15418: Final Project Report

Anupam Pokharel and Lisa Mishra

December 2021

1 Project Summary

The objective of this project was to discover opportunities for parallelism in the usage of Recurrent Neural Networks (RNNs) inference-time forward pass computations. We then exploited these opportunities to speed up the inference layer of RNNs by utilizing synchronization and atomic operations to mitigate the purely-sequential nature to which the data dependencies within and across neural network layers lend themselves. We focused on the speedup that could be obtained with two different methodologies for parralelizaiton: the first one was on a CPU using the OpenMP library, an API that support multi-threading, and the second was on a GPU using CUDA, a parallel computing platform and API.

2 Background

2.1 An Introduction to RNN

A recurrent neural network (RNN) is a neural network that is utilized specifically for processing a sequence of data (denoted as x(t)) over a series of time steps (denoted as t) over a period of time. RNNs essentially contain a "memory" of sorts to keep track of the data and what has been computed over time. The only input into a recurrent neural network is x(t), which contains all the values of x over series of time steps (it can be considered as a vector).

Another key feature of an RNN is the hidden state h(t) which represents the "memory" of the network, storing the hidden state at each time step t. It is calculated based on the current input and the previous time step's (t-1) hidden state.

A third feature of an RNN is the weights that are shared across each of the timesteps, denoted as U, V, and W. U parametrizes the input-to-hidden connections, W parametrizes the hidden-to-hidden connections, and V parametrizes the hidden-to-output connections

Finally, the output of the RNN is denoted as o(t), and is the result of the last layer of the RNN.

2.2 RNN Computation

There are three stages of recurrent neural network computation: the forward pass, the backward pass, and computing gradients. Our focus for this project is on the forward pass segment of recurrent neural network computation (we will describe later why we chose this segment of the computation to focus on). There are four sets of equations that we primarily focused on searching for opportunities for parallelization. The equations are shown below:

```
// x (t) is a vector of inputs over a series of time steps
a[t] = b + W * h[t-1] + U * x[t]
h[t] = tanh(a[t])
o[t] = c + V * h[t]
y[t] = softmax(o[t])
```

where the following variables correspond to layers that are vectors of neurons, indexed by time and ordered by increasing proximity to the output layer, y: a, h, o. RNN computation also consists of the backward pass and computing gradients. We saw, however, the most opportunity for parallelization in the forward pass segment of the RNN algorithm, as the forward pass computation consists of dependencies between time steps (the next time step's calculations depend on the results of the previous one's).

2.3 Parallelization of RNN inference

We decided to analyze the four equations listed above (for a[t], h[t], o[t], and y[t]) to determine where we could implement parallelism. We noticed, in particular, that the computation for a[t] involves the value of h at the previous time step, or h[t-1], creating a dependency at the next time step on the previous time step. This dependency makes the code much slower. It also means that we can't simply break up the work over each of the time steps between threads (i.e. assign each thread one time step) because each thread would rely on the previous thread's work, which is essentially the same as running the code sequentially.

The following figure demonstrates this dependency:

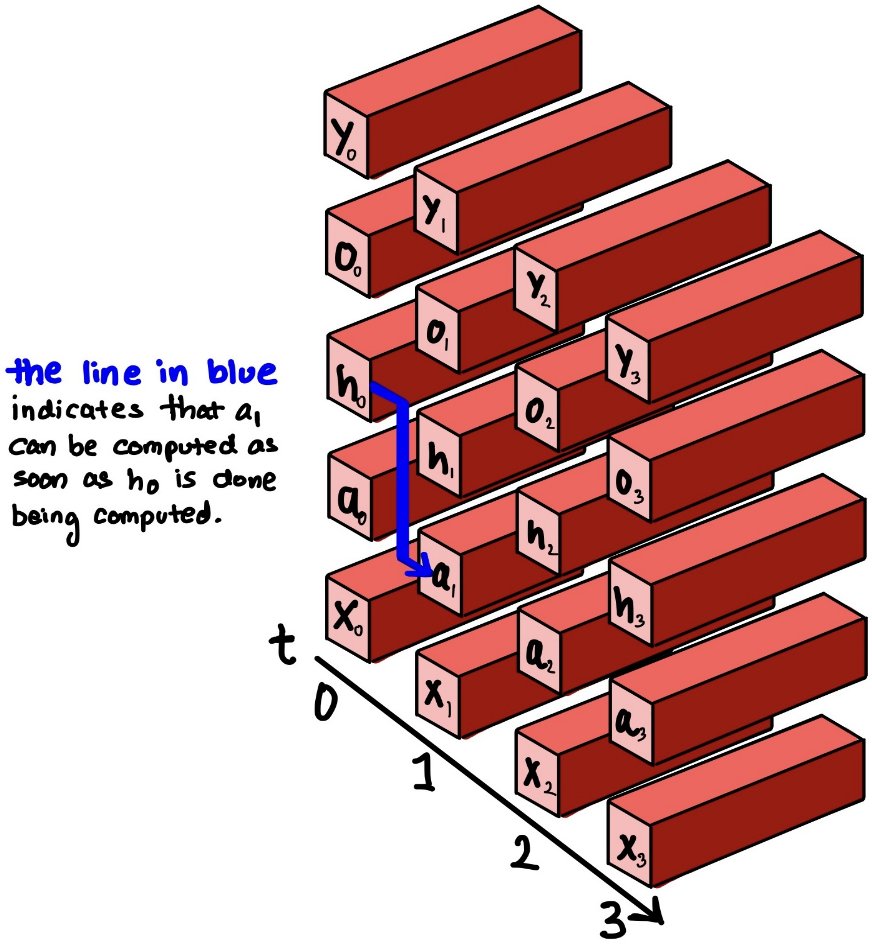


Figure 1

Note that each of x(t), a(t), h(t), o(t), and y(t) are each vectors at each time step, and are thus demonstrated as 3D blocks that are going into the page. At every value of t, there is a different value for each of the five variables. Also, take notice of the line in blue, that highlights the dependency that causes the sequential implementation of an RNN to be quite slow.

If different threads were responsible for the forward-pass computations of adjacent time steps, there would need to be communications of hidden-layer values that would inhibit speedup. Our implementation uses a different thread for the computation at each time step t. For example, thread 0 would be in charge of the computation of a_0, h_0, o_0, and y_0. thread 1 would be in charge of a_1, h_1, o_1, and y_0, and so on. The sequential aspect of this is that thread 1 must wait for h_0 to be computed before beginning to compute a_1. The way we chose to parallelize this is that as soon as thread 0 is done computing h_0, thread 1 immediately starts computing its one set of values.

A demonstration of this aspect of parallelism is shown below. In purple, we have the computation that can be done in parallel (between threads 0 and 1). This notion can carry over into the computations that the rest of the threads do.

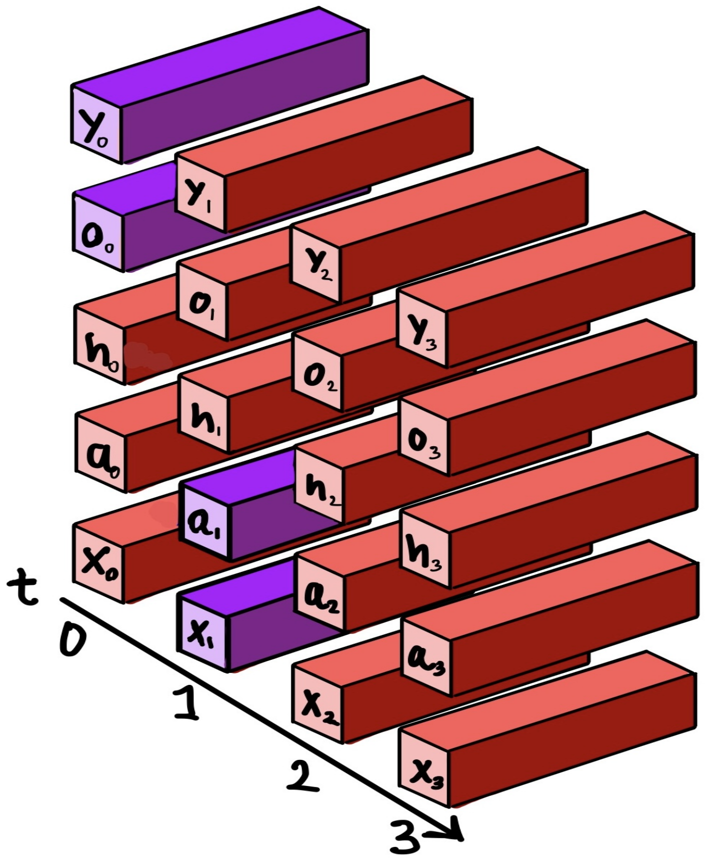


Figure 2

3 Our Approach: CUDA Implementation

3.1 CUDA Implementation: Overview

We began the coding aspect of our project with the CUDA implementation for a GPU-based acceleration. We ran this code on a machine that has a NVIDIA GPU (or is connected to one), like the ghc59 machine that was accessible to us through AFS. We implemented a sequential version on the GPU as a benchmark with which we compared speedups. We also implemented a version that utilized the parallelization scheme explained in the Section 2.3. To compare speedups, we ran both of these implementations and measured the time of each computation, and also verified the correctness by ensuring that the results are the same and correct as well.

3.2 CUDA Implementation: The Code

We will begin by going over the different core functions that we wrote in our CUDA implementation. Each component will be represented with a header of the name of the function and then an explanation of what the code does.

`forwardPassSequential`

We began with implementing a sequential version on the GPU so that we can compare the parallelized program results to that of the sequential program. Upon viewing the code, you will notice a function called forwardPassSequential, which is the sequential implementation of the forward pass component. We utilized CUDA programming to implement this, and ensured that it was sequential by specifying that the kernel (called kernelComputeForward in the code) that is called within the function can only use one block with one thread.

`forwardPass`

The parallel implementation of the forward pass segment of recurrent neural networks is encapsulated by the function called forwardPass, which calls the kernel function kernelComputeForward with 1 block and a number of threads that is equal to the number of timesteps. This ensures that each thread takes care of the computation for each existing timestep.

`kernelComputeForward`

There are two key components to our kernel function in our CUDA implementation. The first component is the basic implementation of the forward pass algorithm for recurrent neural networks. This involves the computation for the equations that we discussed earlier in Section 2.2, but are also shown below:

```
a[t] = b + W * h[t-1] + U * x[t]
h[t] = tanh(a[t])
o[t] = c + V * h[t]
y[t] = softmax(o[t])
```

The second aspect of our kernel function is a spin loop at the beginning. This loop ensures that for each thread that the kernel launches, the thread waits until the previous thread's computation for h is complete so that the current thread can compute a for the current time step (revisit figure 2 in Section 2.3 for a detailed diagram of this dependency). Therefore, each thread that the kernel launches will wait for the necessary computation from the previous thread in the spin loop. The thread will know when the previous thread's computation is complete by accessing a value in shared memory - this value will stay at -1 until the previous thread replaces that value with the correct value for h at that timestep. We will explain in the next section, Section 3.3, what this value was and how we different threads accessed it.

3.3 CUDA Implementation: Utilizing Shared Memory

We had originally written an implementation that involved using global memory to store all the values of h at each time step. However, this resulted in undefined behavior. For example, if thread 0 were to update update the value for h at time step 0, and at the same time thread 1 were to attempt to read it, it would cause a race condition that prevented the RNN computation from proceeding to the next part of the computations. Both threads would be deadlocked, and the rest of the threads would be sitting and waiting at their individual spin loop waiting for the previous time step's value for h[0] to no longer equal to -1.

Due to this issue, we realized that we needed to implement shared memory instead. Another benefit to shared memory is that it is simply much faster than global memory. We used a value that is denoted as h_shared in the code that is of size hsize * timesteps. This is because hsize is the size of the entire vector h at a particular timestep, and to encapsulate all the values of h at all the timesteps, we needed to multiply hsize with timestep (remember that a key component of RNNs, as discussed in Section 2, is that it contains memory).

We then set the value at index zero of h[t] at each timestep equal to -1. Each thread looks at the zeroth index of the previous thread's/time step's value of h to check whether that entire vector h is ready or not, in order to be used for the computation of a[t] for the current thread. This "checking" aspect is, as mentioned in the prior section, section 3.2, done in the spin loop in the kernel function.

The diagram below is a demonstration of how the threads use shared memory to check whether the prior thread is done computing it's value of h[t]:

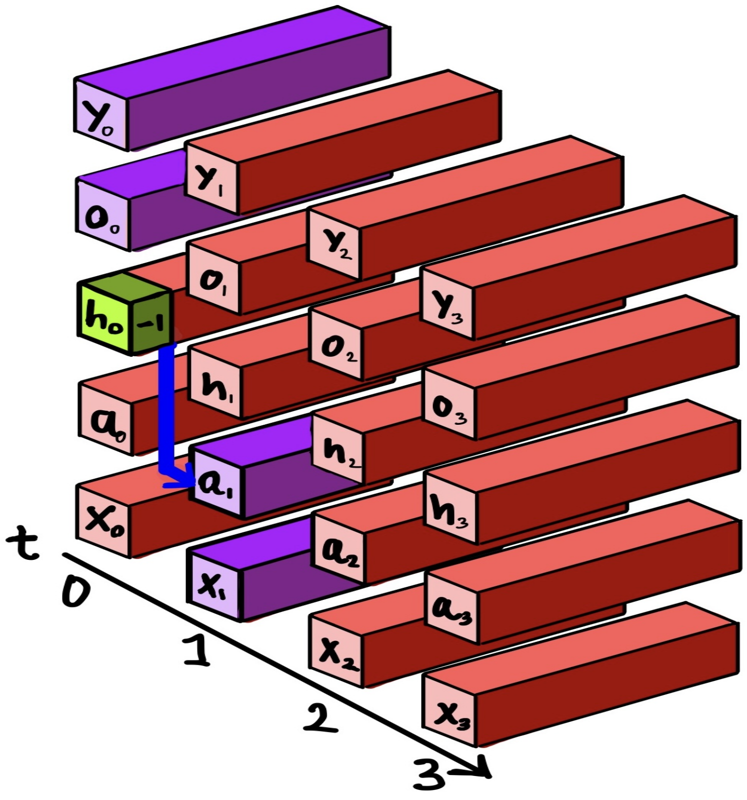


Figure 3

3.4 CUDA Implementation: Using atomicCAS and atomicExch

`atomicCAS`

The general idea is that atomicCAS is used by threads to check whether the prior thread's vaule for h[t] is equal to -1 or not. The thread checks whether the value is -1, and guesses -1 as the value. If this guess is correct (so, if the value is still -1), then the thread "swaps" the value with -1 (which is essentially equivalent to not changing the value there at all). If the value is not -1, then the value that exists there is returned. The reason we used atomicCAS was to ensure atomic reads, to avoid the race conditions that we were getting when simply using global memory without any atomic operations for the reads and writes. The spin loop that was used in the code, with the accompanying atomicCAS (we used a slightly modified version called atomicCAS_f32 that is able to handle floats), is shown below for reference in order to better understand how it was used:

```
while ( atomicCAS_f32(&h_shared[index * (HSIZE - 1)], -1.f, -1.f) == -1 ) ;
```

`atomicExch`

In order to write to its value of h, the thread uses atomicExch. We used this function in order to be able to do atomic writes, so as to prevent a race condition caused by reading and writing to the same place in memory. This is done after the thread knows that previous thread's computation for h is complete, thereby allowing it to change its own value of a and consequently its own value for h (note that atomicExch is only used for updating the value of h, since that is the only piece of information that is needed between threads).

4 Our Approach: OpenMP Implementation

4.1 OpenMP Implementation: Overview

As one would expect, the underlying logic between the two implementations is very similar, as is the guiding principle for parallelization (i.e. we are seeking a way to incur the least synchronization overhead among concurrently-executing computations). The implementation mechanics, however, differ, as the execution environment is a CPU with paralellism facilitated by OpenMP. So, the code format and syntax is adapted from the kernel function, in the following way: we composed a parallelized for loop that protects against data overwrites with some tools provided by OpenMP (an analogous functionality to that provided by the combination of `atomicCAS` and `atomicExch`). More detail on the way we accomplished this is provided in the below section.

4.2 OpenMP Implementation: The Code

The most natural fashion of parallelization for the RNN forward pass is that across the time steps; to realize this intuition, the core logic of the RNN computations (going upward in the figures from Section 3) were housed within a parallel for loop annotated with the syntax `pragma omp parallel default(shared) schedule(static)`. We chose to have static scheduling because the workload of any particular parallelly-executing loop iteration is inconsequentially similar to any other iteration and therefore the benefits of dynamic scheduling would be middling.

Within the loop, a function `openmp_fwd_pass` was called, to outsource the tasks of allocating and freeing memory for intermediate results and, more importantly, carrying out synchronization. In order to replicate the functionality of waiting via spin-loop in our CUDA implementation, we had to create a binary vector `computation_status` that tracks the progress of the RNN's neuron output computations (0 for pending and 1 for complete). Since this vector is a shared memory to which no more than one iteration's execution should be writing at any particular time, writes to the vector were prepended with the annotation `pragma openmp atomic write`. Likewise, we added this annotation before writes to the global memory of the hidden state vectors (denoted h in all of our figures from the prior sections), since they critically dictate the overall computation progress and enforce correctness.

One analytic perk of this implementation is that, unlike in the CUDA implemetation, changing the problem parameters (vocabulary size, hidden size, and number timesteps) does not require changing global constants. This is because the setting of those parameters is not something that has to be affixed unalterably at compile-time and rather can have run-time variation. This is the typically tradeoff with GPU programming: the CPU alternative offers more tractability for runtime variation and more flexibility in coding logic, at the expense of speed. However, we did not observe the tradeoff upside, unfortunately. We explain this foreshadowing of disappointing GPU performance in our results section.

5 Results

Our results are shown below (the CUDA implementation is on the top, whereas the bottom graph is for the OpenMP implementation).

The problem types are defined as follows (we provide a brief definition of what each problem parameter means in the first list):

(1) Large

  (a) time-steps (in our code, this is `TIMESTEPS`), the number of units of time (typically seconds) that the data spans), was set to 15

  (b) hidden size ((in our code, this is `HSIZE`)), the number of features computed in the hidden state of the network (i.e. the dimensionality of the hidden vector h for each time step), was set to 50

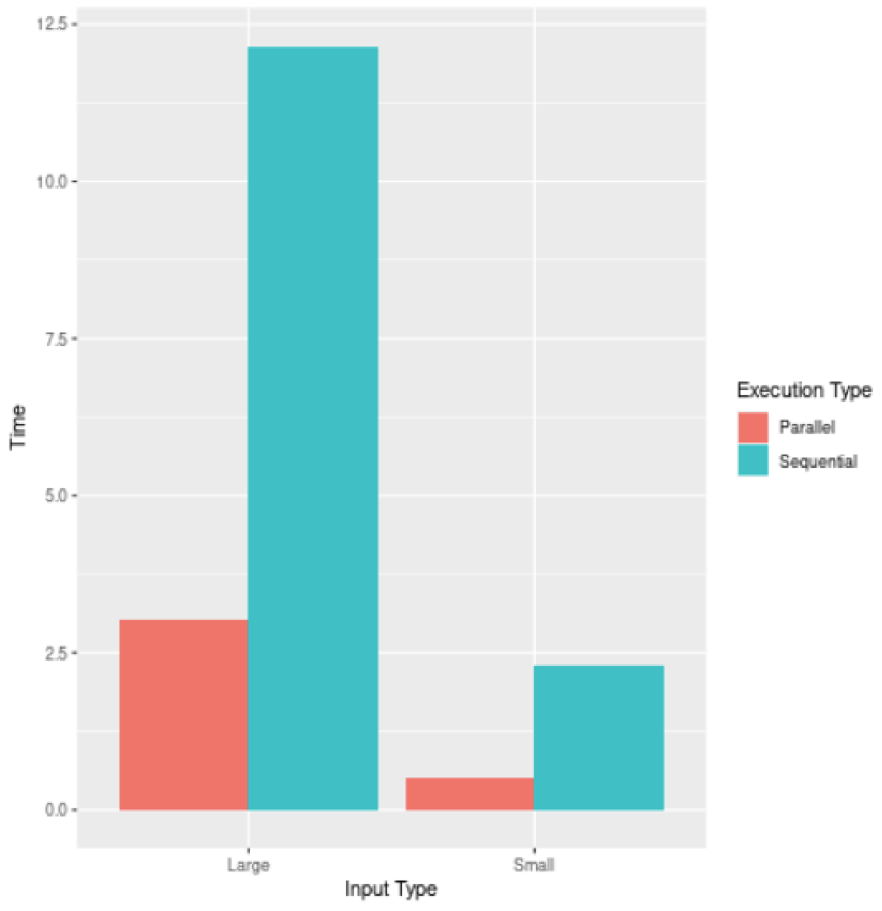
  (c) vocab size (in our code, this is `VSIZE`), the number of outputs over which the distribution is computed (and over which an argmax will yield the prediction), was set to 8000

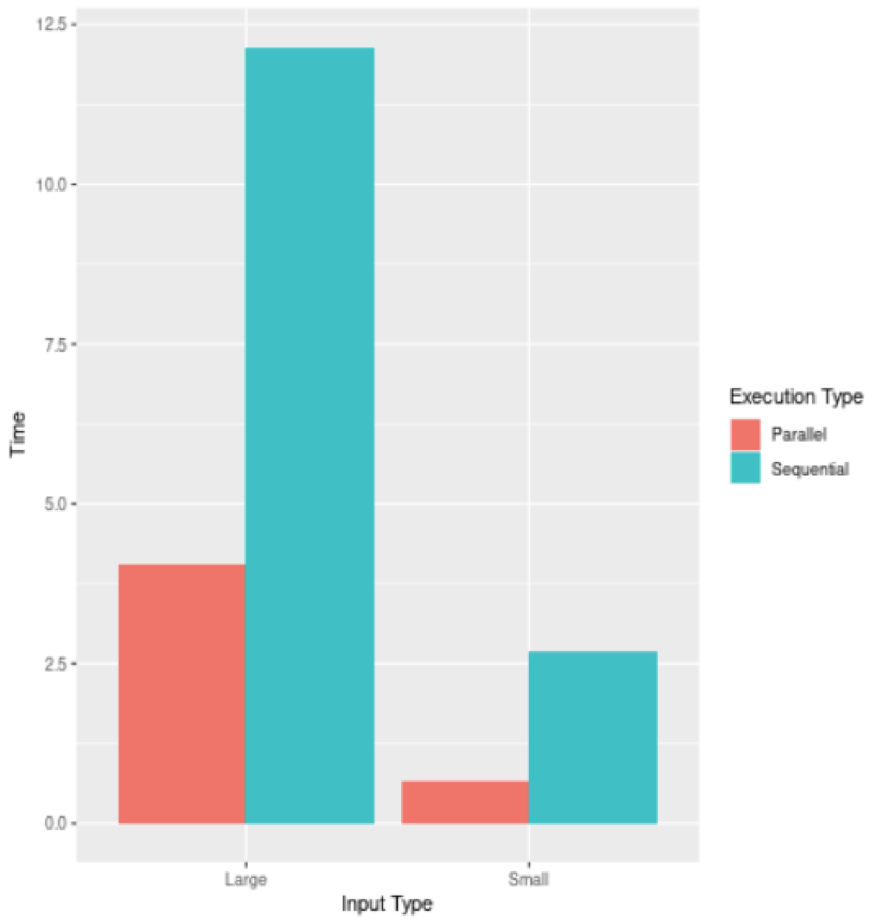
(2) Small

  (a) timesteps was set to 50

  (b) hidden size was set to 120

  (c) vocab size was set to 10000





The overall execution times for both implementations is roughly the same. We believe this is due to the setup overhead that the CUDA implementation required, causing it to slow down significantly. CUDA kernels are most performant when their code involves very little branching and batch operations that differ only marginally, if at all, in the instructions they necessitate. However, the nature of RNNs is inherently incompatible with this: there are different matrix multiplications and conditioning requirements that the logic of the forward pass of a RNN layer entails.

Regardless, we did observe spedup in the CUDA implementation, relative to its own sequential version. This suggests that the overhead was high but does not, fortuntaely, preclude realizing speedup when there are multiple threads. We do, however, anticipate that using more sophisticated techinques (like multiple blocks or grids of blocks) will not be very promising (indeed, we dabbled with this ourselves).

6 Work Done By Each Partner

The total work for the project was evenly distributed - each partner did about 50 percent of the work.

Both Anup and Lisa worked on the CUDA implementation, since it was much more work-intensive to do. Both partners discussed ways to circumvent the race condition issues that the implementation with global memory was facing (before swapping to a shared memory implementation). Anup focused on the OpenMP implementation, and Lisa worked on the report and developing some graphics to be used in both the report and the poster in the meantime. Once Anup completed the OpenMP implementation, he joined Lisa in working on the final report.

7 References

https://towardsdatascience.com/recurrent-neural-networks-rnns-3f06d7653a85

https://nvidia.github.io/libcudacxx/extended_api/synchronization_primitives/atomic/atomic_thread_fence.html

https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/

https://www.bu.edu/pasi/files/2011/07/Lecture31.pdf

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

http://supercomputingblog.com/cuda/cuda-tutorial-4-atomic-operations/

https://on-demand.gputechconf.com/gtc/2013/presentations/S3101-Atomic-Memory-Operations.pdf

https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels

https://datascience-enthusiast.com/DL/Building_a_Recurrent_Neural_Network-Step_by_Step_v1.html