INTRODUCTION

What is a RNN (Recurrent Neural Network) ?

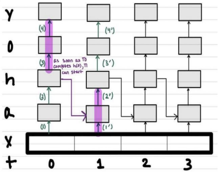
- There are 3 stages of RNN computation, similar to other multi-layer neural networks:
  - the forward pass
  - the backward pass
  - computing gradients during backpropagation

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.



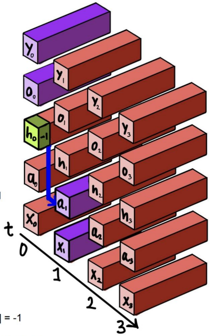
Actions highlighted in purple can occur in parallel via different threads

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

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

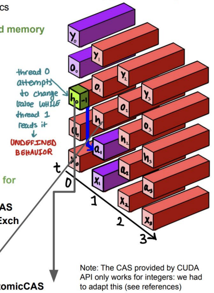
- Allocated within a kernel launch
  - Added perk: faster declaration/initialization

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

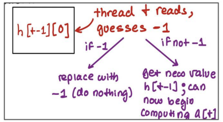
For all timesteps’ `h` vectors,

```
atomicExch(&h[0],
    neroun_ouptut)

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

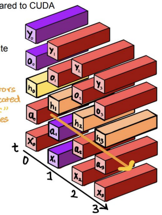
atomicCAS

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



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**



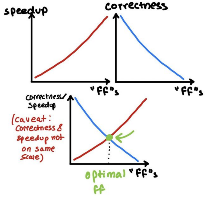
OpenMP Implementation: Challenges

- 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



Results for CUDA Implementation

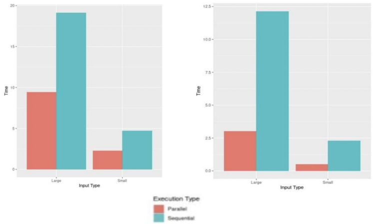
These are the parameters for the two types of problems on which the implementation was tested

Problem Type / Parameters	VSIZE	HSIZE	TIMESTEPS
Small Input Size	8000	50	5000
Large Input Size	8000	125	10000

- TIMESTEPS: number of units of time (typically seconds) that the data spans
- HSIZE: number of features computed in the hidden state of the network (i.e. the dimensionality of the hidden vector `h` for each time step)
- VSIZE: number of values in the output vector (over which an argmax will yield the prediction)
- For all tests, the number of threads launched per block was 5 (empirically found to be a good balance between (under)-utilization and speedup)

Speedup Plots: Before-and-After switch to per-kernel shared memory access (for `h` storage)

Before **After**

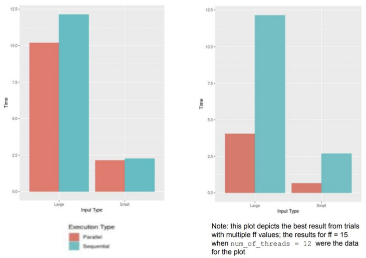


Results for OpenMP Implementation

- We observed middling speedups with a logical replication of the code we had for CUDA kernels.
  - So, we devised the forward-filling strategy

Speedups with and without Forward-Filling

without **with**

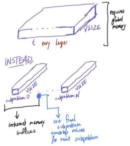


Note: this plot depicts the best result from trials with multiple ff values; the results for ff = 15 when `num_of_threads = 12` were the data for the plot

Because GPUs are much faster than CPUs, we chose the following (smaller) problem parameters to increase comparability with our CUDA impl.

Problem Type / Parameters	VSIZE	HSIZE	TIMESTEPS
Small Input Size	8000	50	500
Large Input Size	8000	125	1000

Discussion of Discoveries, Shortcomings, and Matters for Further Inquiry

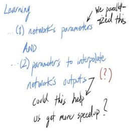


CUDA:

- Using shared memory (per-kernel) was very conducive to speedup
- But we still have some aggregation of results that involves global memory
  - May be some optimizations in that approach we overlooked
    - For sufficiently small problem parameters, don’t bother with global memory at all?
    - Compute fragmented sub-solutions (of contiguous sections of data) in separate executions and combine later, for bigger input sizes?

OpenMP

- Forward-filling turned out to be great for speedup
- However….
  - Its underlying logic is somewhat unsophisticated
    - interpolating data by copying an average, pretty much
  - Did not impinge correctness too much when the ff exceeded num_of_threads by 1-2 (Occam’s Razor!), but correctness declined precipitously thereafter
  - Maybe doing some inference on the values to supply in the forward fill can mitigate this!
    - We are doing machine learning for value prediction after all :)
    - But, might require additional compute power at training time, to also develop neuron-like mechanisms for ff’s values



References

Nabi, J. (2019, July 21). Recurrent neural networks (rnns). Medium. Retrieved December 10, 2021, from https://towardsdatascience.com/recurrent-neural-networks-rnns-3f06d7653a85.

Nvidia. (n.d.). Libcudacxx/atomic_thread_fence.MD at main · NVIDIA/libcudacxx. GitHub. Retrieved December 10, 2021, from https://github.com/NVIDIA/libcudacxx/blob/main/docs/extended_api/synchronization_primitives/atomic/atomic_thread_fence.md.

Using shared memory in CUDA C/C++. NVIDIA Developer Blog. (2021, October 29). Retrieved December 10, 2021, from https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/.

GPU computing with CUDA Lecture 3 - efficient shared ... - bu. (n.d.). Retrieved December 10, 2021, from https://www.bu.edu/pasi/files/2011/07/Lecture31.pdf.

Cuda C++ Programming Guide. NVIDIA Documentation Center. (n.d.). Retrieved December 10, 2021, from https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

Cuda – tutorial 4 – atomic operations. The Supercomputing Blog. (2011, September 11). Retrieved December 10, 2021, from http://supercomputingblog.com/cuda/cuda-tutorial-4-atomic-operations/.

Understanding and using atomic memory operations. Understanding and Using Atomic Memory Operations. (n.d.). Retrieved December 10, 2021, from https://on-demand.gputechconf.com/gtc/2013/presentations/S3101-Atomic-Memory-Operations.pdf.

How to access global memory efficiently in CUDA C/C++ kernels. NVIDIA Developer Blog. (2020, August 25). Retrieved December 10, 2021, from https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels.

Fisseha Berhane, Phd. Building a Recurrent Neural Network - Step by Step - v1. (n.d.). Retrieved December 10, 2021, from https://datascience-enthusiast.com/DL/Building_a_Recurrent_Neural_Network-Step_by_Step_v1.html.