INTRODUCTION

What is a RNN (Recurrent Neural Network) ?

- There are 3 stages of RNN computation, similar to other multi-layer neural networks:
  - the forward pass
  - the backward pass
  - computing gradients during backpropagation

Why use RNNs?

- RNNs are distinguished by **use of previously-computed neuron outputs** across one more axis (not just going "up the layers")
  - RNNs are particularly effective for predictive challenges on data that occur on a segmentable continuum (e.g. time)
    - each segment has intuitive dependencies (that the RNN aims to capture quantitatively) on neighboring segments

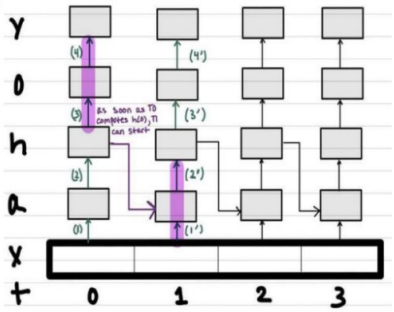
- We focus on the **inference-time forward pass**
  - assume that the training component has completed with parameter values (that are ready for use in inference computations)
  - entails **the below four core formulas for layers' neuronal outputs** that we examine for opportunities to parallelize:

```
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])
```

`a`, `h`, and `o` are (vectors of) intermediate layers' hidden values.

Parallelizing Inference:

- thread for one timestep has to wait for the value of `h[t-1]` to get `a[t]`.
- project's premise is to mitigate parallelization hindrance of this feature of RNNs



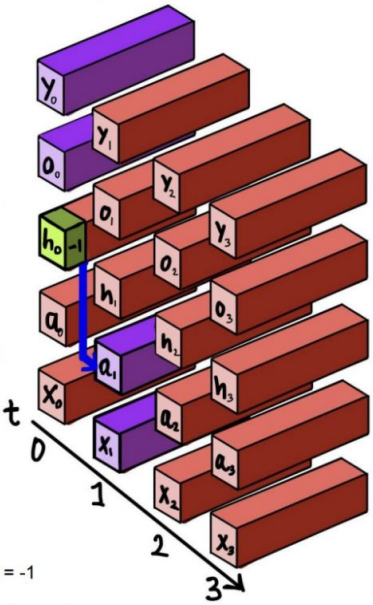
Actions highlighted in purple can occur in parallel via different threads

CUDA Implementation: Logic and Initial Design Decisions

- The CUDA implementation was implemented to analyze the results of **a GPU-based acceleration**.

- **We implemented two versions:**
  - **A sequential version:** this is used as a benchmark to determine the speedup that results from the parallel version
  - **A parallelized version**

- **The Kernel Function:**
  - Sequential version: calls the kernel function with one block with one thread
  - Parallel version: calls the kernel function with one block with *timesteps* number of threads (each thread handles one time step)



OUR DESIGN (see diagram above):

- For each h[t] at each time step: h[0] = -1 (labeled in green in the diagram)
  - Thread 1 handles computation for t = 1
    - Must wait for Thread 0 to finish computing h[0] before thread 1 can compute a[1]
    - Once Thread 0 computes h[0], Thread 1 begins computation for its timestep
    - In purple is the computation that is done in parallel (Thread 0 continues computing o[0] and y[0] while Thread 1 computes a[1]))
    - The blue arrow highlights the dependency between threads

- **The kernel function**
  - **Contains spin loop** that keeps reading value in h[t-1]

CUDA Implementation: Challenges and Iterations

CHALLENGE 1: very slow memory access

- Our original implementation used global memory:
  - many issues with this: one was undefined behavior caused by race conditions (see diagram on below-right)
  - Global memory is much slower than shared memory (15x+ according to NVIDIA developer guide docs

SOLUTION 1: maximally use shared memory local to kernels

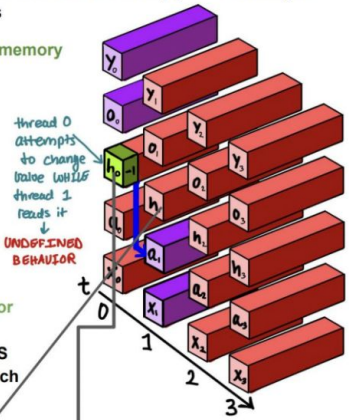
- Allocated within a kernel launch
  - Added perk: faster declaration/initialization

CHALLENGE 2: race conditions in writes and reads

- Pausing thread (reading) might collide with thread that is writing -> problems!

SOLUTION 2: Use CUDA primitives for atomicity guarantees

- **ATOMIC READS:** use **atomicCAS**
- **ATOMIC WRITES:** use **atomicExch**



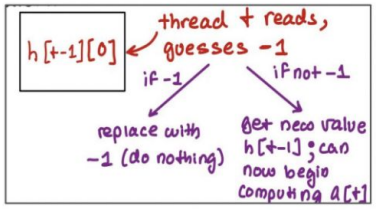
atomicExch

For all timesteps' `h` vectors,

```
atomicExch(&h[0],
    neroun_ouptut)
atomicExch(&h[1],
    neroun_ouptut)
        .
        .
        .
atomicExch(&h[HSIZE-1],
    neroun_ouptut)
```

Note: The CAS provided by CUDA API only works for integers: we had to adapt this (see references)

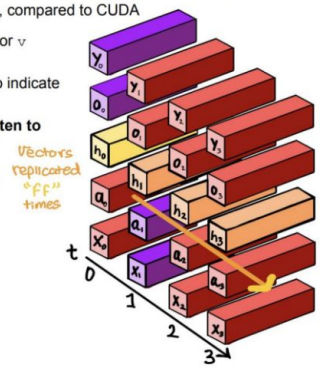
atomicCAS



OpenMP Implementation: Idea

- The OpenMP implementation is composed in CPU code
  - more tractable with logic involving more branching and variation.
  - So, **outsource the mainloop iteration** (that computes the forward pass) to a function with internally-dependent control flow (e.g. conditionals)
    - **but don't have to work on minimization of non-uniformity** in that function's execution pattern, compared to CUDA

- Correctness ensured by tracking a vector `v`
  - initialized to all 0s at timestep 0
  - Previous timestep sets flag to 1 to indicate availability of neuron output
  - `v` **atomically read from and written to**



OpenMP Implementation: Challenges

- cost of threads remaining idle observed to be higher with the OpenMP impl.
- Mitigation
  - compromise some accuracy for speedup
  - by way of the strategy we call "forward-fill"

"Forward Fill"

1. once `h[t]` generated by thread at time `t`, get `average(h[t])`
2. vector of `average(h[t])` replicated across a tunable parameter of ("`ff`") timesteps' hidden neuron outputs
   a. tune to find the optimal point tradeoff between degraded correctness and speedup

Tuning result: found that selecting a `ff` ≈ `num_of_threads + 2` optimal