Parallelizing Recurrent Neural Networks Using Scan

15-418 Final Project

Daniel Wen, Hima Tammineedi

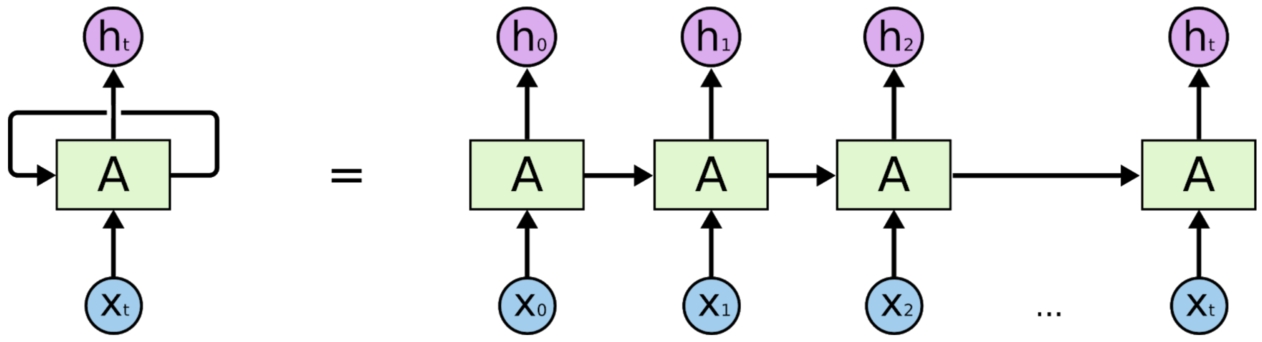
Summary

Fully Connected (FC) and Convolutional Neural Networks (CNNs) are easily parallelized neural networks. Recurrent Neural Networks (RNNs) are also widely popular and excel in sequence tasks such as in natural language processing. However, RNNs are inherently sequential as each step in the RNN computation depends on the previous time step in the sequence. In this project, we implement and evaluate a method to parallelize RNNs over sequence length using the scan algorithm. We improve on a CUDA kernel from previous work to achieve almost 11x speedup in TensorFlow code and 50x speedup in kernel execution time over a serial implementation.

Background

Recurrent Neural Networks offer a powerful means of adding “memory” to conventional neural networks. Normal feedforward and convolutional neural networks are stateless. So they remember nothing between multiple inputs to the network. RNNs, on the other hand, maintain a state for each node in the network which actually gets saved every update. This power is what makes these networks useful in natural language processing, because sentences are composed of words and in order to create a representation for an entire sentence, you need to pass each word through the network one by one and combine all of the word representations.

However, this power also adds a problem because now, the easily parallelizable feedforward and convolutional networks now become non parallelizable. The computation at each time step depends on the previous time step which necessitates sequential execution.



An RNN is presented in this diagram. The arrow going from A back to A on the left-side shows the recurrent connection present in all RNNs. The right-side is the unrolled version of an RNN cell over multiple time steps. Note that the number of time steps is equal to the number of inputs in the sequence. We can clearly see the sequential dependence.

This is why RNNs have always been considered slower than feedforward and convolutional networks. In the recent paper by Martin & Cundy [1], the authors show that it is indeed possible to parallelize RNNs.

To understand their method, a look at the RNN equations is necessary. In the equations, there is a nonlinear function applied to the output of each time step. This means that each time step depends non-linearly on the previous time step, which is how RNNs gain power in their ability to represent arbitrary functions. Just using linear functions would not allow the network to learn much since a composition of linear functions is just a single linear function, which is not very good for representing arbitrary functions.

The paper authors show that by removing the nonlinear dependence between time steps, the RNN equations turn into a linear recurrence rather than a non-linear recurrence. This transformation is what allows the RNN computation to be parallelized.

At each time step *t* in the sequence, a typical RNN computes a nonlinear function of the previous step. For example, in an LSTM, which is the most popular type of RNN, we have

$$\begin{aligned}
f_t, i_t, o_t &= \sigma(U_{f,i,o}h_{t-1} + V_{f,i,o}x_t + b_{f,i,o}) \\
z_t &= \tau(U_z h_{t-1} + V_z x_t + b_z) \\
c_t &= f_t \odot c_{t-1} + i_t \odot z_t \\
h_t &= o_t \odot c_t
\end{aligned}$$

where $\odot$ denotes elementwise multiplication. f_t, i_t, o_t are just functions that are computed in order to improve the expressiveness of the LSTM. The important equation from above, is the one containing h_t. The output h_t depends on nonlinear functions σ, τ of h_{t-1}. To allow for more parallelism, we use linear functions instead and introduce a second output $\tilde{h}_t$ which always depends linearly on its previous timestep [1]:

$$\begin{aligned}
g_t &= \sigma(V_g x_t + b_g) \\
j_t &= \tau(V_j x_t + b_j) \\
\tilde{h}_t &= g_t \odot \tilde{h}_{t-1} + (1 - g_t) \odot j_t \\
f_t, i_t, o_t &= \sigma(U_{f,i,o}\tilde{h}_{t-1} + V_{f,i,o}x_t + b_{f,i,o}) \\
z_t &= \tau(U_z \tilde{h}_{t-1} + V_z x_t + b_z) \\
c_t &= f_t \odot c_{t-1} + i_t \odot z_t \\
h_t &= o_t \odot c_t
\end{aligned}$$

Now, we see that $\tilde{h}_t$ depends linearly on $\tilde{h}_{t-1}$. Note that the original output h_t still involves a nonlinearity, which helps the model retain its expressive power. The new output $\tilde{h}_t$ is the one that is used within a sequence and is the input for the next time step of this cell. But, we continue to use the original output h_t to now be the input to the next *layer*. Neural networks are composed of layers of nodes, and so even though we remove the nonlinearities between time steps in a single node, we continue to keep the nonlinearities between layers in order to preserve the power of the network.

We can simplify and rewrite the linear recurrence in an alternate form as

$$\tilde{h}_t = \lambda_t \odot \tilde{h}_{t-1} + x_t$$

According to Blelloch [2], we can compute all $\tilde{h}_t$ in parallel by applying the scan algorithm on the sequence

$$S = ((\lambda_1, x_1), \cdots, (\lambda_T, x_T))$$

using combining function

$$f((a, b), (c, d)) = (a \odot c, b \odot c + d)$$

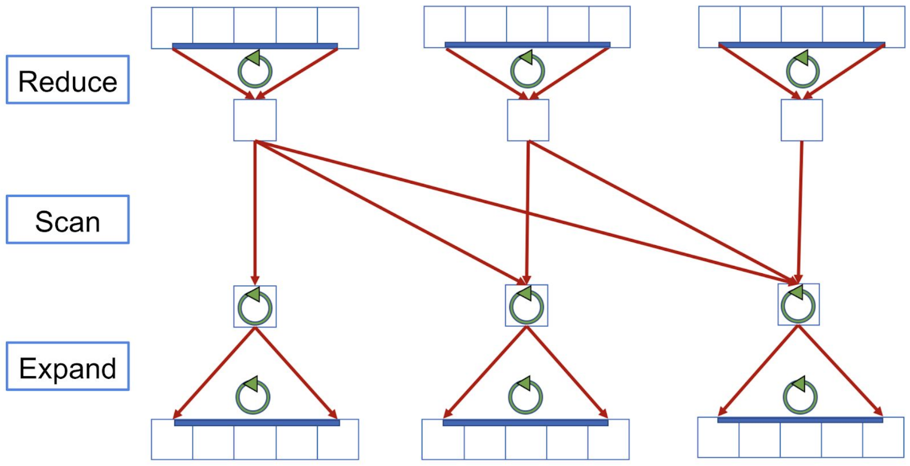
and identity value $(1,\ 0)$. The output of this scan is a sequence of tuples, and the second values of these tuples are the desired $\tilde{h}_t$.

Approach

We decided to implement our approach as a CUDA kernel. The original implementation by the paper authors was also a CUDA kernel so this gives us a good baseline to compare against.

We also had a wrapper around the kernel that allows us to call our CUDA kernel from higher level Python code via the TensorFlow neural network framework. Our CUDA kernel implements the linear recurrence operation in parallel, and then in Python/TensorFlow code, we can define the actual RNNs computing the equations presented earlier by calling the kernels to perform the recurrence computations.

We first present a simplified diagram showing our parallelized approach using scan.



To give a high level idea of our implementation, the sequence is divided into chunks, which are assigned to the parallel “workers” (which we will describe in more detail). First, each worker performs a sequential reduce on its assigned segment. Then, we scan these reduced values to get offsets for each reduced chunk. Finally, each worker sequentially produces its section of the output using the appropriate offset from the reduced chunks.

We now give a more detailed explanation of our algorithm. In order to do this, we first describe how the abstract workers map to the hardware.

Our experiments were conducted on a single GK210 chip in a Nvidia Tesla K80. This chip has 13 Streaming Multiprocessors (SMs) and each SM can simultaneously execute 2 thread blocks. We configure each thread block to have 16 warps.

Our CUDA code is structured in 3 kernels that correspond to the steps described previously: reduce, scan, and expand. The reduce kernel occurs in two phases. In the first phase, each warp reduces its section of the input. In the second phase, each block reduces across the reduced results of its warps. In the scan kernel, we scan the reductions of the blocks to get offsets for each block. The expand kernel occurs in two phases that mirror the reduce kernel’s phases. In the first phase, each block scans over the intermediate values from the first phase of the reduce, using the previous block’s offset, producing offsets for each warp. In the final phase, each warp uses the previous warp’s offset to scan over its section of the input.

To implement the scan kernel in parallel, we modified the inline shared memory scan given in 15-418 Assignment 2 to support our custom sequence data type and custom combining function. We also reduced the amount of shared memory that the scan needed.

To further improve performance, we reduced global memory accesses and tried using shared memory. We found that we originally had a lot of global memory reads/writes (as measured by looking at the bandwidth computed during runtime), so we figured that using shared memory would help.

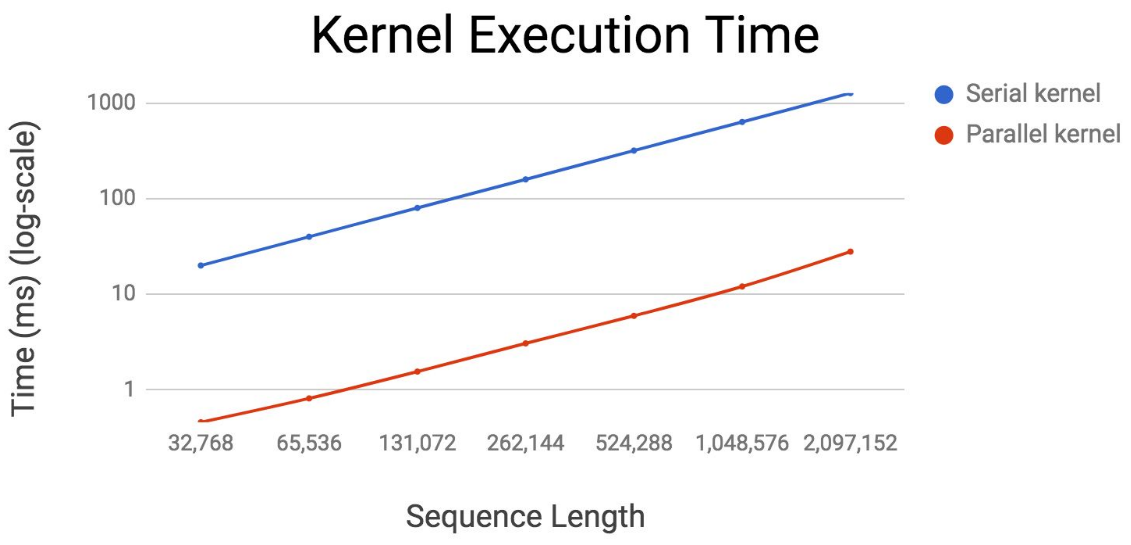
We also improved branch prediction within loops by removing divergent execution statements, and ran profiling to optimize the number of threads.

We used TensorFlow bindings to use our kernel in RNNs constructed in Python code.

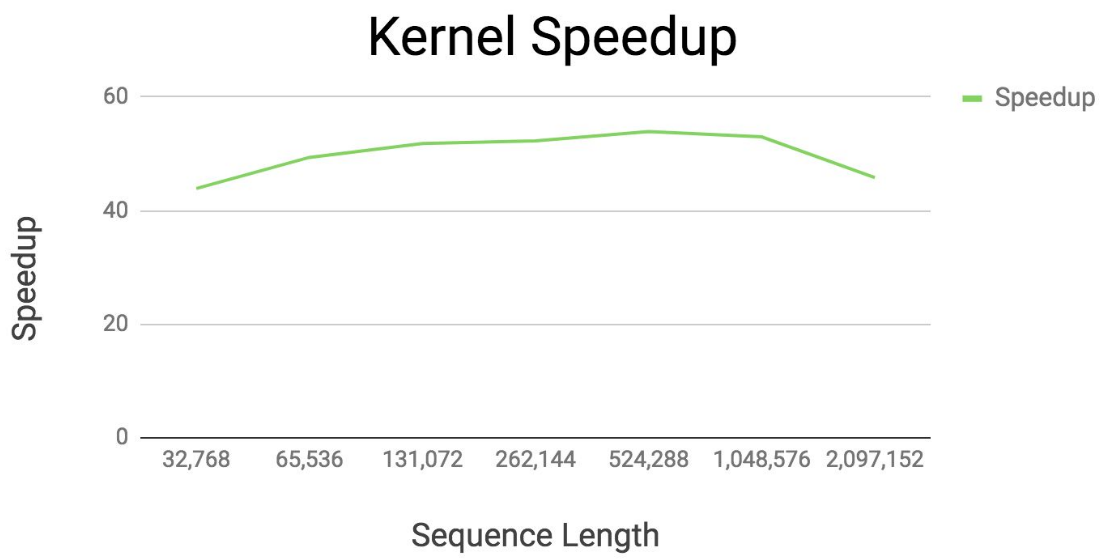
Results

We first show speedup results of our implementation.

The graph below shows the execution of our implemented parallel CUDA kernel versus a serial CUDA kernel using problem scaling. Note that this serial kernel actually does run in parallel as it is parallelized over the hidden dimensions of the inputs. On the other hand, our parallel kernel is also parallelized over the other dimension of the input, which is the sequence length (number of time steps). The hidden dimension size was fixed at 128 for the experiments, but it is a hyperparameter often tuned in machine learning practice.

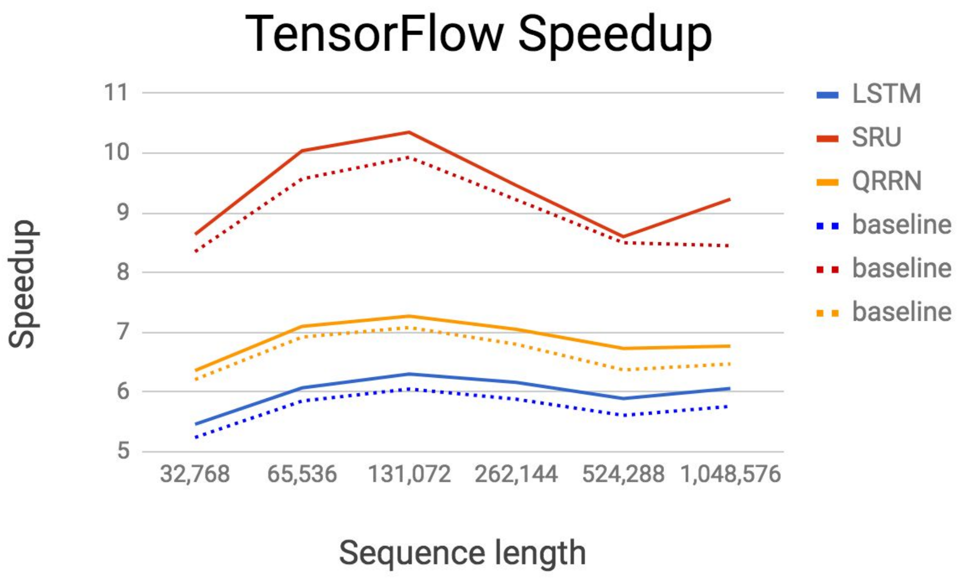


Note that the above is a log-log plot, so the straight lines mean that the parallel kernel maintains a constant speedup over the sequential as we scale the problem size.



The second graph above shows the parallel kernel's speedup over the serial kernel. We see that the speedup is around 50X and is fairly constant across many sequence lengths ranging from tens of thousands of time steps to millions. These great results show that the kernel performs really well across problem sizes. We omit results from smaller sequences as it doesn't make sense to invoke the overhead of the parallel kernel for short sequence lengths.

We also benchmarked the speedup of our kernel when used from Python/Tensorflow code that executes various types of RNNs, namely LSTM, SRU and QRNN.



The baselines (dotted lines) in the graph above are the original code we started with at the first checkpoint for this project. The solid lines show the performance of our final kernel at the end of the project. Every line is showing the speedup of the parallel kernel over the serial kernel when called from TensorFlow code.

LSTMs, SRUs, and QRNNs are all types of RNNs that we implemented in TensorFlow using our linear recurrence kernel.

Note that the speedups are a lot slower as compared to the raw kernel execution speedups because invoking TensorFlow code has a lot higher overhead due to all the memory movement and other computations that are occurring.

This graph shows really great results because, in practice, people will be using TensorFlow code to execute neural networks, so our results show that we can get large speedups for real world usage.

Furthermore, this speedup is achieved while maintaining machine learning accuracy. We ran a synthetic test on a traditional LSTM from the TensorFlow framework and a LSTM using the parallel kernel. Both models have a similar number of parameters. The test gives the model a very long random sequence, and the model must remember the first element and output it. This task is hard because as the number of timesteps grow longer, the LSTM must continue to maintain the input it originally saw.

We measure how long it takes (in wall time and number of training epochs) for the model to achieve 100% prediction accuracy. The results are shown in the table below.

Traditional LSTM	**LSTM using parallel kernel**
92 seconds (49 epochs)	56 seconds (45 epochs)

Another promising result of this work is that due to the highly improved efficiency of the kernel, we are able to train input sequences of never before seen sizes.

Most sequences used in natural language processing research only extend up to the low tens of thousands of time steps. This is due to the computational limitations of using sequential RNNs.

We, however, were able to perform the same computations on sequences that are millions of time steps long, which is unheard of.

Parallelism Limitations

We found that the scan step of the algorithm actually executed in the fastest amount of time. So making the scan faster through more parallelism actually did not improve speedups much more. The most amount of time taken was in the other two kernels as shown below.

On a sequence of length 524,288 with 128 hidden dimensions:

- Reduction kernel: 2.1230ms (35.8%)
- Scan kernel: 24.543us (0.4%)
- Expand kernel: 3.7850ms (63.8%)

Unfortunately, as described by Blelloch [2], in the optimal scan algorithm, each worker must perform sequential processing in the reduce and expand stages. Since we already implemented the optimal scan algorithm, it is not possible to improve execution time via changes to the algorithm. So we can only resort to CUDA implementation improvements.

We also found that in the reduction kernel, the first phase, in which the values inside each warp are reduced takes the most time, and likewise for the phase inside the warp for the expand kernel. This is as expected because the within-warp procedures operate on the most number of elements.

Another limitation that we ran into was that CUDA only supports synchronization between blocks on Pascal and newer architectures, and the K80 GPU has a Kepler architecture. Thus, we had to use three separate kernels in our implementation in order to make sure that all the blocks finished before going onto the next processing step. So this prevented us from using shared memory between the reduce and expand steps, which would have been beneficial because there is a lot of data reuse.

We also found that we were pushing the memory bandwidth of the GPU. For a sequence length of 65,536, the kernel achieved 104 GB/s bandwidth, which is 43% of the 240 GB/s maximum GPU bandwidth. As taught in class, one can never achieve the maximum theoretical memory bandwidth, and in practice, the best one can hope for is about 50%.

Luckily, we had very little divergence in our kernels. Additionally, we definitely see that our choice of implementing this algorithm on a GPU makes a lot of sense due to the large amount of parallel processing units that the GPU provides along with the high speed of computation which is necessary for machine learning workloads.

Conclusion

In conclusion, we have shown that the traditionally un-parallelizable RNNs can actually be parallelized. This leads to large speed gains with minimal loss in learning accuracy, which can really help the field of machine learning in iterating and developing new models much more quickly.

References

[1] Martin, C. & Cundy, C. (2018). Parallelizing Linear Recurrent Neural Nets Over Sequence Length

[2] Blelloch, G. E. (1997). Prefix Sums and Their Applications.

Division of Work

Equal work was performed by both project members.