15418: Final Project Report

Anupam Pokharel and Lisa Mishra

December 2021

1 Project Summary

The objective of this project was to discover opportunities for and examine the performance gains achievable through parallelized speedup of inference-time forward passes of Recurrent Neural Networks (RNNs). The mechanistic steps in realizing speedups took the form of 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 our inquiry on two computing contexts, which are representative of common industrial applications of models like RNNs: the first one was on a CPU using the OpenMP library, an API that supports multi-threading; and the second was on a GPU using CUDA, a parallel computing platform that contains tools mediating communication with NVIDIA GPUs.

2 Background

2.1 An Introduction to RNN

A recurrent neural network (RNN) is an inferential model that is uniquely compatible with input data arranged in a chronology (i.e. with a time-dependency), canonically denoted as `x[t]` for the input at the `t`-th time unit. RNNs' internal parameters are trained in a cumulatively predecessor-dependent fashion, in what is commonly explained as the computational encoding of a "memory". The hidden state (or hidden neuron) `h(t)` is the quantitative snapshot of this "memory" of the network at the `t`-th time step. This state has the foremost of the computational dependencies (namely, on that of the prior time-step) that will present a parallelization hurdle.

Another noteworthy feature of an RNN is the arrangement of the parameters (the "weightings" that dictate the prediction-contributing magnitude of the networks internal neurons) 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

These parameters act as a precise set of levers on the evaluative potency of each neuron of the network. They are combined with the previous time-step's value (which is accumulative), to produce output neuron `o[t]` that is the direct predecessor to the overall network output for the `t`-th time step `y[t]`, separated only by a softmax operation (a vector-wide operation that produces a probability distribution over the output channels, rather than zeroing-out all but one channel containing the maximum value).

2.2 RNN Computation

There are three stages of recurrent neural network computation, across the training and prediction-time phases: the forward pass, loss computation, and the backward pass (also called backpropagation, during which gradients that determine the adjustment of the network's parameters are derived and then applied to the network state). Our focus for this project is on the forward pass segment of recurrent neural network computation. This computational motion is generalized from four equations:

```
// 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. Our pursuit of parallelization opportunities in the forward pass was a decision necessitated mostly by practical constraint: we thought it necessary to first demonstrate a proof-of-concept in the per-time-step dependencies inherent in the progression above (before positing a guideline for the more sophisticated operations involved in, say, backpropagation's nested derivative calculations), and then we found ourselves short on time to rigorously delve into speedups of the other phases.

2.3 Parallelization of RNN inference

The chief dependency inhibiting parallelism, which consumed the majority of our attention on contemplation of mitigation strategies, is that of any time-step's `a` value on the previous time-step's `h` value. We started by acknowledging the non-viability of the rudimentary parallelism protocol that typically offers a baseline for most tasks in consideration for multi-threaded (and/or multi-processor) speedup, which is to assign a single "lane" of computation (which, in this case, would be the `a`, `h`, `o`, and `y` values at each time `t`) on a per-thread/-worker basis. This lends itself to the intuition that there needs to exist some avenue via which any thread/worker for a general `t-1`-st time-step needs to be able to supply values to that for the `t`-th time-step. The following figure demonstrates this dependency and the ensuing challenge :

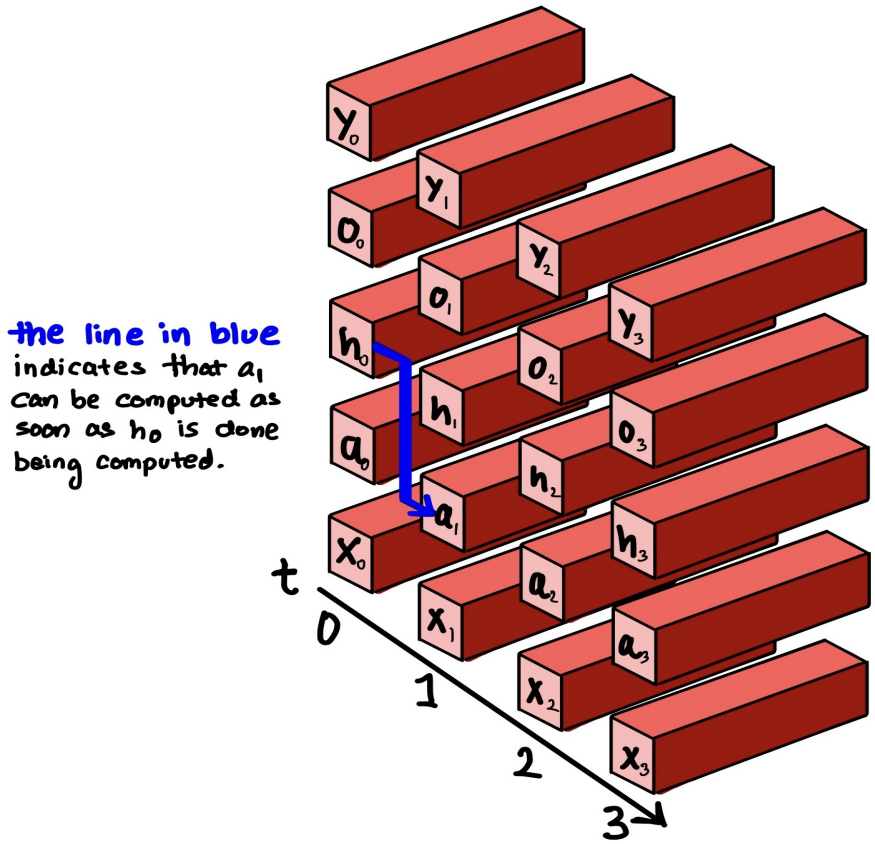


Figure 1

Note that each `a[t]`, `h[t]`, `o[t]`, and `y[t]` is a vector (we have previously referred to these as "values", but the methods and analysis contained in this paper are based on an implementation in which they are vectors–and, indeed, they can be of any dimensionality with the proper corresponding adjustments to the dimensionalities of the network parameters); they are therefore depicted as 3D blocks that are going into the page. The line in blue calls attention to the parallelization hurdle that cannot be addressed without more granular intervention than is found in conventional multi-threaded/GPU-driven workloads.

As we have alluded before, communication between all pairs of adjacent time-steps is necessary to accomplish any nonzero improvement on sequential computation (and, ideally, such communication would be efficient, tidy, and highly specialized, to keep overhead penalties in resource use and speed to a minimum). To this end, we make a modification to the baseline protocol of tasking each processing thread with a single time-step's computation: as soon as any `t-1`-st time step's hidden (`h`) vector is computed, the value is made available to the matrix multiplication needed for the `at]` vector. Notice that the product of `U` and `x`, and the addition to `b` can occur even without the availability of `h[t-1]`, i.e. a nontrivial portion of each time-step's computations can happen while the previous time-step's hidden value is pending.

Refer to the illustration below, with purple highlights on the vectors that can be calculated simultaneously between any two adjacent time-steps (caveat: as explained before, the majority, but not the entirety, of the `a[t]` vector can be computed in the situation captured by Figure 2). This generalizes across all time-steps, to reveal significant promise for at least a linear factor's worth of speedup, in a well-optimized implementation.

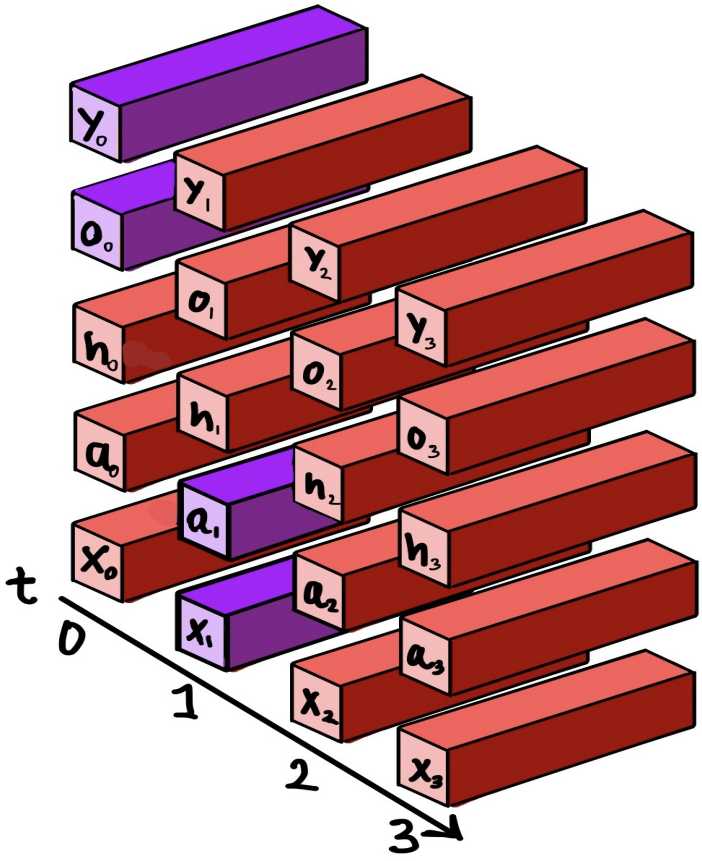


Figure 2

3 Our Approach: CUDA Implementation

3.1 CUDA Implementation: Overview

For the CUDA implementation, we began with a source code document lacking any parallelizing tools (e.g. a maximum of one thread and one block of threads declared), to make sure the matrix-multiplying operations could happen fully correctly when executed seqentially. In the interest of having a high level of granular control on the characteristics of the parallelizing tactics to be implemented in later versions of the code, we wrote all of the per-cell arithmetic for the array-based operations in this beginning phase.

We then proceeded to introduce kernel-launches that brought to implementation actuality the idea posited in Figure 2 above; for this, we began running our code in a remotely-accessed machine from a cluster in the GHC. Unless it was not available due to high demand or some other complication, `ghc59` equipped with an NVIDIA RTX GPU was the machine we defaulted to using (and we ended up using it for the vast majority of the time, as we encountered access problems only sporadically). Once the implementation was complete, we compared execution times between the sequential code and the parallelized CUDA code to obtain the speedups, after a rather strenuous process of verifying correctness in network outputs when derived through CUDA kernel launches (we came into the project without extensive familiarity on the best debugging pratices for highly parallelized code in CUDA, which we found more difficult to apply various debugger instrumentation and forensics to than code run on CPUs, so this was no cakewalk!).

3.2 CUDA Implementation: The Code

The CUDA code's defining logic is captured by the following three top-level functions:

`forwardPassSequential`

This is the function that contains the non-parallelized implementation, adapted from vanilla C++ to include the requisite CUDA syntax, keywords, and on-GPU memory to execute it within a block (of threads). This block consisted of a single thread, which was responsible for computing the entire inference-time phase of the RNN, sequentially for each time-step. It did, however, specify that all memory be allocated on-device (i.e. on the GPU), even though this would not be strictly necessary for a sequential implementation, so that the eventual comparisons to multi-threaded CUDA code would minimize the confounding effect of memory access speed.

`forwardPass`

This function houses the logic for the parallelized counterpart to the above function by invoking 'kernelComputeForward' (which is responsible for the control flow of each individual thread/worker) and completing the administrative overhead required for that: allocating memory for in-transit neuronal computations and network parameters, starting a block of threads with as many threads as timesteps, etc.

`kernelComputeForward`

The crux of the speedup enablement we expected from our implementation was a modified minimal waiting described in section 2.3. So, recalling the neurons and parameters involved in each time step's computation,

```
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])
```

it is evident that `a[t]`'s computation would be pending the 'h' value at time `t-1`. We started out with, and retained throughout the project (though with some critical modifications on the arrangement of the involved memory that will be elaborated upon later), the following waiting protocol: a simple spin loop at the start of each time-step's computation that waits on the completion of the matrix operations that result in `h[t-1][0]`'s value. The ending condition for the loop is the transition from the value of `h[t-1][0]` from `-1`

(the default, which was chosen because negative neuronal outputs are not possible) to some number greater than or equal to zero. The next section delves into greater detail the mechanisms underpinning the detection of that transition, and how the approach we used evolved over the course of the project.

3.3 CUDA Implementation: Utilizing Shared Memory

The initial code for detecting the change in values in `h` was devised such that `h` resided in global memory. But, an unforeseen deadlock risk revealed itself in our testing of this method: two threads–call them `thd1` and `thd2`–attempting to respectively read and write values to the same cell of any shared matrix could, if sufficiently aligned in the start-time of their attempts at execution time, spawn a race condition (rendering the space in memory inaccessible to either). This would halt the entirety of the RNN computation and result in indefinite hanging.

Shared memory, we learned, is in CUDA APIs equipped with the capability, via some primitives that we commentate upon in greater detail later, to sidestep this problem. So, we transitioned our initial code into a structure that would support housing the neurons and network parameters in shared memory. An important perk to this change was that we gained noticeable performance enhancements simply by virtue of the migration (and no other logical alterations), because of the way shared memory is located on hardware (with lower response times in sending values to runtime, as compared to global memory).

We replicated the setup of place-holding, for every `t`, the value of `h[t][0]` with `-1` and spinning until a change. This time, however, the deadlock risk was avoided with CUDA primitives `atomicCas` and `atomicExch`.

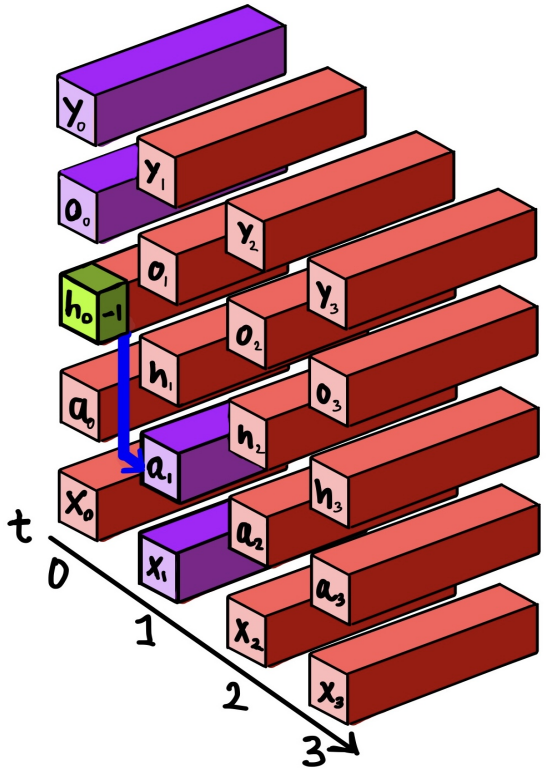


Figure 3

3.4 CUDA Implementation: `atomicCAS` and `atomicExch`

Primitive 1: `atomicCAS`

This primitive, which abbreviates "atomic compare-and-swap", provides safety against the possibility that a reading of a particular location in memory creates a deadlock-inducing contention with a different process that is simultaneously writing at the same location. Compare-and-swap's parameters are used in the following way: the first parameter is the location of memory at which the possible deadlock may occur, the second is a guess for what the value presently is, and the third is a value with which to swap the value at the location given by the first parameter if the guess is correct. The function then returns the value at the specified location in memory. The "swapping" becomes functionally equivalent to nothing happening (a "no-op"), in the way that we included the function in our code (given below), and it will read but not alter the value when it has changed from the default; therefore, a thread-safe reading action is guaranteed:

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

Primitive 2: `atomicExch`

The writing action similarly needed to be transitioned to a thread-safe method, which could be accomplished via `atomicExch`. Since it causes execution to incur higher overhead time, relative to direct assignment, it was used as sparingly as possible (we included this in a function that batch-writes to contiguous memory locations, named `setExcerpt` in the source code).

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 parallelism 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.

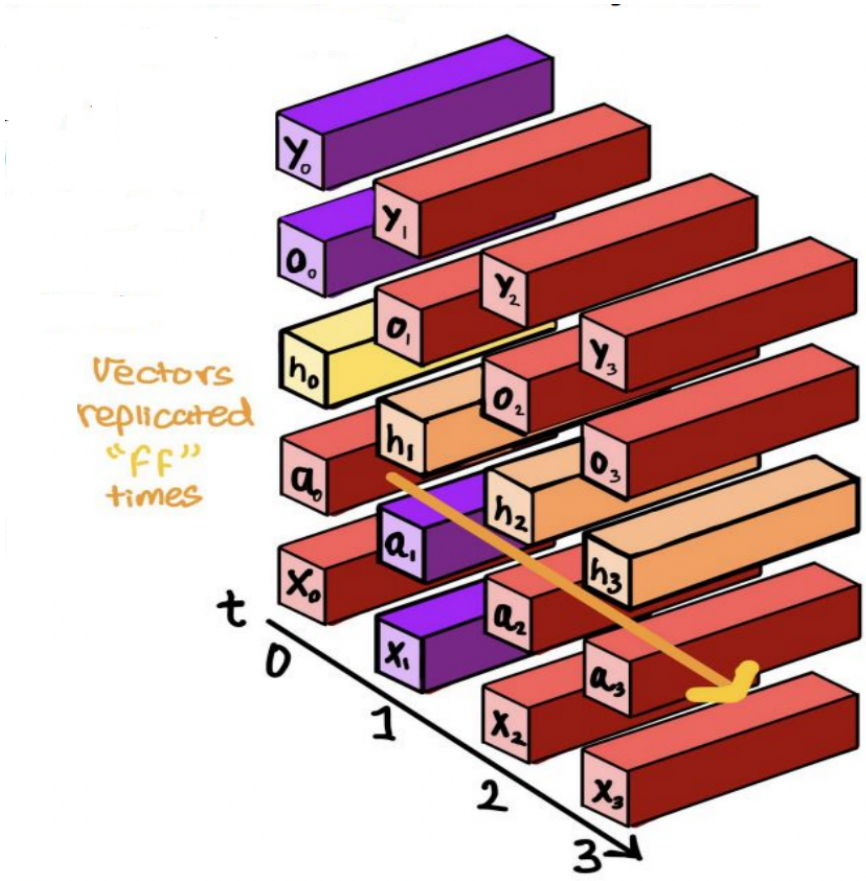
Within the loop, a function `openmp_fwd_pass` was called, to outsource the tasks of allocating and freeing memory for intermediate results and, more importantly, carrying out synchronization. In order to replicate the functionality of waiting via spin-loop in our CUDA implementation, we had to create 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, leading us to reflect on additional tactics to improve performance.

4.3 OpenMP Implementation: "Forward-Filling"

As foreshadowed in the previous section, disappointing performance gains from our first attempts at parallelized execution with OpenMP prompted thinking on possibly more elaborate machinations to achieve greater speedup. The most compelling brainstormed idea we had that passed our vetting for feasibility and well-founded technical reasoning was what we termed "forward-filling", which exchanged computational accuracy for lowering of runtime.

The "forward-filling" strategy's premise was to decrease the number of hidden states computed by a sequentially incremented linear factor, to locate the optimal point of acceptable degradation of accuracy that also notably raised performance. As illustrated in the diagram below, this was accomplished by repeating–rather than computing the actual values for–the hidden state vector, across a tunable "forward-filled" number of time-steps ("ff").



5 Results

The top and bottom figures in this section depict the change in execution times achieved by our parallelizations of the CUDA-based and OpenMP-based implementations, respectively. For the latter, the chart with heading "with" indicates that forward-filling was used and "without" indicates that it was not. Since overall runtimes for a similar-size workload would be much greater in on a CPU as compared to a GPU, due to the nature of the speed of computation occurring on each type of underlying silicon, we accordingly downsized the input parameters for the former (to have relatively similar axis boundaries between the two runtimes' plots).

We devised two problem types, with the following selections for input sizes (the keywords defined in the first list are used identically in the second list):

(1) Small

(a) time-steps (in our code, this is `TIMESTEPS`), the number of units of time (typically seconds) that the data spans), was set to 500 for the OpenMP-based implementation and 5000 for the CUDA-based implementation

(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 for both implementations

(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 for both implementations

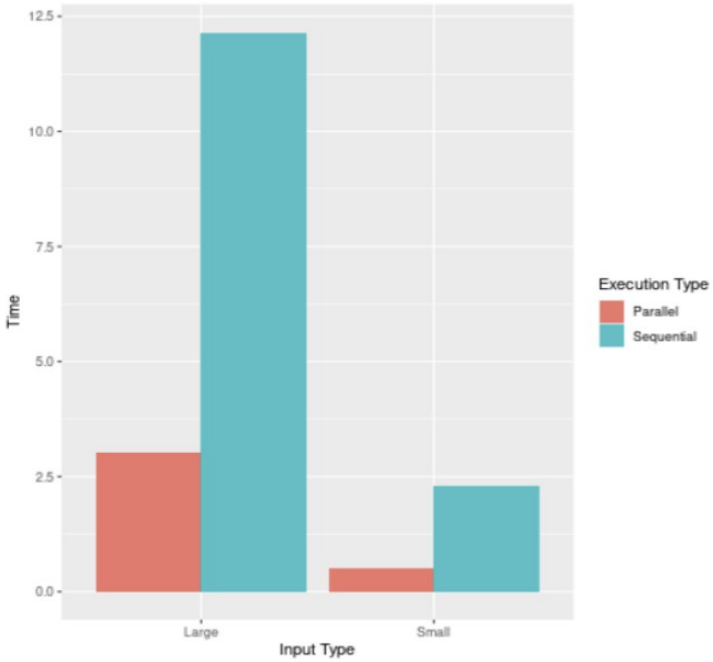
(2) Large

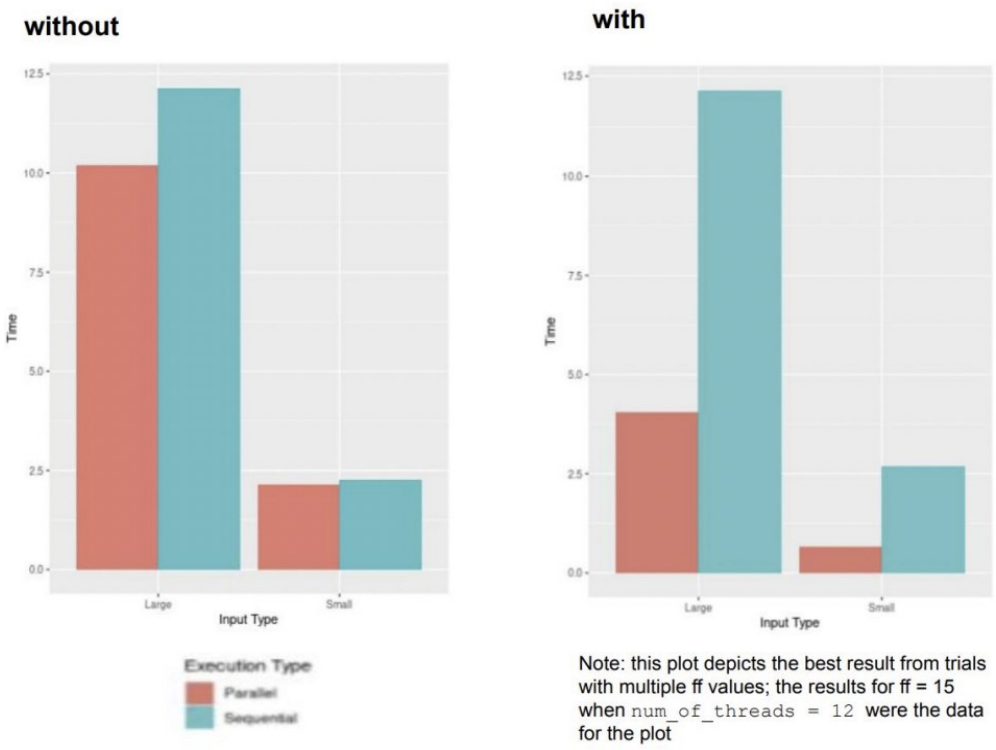
(a) timesteps was set to was set to 1000 for the OpenMP-based implementation and 10000 for the CUDA-based implementation

(b) hidden size was set to 125 for both implementations

(c) vocab size was set to 8000 for both implementations

The plots only capture results from the "Small" problem class, but speedup ratios were very similar for the "Large" problem class.



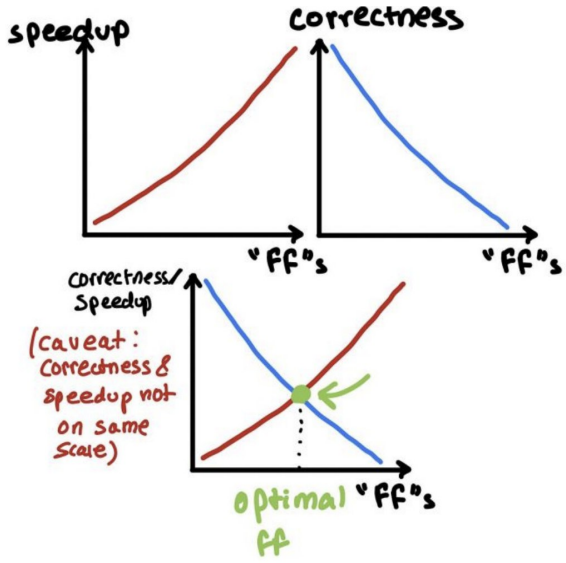


The overall execution times were deliberately designed to be similar, with the selection of different input sizes, but an important factor at play was that the CUDA-based code incurred considerably more setup overhead. CUDA kernels are most performant when their code involves very little branching and 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 noteworthy speedup in the CUDA implementation, especially relative to its own sequential version. This suggests that the overhead was high but does not preclude realizing speedup when there are multiple threads. We do, however, anticipate that using more sophisticated techniques (like multiple blocks or grids of blocks) will not be very promising (indeed, we dabbled with this ourselves).

As has been mentioned in earlier sections, the speedups from a logically analogous implementation of parallelization protocol to that of the CUDA-based code in the OpenMP-based code were unfortunately negligible. This is very evident from no more than a glance at the left-sided plot in the bottom figure

The "forward-filling" strategy we coded into our solution as a response to that discovery yielded a speedup ratio more comparable to that of the CUDA implementation. However, this result must be caveated with the reminder that "forward-filling" imposes a correctness penalty on the overall inference event (i.e. the values derived in the computation captured by the "with"-titled plot in the bottom figure have some nonzero degree of error, as contrasted with those of the CUDA plot that are guaranteed to be correct). This inaccuracy can reach a point of becoming very untenable, especially if the inference occurs along very finely-separated decision boundaries, but we found that, for out problem inputs, correctness penalties were modest if "ff" were carefully tuned. Since the extents of correctness and runtime improvements travel inversely, we were able to select the "ff" that optimizes the compromise on correctness with runtime improvement by finding the intersection of the following (prototypical) graph:



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