TSUC: TNN simulator using CUDA

Hongxuan Li (hongxuan), Zhengfan Zhang (zhengfaz)

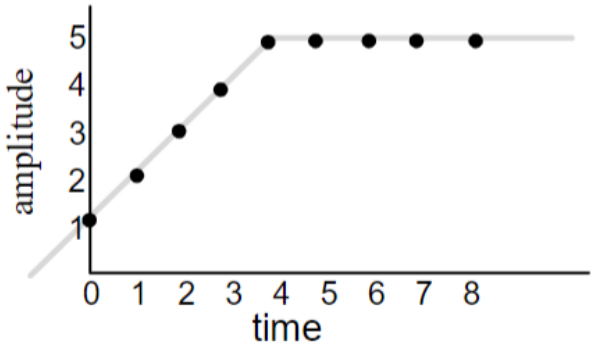
1 SUMMARY

We implemented a Temporal Neural Network simulator in CUDA and compared its performance to a PyTorch-based implementation. The CUDA implementation ran on the GHC machines with RTX2080 while the PyTorch implementation was ran locally on RTX3090. We achieved a >7x speed up compared to the PyTorch implementation.

2 BACKGROUND

Temporal Neural Network is a type of biomorphic neural network which uses a time delay to encode the magnitude of signals. The relative arrival time of a signal (called ‘spike’ in this context) at a synapse, together with its synaptic weight, determines the shape of the response function. The sum of the synapses' response functions determines the body potential curve of the neuron for which the synapses belong to. When a neuron’s body potential reaches a threshold, the neuron will fire an output spike at that time. The neuron’s body potential will reset once per fixed amount of clock cycles, and this longer clock cycle that the resets happen at is called a ‘gamma cycle’, which serves to separate the computational waves. (That is, the delay is with respect to the start of a gamma cycle)

The output function of a synapse in our model follows the Ramp-No-Leak (RNL) function. It does not immediately add the whole weight to the body potential of the neuron, but instead adds 1 to the body potential every clock cycle and stops after weight cycles.



d) ramp no-leak [1]

		t=0	t=1	t=2	t=3	t=4	t=5	t=6	t=7
N1	S1	1	1	1	1				
	S2			1	1	1	1	1	1
	S3				1	1	1	1	1
	pot	1	2	4	7	9	11	13	15
N2	S1	1	1	1	1	1	1	1	
	S2			1					
	S3				1				
	pot	1	2	4	6	7	8	9	9

[2]

The weight on a synapse is trained according to the spike-timing dependent plasticity model (STDP), which modifies the weight of the synapses based on the causality relationships between the input and output spikes:

In the case there is both an input spike and an output spike from the neuron, if the synapse receives a spike before the neuron fires a spike in a gamma cycle, the weight is strengthened, otherwise weakened.

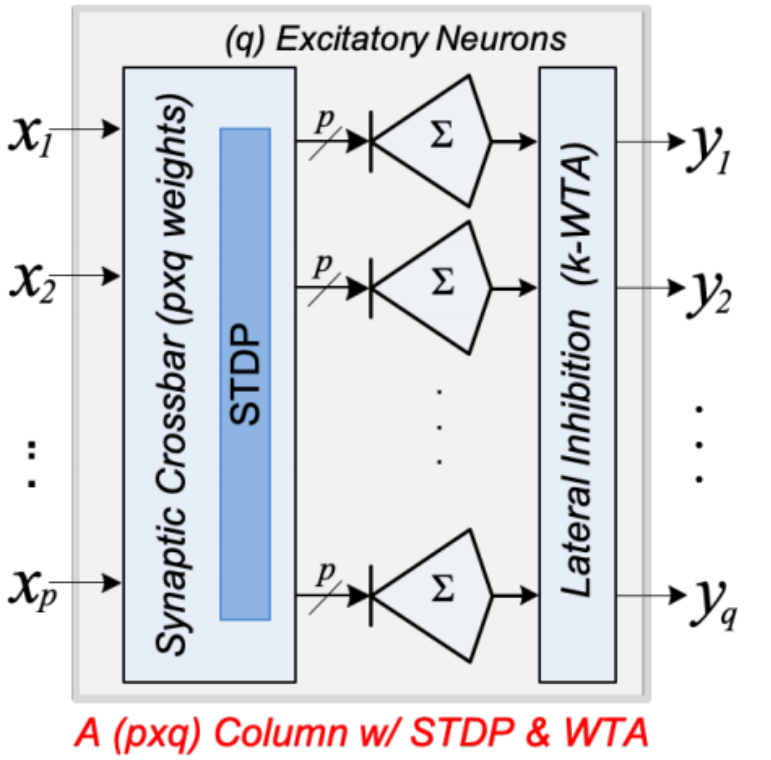
If the synapse receives a spike, while the neuron does not fire, a very small search factor is added to the weight.

If the neuron fires but the synapse did not receive a spike, the weight is weakened. [3]

input conditions		weight update
$x_i \neq \infty$	$x_i \leq y_j$	$\Delta w_{ij} = +\mu_+ \cdot F_+(w_{ij})$
$y_j \neq \infty$	$x_i > y_j$	$\Delta w_{ij} = -\mu_- \cdot F_-(w_{ij})$
$x_i \neq \infty$	$y_j = \infty$	$\Delta w_{ij} = +\mu_s$
$x_i = \infty$	$y_j \neq \infty$	$\Delta w_{ij} = -\mu_- \cdot F_-(w_{ij})$
$x_i = \infty$	$y_j = \infty$	$\Delta w_{ij} = 0$

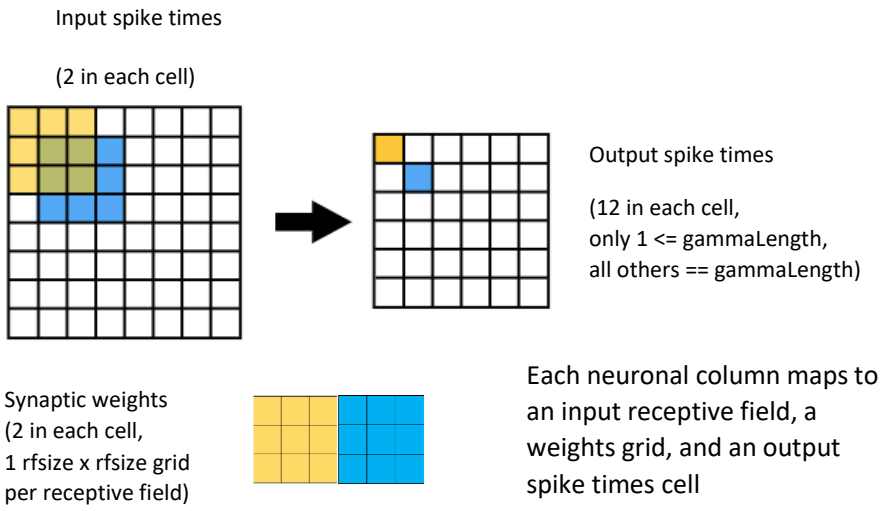
$F_+(w_{ij}) = \mu_+$ if $w_{ij} \geq w_{max}/2$; else $F_+(w_{ij}) = \mu_+/2$
$F_-(w_{ij}) = \mu_-$ if $w_{ij} < w_{max}/2$; else $F_-(w_{ij}) = \mu_-/2$ [3]

Multiple neurons that have their synapses connected to the same set of inputs form a neuronal column. In each column, a winner-takes-all operation is performed over all the neurons: once a neuron fires a spike, the column will pass down that spike and inhibit all other neurons from firing in that gamma cycle.



[4]

Multiple columns form a layer. The TNN may consist of multiple layers.



[5]

2.1 Breakdown of workload

2.1.1 Computation of neuron body potential at each timestep, determination of spiking time

Inputs:

Input spike times corresponding to the neuron's RF, weights corresponding to the neuron's RF

Outputs:

nNeuronsPerColumn body potential at each timestep, output spike times before WTA inhibition.

Algorithm:

1. Compute the timestep at which each synapse starts and finishes contributing to the body potential

2. Compute the body potential at each timestep, by counting the number of synapses that are contributing to the body potential at each timestep, and adding it to the cumulative sum that represents the body potential.

3. At each timestep, compare the current body potential to the threshold potential to determine if the neuron have spiked. Record the timestep and body potential if so.

Dependency:

Weight updates from the previous input is completed; the body potential contribution of all synapses within a neuron has to be collected at the end of each iteration.

Parallelism:

Data parallel across each synapse for start and end time calculation;

data parallel summation of Boolean values across each synapse for body potential

accumulation, but requires sync within each neuron at each timestep boundary;

data parallel across each neuron for spiking determination.

2.1.2 1-WTA (Winner Take All) inhibition

Inputs: Output spike times and body potentials from previous step

Outputs: Inhibited spike times

Algorithm: Find min(column output spike times), and max(neuron body potential at spiking) among the previous min for tie breaking.

Dependency: All spike times has been computed for the column

Parallelism:

Data parallel across columns

2.1.3 STDP learning

Inputs: Input spike time corresponding to the synapse, WTA-inhibited output spike time corresponding to the synapse's neuron, constant STDP learning parameters

Outputs: updated weight of the synapse

Algorithm: as described before

Dependency: WTA inhibition has been done

Parallelism: Data parallel across each synapse

3 APPROACH

3.1 Platform Choice

We build our model on GHC machines because they are easily accessible to us. And we choose to use CUDA for our project. This comes from the intuition that the CUDA threads running in lockstep are well suited to tracking the synapses in our model which also needs to work in sync at every tick cycle.

3.2 Problem mapping

For MNIST with pos-neg encoding with a single large receptive field, we have a single column with 28 * 28 * 2 = 1568 synapses per neuron, and we used 12 neurons per column to capture the features in the handwritten digits in MNIST.

The sheer number of synapses to track is too large to allow us use 1 thread per synapse. So, we had to break the receptive field into 2D batches of pixels, and let each thread process a batch. But still, for ease of computation, we decided to let the number of neurons in the column to take one of the block dimensions (`blockDim.x`), and combine the number of input channel into the third block dimension (`blockDim.z`). This way, we can easily index into different neurons, and also have 2 indices left (one of them is with channel index) to index into the 2D receptive fields.

If there are multiple columns in the layer (a feature we didn't use in our final design), that will use the `gridDim`. If we are using each column in the layer as a convolution filter, we will be using `blockDim.x` and `blockDim.y` to index into these convolution columns.

3.3 Kernel Dimension

We calculate the dimension of the 2D batch by first find out a block dimension that can maximize the use of the 1024 threads:

$$nXYThreads = \left\lfloor \sqrt{\frac{1024}{numNeurons \cdot numInChannels}} \right\rfloor$$

Then we calculate the batch dimension from the receptive field size and `nXYThreads`:

$$batchDim = \left\lceil \frac{rfDim}{nXYThreads} \right\rceil = \frac{rfDim + nXYThreads - 1}{nXYThreads}$$

And we launch columns with block dimension `dim3(numNeurons, nXYThreads, nXYTreads * numInChannels)`. So `threadIdx.x` becomes the `neuronIdx` within the column, `threadIdx.y` becomes `yBatchIdx`, `threadIdx.z / nPrevChan` becomes `xBatchIdx`, and `threadIdx.z % nPrevChan` becomes the `channelIdx`.

Each thread is in charge of the pixels in the 2D range from `yBatchIdx * yBatchSize` to `(yBatchIdx + 1) * yBatchSize`, and from `xBatchIdx * xBatchSize` to `(xBatchIdx + 1) * xBatchSize`. Any pixel that's outside of the receptive field will be skipped.

3.4 Kernel Details

After some initialization, we start to loop through the input images (already translated into spike times).

3.4.1 RNL Start and End Times

Before we start to count the ticks, we check the weights and input spikes for the synapses in the thread's batch, and generate thread local arrays of their RNL function start times and end times.

3.4.2 Generate Neuron Spike Times

Then we loop through the ticks in a gamma cycle and track if any synapse within a thread's batch should contribute to the neuron's body potential during that tick cycle. We add up all these contributions first in a thread local variable `localBodyPot`, and then accumulated using `atomicAdd_block()` to a block-level shared array of neuron body potentials.

After we know the neuron body potentials at the end of that tick cycle, we compare all of them against the spiking threshold (only 1 thread for each neuron is needed), and record the spiking time and its body potential at that time in shared arrays if the neuron spikes.

3.4.3 1-WTA Inhibition

After the tick cycle loop is finished, 1 thread for each neuron checks if the neuron is the earliest to spike. If there is a tie, we favor the neuron with the highest body potential at spike time and the one with lowest index. And only the earliest spiking neuron is allowed to write its spike time to the output. All the other neurons will write a spike time equal to gamma cycle length (means no spike generated) to their corresponding output entries.

3.4.4 STDP Weight Update

If STDP is enabled, each thread will then loop through the synapses in its batch and update the weights according to input and output spike times and using the rules provided in the Section 2.

3.5 Iterations of Optimization

3.5.1 Optimization of problem mapping

Original Direct Mapping of Synapses to Threads

Originally, we were planning to use multiple layers, where each layer has multiple small columns with small receptive fields (3 * 3) and use them like convolution filters. That enabled us to map 1

pixel/synapse to each thread. And a thread block needs only 3 * 3 * 2 * 12 = 216 threads. But latter, since we need to first have a running model, we decided to enlarge the receptive field to the size of an entire image (28 * 28). That means much more synapses per column (28 * 28 * 2 * 12 = 18816), which far exceeds the limit of 1024 threads per column in CUDA. And we have to let each thread process multiple synapses in the way described in Section 3.2.

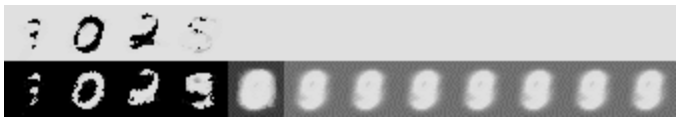
Phases

We thought about mapping neurons to blocks instead of mapping to threads for 2.1.1 and 2.1.3. However, this requires block-wise synchronization for entering and exiting workload phases 2.1.2, and possibly passing of arrays over global memory if multiple kernel launches are used. We were unable to find a satisfactory solution to the problem, and worried about losing out on the speedup for small receptive fields (which we anticipated to occur much more frequently on multiple layer configurations), so we went with the batching approach.

3.5.2 Classification Purity Optimization

Originally, when we are iterating through the tick cycles, we let 1 thread for each neuron to write to 2 column shared variables `earliestSpikingTime` and `earliestSpikingNeuron` if it spikes earlier than what variables indicate. And we allowed race conditions to happen because we believed that the weights of the neurons would quickly divert and generate spikes at different times in most of the cases. But that is not true. And the learnt weights of the neurons are muddy as the image below.

We tried to raised spike threshold from 400 to 3000 to prevent neurons from generating too many spikes and learning trash. But only the first few neurons are learning.



From this picture, we can see that CUDA probably favors lower thread indices on racy writes. In addition, the discretized time unit (tick cycle) we use increases the chance of a tie. And race conditions may happen more than expected. So, having a dedicated "fair" tie-breaking criteria is much more important than anticipated.

Eventually, we used shared arrays to store the spike time and spike body potential of every neuron, and we iterated through these data to break tie in favor of the highest body potential if 2 neurons spike at the same time. And we only allow 1 thread that writes the output spike time for the winning neuron.

Before this final solution, we tried the following:

Using atomic operations

We tried to concatenate the spiking neuron index and the spiking time into 1 single variable and use CAS to update the shared variables `earliestSpikingTime` and `earliestSpikingNeuron`, but that does not change anything at all. Now we know the major problem is unfair tie-breaking.

Using critical sections

We also tried to use CAS based semaphores to protect the shared variables, but that caused the kernel to stuck on the first input image. Maybe it's a deadlock, or maybe it's something else. But we quickly gave up this approach and switched to our current approach.

3.5.3 Resource use optimization

The tie-breaking mechanism we chose lead to `cudaErrorLaunchOutOfResources`, and we had to reduce the number of threads in a block to run our kernel. After a lot of searching, seems like the critical resource could be the local registers. (`ptxas -v` assembling the `ptx` intermediate assembly output can show register usage.)

Before tie-breaking is added, `ptxas` shows 72 registers used, 12*2*6*6*72/1024=60.75K registers for a whole block. After tie-breaking is added, `ptxas` shows 80 registers used, 12*2*6*6*80/1024=67.5K registers for a whole block. And from NVIDIA: Compute capability 6.1 has the limits: max 64K 32-bit registers per block, 255 per thread. [6]

TA Nishanth (nsubram2) suggested adding compile flag "`--maxrregcount 1`" to `makefile`, which reduced the per-thread register usage to 62.

3.6 Starting point & Changes from the Original Algorithm

We have based our algorithm on the PyTorch tensor-based implementation in 18-743 Lab assignment 1. However, the code is entirely rewritten for CUDA, with direct mapping of the neural network structure to blocks and threads, so data structures used are mostly different.

4 RESULTS

4.1 Performance measurement method

We collected the total execution time of the column kernel for the CUDA implementation, and timed the execution of TNNColumnLayer and STDP_deterministic for the Python implementation, for processing 10000 training and testing images respectively.

4.2 Experimental setup

MNIST dataset was obtained from http://yann.lecun.com/exdb/mnist/, and placed in the data/ folder which is in the same directory as the executable, and uncompressed.

A kernel is created to convert the 8-bit greyscale values to PosNeg encoded 3-bit spike delay values. (Pos: bright pixel -> early spike (at time 0), dark pixel -> no spike (at time gammaLength); Neg: bright pixel -> no spike, dark pixel->early spike)

The converted spike time inputs are passed into the kernel directly and launched, and the time before and after launching the kernel is recorded, the difference is taken as the execution time.

For the Python implementation, the TNN column layer and STDP_deterministic layer was ran 10000 times to train the weights matrix for the training phase, and the time taken for the 10000 iteration to complete was recorded. For the testing phase, only the TNN column layer was ran and the time taken was recorded.

4.3 Execution time comparison & speedup

	Python time	CUDA time	speedup
train, 28x28, 10000	43.84	4.09	10.71883
test, 28x28, 10000	46.7	2.83	16.50177
train, 3x3, 60000	545.32	8.1	67.32346
test, 3x3, 10000	91.48	1.08	84.7037

4.3.1 Purity, coverage and confusion matrix

Python implementation:

1	639	35	15	21	36	19	17	30	8
4	0	19	0	274	61	191	54	87	134
0	8	57	614	2	124	11	2	228	12
1	0	17	13	15	15	0	313	28	168
1	464	117	0	9	10	6	40	23	5
106	3	30	69	0	219	138	9	39	4
18	0	51	8	36	23	5	432	6	68
845	0	59	42	10	309	98	29	95	63
4	21	98	56	115	65	28	93	53	124
0	0	421	3	12	6	427	2	33	2
0	0	11	7	488	13	34	29	33	416
0	0	117	183	0	11	1	8	319	5

Purity: 0.5158, Coverage 1

CUDA implementation:

		DIGIT									
		0	1	2	3	4	5	6	7	8	9
NEURON	0	0	0	93	381	0	94	0	0	250	7
	1	23	77	84	32	111	114	85	87	48	43
	2	1	633	62	21	5	9	3	27	37	1
	3	1	45	25	20	68	169	25	23	200	135
	4	22	0	26	23	223	44	5	456	32	482
	5	106	6	53	262	5	165	273	0	53	3
	6	675	0	45	25	1	35	64	2	56	11
	7	10	0	37	6	35	2	0	355	43	156
	8	130	2	37	232	1	244	5	5	160	40
	9	4	0	344	5	294	1	52	32	23	35
	10	1	372	187	1	14	1	6	37	51	2
	11	7	0	39	2	225	14	440	4	21	94

Purity: 0.4513, Coverage 1

4.4 Analysis

We have achieved large speedups across the two extremes of receptive field sizes compared to the PyTorch implementation. This is because the problem is directly mapped to bare metal CUDA structures, instead of being mapped to an intermediate tensor data structure and performing computation using tensors.

Something to take note of here is that the 3x3 runs on Python were done with an additional voting layer, while the CUDA run only include a single neuron layer. This might have made the speedup appear greater than it actually is. Another possibility for the slowdown on Python runs is that the convolutions are implemented by expanding the input data into a larger matrix first using convolution, and then performing the computation on the larger input data set, while on CUDA it was done by using overlapping array indices, making it much more efficient.

4.5 Limits

Currently, we are having pretty good speedup for both training and inferencing. But we think there are still possible improvements we could have done. And the most significant one is the access to weights.

Both our training and testing are using the same number of images. But testing is way faster than training (for 28*28 rf, 4.09/2.83 = 1.45x, for 3*3 rf, 8.1/1.08 = 7.5x). This is caused by the access to the weights when performing STDP updating during training. When we were calculating the RNL start time and end times, we read all the weights once, and during training, after the output spikes are generated, we have to read and update all the weights again. If we could buffer the weights in block shared memory and thus reduce the number of reads from 2 to 1 during training, by speculation, we could get faster training.

But we didn't choose to do so. This is because we are having 28 * 28 * 2 * 12 = 18816 weights per block for 28*28 rfs and 3 * 3 * 2 * 12 * 26 * 26 = 146016 weights per block for 3*3 rfs, and all our weights are floats. So, we would certainly exceed the limit in block shared mem size if we buffer the synapse weights.

5 References

[1] J. E. Smith, "Architecting a Biologically Plausible Silicon Brain (CMU 18-743 S22 Lectures 3 & 4)," 2022.

[2] J. P. Shen and H. Nair, "CMU 18-743 S22 Neuron Example (20220208)," 2022.

[3] 18743 Course Staff, "18-743 Spring'22 Lab Assignment 1," 2022.

[4] J. P. Shen, "CMU 18-743 S22 Lecture 2," 2022.

[5] 15618 Course Staff, "Lecture 23: Deep Neural Networks: Part 1," 2022.

[6] NVIDIA Corporation, "Programming Guide::CUDA Toolkit Documentation," [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications__technical-specifications-per-compute-capability. [Accessed 2022].