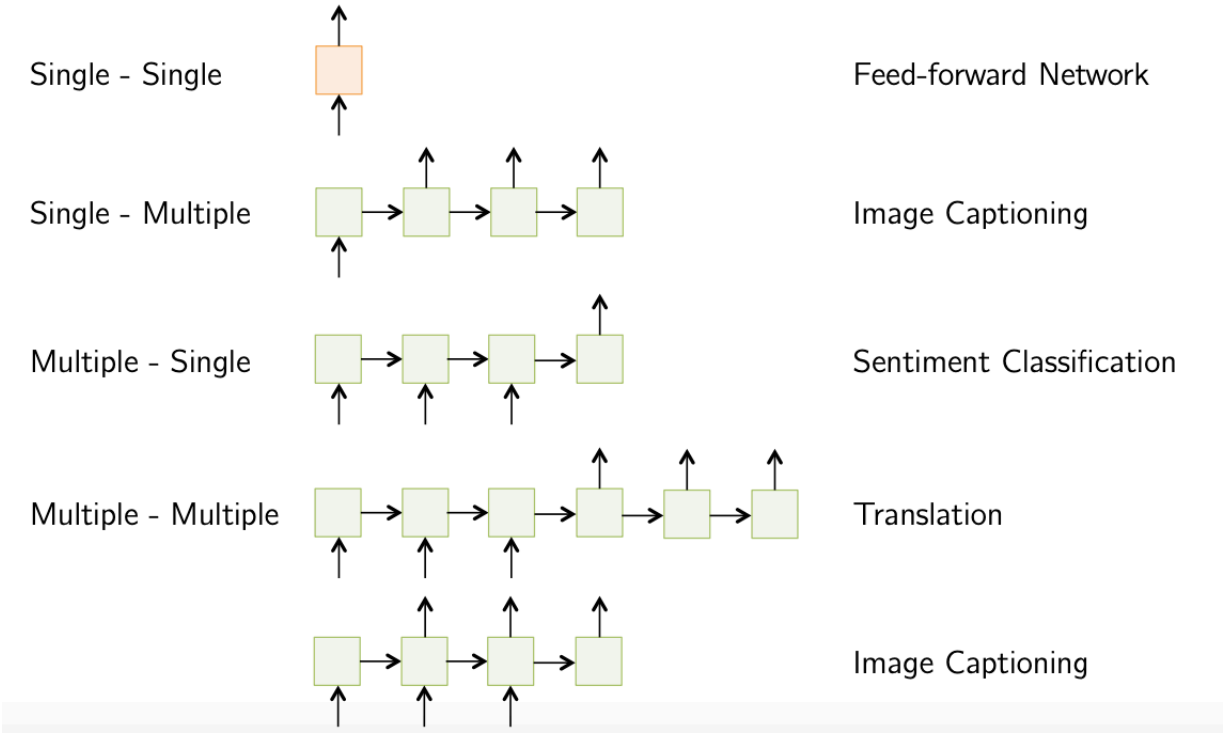
To generate interest in the upcoming topics, we will show a basic, we present to you a very basic diagram explaning RNN used for Language Modelling at Character Level



7

Process sequence of RNN

Depending on the type of application we are going to use RNN for, we have a various process sequence flows as follows:



8

An RNN cell explained! ! !

This is how an RNN cell looks like.

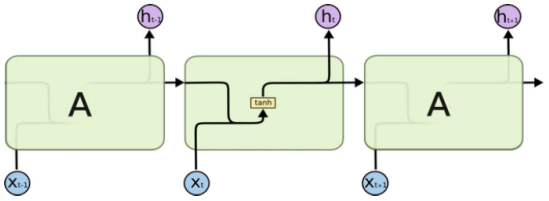


Figura 3: An RNN cell

The value 'h' is computed using the below equations.

$$h_t = f_W(h_{t-1}, x_t)$$

$$\downarrow$$

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

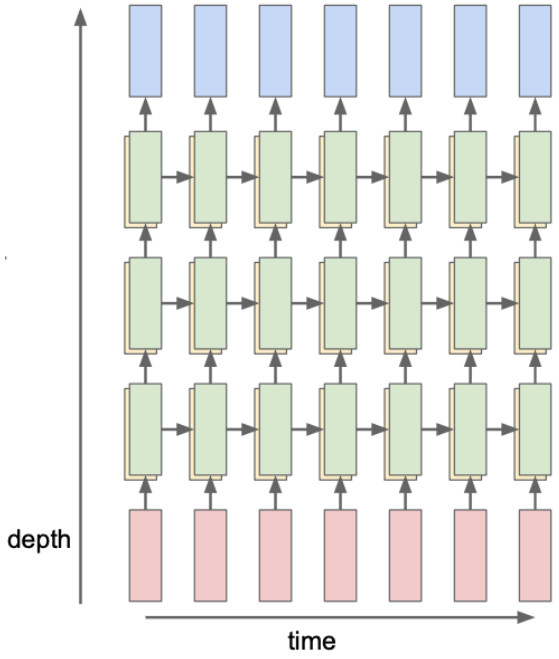
$$y_t = W_{hy}h_t$$

If you are satisfied by the equations for 'one cell' of a 'single layer', let us introduce you to multilayer RNN.

9

Multi-layer RNN

This is a high-level diagram for a Multi-layer RNN. The red ones denote the input 'x', the green ones denote hidden layers 'h^l, and the blue ones are the output 'y'



Multilayer RNNs

$$h_t^l = \tanh W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$

$h \in \mathbb{R}^n$ $\qquad$ W^l $[n \times 2n]$

The way in which we understood the working of a single cell of RNN, same way we can understand working of multiple cells in multiple layers as above computation of 'h'

10

LSTM: Overcoming the shortcoming of RNN

Still, RNN is not a perfect guy. We need to move towards something that will be good for a long run, unlike RNN, that is good for a short run only. Also, as more layers using certain activation functions are added to neural networks, the gradients of the loss function approaches zero, making the network hard to train. This problem, popularly known as 'vanishing gradient' is faced in RNNs too.

LSTM: Long Short Term Memory, comes to our rescue.

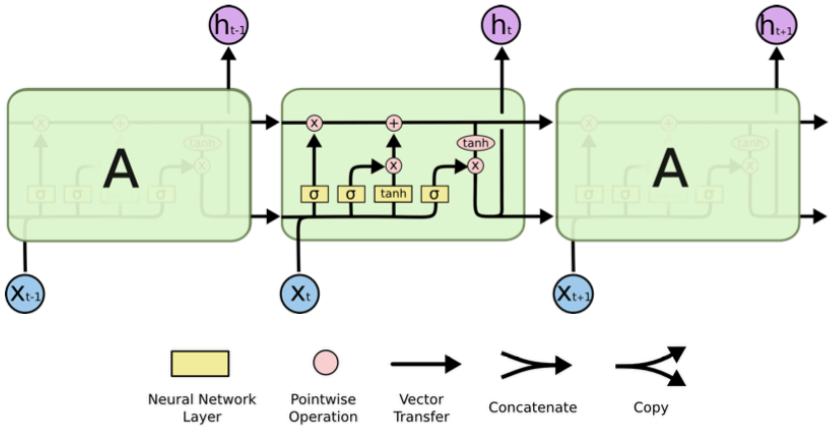
LSTMs are good for long term dependencies.LSTMs have the motto: *We don't want to remember everything, just the important things for a long time.* The LSTM uses this idea of "Constant Error Flow" for RNNs to create a "Constant Error Carousel" (CEC) which ensures that gradients don't decay. The key component is a memory cell that acts like an accumulator (contains the identity relationship) over time. Instead of computing new state as a matrix product with the old state, it rather computes the difference between them. Expressivity is the same, but gradients are better behaved.

Defination of LSTM: A class of artificial neural networks with 'feedback connections'. It is a variant of vanilla RNN which is capable of learning long term dependencies.

11

An LSTM cell explained! ! !

This is how an LSTM cell looks like



The value 'h' is computed using the below equations

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

Notations-

f is forget gate

i is input gate

o is output value

g is input node

c' is temporary variable holding updated cell state

c is cell state

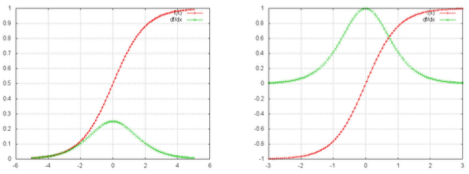
```
void nextHiddenState(vector<vector<float> >& input_t, vector<vector<float> >& h_tmi

    vector<vector<float> > i_t_linear = sum_Wx_Rh_b(input_t, h_tminus1, hiddenSize,
    vector<vector<float> > f_t_linear = sum_Wx_Rh_b(input_t, h_tminus1, hiddenSize,
    vector<vector<float> > o_t_linear = sum_Wx_Rh_b(input_t, h_tminus1, hiddenSize,
    vector<vector<float> > g_t_linear = sum_Wx_Rh_b(input_t, h_tminus1, hiddenSize,

    vector<vector<float> > i_t = matSigma(i_t_linear);
    vector<vector<float> > f_t = matSigma(f_t_linear);
    vector<vector<float> > o_t = matSigma(o_t_linear);
    vector<vector<float> > g_t = matTanh(g_t_linear);

    vector<vector<float> > temp_fOc = matMulElement(f_t, c_tminus1);
    vector<vector<float> > temp_iOg = matMulElement(i_t, g_t);
    vector<vector<float> > c_t = matSum(temp_iOg, temp_fOc);
    vector<vector<float> > tanh_c_t = matTanh(c_t);
    vector<vector<float> > h_t = matMulElement(o_t, tanh_c_t);

    c_tminus1 = c_t;
    h_tminus1 = h_t;
}
 double lstmNaive(int hiddenSize, int miniBatch, int seqLength, int numLayers, int
 ...
 ...
randMat(input_t, range);
   // initializing hidden and latent state
randMat(h_tminus1, range);
randMat(c_tminus1, range);
    int i;
    for (i = 0; i < seqLength; ++i){
        nextHiddenState(input_t, h_tminus1, c_tminus1, hiddenSize, miniBatch);
    }
```

12

Vanishning gradient: LSTM vs RNN

This following graphh explains the issue of vanishing gradient

- When weight or activation functions (their derivatives) are:
  - < 1 Vanishing Gradients
  - > 1 Exploding Gradients



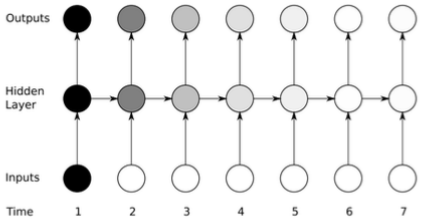


Figure 4.1: **The vanishing gradient problem for RNNs.** The shading of the nodes in the unfolded network indicates their sensitivity to the inputs at time one (the darker the shade, the greater the sensitivity). The sensitivity decays over time as new inputs overwrite the activations of the hidden layer, and the network 'forgets' the first inputs.

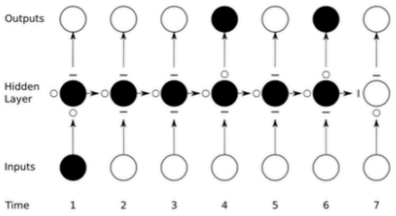


Figure 4.4: **Preservation of gradient information by LSTM.** As in Figure 4.1 the shading of the nodes indicates their sensitivity to the inputs at time one; in this case the black nodes are maximally sensitive and the white nodes are entirely insensitive. The state of the input, forget, and output gates are displayed below, to the left and above the hidden layer respectively. For simplicity, all gates are either entirely open ('O') or closed ('—'). The memory cell 'remembers' the first input as long as the forget gate is open and the input gate is closed. The sensitivity of the output layer can be switched on and off by the output gate without affecting the cell.

13

Implementation of paper: Optimizing Performance of Recurrent Neural Networks on GPUs

In this chapter, we demonstrate that by exposing parallelism between operations within the network, an order of magnitude speedup across a range of network sizes can be achieved over a naive implementation.

For a quick reference, below are the equations for a single cell in LSTM:

$$
\begin{aligned}
i_i &= \sigma(W_i x_i + R_i h_{t-1} + b_i) \\
f_t &= \sigma(W_f x_t + R_f h_{t-1} + b_f) \\
o_t &= \sigma(W_o x_t + R_o h_{t-1} + b_o) \\
c'_t &= \tanh(W_c x_t + R_c h_{t-1} + b_c) \\
c_t &= f_t \circ c'_{t-1} + i_t \circ c'_t \\
h_t &= o_t \circ \tanh(c_t)
\end{aligned}
$$

This is an standard four-gate LSTM network without peephole connections, implemented over 5 optimizations as follows:

13.1 Naive implementation

Code name: *LSTM_naive.cpp*

```
vector<vector<float> > matMul(vector<vector<float> >& mat1, vector<vector<float> >&
```

```
    vector<vector<float> > mat3(mat1.size(), vector<float>(mat2[0].size())) ;

    omp_set_num_threads(8);
    #pragma omp parallel for
    for (int i =0; i < mat1.size(); ++i){
        for (int j = 0; j < mat2[0].size(); ++j){
            for (int k=0; k < mat2.size(); ++k){
                mat3[i][j] += mat1[i][k] * mat2[k][j];
            }
        }
    }
    return mat3;
}
```

Procedure followed: Each individual kernel (ie. matrix multiplication, sigmoid, point-wise addition, etc.) is implemented as a separate kernel. The kernels are executed back-to-back sequentially

Forward pass performance: Poor

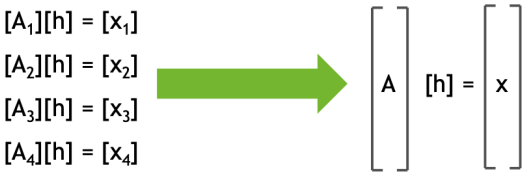
Psuedocode:

```
for layer in layers:
  for iteration in iterations:
     perform 4 SGEMMs on input from last layer
     perform 4 SGEMMs on input from last iteration
     perform point-wise operations
```

13.2 Optimization 1: Grouped GEMMs

Code name: *LSTM_opti_1.cpp*

Procedure followed: Combine matrix operations sharing the same input into a single larger matrix operation. In the forward pass the standard formulation of an LSTM leads to eight matrix matrix multiplications: four operating on the recurrent input (Rh_{t1}), four operating on the input from the previous layer (Wx_t). In these groups of four the input is shared, although the weights are not. As such, it is possible to reformulate a group of four matrix multiplications into a single matrix multiplication of four times the size.

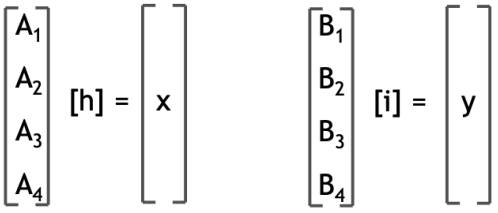


Forward pass performance: Almost two times better than the 'naive implementation'

13.3 Optimization 2: Streamed GEMMs

Code name: *LSTM_opti_2.cpp*

Procedure followed: Execute both matrix multiplications on concurrently((Rh_{t1}) and (Wx_t)) as these matrix multiplications are independent.



Forward pass performance: Almost 1.5x times better than the 'Grouped GEMMs'

13.4 Optimization 3: Fused point-wise

Code name: *LSTM_opti_3.cpp*

Procedure followed: Fuse all of the point-wise kernels into one larger kernel. This step did not give us expected speedup. Reason: CUDA involves memory transfers from CPU->GPU->CPU during CUDA kernel calls. OpenMP involves only context switches during function calls.

Forward pass performance: Almost same as 'Streamed GEMMs', only slight speedup compared to 'Streamed GEMMs'.

Psuedocode for Omptimization 2 and 3 combined:

```
for layer in layers:
  for iteration in iterations:
     perform sgemm on input from last layer in stream A
     perform sgemm on input from last iteration in stream B
     wait for stream A and stream B
     perform point-wise operations in one kernel
```

13.5 Optimization 4: Pre-transpose the weight matrix

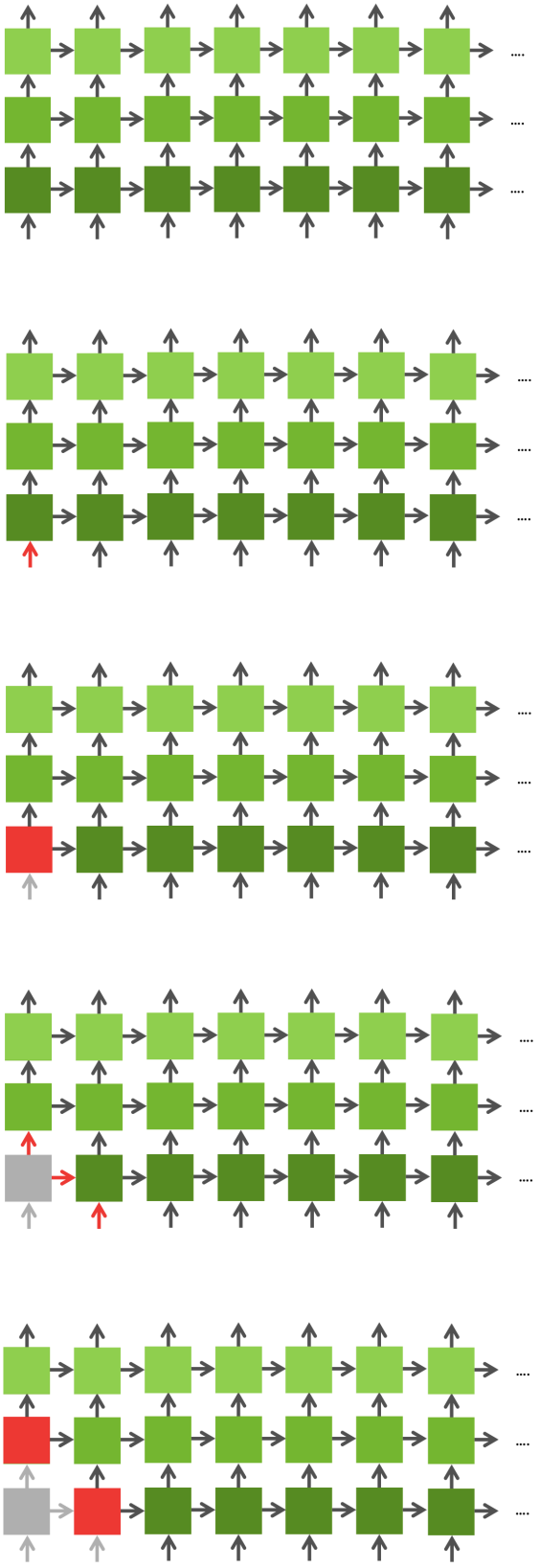
Code name: *LSTM_opti_4.cpp*

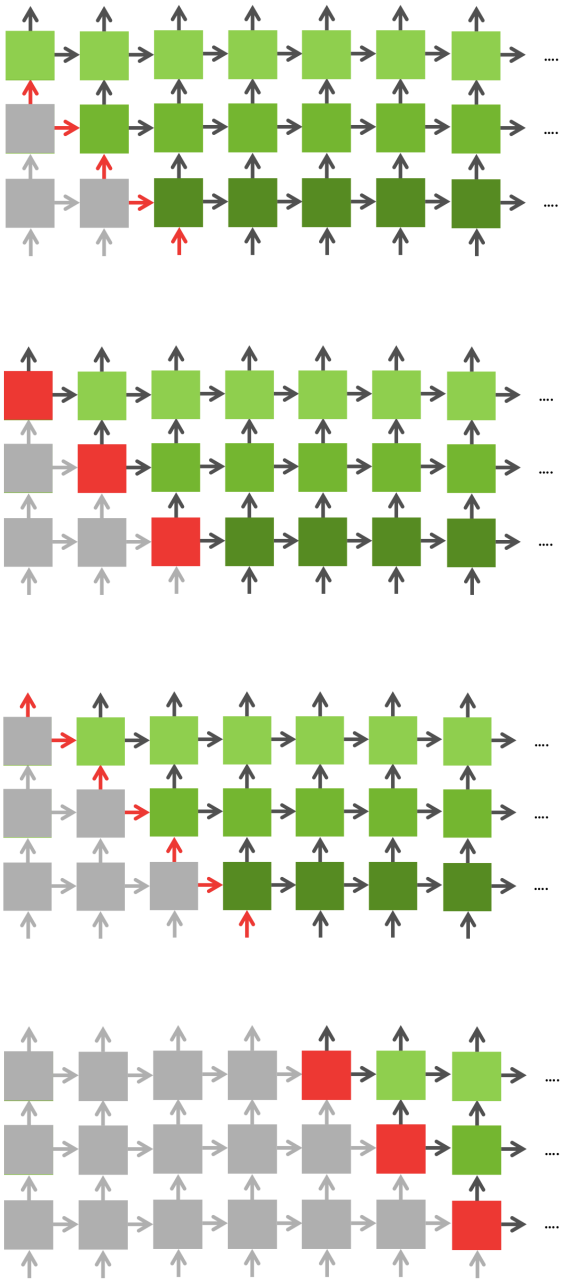
Procedure followed: All weight matrices in a layer are the same. A pre-processing step before we start can make the overall cost cheaper.

Forward pass performance: We expected a very good speedup, but we got a very high time using this optimization. Reason: The logic we tried to implement was perform transpose of weight matrix and multiply it with input matrix, as mentioned in the paper. We successfully implemented this step once for a whole layer, hence saved a lot of time. However, to multiply transpose of weight matrix with the input matrix, it took a very long time(particularly, $O(n^3 + n^2 + n^2)$ time. We tried to make this efficient, but we couldn't come out with an efficient approach(the n^3 way would access the matrix column wise, making this approach very inefficient)

13.6 Multiple Layers

Procedure followed: By the following sequence of diagrams, we can observe that Wx of an LSTM cell can be calculated without waiting for previous hidden output. This can enhance the speedup extensively. We have tried to implement multiple layers naive, opti1, opti2, opti3, but we couldn't effectively write the overlapping layers code.





14

Tabular Results

We obtained the following results by implementing the above mentioned.

Comparision of our speedups and original speedups			
Optimization Name	Time Taken	Our code Speedup	Original code Speedup
Naive	0.40m	1x	1x
Groupped GEMMs	0.25m	1.6x	1.9x
Streamed GEMMs	0.1783m	2.324x	2.8x
Fused Point-wise	0.1769m	2.23	5.3x
Pre-Transpose	1.18m	-	6.2x

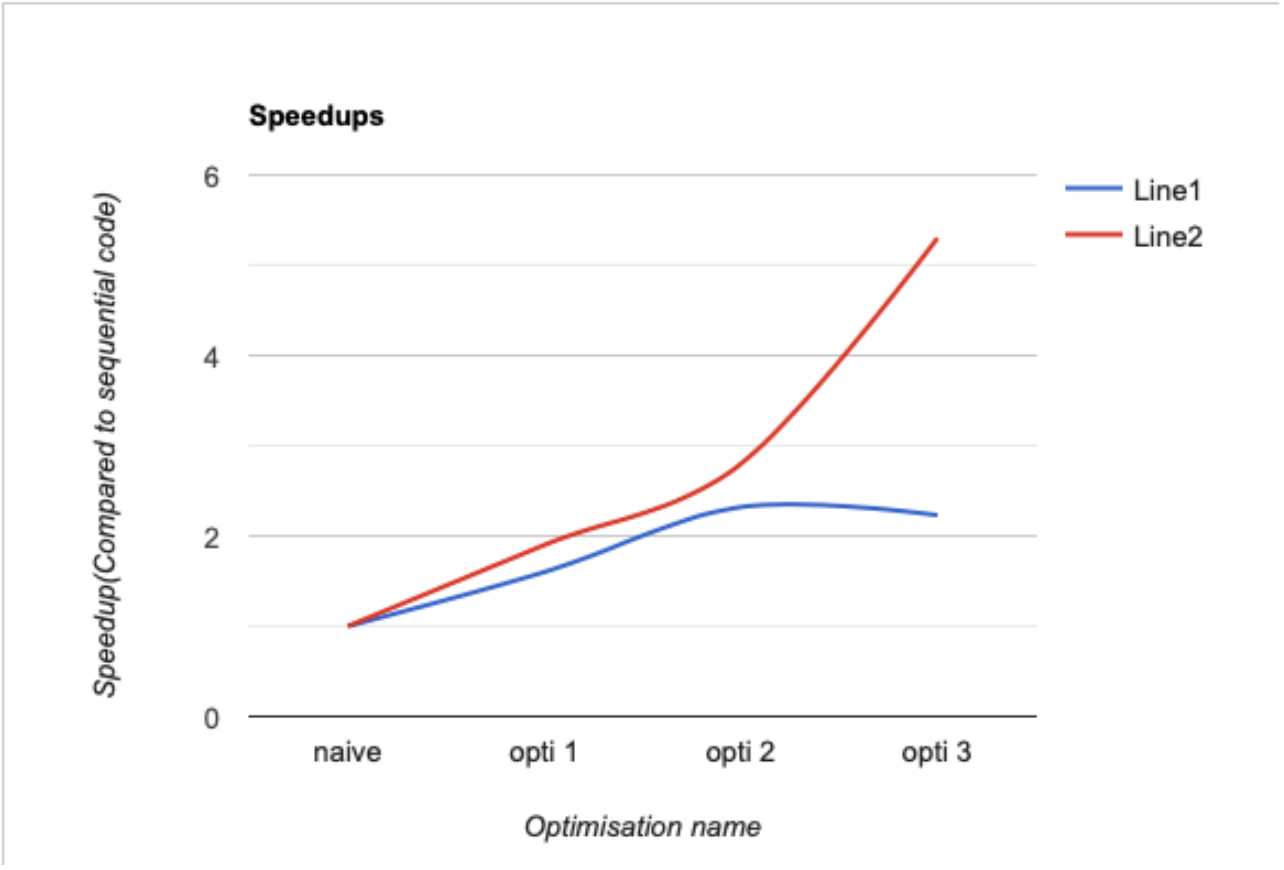
15

Visual Results

```
[manika@indus Project]$ g++ -fopenmp LSTM_naive.cpp
[manika@indus Project]$ time ./a.out
Running with default settings
Time for the run number 0 :  0.14747200 us

Average Runtime for LSTM naive is 0.14747200ms

real    0m40.859s
user    5m14.286s
sys     0m0.101s
[manika@indus Project]$ g++ -fopenmp LSTM_opti_1.cpp
[manika@indus Project]$ time ./a.out
Running with default settings
Time for the run number NAIVE EFFICIENT 0 :  -0.30117200 ms

Average Runtime for LSTM NAIVE EFFICIENT is -0.30117200 ms


real    0m25.707s
user    5m6.855s
sys     0m0.166s
[manika@indus Project]$ g++ -fopenmp LSTM_opti_2.cpp
[manika@indus Project]$ time ./a.out
Running with default settings
Time for the run number NAIVE EFFICIENT 0 :  -0.18433400 ms

Average Runtime for LSTM NAIVE EFFICIENT is -0.18433400 ms


real    0m17.823s
user    3m3.730s
sys     0m0.123s
[manika@indus Project]$ g++ -fopenmp LSTM_opti_3.cpp
[manika@indus Project]$ time ./a.out
Running with default settings
Time for the run number NAIVE EFFICIENT 0 :  -0.31468400 ms

Average Runtime for LSTM NAIVE EFFICIENT is -0.31468400 ms


real    0m17.692s
user    3m0.422s
sys     0m0.246s



[manika@indus Project]$ g++ -fopenmp LSTM_opti_4_code.cpp
[manika@indus Project]$ time ./a.out
Running with default settings
Time for the run number 0 :  0.38534300 us

Average Runtime for LSTM naive is 0.38534300ms

real    1m18.628s
user    5m50.227s
sys     0m30.678s
[manika@indus Project]$
```

16

Graphical Results



Riferimenti bibliografici

[1] Optimizing Performance of Recurrent Neural Networks on GPUs, Jeremy Appleyard, Tomáš Kocˇiský, Phil Blunsom

[2] "Anyone Can Learn To Code an LSTM-RNN in Python (Part 1: RNN) - I Am Trask."

[3] "Recurrent Neural Networks Tutorial, Part 1 – Introduction to RNNs." WildML

[4] Deep Learning, Ian Goodfellow

[5] RNN, Md Shad Akhtar, IIT Patna

[6] RNN, Fei-Fei Li , Justin Johnson ,Serena Yeung