LARGE-SCALE SIMULATION OF NEURAL NETWORKS WITH BIOPHYSICALLY ACCURATE MODELS ON GRAPHICS PROCESSORS

A Thesis

by

MINGCHAO WANG

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

May 2012

Major Subject: Electrical Engineering

Large-Scale Simulation of Neural Networks with Biophysically

Accurate Models on Graphics Processors

LARGE-SCALE SIMULATION OF NEURAL NETWORKS WITH BIOPHYSICALLY ACCURATE MODELS ON GRAPHICS PROCESSORS

A Thesis

by

MINGCHAO WANG

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Chair of Committee,	Peng Li
Committee Members,	Gwan S. Choi
	Samuel Palermo
	Yoonsuck Choe
Head of Department,	Costas N. Georghiades

May 2012

Major Subject: Electrical Engineering

ABSTRACT

Large-Scale Simulation of Neural Networks with Biophysically
Accurate Models on Graphics Processors. (May 2012)
Mingchao Wang, B.S., Zhejiang University
Chair of Advisory Committee: Dr. Peng Li

Efficient simulation of large-scale mammalian brain models provides a crucial computational means for understanding complex brain functions and neuronal dynamics. However, such tasks are hindered by significant computational complexities. In this work, we attempt to address the significant computational challenge in simulating large-scale neural networks based on the most biophysically accurate Hodgkin-Huxley (HH) neuron models. Unlike simpler phenomenological spiking models, the use of HH models allows one to directly associate the observed network dynamics with the underlying biological and physiological causes, but at a significantly higher computational cost. We exploit recent commodity massively parallel graphics processors (GPUs) to alleviate the significant computational cost in HH model based neural network simulation. We develop look-up table based HH model evaluation and efficient parallel implementation strategies geared towards higher arithmetic intensity and minimum thread divergence. Furthermore, we adopt and develop advanced multi-level numerical integration techniques well suited for intricate dynamical and stability characteristics of HH models. On a commodity CPU card with 240 streaming processors, for a neural network with one million neurons and 200 million synaptic connections, the presented GPU neural network simulator is about 600X faster than a basic serial CPU based simulator, 28X faster than the CPU implementation of the proposed techniques, and only two to three times slower than the GPU based simulation using simpler spiking models.

ACKNOWLEDGMENTS

I would like to thank my committee chair, Dr. Li, and my committee members, Dr. Choi, Dr. Palermo and Dr. Choe, for their guidance and support throughout the course of this research. Thanks also to the other members in our research group, especially Boyuan Yan, Yong Zhang and Jingzhen Hu, for their help and encouragement. Finally, thanks to my mother, father and my wife for their patience and love.

TABLE OF CONTENTS

Page

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

FIGURE Page

CHAPTER I

INTRODUCTION

Computer simulation is a critical enabler for understanding complex functions and neuronal dynamics of mammalian brains. Brain behaviors are controlled by large networks of neurons. In a neural network, the information is processed in the form of electrical signals, which are passed from one neuron to another via synaptic connections. When coupled with experimental studies, numerical simulation of large scale nervous systems can provide profound understanding of brain functions, offering insights that are practically or ethically impossible to acquire purely through experimental means [26]. Due to the sheer complexities, efficient computational techniques, capable of simulating large neural networks with biophysically accurate neuron models, are highly desirable. Such capability will fundamentally enable the test of hypotheses of neurological disorders and development of therapeutic treatments, as well as stimulate new engineering applications [1, 11, 16].

Recently, brain-scale simulation of the neocortex has been demonstrated on supercomputers and/or large computer clusters in several research efforts [1,11,16,24]. The compute power of thousands of processors or more is utilized to cope with the computational challenges in simulating large neural networks.

On the other hand, the emergence of passively parallel single-instruction-multiple-data (SIMD) graphics processors, or GPUs, has drawn a significant interest in the computing community (see for example [21]). In 1965, Gordon Moore, the co-founder of Intel, foresaw that every year in the future, the transistors on an integrated circuit would be twice as many in number. [19]. Fig. 1 shows plot of CPU transistor counts against dates of introduction. History has proven the trueness of this prediction. However, there's a limitation

---

The journal model is *IEEE Transactions on Automatic Control.*

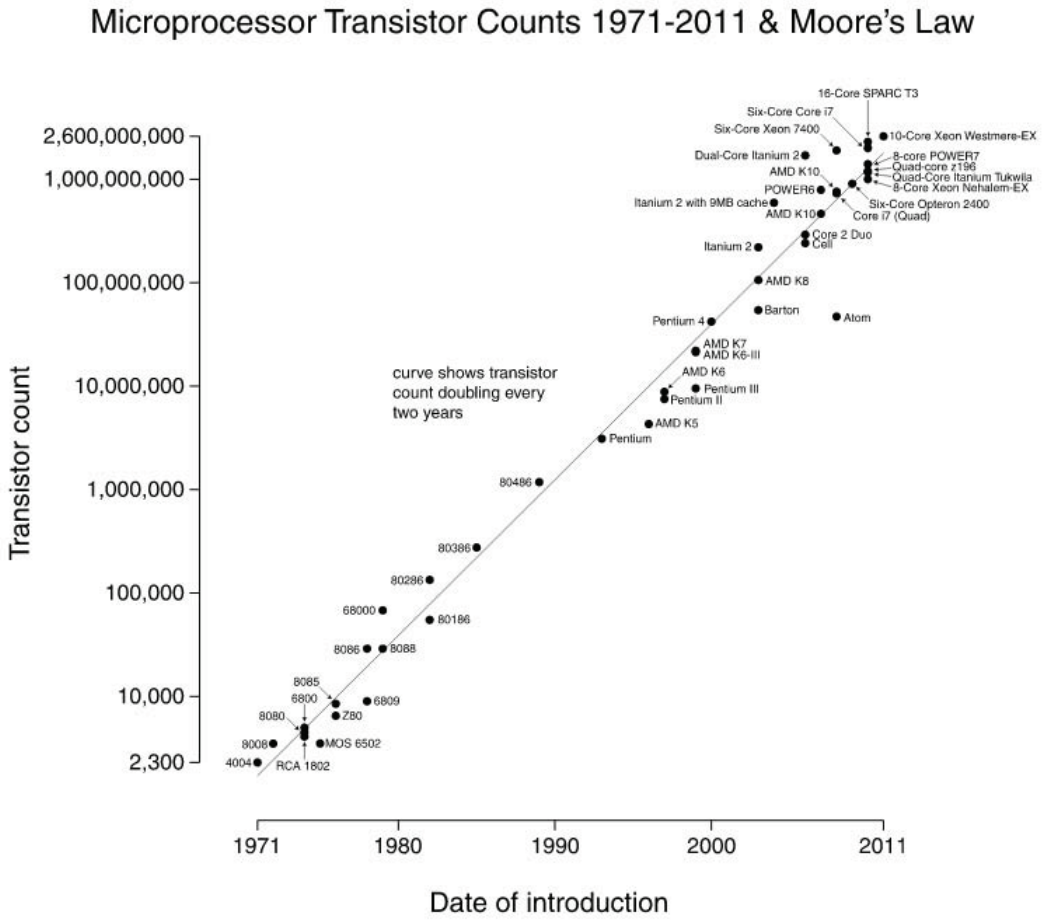

Fig. 1. Plot of CPU transistor counts against dates of introduction [25].

with one or more traditional CPUs. The solution to this problem is assigning multiple cores (Processing unit) to the same processing chip, which gives the field of Graphic Processor Units (GPU) a huge advantage. During the past twenty years, GPU has shown great improvements in performance and capability while developing the large parallelism intrinsic in graphics processing. Due to the evolution of GPU, the consumption of transistors has increased dramatically, even exceeded the requirement of Moore's Law [8, 19]. There have been early attempts to use GPUs for neural network simulation. In particular, in [20] a GPU based neural simulator using simple neuron models is demonstrated. In [4], the multi-core and GPU platforms are compared for neural network simulation using simple two-level network examples.

Most of these works have dealt with the simulation of spiking neuron networks (SNNs) based upon phenomenological spiking neuron models developed by Izhikevich [16]. A spiking neuron compartment model consists of only two differential equations. Its simplicity makes it attractive for large-scale neural simulation where computational complexity is a significant challenge. In comparison, the well established Hodgkin-Huxley (HH) neuron models [15] are more complex; a compartment model consists of four differential equations and a larger number of parameters. While being more biophysically plausible, the use of HH type neuron models can make the network simulation 100X more expensive.

In this work, we attempt to address the significant computational challenge in simulating large-scale neural networks based on the most biophysically accurate Hodgkin-Huxley (HH) neuron models. The use of HH models provides crucial benefits. With essential neuron characteristics such as membrane potentials, ion channel currents and gating of ion channels modeled on a biophysical basis, one may be able to directly associate the observed network dynamics with the underlying biological and physiological causes. From a computational perspective, we exploit recent commodity massively parallel graphics processors (GPUs) to alleviate the significant computational cost in HH model based neural network

simulation.

Simulating large networks of HH neuron models on GPUs presents interesting challenges and opportunities. Special care must be taken in algorithm development and code implementation in order to fully take the advantages of SIMD GPU compute power. The evaluation of HH models is significantly more complex than the spiking model counterparts and stresses the GPU implementation. We develop look-up table based HH model characterization and efficient evaluation on GPU to alleviate such cost. Apart from this, the intricacy of complex nonlinear dynamics captured in HH models introduces further computational overhead; to maintain time-domain simulation accuracy and stability, it is observed that on average a 25X smaller times step size must be used in the HH model based simulation compared with its spiking model counterpart. To address this problem, we develop efficient parallel implementation strategies geared towards higher arithmetic intensity and minimum thread divergence, critical for the GPU implementation. Furthermore, we adopt and develop advanced multi-level numerical integration techniques well suited for intricate dynamical and stability characteristics of HH models, which leads to reduction of effective step size.

On a commodity NVIDIA Tesla GPU card with 240 streaming processors, for a set of neural network examples, the presented GPU neural network simulation approach is about 600X faster than a basic serial CPU based simulator, 28X faster than the CPU implementation of the proposed techniques, and is only two to three times slower than the GPU based simulation using spiking models.

CHAPTER II

BACKGROUND

Brain activities are controlled by large networks of neuron cells which process information in the form of electrical signals passed from one neuron to another via synaptic connections. To describe this process, models with different complexities have been proposed in the past several decades, to simulate different neural spiking patterns and synaptic connections behaviors.

A. Brain Structure

Human's central nervous system locates at the most complex organ-the human brain. All of human's thoughts, emotions, feelings and experiences come from the brain. The brain is formed with more than one hundred billion neurons (nerve cells). Each neuron can communicate directly with nearly ten thousand others in electrical and chemical ways. Among different pairs of neurons, there are connection gaps that are called synapses. A human brain contains more than one quadrillion synaptic connections. The complex structure of a human brain makes it possible to store huge amount of information. Fig. 2 shows the six different structures of a human brain: brainstem, cerebellum, frontal lobe, occipital lobe, parietal lobe and temporal lobe.

Neurons play the most important role in the brain. The uniqueness of neurons is that they can reach nearby or distant target cells by sending signals via an axon [17]. An axon, which usually has many branches, is a thin protoplasmic fiber that extends from the cell body to other areas. Synapses are specialized junctions that work as the media of axons transmission of signals to other neurons. The synaptic connections between an axon and other cells can be thousands. When an action potential travels along an axon and arrives at a synapse, a chemical called a neurotransmitter will be released.

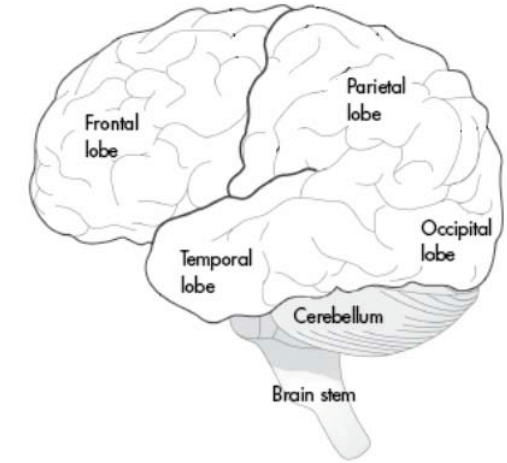

Fig. 2. The human brain structures.

Synapses are the essential functional elements of the brain. They provide the points at which brain's cell-to-cell communication occurs. There are several synapses functions: the excitatory action (excite the target cell), the inhibitory action, and the activation of second messenger systems that change the internal chemistry of their target cells in complex ways [23].

B. Spiking Neural Models

In this section, different spiking models are presented, ranging from the straightforward integrate-and-fire (I&F) model [22] to the most biologically plausible Hodgkin-Huxley (H-H) models [15].

1. Leaky Integrate-and-Fire Model

Detailed conductance-based neuron models are difficult to analyze because of their intrinsic complexity. Therefore, phenomenological spiking neuron models are widely used in computational science for studies of neural network dynamics. Leaky integrate-and-fire

(I&F) neuron model is one of the most popular phenomenological models. The standard form of Leaky I&F model is shown in the following equation [22]

$$\dot{v} = I + a - bv \tag{2.1}$$

where variable v is the membrane potential, I is the input current, and a, b are parameters. When the membrane potential v reaches a threshold criterion, the potential is reset to a new value according to

$$v = c \tag{2.2}$$

where c is a model parameter.

The I&F neuron model is one of the simplest models to implement. However, because the I&F model has only one variable and fixed threshold, it cannot simulate many neuronal spike properties, such as bursting, rebound response, and latencies. Some advanced I&F models, such as nonlinear I&F model, I&F with Adaptation, are proposed but they have their own limitations and are not able to simulate most of the neuron spiking patterns [22]. Those I&F models will not be presented in this article.

2. Izhikevich Spiking Neural Models

Recently, phenomenological spiking neuron models proposed by Izhikevich have been widely used for large-scale neuron network simulation [16, 22]. In Izhikevich models, spiking dynamics of each neuron can be described by the following two differential equations

$$\begin{aligned} \dot{\mathrm{v}} &= 0.04\mathrm{v}^2 + 5\mathrm{v} + 140 - u + I \\ \dot{u} &= a(b\mathrm{v} - u) \end{aligned} \tag{2.3}$$

where variable v represents the membrane potential of the neuron, I represents the synaptic input current, u represents a membrane recovery variable, and (a, b, c, d) are the four

dimensionless model parameters. When the membrane potential ν reaches 30 mV, the membrane voltage and the recovery variable are reset according to

$$\begin{aligned} \nu &= c \\ u &= u + d \end{aligned} \tag{2.4}$$

The model can exhibit firing patterns of all known types of cortical neurons with the suitable choice of parameters (a, b, c, d).

While highly appreciated by the simplicity, Izhikevich models have limitations in applications. Although they can reproduce typical firing phenomena of cortical neurons, their parameters are not necessarily biophysically meaningful. In order to directly associate the observed network dynamics with the underlying biological and physiological causes, the classical Hodgkin-Huxley type models are needed.

3. Hodgkin-Huxley Models

The equivalent circuit of Hodgkin-Huxley model [15] is shown in Fig. 3. The behavior of an electrical circuit can be described by a differential equation of the form

$$c\dot{\nu} = -g_L(\nu - E_L) + I_{ion} + I_{syn} \tag{2.5}$$

where ν is the intracellular potential (membrane potential), c is the membrane capacitance, g_L is the leakage conductance, E_L is the leakage reversal potential, I_{ion} is the ionic current flowing across the membrane, and I_{syn} is the input synaptic current.

The total ionic current I_{ion} is the sum of individual contributions from all participating ion types. For example, for regular-spiking pyramidal cells [9], there are three ion types: a sodium current I_{Na}, a potassium current I_K, and a slow voltage-dependent potassium current

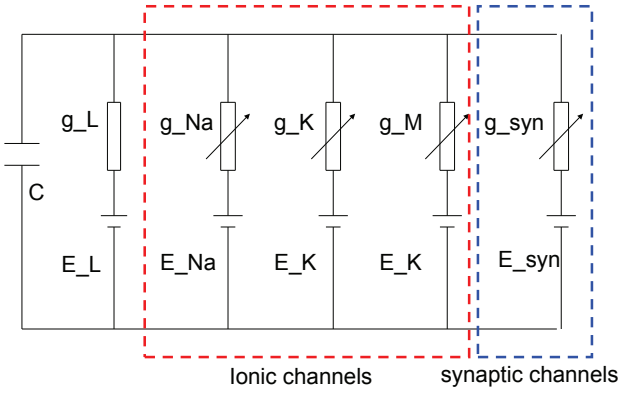

Fig. 3. Equivalent circuit of HH model.

I_M , which can be described by the following equations

$$\begin{aligned} I_{Na} &= g_{Na} m^3 h(\nu - E_{Na}) \\ I_K &= g_K n^4 (\nu - E_K) \\ I_M &= g_M p(\nu - E_K) \end{aligned} \tag{2.6}$$

where g_{Na}, g_K, g_M are maximum conductance values, E_{Na}, E_K are reversal potentials, and m, h, n, p are gating variables describing the probability that an ion channel is open. The gating variables evolve as follows

$$\begin{aligned} \tau_m \dot{m} &= -m + m_\infty \\ \tau_h \dot{h} &= -h + h_\infty \\ \tau_n \dot{n} &= -n + n_\infty \\ \tau_p \dot{p} &= -p + p_\infty \end{aligned} \tag{2.7}$$

where the steady-state values m_∞, h_∞, n_∞, p_∞ and the time constants τ_m, τ_h, τ_n, τ_p are dependent on the membrane potential ν.

Compared with Izhikevich models, it is easy to see that Hodgkin-Huxley type models

are much more computationally expensive. However, as shown in the following sections, with suitable parallel computing and advanced numerical integration techniques, such computational gap can be narrowed dramatically by taking advantage of the properties of the cortex neuron network.

C. Synaptic Models

The electrical or chemical transmission between neuron cells relies on the structure of a synapse in the nervous system. When an action potential is generated at synapses near the cell body, it will propagate down the axon and reach the connected neuron cell. Neuronal function cannot perform without synapses, because synapses are the means by which neurons can pass signals to individual target cells [3]. In this work, alpha-function synapse model is used.

The total synaptic current at each compartment is simulated as

$$I_{syn} = g_{AMPA}(v-E_{AMPA})+g_{NMDA}(v-E_{NMDA})+g_{GABAA}(v-E_{GABAA})+g_{GABAB}(v-E_{GABAB}) \quad (2.8)$$

where v is the postsynaptic membrane potential, and the subscript indicates the receptor type. E is reverse potential of each receptor type. Each conductance g has first-order linear kinetics

$$g = \frac{t}{\tau}e^{-\frac{t}{\tau}} \quad (2.9)$$

where time constant τ specifies the duration of the response and can be used to distinguish for instance between fast and slow transmission.

D. Numerical Methods for Ordinary Differential Equations

Numerical ordinary differential equations is the part of numerical analysis which studies the numerical solution of ordinary differential equations (ODEs) [5,14,18]. Analytical solution

of many differential equations are hard to compute. However, in science and engineering, analytical solution is not necessary to solve a problem since a numerical approximation to the solution is often accurate enough.

The problem can be described as follow. We want to compute the approximation of solation of the differential equation

$$y'(t) = f(t, y(t)), \quad y(t_0) = y_0 \tag{2.10}$$

where f is a function and the initial condition y_0 is a given vector.

In this section, three elementary methods will be studied which can be used to compute a numerical approximation of the solution of the differential equation above.

1. The Forward Euler Method

We denote the time at the nth time-step by t_n and the computed solution at the nth time-step by y_n. Given (t_n,y_n), the forward Euler method (FE) computes y_{n+1} as

$$y_{n+1} = y_n + hf(t_n, y_n) \tag{2.11}$$

where h is the step size which is given by h=t_n-t_{n-1}. A simple example of forward Euler approximation is shown in Fig. 4. The forward Euler method is an explicit method, i.e., y_{n+1} is given explicitly as known quantities such as y_n and $f(y_n, t_n)$. Despite of the easy implementation of explicit methods, they have limitations on the time step size to ensure numerical stability.

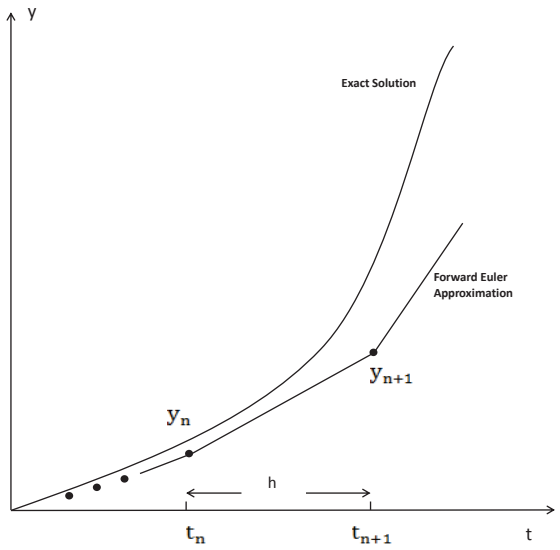

Fig. 4. An example of forward Euler method.

2. The Backward Euler Method

The implicit analogue of the explicit FE method is the backward Euler (BE) method. The backward Euler method computes y_{n+1} as

$$y_{n+1} = y_n + hf(y_{n+1}, t_{n+1}) \tag{2.12}$$

Note that $f(y_{n+1}, t_{n+1})$ is unknown, therefore it shows us an implicit equation to compute y_{n+1}. Fixed point iteration or the Newton-Raphson method is commonly used to accomplish this. However, the iteration methods often cost more time to solve this equation. The implicit methods have a benefit in stability when solving a stiff equation, so a larger step can be used.

3. The Exponential Euler Method

If the differential equation is of the form

$$y'(t) = A - By(t) \tag{2.13}$$

then an approximation of solution can be given by

$$y_{n+1} = y_n e^{-Bh} + \frac{A}{B}(1 - e^{-Bh}) \tag{2.14}$$

The exponential Euler method is numerically stable but slower than the forward Euler method [2].

In our work, the forward Euler method is used, because comparing to the other two methods, the forward Euler method is much easer to implement and faster to simulate the neural network. However, the limitation of forward Euler method is obvious, that a large time step needs to be used in order to ensure numerical stability. In the next several sections, we will show you how different techniques and approaches are adopted to narrow this computational gap to take advantage of the forward Euler method.

CHAPTER III

GPU ARCHITECTURE

Single-core microprocessor is facing limitations on performance growth due to the semiconductor technology scaling limits, power and thermal challenges. Parallel computing became popular and is developing very quickly. New parallel computing platforms such as GPUs and multi-core CPUs have become the domination of the market. In the past decade, the competition in floating-point performance has been led by a group of many-core processors named Graphics Processing Units (GPUs). The GPU, as a modified stream processor, is increasingly commonly used, because of its massive parallelism, high memory bandwidth, and general purpose instruction sets, including support for both single- and double-precision IEEE floating point arithmetic [7].

Performance growth for single-core microprocessors is becoming more and more limited because of semiconductor technology scaling limits, power and thermal challenges, and difficulty of exploiting greater levels of instruction level parallelism. Parallel computing became popular and is developing very quickly. New parallel computing platforms such as GPUs and multi-core CPUs have come to dominate the market. In the past decade, a class of many-core processors called Graphics Processing Units (GPUs), have led the race for floating-point performance. And because of its massive parallelism, high memory bandwidth, and general purpose instruction sets, including support for both single- and double-precision IEEE floating point arithmetic, as a modified form of stream processor, it is becoming increasingly common to use a general purpose graphics processing unit. This concept turns the massive floating-point computational power of a modern graphics accelerator's shader pipeline into general-purpose computing power, as opposed to being hard wired solely to do graphical operations.

A. GPU Architecture

The NVIDIA's Compute Unified Device Architecture (CUDA) is designed to support both graphics and general purpose computing. The programmable processing elements, which use a general-purpose instruction set, are built around a scalable array of multi-threaded Streaming Multiprocessors (SMs) [21]. In our experiments, we use a commodity NVIDIA Tesla C1060 GPU card, which consists of 240 streaming scalar processor cores grouped into 30 streaming multiprocessors (SMs), each operating at 1.3 GHz. Each SM consists of eight scalar processor (SP) cores, two special function units for transcendental and other special functions, a multi-threaded instruction unit, and on-chip shared memory. This GPU card has 4 GB global memory, 64 KB constant memory, and 16 KB shared memory. The system provides a maximum bus bandwidth of over 100 GB/s. A simplified GPU architecture is shown in Fig. 5.

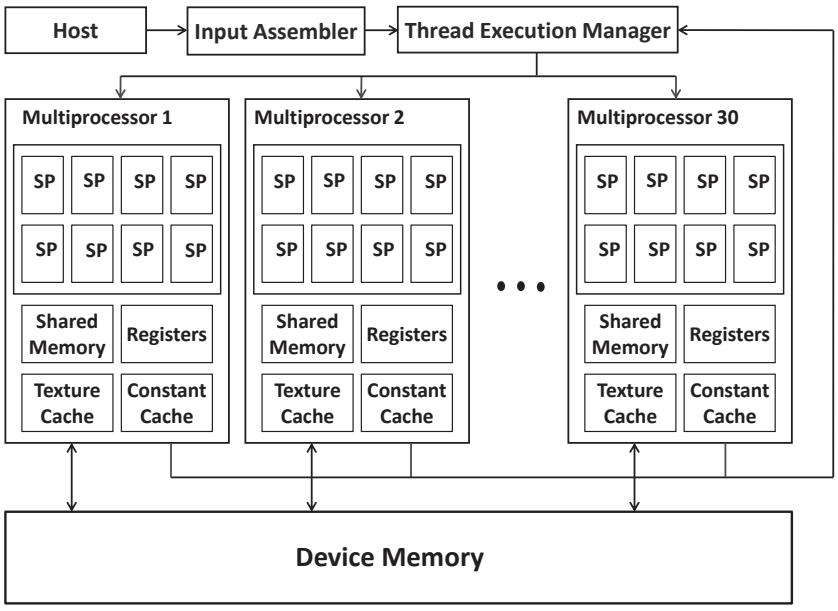

Fig. 5. CUDA architecture.

To a CUDA programmer, the computing system consists of a host and one or more

devices. Host is a traditional CPU, such an Intel Architecture microprocessor in personal computers today, and devices are the massively parallel processors on the GPUs. For a typical CUDA program, when the data is ready in the host memory, they are transferred to the memory of graphic cards, then processed by the program in the device. After it finishes, the results will be transferred back to the host.

CUDA programming is done with data-parallel functions, called kernels or grids. These kernels describe the work of a single thread and typically are invoked on thousands of threads. A grid consists of many blocks which have the same program and each block consists of a large number of threads. A figure of a grid of thread blocks is shown in Fig. 6. Within each block, the threads can share their data and synchronize their actions. Threads in different blocks cannot access the same memory block at the same time, which means they cannot communicate and synchronize with each others. The multiprocessor maps each thread to one scalar processor core, and each scalar thread executes independently with its own instruction address and register state. Multiprocessors schedule threads on the basis of 32-thread groups called warps.

B. CUDA Programming

Compute Unified Device Architecture (CUDA) [21] is a parallel computing architecture developed by Nvidia. CUDA programming language is an extension of standard C language with a few additions to the C syntax. For developers, the GPUs could be accessed for computation like CPUs, even thought GPUs' architecture is not like CPUs. GPUs' parallel throughput architecture is to execute many threads slowly but concurrently. As the GPUs has hundreds of cores, with proper implementation, the speedup of CUDA program could be dramatically high, even each core is much slower than CPU core.

GPU computing has several advantages over traditional CPU parallel computing. First,

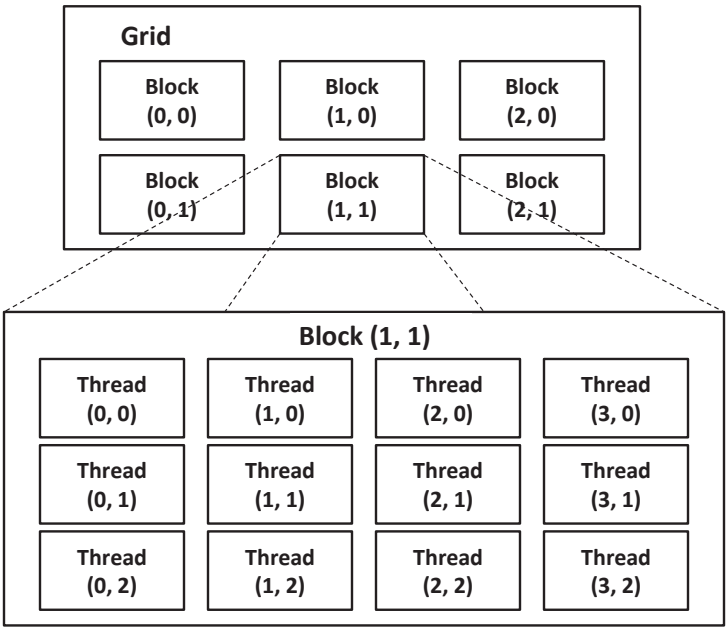

Fig. 6. Grid of thread blocks.

normally graphic cards have large bandwidth. For example, NVIDIA GeForce 8800GTX has more than 50GBs memory bandwidth, while recent CPUs mostly have only 10GBs bandwidth. Second, graphic cards have much more processor cores than CPU. The latest NVIDIA GPUs can have more than 256 cores. Third, comparing to expensive CPUs, GPUs is more affordable for personal use.

However, GPUs have limitations. CUDA programming is not suitable for tasks which cannot be highly parallelized. And recent GPUs have very limit support for double-precision IEEE floating point arithmetic. Also, GPUs normally have no complex conditional branch control unit. Therefore, CUDA program is not good at dealing with conditional branch as warp divergence could occur.

Several basic rules and principles should be followed when developing general-purpose GPU computing applications [4, 21]:

1. A good way to manage global memory latency is to create sufficient number of ac-

tive threads to keep SPs occupied while other threads are waiting on global memory accesses.

2. In order to achieve peak memory bandwidth, various access strategies should be taken for different memory types. For the global memory, peak memory bandwidth is achieved when accesses by threads in a half warp are coalesced. For the shared memory, bank conflicts, caused by multiple threads accessing the same bank, should be avoided. The texture memory is read only, but cached with a significantly smaller latency than the global (device) memory. Therefore, it can be used for efficient storage of constant values.

3. Conditional branches should be avoided to minimize warp divergence so as to avoid serialization. If a warp divergence is unavoidable, tasks within each branch should be minimized. Some simple programming tricks could help in these cases, such as regrouping the tasks to balance the work load. Arithmetic intensity (amount of calculation relative to memory access) should be maximized to effectively reduce memory latency.

CHAPTER IV

CORTICAL NETWORK SIMULATION

A. The Multi-Scale Behaviors of Cortex Networks

A portion of the cortex network is illustrated in Fig. 7. The tree-like structures represent pyramidal cells and the ball-like structures represent interneurons. In this figure, there are six pyramidal cells and only one interneuron. The typical firing pattern of both pyramidal neurons and interneurons are shown in the figure. Generally speaking, interneurons have larger frequency than pyramidal neurons.

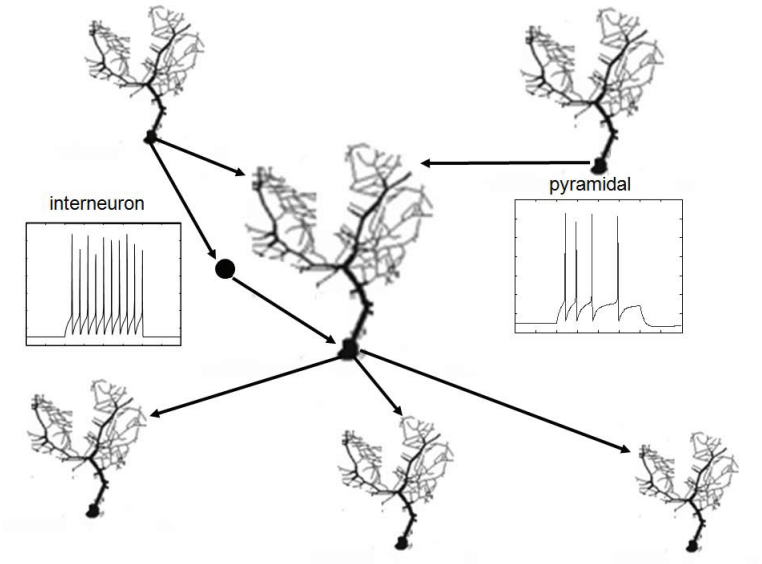

Fig. 7. An illustration of cortex network.

In the network, each neuron receives synaptic inputs from other neurons and integrates those inputs in the cell body. If those inputs are large enough, a train of spiking will be initialized in the cell body and propagates through axons. At the end of axons, the train of spikes will lead to the release of neurotransmitters. By binding the synaptic receptors of the receiving neurons, the neurotransmitters may increase (excite) or decrease (inhibit) the

membrane potential of the receiving neurons.

An important characteristic of the neuron network is that it exhibits a variety of time scales:

1. Different Time Scales for Active and Inactive Neurons

In practical networks, neurons do not fire all the time. Indeed, a neuron is in a resting state most of the time. When there is no external stimuli, the thalamus-cortical network typically oscillates at about 8-10Hz [10], which means a neuron only fires 8-10 times in the period of one second. As the duration of each firing is on the order of milliseconds, the neuron is inactive during most of the time.

When a neuron fires action potentials, strong nonlinear dynamics happens in a short time period. In this case, small time steps (e.g. 0.02 milliseconds) must be taken in transient simulation to satisfy both accuracy and stability requirements, especially when Hodgkin-Huxley models are used. However, in the inactive state, the changes of membrane potentials are small and it is possible to use much larger time steps if advanced numerical integration techniques are adopted. This fact opens an opportunity to reduce the costs of the simulation associated with Hodgkin-Huxley type models.

2. Different Time Scales for Intra-neuron and Inter-neuron Activities

During most of the time, a neuron is at the resting state. However, even if the neuron is about its resting state, a sufficiently small time step is still needed (especially, for HH models) to accurately track the membrane potentials and avoid numerical instability.

On the other hand, at the network level, neurons could not interact with others until the membrane potentials are large enough to trigger action potentials. Even if a neuron fires, it takes some time to reach others due to axon delays, which typically range from a few milliseconds to hundreds of milliseconds.

As a result, compared with the time resolution (simulation step size) required to track the internal activities of individual neurons (intra-neuron activity), the interactions between different neurons (inter-neuron activity) occur much less frequently. Typically, a time resolution of one millisecond is enough for tracking network activities [16]. This property makes it possible to apply a smaller time step to simulate individual neuron dynamics and a much larger time step to update network-level interactions, as we exploit in this work. As shown in later sections of this article, exploiting the multi-scale nature of the neural network in a numerically and computationally efficient way can lead to significant runtime benefits in simulating large-sale neocortex models.

B. Basic Steps for Dynamical Simulation of Cortex Networks

Before presenting proposed techniques, we briefly introduce the basic cortex network simulation steps. The cortex networks are described by a set of coupled differential equations using neuron models such as in equations (2.5), (2.6) and (2.7), and additional synaptic and axon delay models. The state variables are the membrane potentials associated with neurons and the gating variables associated with ion channels. All the neurons in the network receive synaptic current inputs and some of the neurons also receive inputs from thalamus. To simulate the network is essential to solve the set of differential equations in a numerically efficient way. Given the initial values of all the state variables, the simulation can be divided into the following steps:

1. Set thalamic inputs. Some neurons in the network are chosen to receive current inputs to initiate the simulation, which mimics the effects of the thalamus.

2. Find firing neurons. Scan through the network to find all firing neurons. Firing or not is determined by whether their membrane voltages exceed a threshold value. When these neurons are found, a spike will fire from each of them and propagate through

axons, which are modeled as delays, to the receiving neurons.

3. Update synaptic conductances. Go through the network to update the synaptic conductances corresponding to the excitatory and inhibitory inputs they receive at the current time.

4. Calculate synaptic input currents. Calculate the excitatory and inhibitory input currents, respectively, for each neuron.

5. Update state variables. Use a numerical integration method, such as Forward Euler, to update the state variables associated with a neuron model for the current time. Note that for HH models, the update of state variables is very expensive.

The above five-step simulation loop iterates repeatedly until the simulation time reaches the end. The increment time step here is determined by the numerical accuracy and stability of the numerical method in step (5), which is no more than 0.02ms. The overall flow of SNN simulation is shown in Fig. 8.

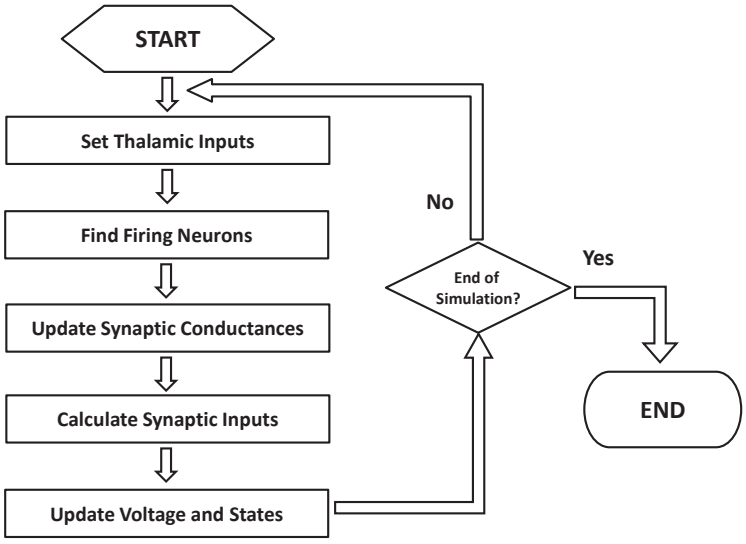

Fig. 8. Flowchart of the basic simulation steps.

CHAPTER V

IMPLEMENTATION

Before discussing the proposed techniques in details, we first briefly discuss the main data structures and implementation in our GPU simulator, in order to show a clear view of the key pieces of the simulator.

A. Data Structure

There are two major types of data structures, one for storing network connection, and one for storing network dynamics.

1. Network Connection

The network connection is created by the CPU code and is then transferred onto the GPU memory. Once the network information is created, it remains unchanged during simulation. The network information stored in these data structures are pre-neurons IDs, post-neurons IDs, and post-synaptic delays. These key data structures are mostly organized as compact one-dimensional arrays. Such organizations lead to coalesced memory accesses, which are important for minimizing memory access latency. For each neuron, its neuron id, the number of post-synaptic connections are unique. Since the numbers of post-synaptic connections of all neurons are different, two data pools are used to store the post-neuron ids and post-synaptic delays for all neurons. Indexes to these data pools are also stored in an array for easy access. A schematic of the data structures is shown in Fig. 9. In this network, there are totally six neurons, represented by the circles. The numbers inside the circles are their neuron ids. The arrows represent synaptic connections between two neurons and the axonal delays (in ms) are annotated on the arrows. When a neuron fires action potentials,

the spike can not reach the post-synaptic neuron immediately, buy after a certain axonal delay. All the information of the network connection can be obtained from the data structures shown in the figure. Take neuron 3 as an example. Neuron 3 has two post-synaptic neurons, and this number is stored in the second array shown in the figure. Its post-synaptic neurons (neuron 5 and neuron 6) and corresponding delays (4ms and 2ms) are stored in the data pools. Using these data structures, one can efficiently find all the post-neurons connected

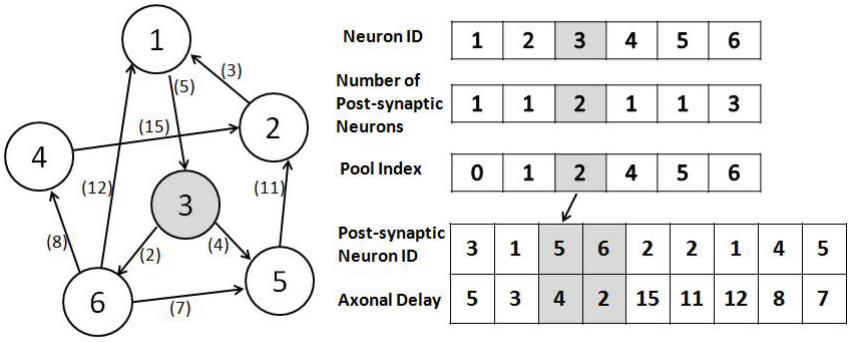

Fig. 9. Connectivity graph and data structures for a simple network.

to a neuron, and also the axon delays to those post-neurons.

2. Network Dynamics

The network dynamics data structures store state variables and event tables. The state variables, which are updated every time step by the model dynamics, include membrane potential, gating variables, time constants in the HH model differential equations and states in the synapse model. The state variables are organized as arrays for each access in GPU.

The event tables, which work in a circular event queue fashion, store the timing information and the IDs of fired neurons. Each neuron cell has its own event table to store number of spikes it will receive in the future. For the lookup table technique which will be

discussed in the following section, the event table also needs to store previous firing information in order to quickly compute its synapse input at the current time point. Whenever a neuron fires, its neuron id is recorded and the following synaptic events are updated. Let's still use the the neuron network in Fig. 9 as an example. If only two neurons (neuron 3 and neuron 4) fire in the current time step, recalling the flowchart of simulation steps shown in Fig. 8, their ids are recorded in the "Find Firing Neurons" stage. The next step is to find all their post-synaptic neurons and when the spikes will reach them. And then, the numbers in the corresponding slots in the event tables of those post-synaptic neurons are incremented by one. As neuron 3 has two post-synaptic neurons (neuron 5 and neuron 6), their event tables will be updated. Fig. 10 shows an example of updating the event tables. (t_0 is the current time point). Since the axonal delays from neuron 3 to neuron 5 and neuron 6 are 4ms and 2ms, respectively, the corresponding slots in the event tables are incremented by one. With these event tables, we can efficiently find fired neurons and firing times and then use this information to update the network-level activities.

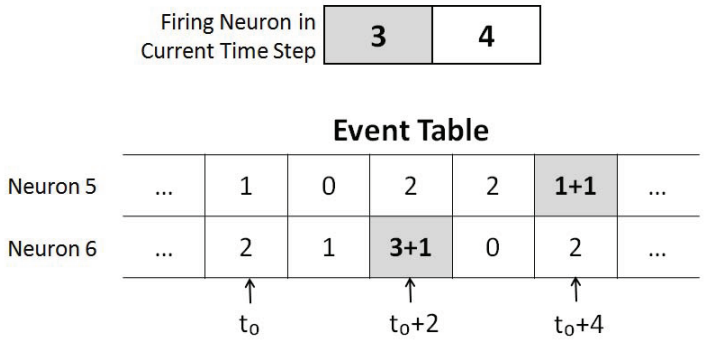

Fig. 10. Example of updating event tables.

B. GPU Mapping

In this thesis work, the neuron network is mapped onto GPU. At different stages, the GPU mapping could be different. When updating neuron level information, each neuron is mapped on a processor. Different neuron cells are updated in parallel by different threads. Whenever a spike is generated, it will propagate through the post-synaptic connections. In this stage, each synaptic connection is mapped in parallel. As we store most of the information as one-dimension arrays, these different mapping can be easily applied within SMs. Fig. 11 shows a piece of representative cuda code in the simulator. In this function, each neuron is mapped on one thread. With such organization, memory bandwidth performance is improved due to coalesced memory access. Note that the shared memory is also used in the function to act as a local buffer. Shared memory is a type of memory that can be shared within a thread block. In this case, shared memory helps to avoid frequent global memory accesses and reduce warp divergence.

More implementation details of the advanced techniques we use in this thesis work will be discussed in the following sections.

```
1.  __global__ void kernel_get_firing() {
2.      __shared__ volatile int cnt;
3.      __shared__ int local_buffer[256];
4.      if(0==threadIdx.x) cnt=0;
5.      for(int i=blockDim.x*blockIdx.x; i<num_neuron;
    i+=blockDim.x*gridDim.x) {
6.          int id = i+threadIdx.x;
7.          if (id<num_neuron && V[id]>Threshold ){
8.              int index=atomicAdd(&cnt,1); //update local buffer
9.              local_buffer[index] = id;
10.         }
11.         ... // update global buffer when local buffer is full
12.     }
13. }
```

Fig. 11. Sample CUDA code in the simulator.

CHAPTER VI

SIMULATION PERFORMANCE OPTIMIZATION ON GPU

Due to the high computational cost of HH models, the update of synaptic currents and neuron state variables completely dominates the overall simulation runtime. Among these, (coalesced) access to global memory also contributes to an important overhead. Recall the simulation flow shown in Fig. 8, in each iteration, the simulator needs to update synaptic current and HH model state variables. Each of such updates involves one global memory access and complex calculations. As we have mentioned in previous sections, a very small time step must be used in order to satisfy both accuracy and stability requirements, meaning that the numbers of both global memory accesses and expensive calculations are relatively large. To improve the simulation runtime efficiency, we introduce two optimization strategies. The first approach aims to enhance arithmetic intensity, hence increasing the ratio between calculation and memory access and effectively reducing the overhead of memory access. The second approach uses precharacterized lookup table based HH models and efficient GPU implementation to speed up the evaluation of HH models.

A. Enhancement of Arithmetic Intensity

Due to the complex dynamics associated with the HH models, the step size of an explicit integration method such as Forward Euler or exponential Euler, commonly used in neural simulation [1, 4, 16, 20] and public-domain neuron simulators [2, 6], is significantly constrained by stability and accuracy requirements. We have observed that a step size beyond 0.02 ms can simply produce erroneous results.

However, from a network level point of view, since a typical firing pattern lasts more than 20 ms and axon delays range from a few milliseconds to hundreds of milliseconds, a time resolution of 1 ms is small enough to remain sufficient system level simulation ac-

curacy. Therefore, we propose a technique to enhance arithmetic intensity. The proposed technique applies a larger time resolution, which is 1 ms, at the network level while adopting a much smaller time resolution (or inner time step size), 0.02 ms, to track individual neuron dynamics to guarantee numerical accuracy and stability. As shown in Fig. 12, the system updates its firing information and synaptic conductance of each neuron once per millisecond (we call it outer time step). At the very beginning of each outer time step, the synaptic inputs of each neuron are updated only once based upon received action potentials from other neurons. Then, multiple (around 50) smaller inner time steps are taken to integrate the dynamics of the neuron for one outer time step.

Note that even with proper management of global memory accessing, a coalesced access takes at least 200 clock cycles. To integrate a neuron in time, six data including the input current, membrane potential and four state variables, need to be loaded from the global memory. At the completion of the integration, the results need to be written back to the global memory. As such, the advantage of this arithmetic intensity enhancement approach is obvious. By exploiting the multi-scale nature of the network, global memory accesses are significantly reduced because for each neuron one only needs to load and store its state variables in the global memory once in 1 ms instead of, say, 50 times. According to our experiments, the use of this arithmetic intensity enhancement can maintain good accuracy for the network simulation.

B. Lookup Table Acceleration for HH Model Evaluation

HH models are more complex compared with spiking models. The model formulae contain both exponential expressions and conditional branches. On GPU, an evaluation of an exponential function is very expensive and can take 32 clock cycles. Conditional branches are also detrimental as they can easily cause wrap divergence and serialization of diverging

threads. The proposed lookup table (LUT) approach addresses both issues.

Using equation (2.5) as an example, voltage-dependent steady-state activations m_∞, h_∞, n_∞, p_∞ and time constants τ_m, τ_h, τ_n, τ_p of pyramidal cells are given by the following expressions:

$$
\begin{aligned}
m_\infty &= \frac{1}{1 + exp((32 - V)/8)} \\
\tau_m &= \begin{cases} 0.0125 + 0.007exp((V - 40)/8) & V \leq 38 \\ 0.01 + 0.0725exp((40 - V)/8) & V > 38 \end{cases} \\
h_\infty &= \frac{1}{1 + exp((V - 12.6)/7)} \\
\tau_h &= 0.75 + \frac{5.75}{1 + exp((V - 36.5)/10)} \\
n_\infty &= \frac{1}{1 + exp((40.5 - V)/10)} \\
\tau_n &= \begin{cases} 0.75 + 13.05exp((V - 60)/10) & V \leq 60 \\ 0.75 + 13.05exp((60 - V)/10) & V > 60 \end{cases} \\
p_\infty &= \frac{1}{1 + exp((35 - V)/10)} \\
\tau_p &= \frac{1000}{3.3exp((V - 35)/20) + exp((35 - V)/20)}
\end{aligned}
\tag{6.1}
$$

For each neuron, these eight variables need to be computed in every HH state variables update step. In order to avoid these exponential calculations and warp divergence, eight one-dimensional tables are built correspondingly. Each table stores the pre-calculated values of a variable at a given set of membrane potentials. Proper data stored in the table are fetched for HH model evaluation during simulation.

Note that, for regular-spiking pyramidal cells [9], there are three ion channel types: a sodium current I_{Na}, a potassium current I_K, and a slow voltage-dependent potassium

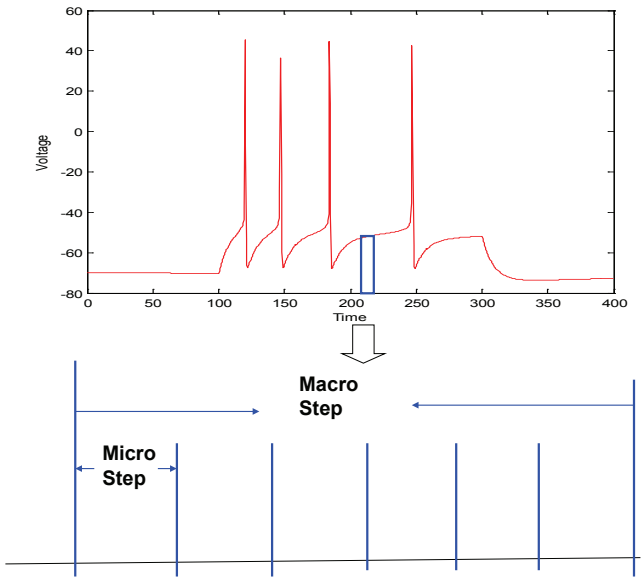

Fig. 12. Macro and micro time steps.

current I_M. Fast-spiking interneurons have been modeled to have only the first two ion channels [9]. Due to the fact that in most network level simulations [10], only a limited number of neurons and ion channels are included, it is possible to construct the associated lookup tables using several one-dimensional arrays.

The main data flow in the state variables updating process is shown in Fig. 13. The lookup tables are stored in the texture memory, which is cached. In the case of cache hit, a texture fetch is as fast as accessing a register. An important fact is that in 80% of time, neurons are in the resting mode. In this mode, the membrane potentials fluctuate within a small range. As a result, the cache hit rates are high because only a very small part of the lookup tables are frequently accessed, hence a strong locality. In contrast to an exponential function evaluation, which costs more than 32 cycles on GPU, a texture cache hit only takes one clock cycle. The proposed technique takes full advantage of the texture memory, eliminates the expensive exponential calculations, and avoids warp divergence. Fig. 14 shows an example of evaluating time constant τ_m using the lookup tables.

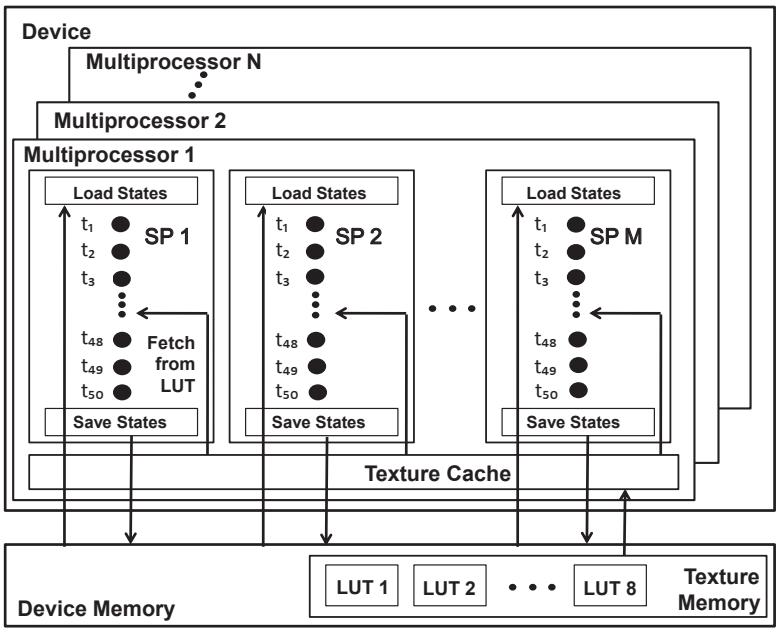

Fig. 13. State variables updating with lookup tables.

(a) Calculate τ_m without Lookup Table

1. i=threadIdx.x+blockIdx.x*blockDim.x
2. If(V[i]<=38) //Warp divergence
3. τ_m=0.0125+0.007*exp((V[i]-40)/8); //Exponential
4. else
5. τ_m=0.01+0.075*exp((40-V[i])/8)
6. . . .

(b) Calculate τ_m with Lookup Table

1. BindTextureToArray(Table_tex,Table_array); //bind to texture at the beginning of simulation
2. i=threadIdx.x+blockIdx.x*blockDim.x
3. index=get_index(V[i]);
4. τ_m(i)=tex2D(Table_tex,index,1); //Fetch result

Fig. 14. Pseudo code showing that using Lookup Table to avoid exponential calculation and warp divergence.

CHAPTER VII

TELESCOPIC PROJECTIVE NUMERICAL INTEGRATION

Biophysical plausible Hodgkin-Huxley (HH) models capture rich and complex neuronal dynamics. From a simulation point of view, different modes of neuron operations impose distinctive requirements on accuracy and stability. Proper use of advanced numerical integration methods can lead to larger time step sizes and hence further improves the efficiency of neural simulation.

A. Detection of Active and Inactive States

In the generation of action potentials, the membrane potential and other state variables evolve very quickly in time so as to produce sharply rising and falling spiking patterns. In this case, the accuracy requirement is often much more stringent and very small step sizes need to be used to ensure accuracy. On the other hand, around the resting potential, no significant activity takes place in a neuron. This may allow one to use larger time sizes to boost the simulation efficiency. As mentioned in section 6, on average a neuron sits around the resting potential, or is inactive, around 80% of time under typical network conditions. Properly, in such low-activity periods, exploiting large time steps is beneficial in terms of boosting the overall simulation efficiency.

In this work, as shown in Fig. 15, we detect active and inactive states by comparing the membrane potential of each neuron with a predefined threshold voltage. The threshold voltage is chosen conservatively to capture all potential fire activities. As illustrated in Fig. 15, since such detection is purposely made conservative, a period of time which is detected as "active" may not be immediately followed by the generation of an action potential.

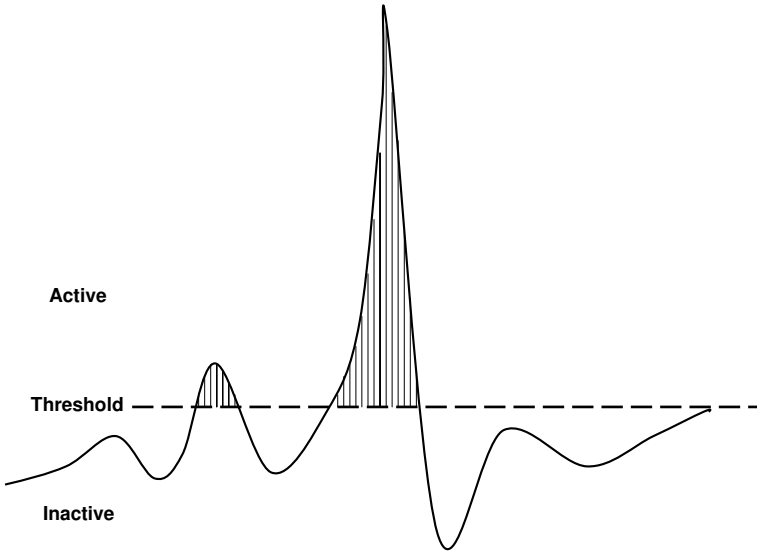

Fig. 15. Detecting active and inactive states using a threshold voltage.

B. Acceleration of Simulation in Inactive States

Since there are no firing activities during identified inactive periods, the simulation accuracy requirement becomes relaxed. However, with large time steps, the issue of stability comes into play. As mentioned in the previous sections, a small time step size not exceeding 0.02 ms is required by the explicit integration method such as Forward Euler used to evaluate HH model variables.

To boost the step size, we adopt and develop more advanced stable numerical integration methods based on the recently developed telescopic projective integration framework [12, 13]. Often, the long term transient responses of a system are mainly determined by slow components (corresponding to large time constants) in the network. Fast components only exist for a short period of time and dissipate quickly. Since it is often sufficient to only track the slow components in the transient responses, it is desirable to use a time size that is comparable to large time constants. In this regard, the problem with a standard explicit integration method (e.g. Forward Euler) is that the step size has to be comparable to

the smallest time constant to maintain stability, a significant constraint on efficiency. This problem can be alleviated by adopting the projective integration method shown in Fig. 16

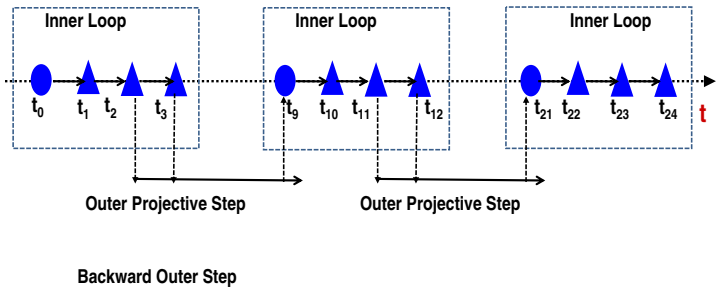

Fig. 16. A one-level projective integrator(k=2).

Intuitively, to ensure stability, k+1 explicit integration (e.g. Forward Euler) steps are taken at the inner loop to integrate the system from, say from time $t_n$to time t_{n+k+1}. Here, a small step size is used to sufficiently damp fast transient responses. Then, a large projective or extrapolation step with a step size commensurate with the slow time constants is taken to project into the forward direction of time to derive the response at time $t_{n+k+1+M}$ (M is desired to be large)

$$x_{n+k+1+M} = (M+1)x_{n+k+1} - Mx_{n+k} \tag{7.1}$$

where x_{n+k+1} and x_{n+k} are the solutions at the last two time points computed at the inner loop.

From a stability point of view, the sequence of k+1 integration steps with a small step size exponentially (in k) damps the numerical integration error, while the potential error amplification incurred by the large projective step is only linear in step size. This makes

it possible to maintain a large M without sacrificing stability. Hence, the combination of several small integration steps and one large extrapolation step boosts the effective step size of the overall integration scheme.

It can be shown that with proper choices of k and M the projective integrator maintains the so-called [0, 1] stability [13]. However, in practice, when the eigenvalues of the system are widely distributed with no clear clustering or the eigenvalue distribution is not known a priori, the step size of the outer projective step must be conservatively controlled to ensure stability. In other words, M needs to be chosen conservatively small to ensure the [0, 1] stability, leading to reduced step size amplification. To maintain good efficiency in general practical cases, the concept of projective integration has been generalized to a multi-level telescopic scheme [13]. Fig. 17 illustrates a two-level telescopic projective integration scheme. The steps viewed at the top level are similar to those of a projective integrator except that each basic integration step is expanded into a projective integration step at the bottom level. As such, while at each individual projective integration level, limited step size amplification is obtained with a relatively small M, significant overall step size amplification may be obtained in the multi-level telescopic framework.

C. Implementation of Telescopic Projective Integration on GPU

In our implementation, we choose to adopt a two-level telescopic framework with k=3, M=6. With such choices, the effective step size boost is 2.5X without incurring instability and any significant numerical error.

In order to manage work loads and minimize wrap divergence, at the beginning of each outer time step, all the neurons are divided into two groups according to their current states. The identified active and inactive groups are subsequently processed by all the threads. During this process, the neurons in the inactive group are integrated using the

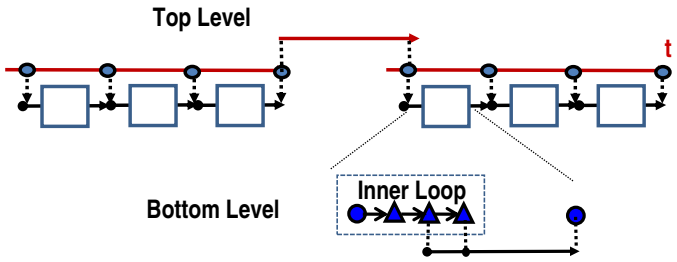

Fig. 17. A two-level telescopic projective integrator.

faster telescopic projective integration while ones in the other group are simulated using the conventional forward Euler with 50 small time steps, as shown in Fig. 18.

More specifically, as shown in the pseudocode in Fig. 19, without grouping of neurons, significant wrap divergence can be resulted. In this case, the execution path for regular integration takes more than 200 clock cycles to complete. With grouping, each neuron is first checked to determine its state and put into the corresponding group table. This process involves conditional branches, however, with each branch takes only about 5 to 10 clock cycles to complete. After the two group tables are built, they are processed subsequently without any wrap divergence.

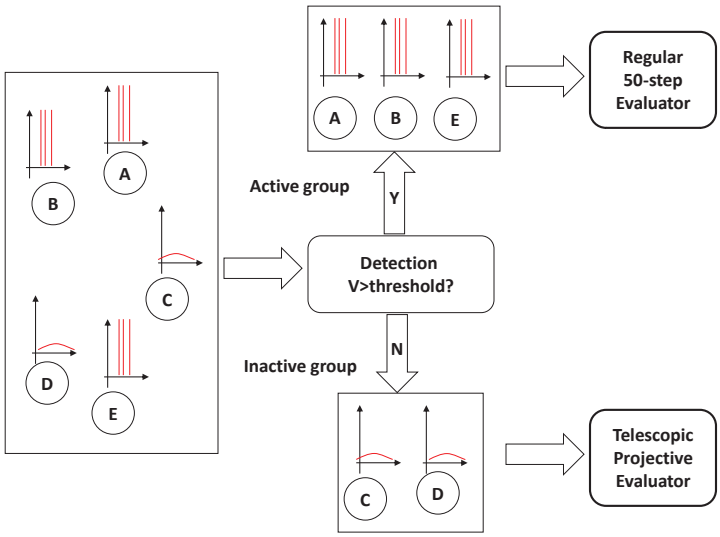

Fig. 18. Neuron dispatch for telescopic projective integration.

(a) Update state variables without grouping

```
1.  i=threadIdx.x+blockIdx.x*blockDim.x
2.  If(V[i]>THRESHOLD)
3.      regularUpdate(i); //more than 200 cycles
4.  else
5.      telescopicUpdate(i);
6.  ...
```

(b) Update state variables with grouping

```
1.  cnt1=0;cnt2=0;
2.  i=threadIdx.x+blockIdx.x*blockDim.x
3.  If(V[i]>THRESHOLD){
4.      p=atomicAdd(&cnt1,1); //5-10 cycles
5.      active_table[p]=i;
6.  }
7.  else{
8.      p=atomicAdd(&cnt2,1);
9.      inactive_table[p]=i;
10. } // go through the whole network
11. ... __syncthreads();
12. regularUpdate(active_table, cnt1);
13. telescopicUpdate(inactive_table, cnt2);
```

Fig. 19. Pseudo code showing that using Telescopic Projective method with grouping to minimize thread divergence and reduce simulation steps.

CHAPTER VIII

EXPERIMENTAL RESULTS

A. Experiments Setup

To demonstrate the efficiency of our neural network simulator, we use a neocortical network as an example, which consists of two types of neurons: excitatory pyramidal neurons and inhibitory interneurons. In the network, 80% neurons are pyramidal neurons and 20% neurons are interneurons according to biophysical facts. The neuronal dynamics are modeled using Hodgkin-Huxley (HH) models. The parameters we use for the two types of neurons are shown in Table I [10], where all conductances are in mS/cm^2 and all potentials are in mV. We vary the number of neurons to get network models with different complexities. Each model is simulated to track one second of network dynamics and the simulation runtime of different methods are reported.

Table I. Neuron parameter used in this implementation.

Neuron Type	g_L	g_{Na}	g_K	g_M	E_L	E_{Na}	E_K
Pyramidal	0.1	50	5	0.07	-70	50	-100
Interneuron	0.15	50	10	0	-70	50	-100

B. Performance of the Proposed Simulation Techniques

In the first part of the experimental results, we show the benefits resulted from the three techniques proposed in the previous sections. We include three techniques into the basic implementation one after the other and show the corresponding performance improvement. The results are summarized in Table II. In the following, we use "EAI", "LUT" and "TPM"

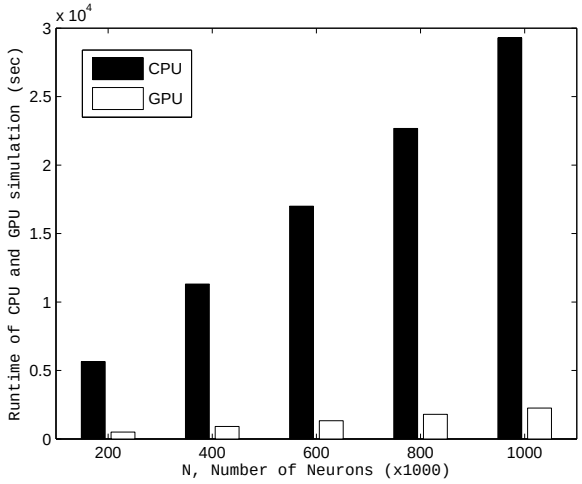

Fig. 20. Runtime comparison between GPU and CPU basic simulations with a time step of 0.02 ms (M=200).

to denote the proposed arithmetic intensity enhancement, GPU look-up table based HH models and telescopic projective integration method, respectively. We also use N to denote the number of neurons of the network and M to denote the maximum number of post-synapses of each neuron.

1. Basic CPU vs. GPU Implementations

First, we compare runtime between the basic GPU and CPU implementations. In basic implementations, we did not apply any advanced techniques proposed in this thesis work. The simulation is performed with a time step of 0.02 ms. The time step is also the timing resolution of the network activity. The runtime comparison is shown in Fig. 20. The simulation runtime of the two implementations is also reported in Table II and Table III.

Table II. Runtimes of the CPU and GPU implementations. Simulation runtimes are in seconds. Each network model is simulated for one second (real time). GPU_All denotes the GPU implementation with three proposed techniques included (GPU_EAI_LUT_TPM).

# of Neurons	Basic CPU Runtime	Basic GPU Runtime	GPU_EAI Runtime	GPU_EAI_LUT Runtime	GPU_All Runtime
200k	5641.46	501.74	203.62	11.73	9.42
400k	11311.92	910.43	406.14	23.44	19.85
600k	17000.33	1329.04	613.26	35.8	29.82
800k	22673.95	1795.68	815.86	47.68	39.8
1M	29285.34	2255.84	1023.11	60.1	48.96

Table III. Speedups of the CPU and GPU implementations.

# of Neurons	GPU_EAI vs. GPU Basic Speedup	GPU_EAI_LUT vs. GPU_EAI Speedup	GPU_All vs. GPU_EAI_LUT Speedup	GPU_All vs. Basic CPU Speedup
200k	2.46X	17.34X	1.25X	599X
400k	2.24X	17.33X	1.18X	570X
600k	2.17X	17.13X	1.20X	570X
800k	2.20X	17.11X	1.20X	571X
1M	2.20X	17.02X	1.23X	599X

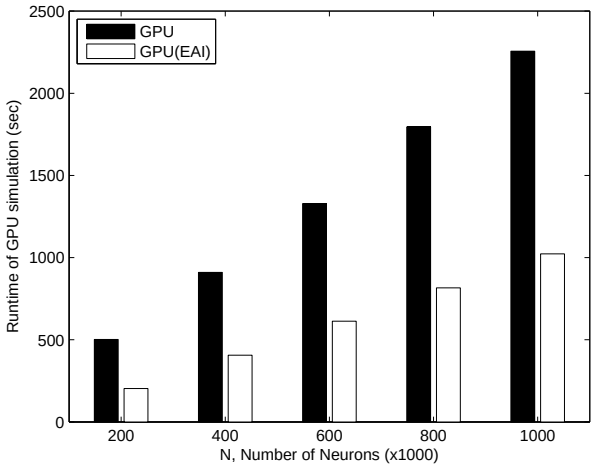

Fig. 21. Runtime performance improvement with EAI (M=200).

2. Speedups of Arithmetic Intensity Enhancement

Second, the proposed Enhancement of Arithmetic Intensity (EAI) technique is applied to the basic GPU implementation. Fig. 21 shows the runtime performance improvement of the GPU simulator with EAI. For GPU simulation with EAI, the timing resolution for the network activity is 1 ms.

Table III shows that the proposed EAI technique provides an average speedup of 2.2X.

3. Speedups of GPU Lookup based HH Models

Based on the GPU based implementation with EAI, the proposed lookup table (LUT) technique is further introduced. Fig. 22 shows the runtime performance improvement with LUT models compared with the GPU simulation with only EAI.

Table III shows that the proposed Lookup Table technique provides an additional speedup of 17X on average.

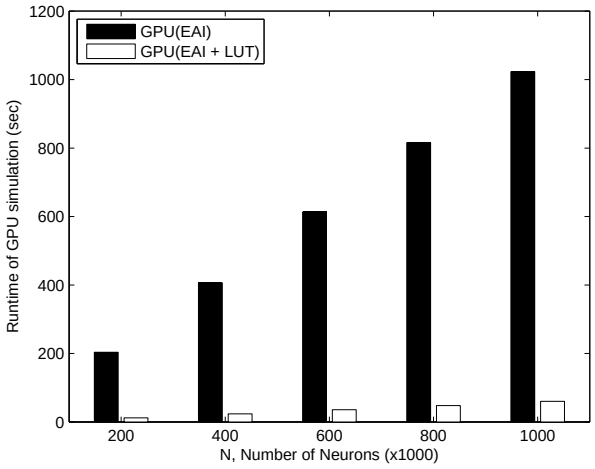

Fig. 22. Runtime performance improvement with LUT (M=200).

4. Speedups of Telescopic Projective Integration

Finally, we introduce the proposed telescopic projective integration method (TPM) into the GPU implementation. The performance improvement introduced by TPM compared with the previous implementation with EAI and LUT is shown in Fig. 23.

Table III shows that the proposed TPM method can provide an average speedup of 1.2X. A more careful examination reveals that the use of TPM can introduce a more than 2X boost of effective step size for each neuron simulation, hence cutting down that part of simulation cost by the same factor. However, after applying the previous two techniques (EAI and LUT), the percentage of runtime spent on transient simulation of individual neurons over the total runtime of network simulation has been significantly reduced. In this case, the effectiveness of TPM in reducing the total simulation time is somewhat limited. However, when used as stand alone, it can produce useful runtime speedups. By exploiting all the three proposed techniques, the GPU simulator is about 600 times faster than the basic CPU implementation. This speedup reflects the use of graphics processors and also the proposed algorithmic/implementation improvements.

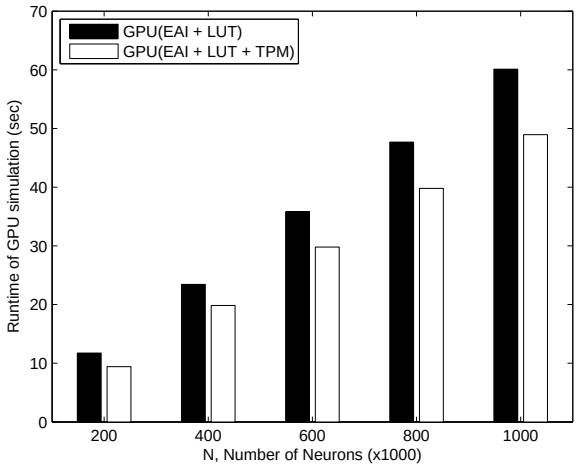

Fig. 23. Runtime performance improvement with TPM (M=200).

C. GPU Implementation Speedup Over CPU Implementation

We show the overall runtime performance improvement of the GPU implementation with all three proposed techniques. The results are compared with the CPU implementation also with those three techniques applied, as shown in Fig. 24, Fig. 25, and Table IV.

D. Comparison on HH and Spiking Model based GPU Simulations

We compare the GPU-based network simulations with HH and Spiking (Izhikevich) neuron models. The Izhikevich spiking model simulator comes from another publication [20]. All the experiments for comparison between the two simulators are based on the same neural network setting and run on the same platform. The results show that the simulation efficiency gap between the two models can be narrowed dramatically with the proposed three techniques. For the same set of neural networks, the runtime results are given in Fig. 26 and Table V.

As shown in Table V, the GPU simulation with HH models is two to three times slower than that with Izhikevich spiking neuron models. It is important to note that this represents

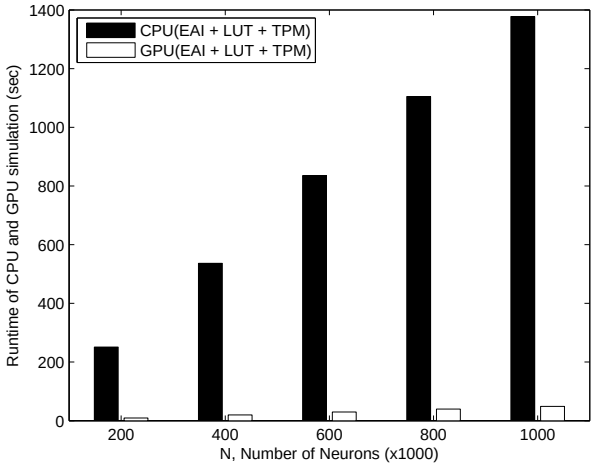

Fig. 24. Runtime performance improvement of CPU implementation with all three techniques over the CPU implementation (M=200).

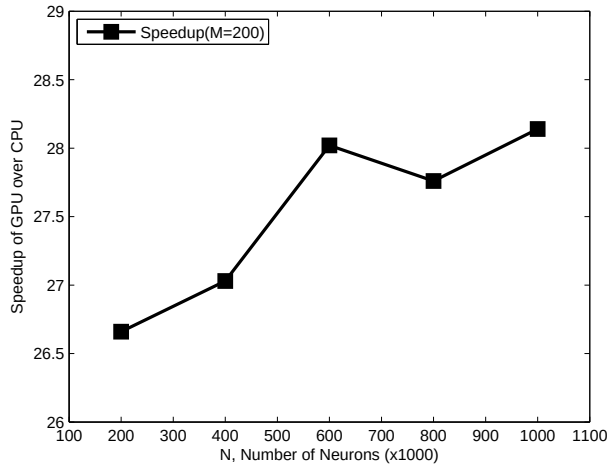

Fig. 25. Speedups of the GPU implementation with all three techniques over its CPU counterpart implementation.

Table IV. Runtimes and speedups of the GPU simulator over its CPU counterpart. Runtimes are in seconds. The proposed three techniques are used in both implementations.

N	CPU_EAI_LUT_TPM	GPU_EAI_LUT_TPM	Speedup
200k	251.13	9.42	26.66X
400k	536.53	19.85	27.03X
600k	835.51	29.82	28.02X
800k	1104.66	39.8	27.76X
1M	1377.69	48.96	28.14X

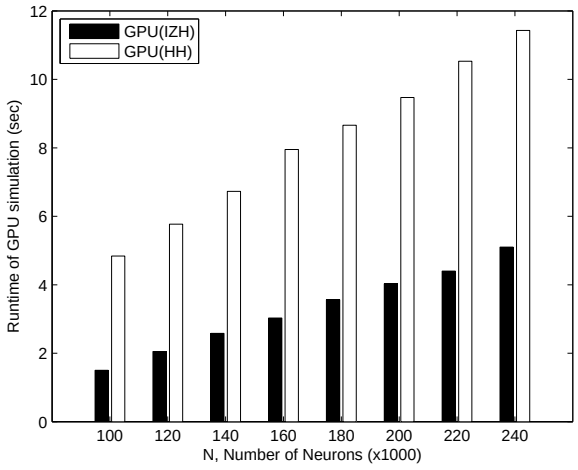

Fig. 26. Runtime comparison between GPU simulations with HH and Izhikevich neuron models (M=100). Runtimes are in seconds

Table V. Runtimes and speedups of the HH neuron model based simulation over its Izhikevich counterpart on GPU(M=100). Runtimes are in seconds.

N	GPU_IZH	GPU_HH	Speedup
100k	1.5	4.84	1/3.23X
120k	2.05	5.77	1/2.81X
140k	2.58	6.73	1/2.61X
160k	3.03	7.95	1/2.62X
180k	3.56	8.66	1/2.43X
200k	4.03	9.47	1/2.35X
220k	4.4	10.53	1/2.39X
240k	5.1	11.43	1/2.24X

a significant efficiency improvement from the original efficiency gap of 100X, achieved by using the proposed techniques. As a result, with the proposed techniques, we expect that HH models may be used to replace Izhikevich models in large-scale network simulations to explore the relationship between observed network dynamics and the underlying biological and physiological causes.

CHAPTER IX

CONCLUSIONS

In this thesis work, we have exploited recent commodity massively parallel graphics processors to alleviate the significant computational costs in HH model based neural network simulation. We have developed look-up table based HH model evaluation and efficient parallel implementation strategies geared towards higher arithmetic intensity and minimum thread divergence. Furthermore, we have adopted and developed advanced multi-level numerical integration techniques well suited for intricate dynamical and stability characteristics of HH models. With the above techniques, our experimental results have demonstrated that the presented GPU neural network simulator is about 600X faster than a basic serial CPU based simulator, 28X faster than the CPU implementation of the same three techniques, and is only two to three times slower than the GPU based simulation using simpler spiking models for a neural network with one million neurons and 200 million synaptic connections.

REFERENCES

[1] R. Ananthanarayanan, S. K. Esser, H. D. Simon, and D. S. Modha, "The cat is out of the bag: cortical simulations with 10^9 neurons, 10^{13} synapses," *Proc. of Conference on High Performance Computing Networking, Storage and Analysis*, pp. 143–175, 2009.

[2] B. Arnold, *Encountering the Book of Genesis*, 1st ed. Grand Rapids: Baker Books, 1998.

[3] C. Bernard, Y. C. Ge, E. Stockley, J. B. Willis, and H. V. Wheal, "Synaptic integration of nmda and non-nmda receptors in large neuronal network models solved by means of differential equations," *Biological Cybernetics*, vol. 70, pp. 267–273, 1994.

[4] M. A. Bhuiyan, V. K. Pallipuram, and M. C. Smith, "Acceleration of spiking neural networks in emerging multi-core and gpu architectures," *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and PhD Forum*, pp. 1–8, May 2010.

[5] J. C. Butcher, *Numerical methods for ordinary differential equations*, 2nd ed. Hoboken, NJ: Wiley, 2008.

[6] N. T. Carnevale and M. L. Hines, *The NEURON book*, 1st ed. New York, NY: Cambridge University Press, 2009.

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, "A performance study of general-purpose applications on graphics processors using cuda," *Journal of Parallel and Distributed Computing*, vol. 68, pp. 1370–1380, 2008.

[8] J. Y. Chen, "Gpu technology trends and future requirements," *The 2009 IEEE International Electron Devices Meeting*, pp. 1–6, 2009.

[9] B. W. Connors and M. J. Gutnick, "Intrinsic firing patterns of diverse neocortical neurons," *Trends Neuroscience*, vol. 13, pp. 99–104, 1990.

[10] A. Destexhe, D. Contreras, and M. Steriade, "Mechanisms underlying the synchronizing action of corticothalamic feedback through inhibition of thalamic relay cells," *Journal of Neurophysiology*, vol. 79, pp. 999–1016, 1998.

[11] M. Djurfeldt, M. Lundqvist, C. Johansson, and M. Rehn, "Brain-scale simulation of the neocortex on the ibm blue gene/l supercomputer," *IBM Journal of Research and Development*, vol. 52, pp. 31–41, April 2010.

[12] W. Dong and P. Li, "Parallelizable stable explicit numerical integration for efficient circuit simulation," *Proc. of IEEE/ACM Design Automation*, pp. 382–385, July 2009.

[13] C. Gear and I. Kevrekidis, "Telescopic projective methods for parabolic differential equations," *Journal of Computational Physics*, vol. 13, pp. 99–104, 1990.

[14] E. Hairer and G. Wanner, *Solving ordinary differential equations II: Stiff and differential-algebraic problems*, 2nd ed. Berlin: Springer, 2004.

[15] A. Hodgkin and A. Huxley, "A quantitative description of membrane current and its application to conduction and excitation in nerve," *Journal of Physiology*, vol. 117, pp. 500–544, 1952.

[16] E. M. Izhikevich and G. Edelman, "Large-scale model of mammalian thalamocortical systems," *Proc. of the National Academy of Sciences of the United States of America*, vol. 105, pp. 3593–3598, 2008.

[17] E. Kandel, J. Schwartz, and T. Jessell, *Principles of Neural Science*, 4th ed. New York, NY: McGraw-Hill Medical, 2000.

[18] J. D. Lambert, *Numerical Methods for Ordinary Differential Systems: The Initial Value Problem*, 1st ed. Chichester, UK: Wiley, 1991.

[19] G. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, p. 8, 1965.

[20] J. M. Nageswaran, N. Dutt, J. L. Krichmar, A. Nicolau, and A. Veidenbaum, "Efficient simulation of large-scale spiking neural networks using cuda graphics processors," *Proc. of International Joint Conference on Neural Networks*, pp. 3201–3208, 2009.

[21] *Nvidia GPU programming guide*, Nvidia Corporation.

[22] E. M. Odabasioglu, "Which model to use for cortical spiking neurons," *IEEE Transaction on Neural Networks*, vol. 15, pp. 1063–1070, 2004.

[23] G. M. Shepherd, *The Synaptic Organization of the Brain*, 5th ed. USA: Oxford University Press, 2003.

[24] M. Wang, B. Yan, J. Hu, and P. Li, "Simulation of large neuronal networks with biophysically accurate models on graphics processors," *The 2011 International Joint Conference on Neural Networks*, pp. 3184–3193, 2011.

[25] Wgsimon, "File:transistor count and moore's law," Wikipedia, retrieved on 10 April 2012, <http://en.wikipedia.org/wiki/File:Transistor_Count_and_Moore%27s_Law_-_2011.svg>.

[26] B. Yan and P. Li, "Reduced order modeling of passive and quasi-active dendrites for nervours system simulation," *Journal of Computational Neuroscience*, 2011.

VITA

Name: Mingchao Wang

Address: Department of Electrical and Computer Engineering,
Texas A&M University,
214 Zachry Engineering Center,
TAMU 3128
College Station, TX 77843

Email Address: mingchaowang@gmail.com

Education: B.S., Automation, Zhejiang University, 2007
M.S., Electrical Engineering, Texas A&M University, 2012