15-418 Final Report

Xiangkai Zeng `xiangkaz`

Matthew Lipari `mlipari`

Summary

We implemented an Encoder-Decoder Neural Network in CUDA on Nvidia GPUs. We optimized the various steps and layers of the network against naive and CPU methods.

Background

Deep Learning and Neural Networks are becoming increasingly popular in the areas of Machine Learning, Artificial Intelligence, Computer Vision, Natural Language Processing and many more. However, in order to train a deep neural network, you often need a large amount of data and time. Since many implementations for neural networks involve large matrix multiplications, people often use GPUs to train their models.

A widely used neural network structure for Natural Language Processing and Computer Vision is called a Recurrent Neural Network. The original RNN suffers from the gradient vanishing and the gradient exploding problem. Both the LSTM RNN and GRU RNN have been invented to overcome the problem. Similar to other layers in neural networks, training these parts is often time-consuming. Fortunately, different elements in the weight matrices can be computed in parallel. Therefore, GPUs are a good way to speedup the training and testing process of neural network models. Specifically, a special type of RNN is called Encoder-Decoder Network which is designed to address the sequence-to-sequence problems. It consists of two RNN and the outputs of the first RNN are the inputs to the second RNN, which makes it more complicated to implement and parallelize. Besides that, the Encoder-Decoder could also have a attention mechanism, which tries to attend to different outputs of the encoder when the decoder is predicting the sequences. The attention mechanism gives the model direct access to all the previous outputs so that it does not need to rely only on the last output of the encoder. In this way, the model has more information and therefore generally has better performance compared to a simple Encoder-Decoder Network.

Model Definition

Here we define the Long Short Term Memory networks (LSTM), which is a widely used neural network model. Note that the following formulas are taken from `http://arunmallya.github.io/writeups/nn/lstm/index.html` (Listed in the References section).

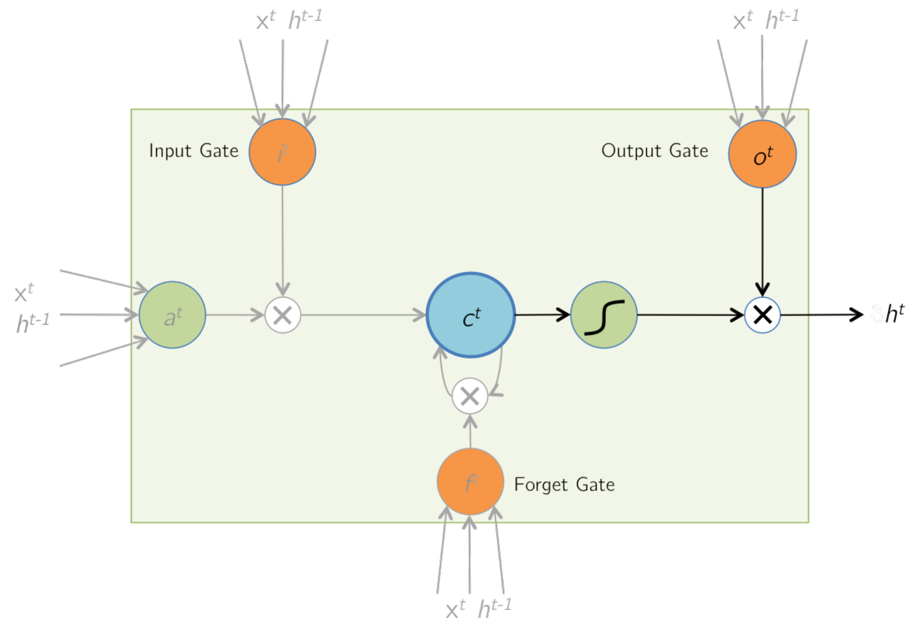

Figure 1: LSTM Network

Figure 1 shows the architecture of LSTM Network. For forward computation, essentially, at time t the LSTM computes the hidden state and cell state based on the input, the hidden state and cell state at time $t1$. At time step t, let x^t be an input vector, h^{t-1} be the hidden state at time step $t-1$, c^{t-1} be the cell state at time step $t-1$.

$$\begin{aligned}
a^t &= \tanh(W_c x^t + U_c h^{t-1}) = \tanh(\hat{a}^t) \\
i^t &= \sigma(W_i x^t + U_i h^{t-1}) = \sigma(\hat{i}^t) \\
f^t &= \sigma(W_f x^t + U_f h^{t-1}) = \sigma(\hat{f}^t) \\
o^t &= \sigma(W_o x^t + U_o h^{t-1}) = \sigma(\hat{o}^t)
\end{aligned}$$

Ignoring the non-linearities, we have

$$
\begin{aligned}
z^t = \begin{bmatrix} \hat{a}^t \\ \hat{i}^t \\ \hat{f}^t \\ \hat{o}^t \end{bmatrix} &= \begin{bmatrix} W^c & U^c \\ W^i & U^i \\ W^f & U^f \\ W^o & U^o \end{bmatrix} \times \begin{bmatrix} x^t \\ h^{t-1} \end{bmatrix} \\
&= W \times I^t
\end{aligned}
$$

Then we compute the cell state c^t and the hidden state h^t. Here $\odot$ represents elementwise multiplication.

$$
\begin{aligned}
c^t &= i^t \odot a^t + f^t \odot c^{t-1} \\
h^t &= o^t \odot \tanh(c^t)
\end{aligned}
$$

For backward computation, given the previous gradient $\delta h^t = \frac{\partial J}{\partial h^t}$,

$$
\begin{aligned}
\frac{\partial E}{\partial o_i^t} &= \frac{\partial E}{\partial h_i^t} \cdot \frac{\partial h_i^t}{\partial o_i^t} \\
&= \delta h_i^t \cdot \tanh(c_i^t) \\
\therefore \delta o^t &= \delta h^t \odot \tanh(c^t)
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial E}{\partial c_i^t} &= \frac{\partial E}{\partial h_i^t} \cdot \frac{\partial h_i^t}{\partial c_i^t} \\
&= \delta h_i^t \cdot o_i^t \cdot (1 - \tanh^2(c_i^t)) \\
\therefore \delta c^t + &= \delta h^t \odot o^t \odot (1 - \tanh^2(c^t))
\end{aligned}
$$

Notice here J denotes the loss function and δc^t is computed at time step $t+1$, which is defined later.

$$
\begin{aligned}
\frac{\partial E}{\partial i_i^t} &= \frac{\partial E}{\partial c_i^t} \cdot \frac{\partial c_i^t}{\partial i_i^t} & \frac{\partial E}{\partial f_i^t} &= \frac{\partial E}{\partial c_i^t} \cdot \frac{\partial c_i^t}{\partial f_i^t} \\
&= \delta c_i^t \cdot a_i^t & &= \delta c_i^t \cdot c_i^{t-1} \\
\therefore \delta i^t &= \delta c^t \odot a^t & \therefore \delta f^t &= \delta c^t \odot c^{t-1}
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial E}{\partial a_i^t} &= \frac{\partial E}{\partial c_i^t} \cdot \frac{\partial c_i^t}{\partial a_i^t} & \frac{\partial E}{\partial c_i^{t-1}} &= \frac{\partial E}{\partial c_i^t} \cdot \frac{\partial c_i^t}{\partial c_i^{t-1}} \\
&= \delta c_i^t \cdot i_i^t & &= \delta c_i^t \cdot f_i^t \\
\therefore \delta a^t &= \delta c^t \odot i^t & \therefore \delta c^{t-1} &= \delta c^t \odot f^t
\end{aligned}
$$

We can see that for the above gradient, all we need to do it elementwise multiplication.

$$
\begin{aligned}
\delta \hat{a}^t &= \delta a^t \odot (1 - \tanh^2(\hat{a}^t)) \\
\delta \hat{i}^t &= \delta i^t \odot i^t \odot (1 - i^t) \\
\delta \hat{f}^t &= \delta f^t \odot f^t \odot (1 - f^t) \\
\delta \hat{o}^t &= \delta o^t \odot o^t \odot (1 - o^t) \\
\delta z^t &= \left[\delta \hat{a}^t, \delta \hat{i}^t, \delta \hat{f}^t, \delta \hat{o}^t\right]^T
\end{aligned}
$$

After the above gradients, we can now define the gradients to weights.

$$
\begin{aligned}
\delta I^t &= W^T \times \delta z^t \\
\text{As } I^t &= \begin{bmatrix} x^t \\ h^{t-1} \end{bmatrix}, \\
\delta h^{t-1} & \text{ can be retrieved from } \delta I^t \\
\delta W^t &= \delta z^t \times (I^t)^T
\end{aligned}
$$

Finally, we have

$$\delta W = \sum_{t=1}^{T} \delta W^t$$

Besides the basic definition, we also added the attention mechanism. The attention mechanism is essentially adding one extra input to the LSTM at each time step. we denote this extra input as c^i at decoding time step i. It is defined as:

$$
\begin{aligned}
c^i &= \sum_{j=1}^{T} {\alpha_i}^j h^j \\
{\alpha_i}^j &= \frac{\exp a(s^{i-1}, h^j)}{\sum_{k=1}^{T} \exp a(s^{i-1}, h^k)}
\end{aligned}
$$

Here, $a(s^{i-1}, h^j)$ is often defined as the dot product of two vectors. Now the input is the concatenation of x^t, h^{t-1} and c^t.

Data Structure and Key Operations

To organize and structure our network, we broke it down into several reusable components. We utilized a matrix data structure that allowed us to convert the matrix data from host to device memory (and the other way around) using only a method call. We also created data structures for each type of layer in the neural network. Each of these layers can take in an input and compute the forward and backpropagation results. We also create class for each layer so that we can access their weights and attributes easily and make our code more organized.

Within our matrix data structure, we have defined methods for every matrix operation we need within our layers. These operations include multiplication, transpose, add/subtract, concatenate, split, and element-wise multiplication.

The methods defined for each type of layer contain the bulk of our algorithmic complexity. Each layer can take in some input (usually a matrix or a vector) and do the forward and backward computation. The most computationally expensive parts are the matrix operations, especially for matrix multiplication, add/subtract, element-wise multiplication, etc. Fortunately, these operations can be easily paralleled by GPUs, as there are few dependencies within them.

There are several dependencies in the model. First is that we cannot parallel between time steps or layers since they depend on the previous results. We also cannot parallel between forward and backward computation since they rely on each other. Lastly, we cannot parallel some computations defined in LSTM as they depend on the previous results. We can parallel most of matrix operations and there is locality in many matrix operations. Matrix multiplication can be sped up by SIMD execution.

Approach

We used CUDA to implement our neural network, targeting deployment on the GHC machines. We decided on CUDA because we initially thought that many of the operations done within our implementation could be tremendously sped up with a large number of processors, which we found to be true.

The code we started modifying was found on github (zhfxl/CUDA-CNN). This CNN implementation contained the matrix data type as well as structural blueprints for each of our layers. The matrix class originally contained all of the code needed to store and transfer data from Host and Device memory. It also had methods for matrix multiplication and multiplication with transpose.

To this class definition we added all of our additional matrix and layer operations. We go into the implementation of each of these operations in detail below. Most of these operations are executed in parallel in different threads and warps on GPU.

Foward and Backward computation using Matrix Multiplication

As we can see from our model definition, most of the forward computation can be done using matrix multiplication. We initially tried to write a naive kernel implementation, but then we found that we can use cuBLAS to do the computation for us. For the backward computation, at first we thought we could not form them as matrix multiplication problems and that we needed to write additional CUDA kernels and copy data around. Then we figured out that the backward computation can actually be done using matrix transpose and matrix multiplication. As such, we also utilized cuBLAS to speedup the model. Specifically, we use the *cublasSgemm* for the multiplication. Matrix transpose is introduced in the next subsection.

Matrix Transpose

Originally we decided to use the *cublasSgeam* function from the cuBLAS library to perform our transpose. Normally this function takes two matrices A and B, operates on them based on some additional arguments, and sums them to some matrix C. However, you can specify some specific parameters to instead have the function compute the transpose of A and store that to the output matrix C. In order to utilize this functionality, we found that we had to *cudaMalloc* some temporary array to store the results of our transpose (we could not use *cublasSgeam* to transpose in place).

After timing our code, we realized that *cudaMalloc* was taking a large amount of time, outweighing the speedup we got by using the *cublasSgeam* function. As a result of this, we modified our tranpose function to instead utilize CUDA kernels. In each kernel function our thread would be responsible for swapping elements of our matrix $A[i][j] = A[j][i]$. Although the cuBLAS function itself is faster than this, we could not find a way to use the function without *cudaMalloc*, so we settled for the kernel implementation, which was still much faster than any sequential implementation.

Matrix Add/Subtract

This operation is very common in our model since we need to update our weights and add/subtract the gradients in each iteration many times. For this function we also used *cublasSgeam*. In addition to the matrices A, B and C, the function also accepts some floats α, β, such that $\alpha A + \beta B = C$. By specifying $\alpha = 1$, we could pass in $\beta = 1$ or $\beta = -1$ to the function to add or subtract, respectively.

LSTM computation using Matrix Elementwise Multiplication

This operation is a frequent one in LSTM definition, both forward and backward. For this operation we could not find any relevant cuBLAS functions, so we decided to implement the function using CUDA kernels. Each thread would be responsible for computing an entry in the resulting matrix, such that $A[i][j] \cdot B[i][j] = C[i][j]$.

Matrix Concatenation and Split

These two operations are used to combine and separate inputs and gradients for the LSTM. The concatenation stacks two matrices, A and B on top of each other to form a new matrix, C. If A has n rows and m columns and B has n' rows and

m columns, C has $n + n'$ rows and m columns. The function implementation is just a series of two *cudaMemcpy* calls that place A and B correctly into C. Th split will take an input matrix C and split it into two matrices, A, B, such that if we used our Matrix Concatenation function on A and B, we would get C. This function is also implemented with a series of two *cudaMemcpy* calls.

Fusing Activation and Bias Kernels for Fully Connected Layers

As part of the FullyConnected layer of the neural network, we must apply some activation function to each element in our output matrix after forward computation. We also have some bias vector, which we must apply to each column of our output matrix. We were able to combine these operations into a single kernel function. Each thread is responsible for applying the activation function to a single element of the output matrix, and then adding the corresponding bias term to this element. This reduces the overhead of launching two different kernels and give us some speedup.

We also use a similar kernel function in FullyConnected backpropagation. We need access to the derivative of each element with respect to this activation function, and we can get each in parallel using our kernel call.

Implementing Attention Mechanism

For the attention mechanism, it actually requires a batched vector dot product, but we could not find good ways to implement it, and simply doing one vector dot product at a time is too time-consuming. cuBLAS only provided batched matrix multiplication and was not suitable for our task. After a deep analysis, we found that it can actually be decomposed into element-wise multiplication with a kernel computation and matrix addition. This is because for batched vector dot product, it is basically element-wise product with an addition. Therefore, we can do the element-wise multiplication first and compute the sum later in a kernel. We can also directly modify the matrix in that kernel without doing it separately.

Results

We measured the performance of our implementation via run time. A neural network should aim to be as efficient as possible, so we strived for the same with our code. In general, we tested our code with varying sizes of matrices in order to see how much time particular sections would take. Since neural networks must be trained on thousands and thousands of data points, correlating to very large matrices, we felt that this was the most direct way to test whether or not our code was as efficient as we wanted it to be.

We tested to see if changing the grid dimensions for each kernel function changed our run time, but it did not. As a result, we decided to keep a standard grid size of 16×16 blocks per grid. Each block was then broken up into just enough threads so that each thread could perform a computation for an individual element in our matrix. For any kernel function aforementioned it should be assumed that this standard holds.

All of the matrices we generated contained float values ranging from -1 to 1, as initializing the matrices to 0 would not give us accurate run time results.

Below we will discuss various sections of our neural network and the results we have to support why we decided on a particular implementation. We also investigated the importance of the problem size in the `Varying Batch Size` section.

Matrix Multiplication

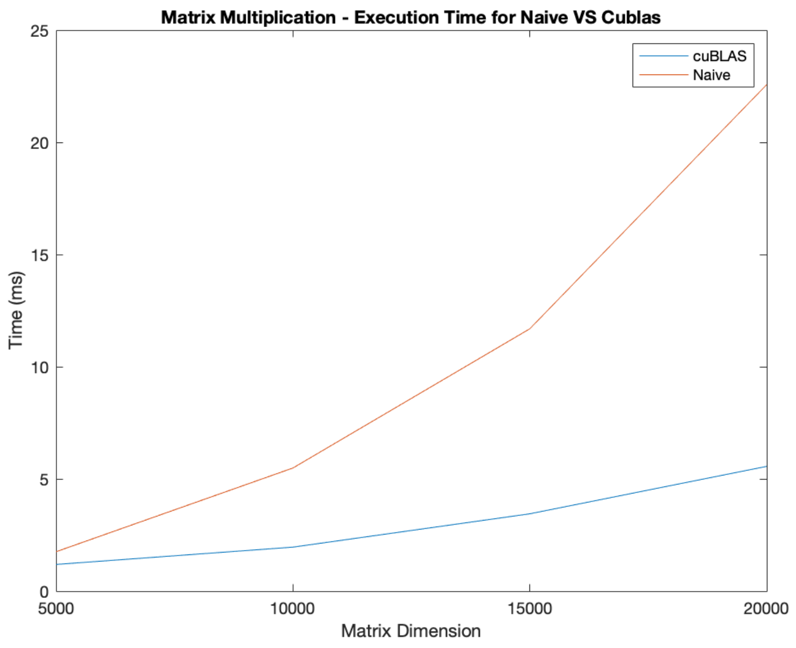

Figure 2: Execution times for Matrix Multiplication

	Matrix Dimension			
Version	5000	10000	15000	20000
cuBLAS	1.19 ms	1.961 ms	3.446 ms	5.559 ms
Naive	1.756 ms	5.487 ms	11.691 ms	22.57 ms

From Figure 2, we can see that the cuBLAS implementation fared much better than the naive implementation, especially with larger matrices. The naive implementation was a kernel function call that computed one element of the resulting matrix per thread.

Matrix Transpose

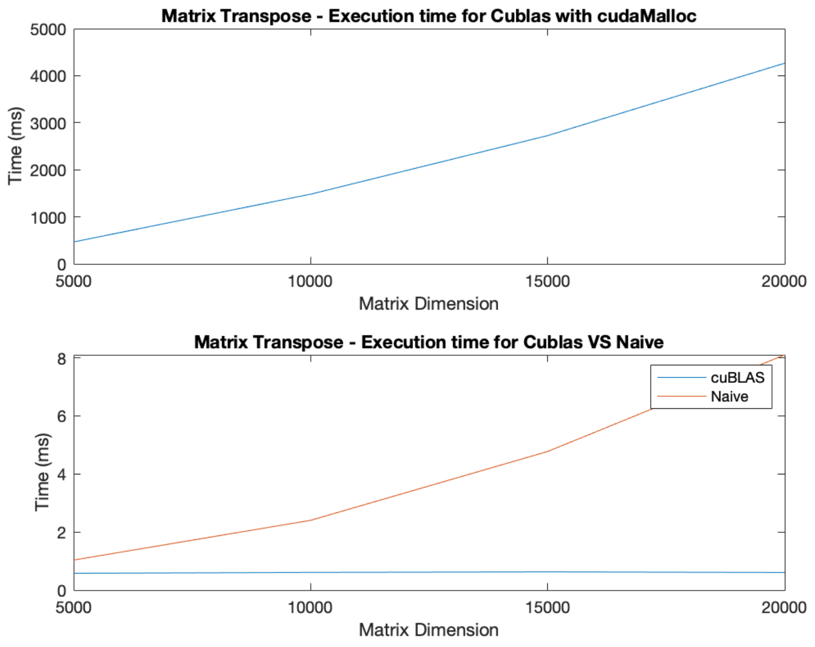

Figure 3: Execution times for Matrix Transpose

	Matrix Dimension			
Version	5000	10000	15000	20000
cuBLAS + cudaMalloc	466.515 ms	1480.878 ms	2726.227 ms	4264.585 ms
cuBLAS	0.578 ms	0.61 ms	0.628 ms	0.607 ms
Naive	1.034 ms	2.403 ms	4.774 ms	8.102 ms

From Figure 3, we can see the huge difference in run time when we include the *cudaMalloc* call, and when we leave it out. Even though the cuBLAS function call itself is much faster than the naive implementation, we cannot use the cuBLAS call without *cudaMalloc*, so we were forced to use the naive implementation (the naive implementation is a kernel function where each thread swaps corresponding indices by the definition of matrix transpose).

The reason that *cudaMalloc* takes so much time is because we are allocating memory for such a huge matrix. If we were strictly dealing with smaller matrices, we may have found favorable results for using cuBLAS. Since we want our neural network to efficient for very large matrices, we have to use the naive implementation.

Matrix Addition/Subtraction

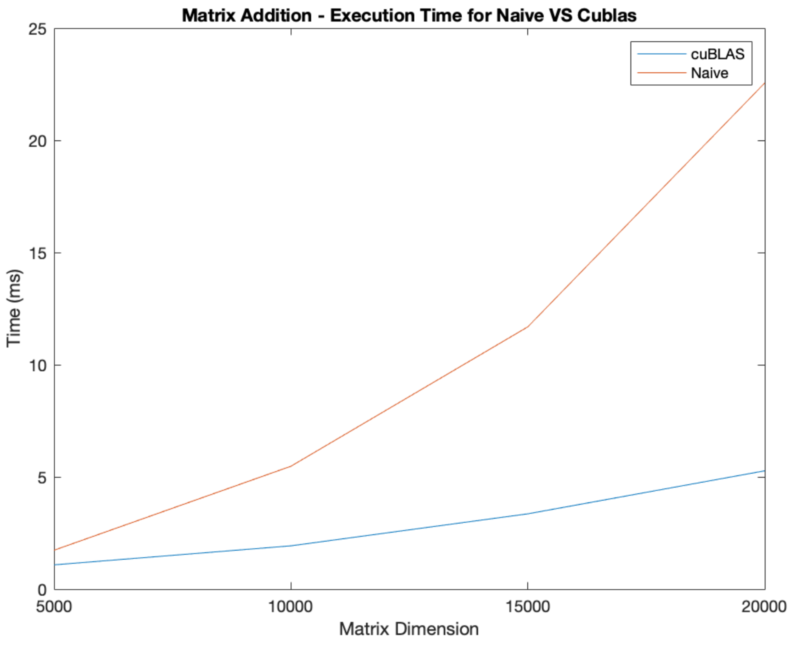

Figure 4: Execution times for Matrix Addition/Subtraction

	Matrix Dimension			
Version	5000	10000	15000	20000
cuBLAS	1.082 ms	1.93 ms	3.356 ms	5.272 ms
Naive	1.734 ms	5.477 ms	11.688 ms	22.552 ms

Figure 4 shows us that the cuBLAS implementation of matrix addition is faster than the naive implementation. The naive implementation is a kernel function where each thread computes the sum of a specific index in the resulting matrix.

Varying Batch Size

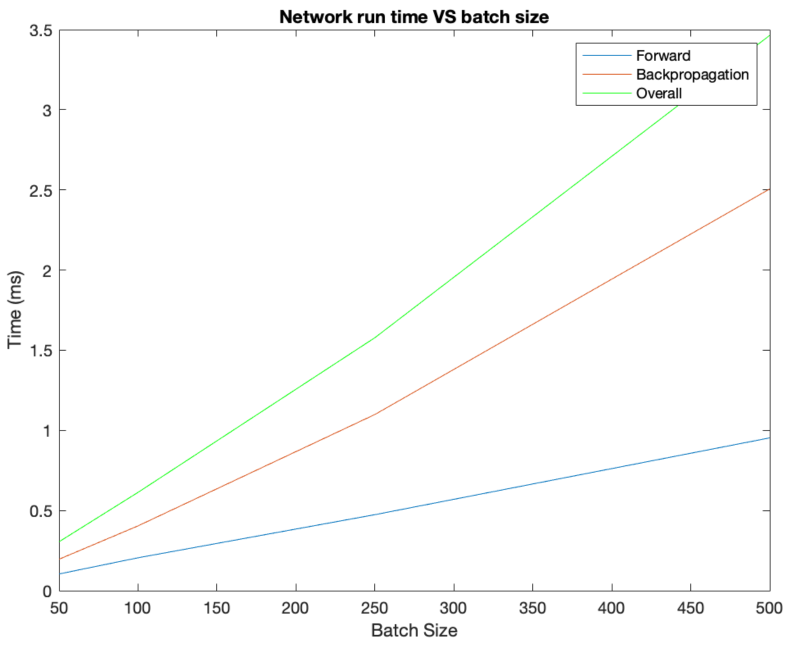

Figure 5: Run time with varying Batch Sizes

	Batch Size			
Computation	5000	10000	15000	20000
Forward	.104 ms	.205 ms	.474 ms	.953 ms
Backward	.196 ms	.404 ms	1.099 ms	2.506 ms
Overall	.305 ms	.613 ms	1.578 ms	3.464 ms

Figure 5 does not necessarily show any optimizations, we simply wanted to see how our implementation would scale given an increased batch size. In this context, Batch Size represents the number of input matrices our LSTM layer computed. Our results show that the runtime scales linearly with Batch Size, which is what we expected. This means that we could feed in as much data as possible to our LSTM without fear of losing any performance.

Our speedup was limited by the inherent dependencies of the Encoder-Decoder neural network. There are a number of dependencies within the network. The first of which is between the forward and backpropagation steps. In order to run backpropagation, we need to have the results from forward propagation.

We have similar dependencies between layers. Our network is made from a sequence of layers, each feeding their output into the next as input. Without the output of the previous layer, the current layer has nothing to compute. This is another dependency which we really cannot do anything about. As such, we were forced to look for parallelism within each of the layers.

Even within the layers we have some dependencies. There are several computations that rely on the results of other computations (most notably within the LSTM layer, computation of the cell state c^t, seen on page 3).

Our performance is also reduced by the number of memory operations we have to perform. We are constantly allocating memory for various matrices and accessing individual entries in these matrices, and most of them are global memory access. The performance lost to memory accesses only increases as we scale the size of our inputs. There are likely things we could do to improve the number or the manner of our accesses, but even an optimal implementation will still be less than ideal.

References

zhxfl, CUDA-CNN, (2017), GitHub repository,
https://github.com/zhxfl/CUDA-CNN

Arun Mallya, LSTM, GitHub repository,
http://arunmallya.github.io/writeups/nn/lstm/index.html/

Distribution of Credit

We believe that credit should be distributed evenly for this project (50-50). We worked together simultaneously on most parts of the project, but we've listed some individual tasks below.

Xiangkai	**Matthew**
Matrix Element-wise Multiplication	Matrix Concat/Split
Encoder-Decoder Layer with Attention Mechanism	Run time Analysis