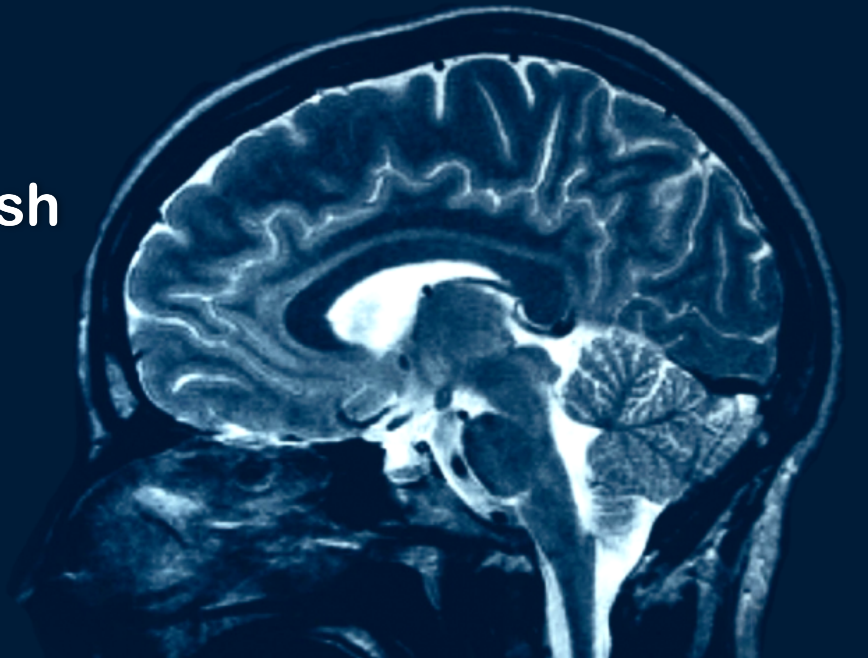
Simulation of Izhikevich's Large-Scale Neuronal Model on GPU

Nima Mohammadi
Under Supervision of **Prof. Ganjtabesh**



Large-Scale Modeling

- Principle of hierarchical reductionism: study smaller and smaller parts and attempt to understand the whole .
- While it remains uncertain whether a brain system can be understood as the interaction between independently describable subsystems, the brain does display a hierarchy of spatial scales with repeated structure such as molecules, synapses, neurons, microcircuits, networks, regions and systems.
- How the advent of large-scale modeling in computational neuroscience opens the possibility to study the dynamics of models simultaneously incorporating various levels from molecules to regions?

Djurfeldt (2008). **Large-scale modeling–a tool for conquering the complexity of the brain.** Frontiers in neuroinformatics

Large-Scale Modeling | Mathematical Modeling

- Experience shows that a model should be as simple as possible in order to be tractable (possible to analyze, easy to do computations with) and in order to make strong statements about the physical system being modeled.

> *"It can scarcely be denied that the supreme goal of all theory is to make the irreducible basic elements as simple and as few as possible without having to surrender the adequate representation of a single datum of experience."*
>
> -Einstein

- With increased complexity of model, uncertainty of modeling results increases. Also the explanatory power of model is lost. Of course the degree of simplicity that is achievable is dependent on the scientific question posed.

Is a Detailed Model More Realistic?

- It seems that a higher level model should be composed of component models at a lower level, and in turn must be much more complex and have more parameters than a model of a neuron. If true, one might ask what is the proper lowest level? After all, the deeper the level, the more realistic, right?!

  Wrong! Including more details from lower levels leads to more model parameters which makes it harder to obtain a realistic model since the realism of a model is related to how well-constrained it is by experimental data.

- More model parameters means that more data is required to determine them, data which can often contain uncertainties and be hard to acquire.

Example: Describing the propagation of sound through air.

- Modeler's wrench, abstraction: taking away aspects not important for answering the scientific questions which the model is designed to address, useful models can be formulated at different levels of organization without loss of tractability.

- Numerical simulations of brain network models have typically been based on either abstract connectionist-type units or integrate-and-fire units.

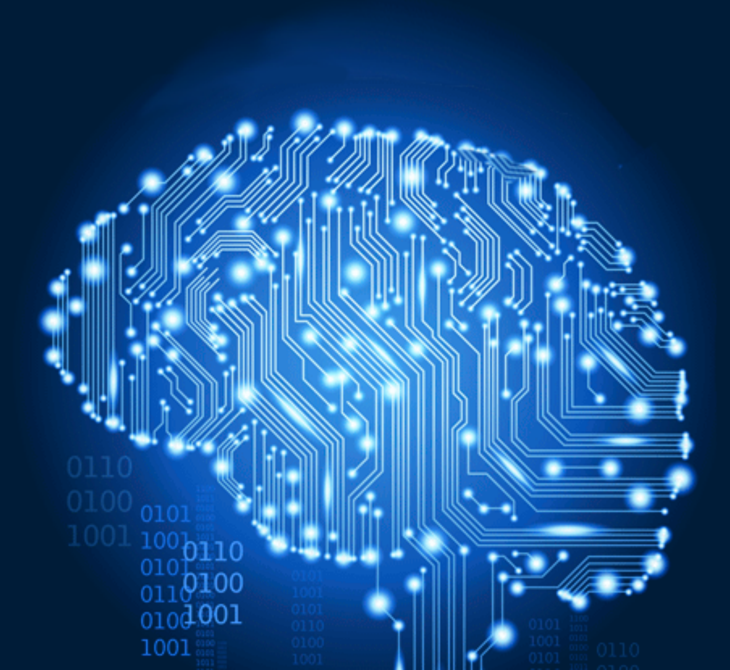
- Subsampling is often employed to decrease model size. With fewer pre-synaptic units providing synapses, it becomes necessary to <u>exaggerate</u> connection density or synaptic conductance. This results in a network with unnaturally few and strong signals circulating, in contrast to the real network, where many weak signals interact. Therefore <u>differences</u> might arise such as artificial synchronization which is a problem especially since synchronization is one of the more important phenomena.

Why Large-Scale Neuronal Networks?

- Improve understanding of brain functionality involving interactions of billions of neuronal and synaptic processes.
- Perform experiments (on a computer) that are impossible (experimentally or ethically) to be done on humans or animals.
- Eventually improve and test hypotheses about complex behaviors:
  Perception • Attention • Learning • Memory • Consciousness • Sleep and wakefulness

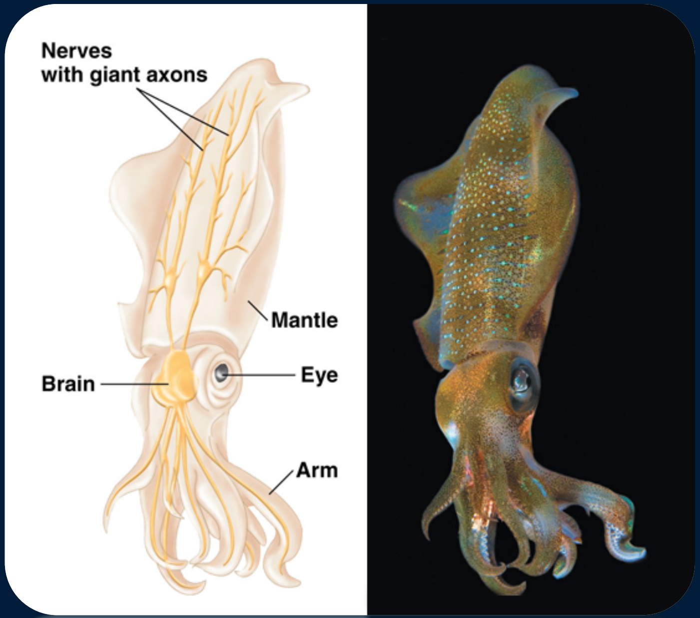
Large models need simple neuron models:

- Integrate-and-Fire types of models are obligatory because of their efficiency
- Izhikevich model is a wise choice because it exhibits a wide range of spiking behaviors and allows about 100 times faster computation runs than Hodkin-Huxley

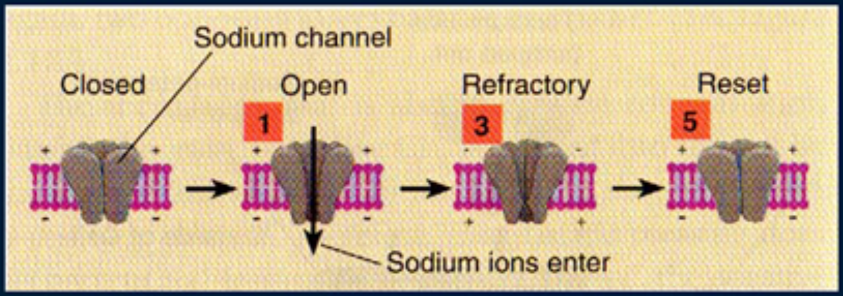


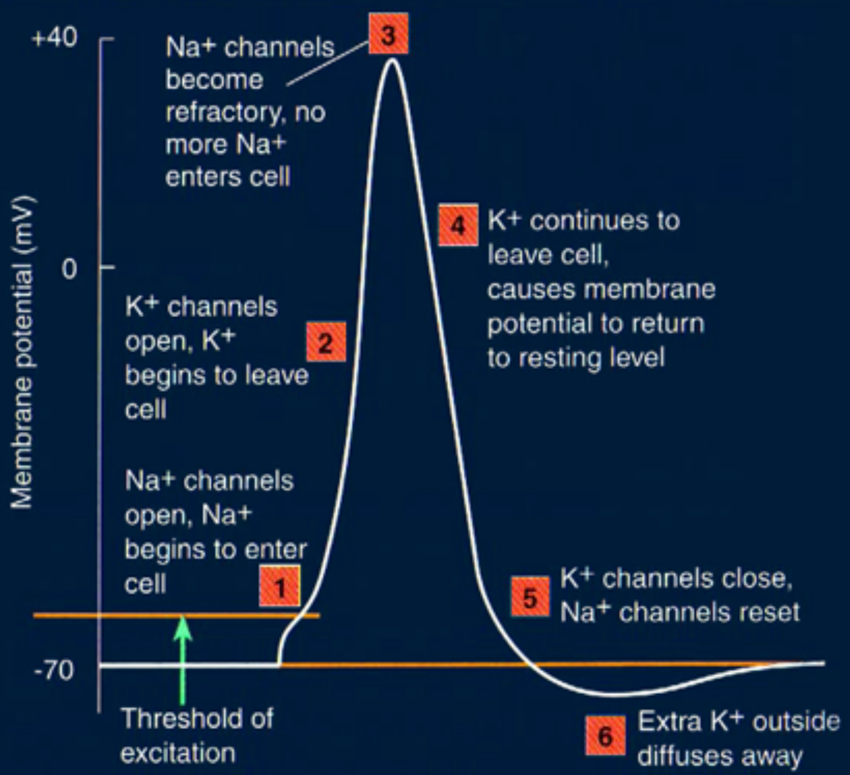
Neuron Models

Hodgkin Huxley



- Proposed model in 1952
- Based on voltage-clamp experiments on the squid giant axon
- Ion channels modeled as resistors and capacitors
- Membrane modeled as capacitor
- Received the 1963 Nobel Prize





Neuron Models — Hodgkin Huxley (cont.)

$$C_m \frac{dV}{dt} = \overbrace{-g_L\left(V - V_L\right)}^{I_L} - \overbrace{\bar{g}_{Na} m^3 h\left(V - V_{Na}\right)}^{I_{na+}} - \overbrace{\bar{g}_K n^4\left(V - V_K\right)}^{I_{K+}}$$

$$\frac{dm}{dt} = \alpha_m(V)(1-m) - \beta_m(V)m$$

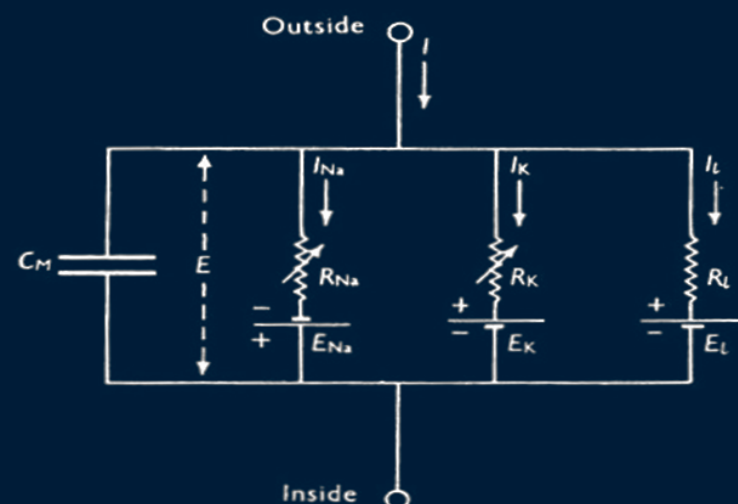
$$\frac{dh}{dt} = \alpha_h(V)(1-h) - \beta_h(V)h$$

$$\frac{dn}{dt} = \alpha_n(V)(1-n) - \beta_n(V)n$$

- Computationally complex
- Defined by 4 differential equations:
  - n controls potassium channel opening
  - m controls sodium channels opening
  - h controls sodium channels closing

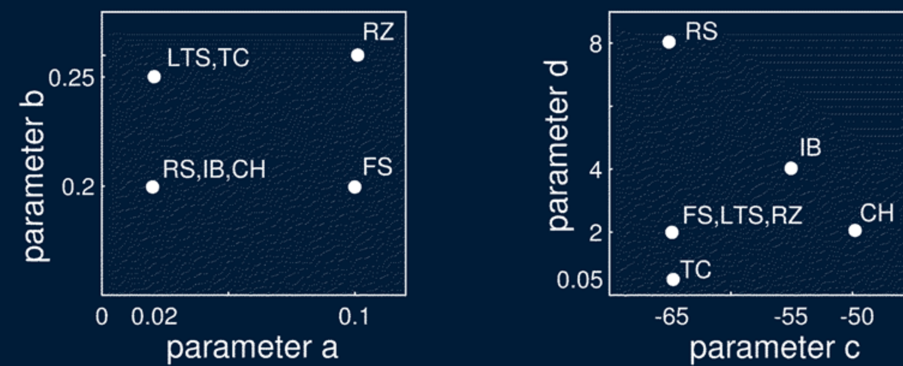
Three major currents:

- voltage-gated persistent K^+ current I_{K+} with 4 activation gates (n_4)
- voltage-gated transient Na^+ current I_{Na+} with 3 activation gates (m^3) and 1 inactivation gate (h)
- Ohmic leak current I_L (carried mostly by Cl^-)



Neuron Models — Izhikevich's

- *v* represents the membrane potential
- *u* represents the membrane recovery variable (a negative feedback to v)
- *I* is a variable that represents synaptic currents



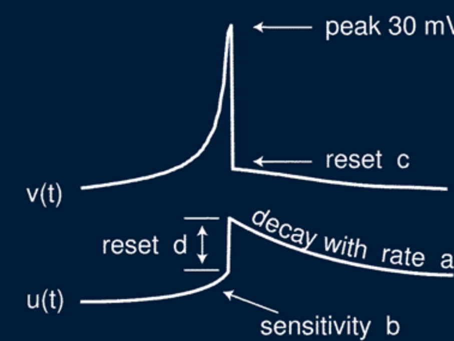
Parameters:

- *a* describes the time scale of the recovery variable u. Smaller values result in slower recovery (typically a a=0.02)
- *b* describes the sensitivity of the recovery variable u to the sub-threshold fluctuations of the membrane potential v (typically b=0.2)
- *c* describes the after-spike reset value of the membrane potential v (typically c=-65mV)
- *d* describes the after-spike reset of the recovery variable u (typically d=2)

$$\frac{dv}{dt} = 0.04v^2 + 5v + 140 - u + I$$

$$\frac{du}{dt} = a(bv - u)$$

$$\text{if } v = 30 \text{ then } v = c$$
$$u = u + d$$

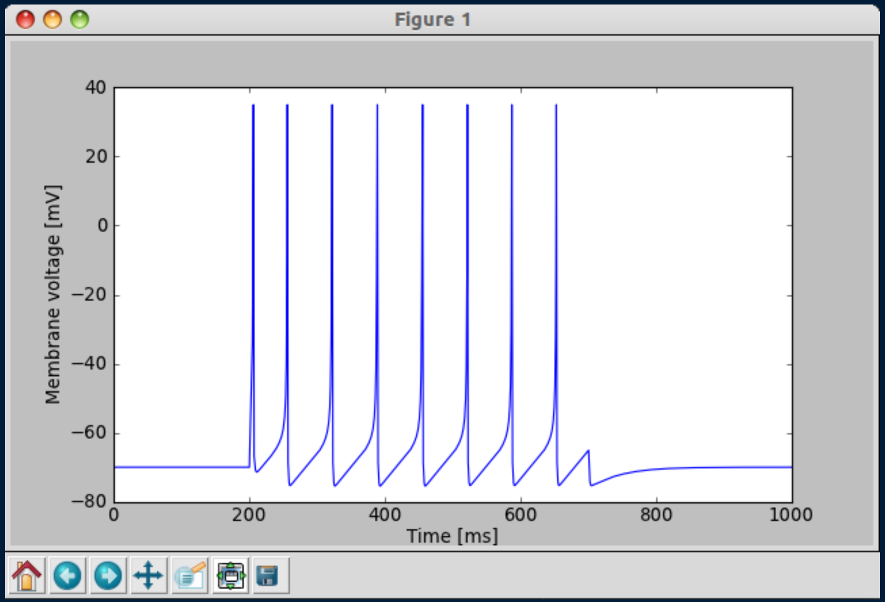


Izhikevich (2003). **Simple model of spiking neurons.** IEEE Transactions on neural networks

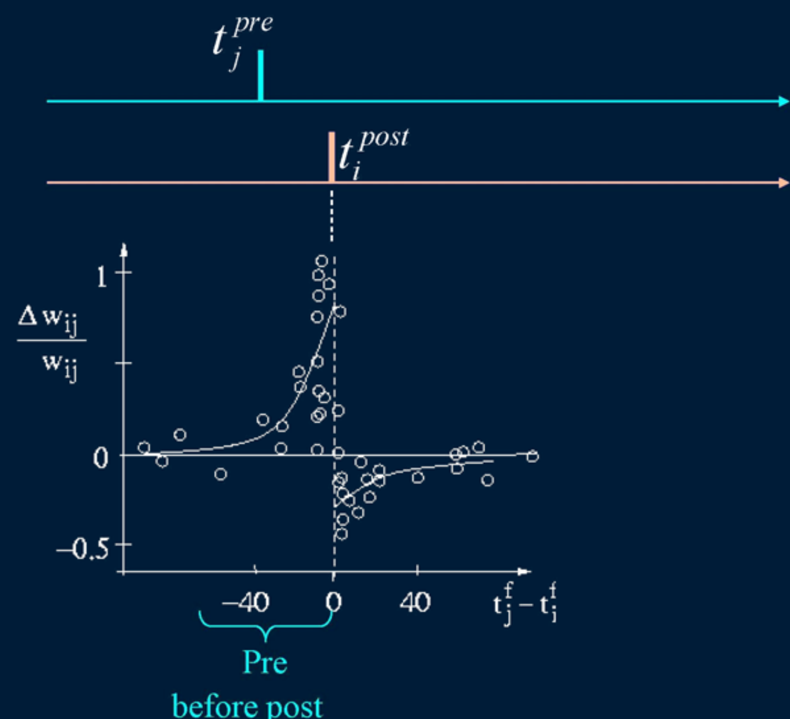
Neuron Models | Izhikevich's | Implementation in Python

```
tmax = 1000.
dt = .5

neuron_model = 'RS'
if neuron_model=='RS':
    a = .02; b = .2; c = -65; d = 8.
elif neuron_model=='IB':
    a = .02; b = .2; c = -55; d = 4.
elif neuron_model=='FS':
    a = .1;  b = .2; c = -65; d = 2.

Iapp = 7
tr = array([200., 700.])/dt

T = ceil(tmax/dt)
v = zeros(T)
u = zeros(T)

v[0] = -70
u[0] = -14

for t in arange(T-1):
    if t>tr[0] and t<tr[1]:
        I = Iapp
    else:
        I = 0
```

```
    if v[t]<35:
        dv = (.04*v[t]+5)*v[t]+140-u[t]
        v[t+1] = v[t]+(dv+I)*dt
        du = a*(b*v[t]-u[t])
        u[t+1] = u[t]+dt*du
    else:
        v[t] = 35
        v[t+1] = c
        u[t+1] = u[t]+d
```



- Spike Timing Dependent Plasticity (STDP) is a temporally asymmetric form of Hebbian learning induced by tight temporal correlations between the spikes of pre- and post-synaptic neurons.
- Repeated pre-synaptic spike arrival a few milliseconds before postsynaptic action potentials leads in many synapse types to long-term potentiation (LTP) of the synapses, whereas repeated spike arrival after postsynaptic spikes leads to long-term depression (LTD) of the same synapse.
- The weight change Δw_j of a synapse from a presynaptic neuron j depends on the relative timing between presynaptic spike arrivals and postsynaptic spikes.
- $t_j^{\,f}$ and $t_i^{\,f}$ count pre-synaptic and post-synaptic spikes respectively:



$$\Delta w_j = \sum_{f=1}^{N} \sum_{n=1}^{N} W(t_i^n - t_j^f)$$

$$W(x) = A_+ \exp(-x/\tau_+) \qquad for \quad x > 0$$
$$W(x) = -A_- \exp(x/\tau_-) \qquad for \quad x < 0$$

Sjöström (2011). **Spike-timing dependent plasticity**. Frontiers in Synaptic Neuroscience

Large-Scale Model | Sequential Code | Implementation in Python

```
from numpy import *
from pylab import *

Ne = 800 #excitory neurons
Ni = 200 #inhibitory neurons
N = Ne+Ni #total neurons
M = 100   #synapses per neuron
D = 20    #maximal conduction delay
sm = 10.0 #maximal synaptic strength

class Izh(object):
   def __init__(self, a, d, v, u, init_s):
      self.a = a
      self.d = d
      self.I = 0
      self.v = v
      self.u = u
      self.post = None
      self.pre = []
      #synaptic weights
      self.s = [init_s] * M
      #derivatives of synaptic weights
      self.sd = [0.0] * M
      #distribution of delays
      self.delays = [[] for i in xrange(D)]
      #STDP functions
      self.LTP = [0.0]*(1001+D)
      self.LTD = 0.0
```

Disclaimer: This code is horribly slow! Though it is much more readable than the original C code, hence the implementation in Python for this presentation.

Izhikevich (2006). Polychronization: computation with spikes. Neural computation

Large-Scale Model | Sequential Code | Implementation in Python (cont.)

```
#population of RS and FS type neurons
neurons = [Izh(.2, 8, -65, -13, 6.0) for i in xrange(Ne)] + \
   [Izh(.1, 2, -65, -13, -5.0) for i in xrange(Ni)]

#wiring neurons to each other
for i, neuron in enumerate(neurons):
   if i<Ne:
      neuron.post = list(permutation(N))[:M]
      for j in xrange(M):
         neuron.delays[randint(D)].append(j)
   else:
      neuron.post = list(permutation(Ne))[:M]
      neuron.delays[0] = range(M)

#find presynaptic neurons to each neuron
for neuron in neurons[:Ne]:
   for d in xrange(D):
      for syn in neuron.delays[d]:
         neurons[neuron.post[syn]].pre.append({'neuron':neuron, 'd':d, 'syn':syn})

firings = [[-D, 0]]
N_firings = 1
```

Large-Scale Model | Sequential Code | Implementation in Python (cont.)

```
for sec in xrange(60*60*24): #1 day
    for t in xrange(1000):    #1 sec
        for neuron in neurons:
            neuron.I = 0 #reset input
        neurons[randint(N)].I = 20 #random thalamic input
        for i, neuron in enumerate(neurons):
            if neuron.v >= 30: #did it fire?
                neuron.v = -65.0
                neuron.u += neuron.d
                neuron.LTP[t+D] = .1
                neuron.LTD = .12
                for pre in neuron.pre:
                    #this spike was after pre-synaptic spikes
                    pre['neuron'].sd[pre['syn']] += pre['neuron'].LTP[t+D-pre['d']-1]
                firings.append([t, i])
                N_firings += 1
            for firing in reversed(firings[:N_firings-1]):
                if t-firing[0] >= D:
                    break
                for syn in neurons[firing[1]].delays[t-firing[0]]:
                    i = neurons[firing[1]].post[syn]
                    neurons[i].I += neurons[firing[1]].s[syn]
                    if firing[1]<Ne:
                        #this spike is before postsynaptic spikes
                        neurons[firing[1]].sd[syn] -= neurons[i].LTD
            for neuron in neurons:
                neuron.v += ((.04*neuron.v+5)*neuron.v+140-neuron.u+neuron.I)
                neuron.u += neuron.a*(.2*neuron.v-neuron.u)
                neuron.LTP[t+D+1] = .95*neuron.LTP[t+d]
                neuron.LTD *= .95
```

```
print 'sec:', sec, 'firing rate:', float(N_firings)/N
   #prepare for the next sec
   for neuron in neurons:
      neuron.LTP[:D+1] = neuron.LTP[1000:1000+D+1]
   k=N_firings-1
   while 1000-firings[k][0]<D:
      k -= 1
   for i in xrange(1, N_firings-k):
      firings[i][0] = firings[k+i][0]-1000
      firings[i][1] = firings[k+i][1]
   N_firings = N_firings-k
   #modify only exc connections
   for neuron in neurons[:Ne]:
      for j in xrange(M):
         neuron.s[j] += .01+neuron.sd[j]
         neuron.sd[j] *= .9;
         if neuron.s[j]>sm:
            neuron.s[j] = sm
         elif neuron.s[j]<0:
            neuron.s[j] = 0.0
```

GPU Based Simulation

- The GPU-SNN model (running on an NVIDIA GTX-280 with 1GB of memory), is up to 26 times faster than a CPU version for the simulation of 100K neurons with 50 Million synaptic connections, firing at an average rate of 7Hz.

- For simulation of 100K neurons with 10 Million synaptic connections, the GPU-SNN model is only 1.5 times slower than real-time.

Challenges:

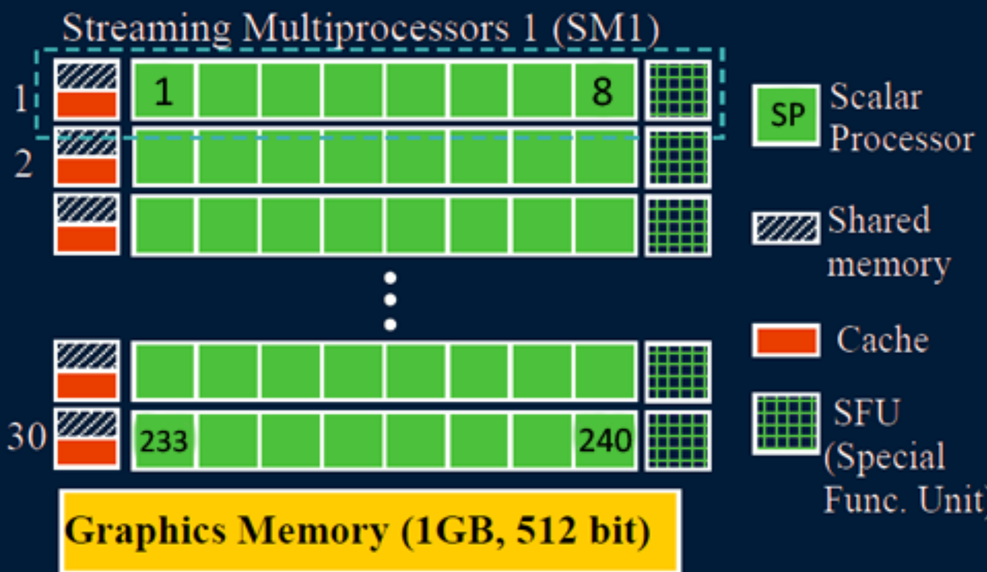
- Effective parallelism to optimize the GPU resources (processors, shared memory and memory bandwidth)
- Effective handling of large fan-in/fan-out connections to neurons
- Efficient usage of limited GPU memory for simulating large networks using sparse representations.

Nageswaran (2009). **Efficient simulation of large-scale spiking neural networks using CUDA graphics processors.** IJCNN 2009

GPU Architecture

- Each Streaming Multiprocessors (SM) consists of eight floating-point Scalar Processors (SPs), a Special Function Unit (SFU), a multi-threaded instruction unit, a 16KB user-managed shared memory, and 16KB of cache memory.
- A single NVIDIA GTX280 GPU card consisting 240 scalar processors grouped into 30 SMs, each operating at 1.2 GHz, is used (350 GFLOPS).

- Each SM has a hardware thread scheduler that selects a group of threads, a.k.a warp, for execution.
- If any one of the threads in the group issues a costly external memory operation, then the thread scheduler automatically switches to a new thread group.



GPU Mapping | Parallelism Analysis

- Neuronal parallelism (N-parallel): Each neuron is mapped on a processing element and computed in parallel. The synaptic computation for each neuron is carried out sequentially on its processing element. This mapping leads to warp divergence and is ineffective for GPUs.

- Synaptic Parallelism (S-parallel): For a given neuron each synaptic connection is updated in parallel by different processing element. Thus synaptic information is distributed over all processing elements. The neuron computation is carried out sequentially. The maximum parallelism is limited by the number of synaptic connection that need to be updated in a given time step.

- Neuronal-Synaptic Parallelism (NS-parallel): Uses both N-parallel and S-parallel techniques but at different stages in the simulation. At each time step where the neuron information needs to be updated, the N-parallel strategy is adopted. Thus, every thread within the GPU updates different neuron information in parallel. Whenever a spike is generated, the S-parallel mapping is deployed where the synapses need to be updated.

Minimizing Impact of Warp Divergence

Large computation within diverging loop

```
i = threadIdx.x + blockIdx.x*blockDim.x
if ( membraneV[i] >= 30.0 ) {
   do_firing (i)        // 100-200 cycles
}                       // repeat for other neurons
```

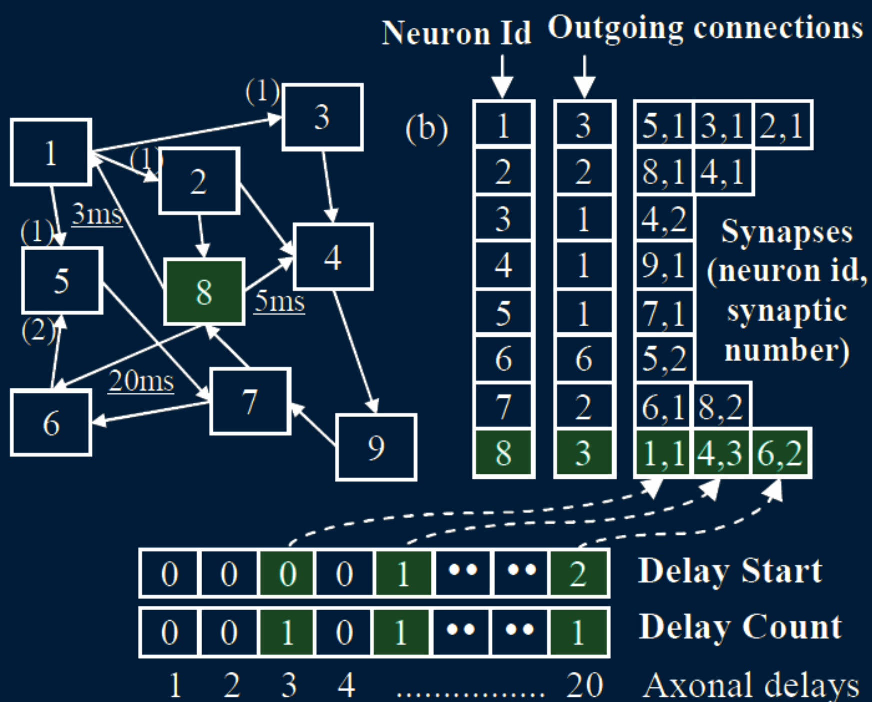
Warp divergence can occur if different threads within the same warp take different paths after a branch condition. If the diverging condition takes a large number of cycles, then other threads in the warp go into a busy waiting mode.

Small computation within diverging loop

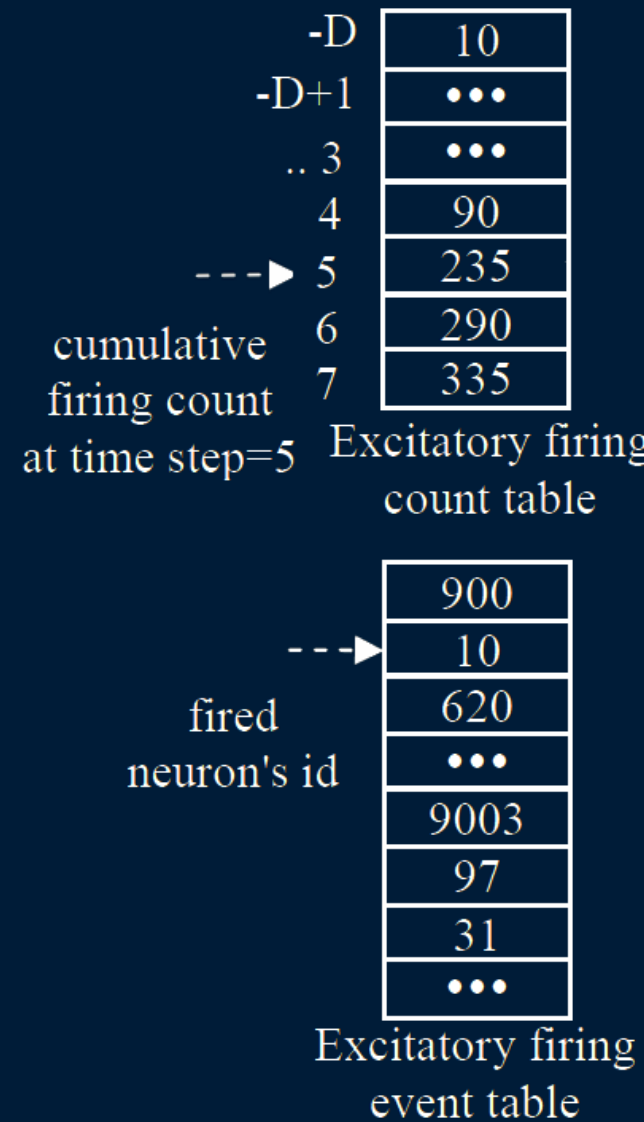
```
k=0;i= threadIdx.x + blockIdx.x*blockDim.x
if ( membraneV[i] >= 30.0 ) {
p=atomicAdd(&k,1);buffer[p]=i //5-10 cycles
} // repeat for other neurons
 __syncthreads();
offset = threadIdx.x;
while (offset < k)
     do_firing (buffer[offset])
     offset=offset+blockDim.x
```

GPU Based Simulation

Data Structure



Normally required memory is O(NMD) but we bring that down to O(N(M+D)) with sparse representation.



Address-Event-Representation (AER)

GPU Based Simulation

Inputs: nid => Neuron Id,
I_fire => Input Fired Vector
s[i][j] => weight of i^{th} neuron, j^{th} synapses
len[i] => number of synaptic connections
Output: I_sum => Total synaptic current
Require: find_one[x] => pre-computed 256 entry table that returns the position of the first set bit in a given byte (e.g. find_one[0x10]=4, find_one[0x77]=0

```
I_sum=0, y_end = ceil(len[i]/32)
for y=0:(y_end-1)
    part_I = read32(I_fire, y)     // Read y^th 32 bit
                                   // from I_fire vector
    x = 0;
    while part_I ≠ 0
       byte_I = byte(part_I, x)    // Read x^th byte
       while byte_I ≠ 0
          idx = find_one[byte_I]
          set byte_I(idx) ← '0'
          I_sum = I_sum + s[nid][y*32+x*8+idx]
       part_I(x) ← 0; x = x+1;
return I_sum
```

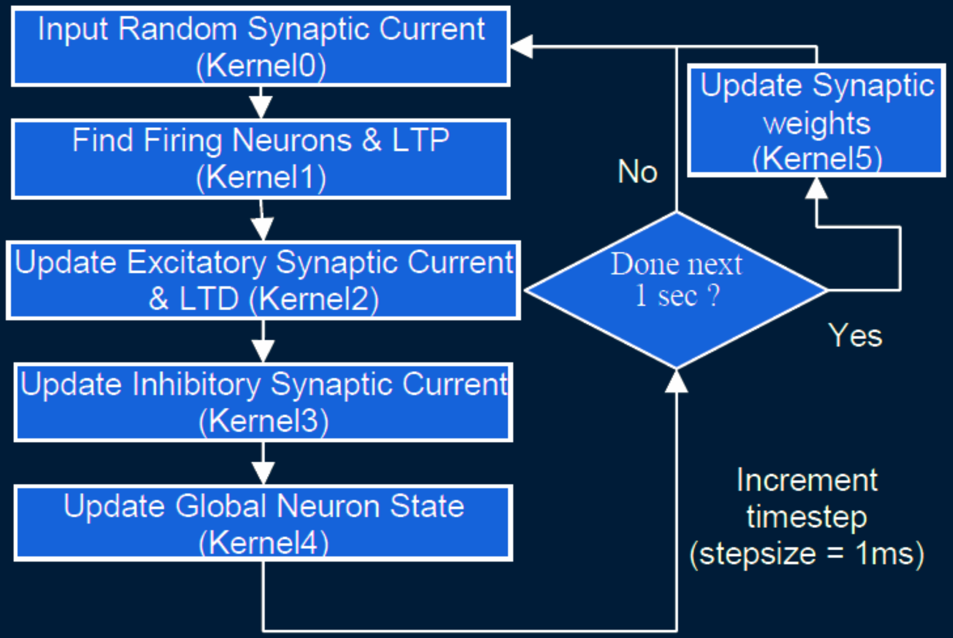
Large Fan-in

- Large fan-ins of each neuron need to be calculated concurrently. Updating the synaptic current of post-synaptic neurons atomically is infeasible since due lack of atomic floating point operations in GPUs.
- Bit vector I_fire represents the input firing status of each neuron, whose up to 2-3 bits are set. The algorithm first scans the I_fire at the word level, then at the byte level and finally at bit level.
- If no bit is set, this approach incurs a small overhead of about 8 instruction cycles. This approach is memory efficient and has low computation overhead for a moderate number of input connections.

GPU Based Simulation	Simulation Flow

- For each kernel, various block sizes are tested in range of 30 to 120, with 128 threads in each block. The change in performance was very small for block sizes greater than 60.

- The network consist of N randomly connected excitatory (80%) and inhibitory (20%) neurons. For our experiments N ranges from 50K to 225K neurons. Number of synapses per neuron, M, ranges from 100 to 1000. The amount of change in the synaptic weight (using STDP rule) is accumulated during each time step; and the weight is updated once a second by Kernel5 (Figure 7) such that synaptic weight changes at a slower-rate than the neurons.



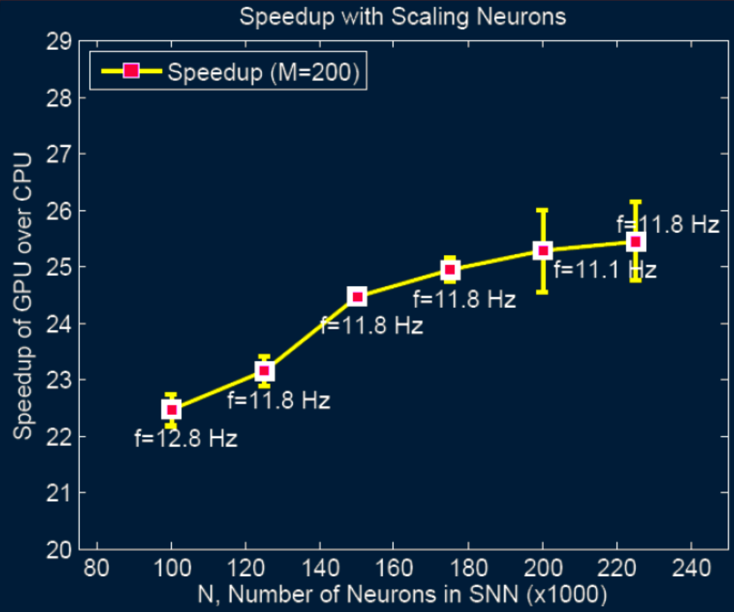
Results

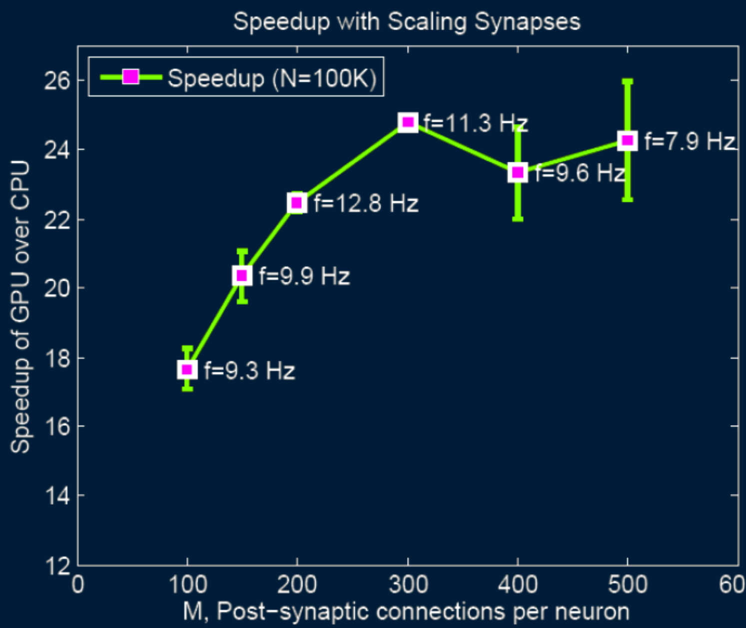
Scalability Analysis

- The speedup curves were obtained by dividing the time taken by the CPU only mode and GPU mode for simulating 10 seconds of model time (10,000 times steps with 1ms resolution) from the steady condition.

- We can observe that the overall speedup does not vary significantly for various values of N (N>105). The variation in the speedup curve is mainly due to the variation in the firing rate. An increase in the firing rate causes slight improvement in the speedup.

- For M=100 and N=105, the speedup is limited to 18. The GPU takes 15 seconds to simulate 10 seconds of model time. For larger values of M the speedup jumps from 18 to around 25 due to increases in the available synaptic parallelism.





Results | Fidelity Test

- The GPU model differs from the reference model in the following ways: implementation of STDP calculations, network representation, firing information representation, etc.
- Thus direct comparison is difficult because the SNN state can change significantly even if one spike is altered.
- To ensure the accuracy and fidelity of GPU implementation various neuronal metrics are considered:
  - difference in average firing rate
  - difference in the synaptic weights of excitatory connections
  - difference in the inter-spike intervals (ISI) for excitatory neurons and for inhibitory neurons

Metrics	N=1000, M=100	N=3000, M=100
Synaptic Weights	0.992	0.099
ISI (Excitatory)	0.799	0.144
ISI (Inhibitory)	0.677	0.261

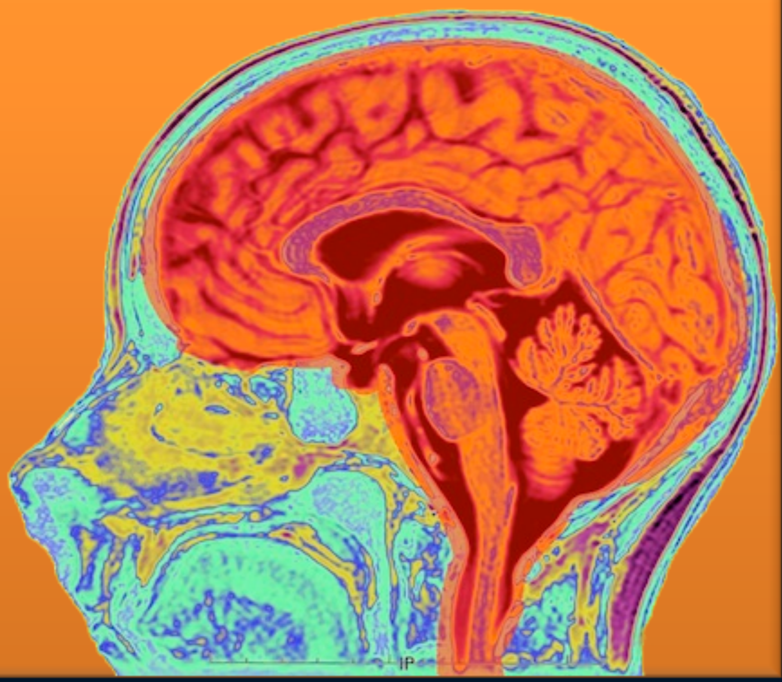
Comparison of distribution of synaptic weights and ISI

Firing Rate Metrics (Hz)	N=1000, M=100		N=3000, M=100	
	Matlab	GPU	Matlab	GPU
Excitatory Neurons	3.1423 (0.4934)	3.1693 (0.7034)	3.8242 (0.1413)	3.3855 (0.0843)
Inhibitory Neurons	24.95 (3.3683)	22.0345 (4.5293)	31.5863 (1.0140)	24.9593 (0.5713)

Comparison of firing rate between MATLAB and GPU implementations

Thanks for your attention :)

Any Questions?!



Contact me at nima.mohammadi@ut.ac.