Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2013

MSc THESIS

GPU-BASED SIMULATION OF BRAIN NEURON MODELS

DU NGUYEN HOANG ANH

CE-MS-2013-10

Abstract

The human brain is an incredible system which can process, store, and transfer information with high speed and volume. Inspired by such system, engineers and scientists are cooperating to construct a digital brain with these characteristics. The brain is composed by billions of neurons which can be modeled by mathematical equations. The first step to reach that goal is to be able to construct these neuron models in real time. The Inferior Olive (IO) model is a selected model to achieve the real time simulation of a large neuron network. The model is quite complex with three compartments which are based on the Hodgkin Huxley model. Although the Hodgkin Huxley model is considered as the most biological plausible model, it has quite high complexity. The three compartments also make the model become even more computationally intensive. A CPU platform takes a long time to simulate such a complex model. Besides, FPGA platform does not handle effectively floating point operations. With GPU's capability of high performance computing and floating point operations, GPU platform promises to facilitate computational intensive applications successfully. In this thesis, two GPU platforms of the two latest Nvidia GPU architectures are used to simulate the IO model in a network setting. The performance is improved significantly on both platforms in comparison with that on the CPU platform. The speed-up of double precision simulation is 68.1 and 21.0 on Tesla C2075 and GeForce GT640, respectively. The single precision simulation is nearly twice faster than the double precision simulation. The performance of the GeForce GT640 platform is 67% less than that on the Tesla C2075 platform, while the cost efficiency on the GeForce GT640 is eight times higher than that on the Tesla C2075 platform. The real time execution is achieved with approximately 256 neural cells. In conclusion, the Tesla C2075 platform is essential for double precision simulation and the GeForce GT640 platform is more suitable for reducing execution time of single precision simulation.

Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

GPU-BASED SIMULATION OF BRAIN NEURON MODELS

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

DU NGUYEN HOANG ANH
born in DANANG, VIETNAM

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

GPU-BASED SIMULATION OF BRAIN NEURON MODELS

by DU NGUYEN HOANG ANH

Abstract

The human brain is an incredible system which can process, store, and transfer information with high speed and volume. Inspired by such system, engineers and scientists are cooperating to construct a digital brain with these characteristics. The brain is composed by billions of neurons which can be modeled by mathematical equations. The first step to reach that goal is to be able to construct these neuron models in real time. The Inferior Olive (IO) model is a selected model to achieve the real time simulation of a large neuron network. The model is quite complex with three compartments which are based on the Hodgkin Huxley model. Although the Hodgkin Huxley model is considered as the most biological plausible model, it has quite high complexity. The three compartments also make the model become even more computationally intensive. A CPU platform takes a long time to simulate such a complex model. Besides, FPGA platform does not handle effectively floating point operations. With GPU's capability of high performance computing and floating point operations, GPU platform promises to facilitate computational intensive applications successfully. In this thesis, two GPU platforms of the two latest Nvidia GPU architectures are used to simulate the IO model in a network setting. The performance is improved significantly on both platforms in comparison with that on the CPU platform. The speed-up of double precision simulation is 68.1 and 21.0 on Tesla C2075 and GeForce GT640, respectively. The single precision simulation is nearly twice faster than the double precision simulation. The performance of the GeForce GT640 platform is 67% less than that on the Tesla C2075 platform, while the cost efficiency on the GeForce GT640 is eight times higher than that on the Tesla C2075 platform. The real time execution is achieved with approximately 256 neural cells. In conclusion, the Tesla C2075 platform is essential for double precision simulation and the GeForce GT640 platform is more suitable for reducing execution time of single precision simulation.

Laboratory	:	Computer Engineering
Codenumber	:	CE-MS-2013-10

Dedicated to my parents, who gave me a dream
and my love, who encourages me fulfill it

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank Dr. Zaid Al-Ars for his supervision to my work and patience in improving my analyzing and writing skill. I would like to thank Georgios Smaragdos for his enthusiastic help on neural science knowledge. I would like to thank Eef Hartman for his support on simulation platforms. I would like to thank Josje Kuenen for improving my English presentation skill and my confidence in general. I would like to thank Dr. Koen Bertels, Dr. Said Hamdioui, and Dr. Jeroen de Ridder for being on my graduation committee.

DU NGUYEN HOANG ANH
Delft, The Netherlands
August 26, 2013

Introduction 1

For hundreds of years, neural sciences have been accumulating huge amount of detailed knowledge to support researchers understand brain activity. The human brain has plenty of desirable characteristics such as rapid processing, low energy consumption, large memory storage etc. Hence, engineers are interested to mimic a system which can function as efficiently as the human brain. However, to carry out experiments on human brain is difficult and sometimes immoral. Therefore, it is essential to build a brain simulation system for research purposes. Constructing a digital brain starts from simulating successfully spiking neural network (SNN) models. The SNN model is the neuron model in a network setting. Despite its complexity, SNN models have been well-developed so that simulation on those models is providing plenty of insight on brain operations. Depending on the information needed to extract from the simulation, a different model of neuron should be employed [1]. Simulations based on those models could have a large impact on society such as the repair of damaged part of the human brain. However, this simulation is costly because of the high-demanding requirements of a large-scale SNN. A meaningful simulation does require to be performed on a sufficient number of neurons. Moreover, the simulation time should be fast enough to represent real time brain processing. With the above reasons, research on how to simulate neural networks efficiently has both practical and scientific impact.

1.1 Problem statement

Recently, neural science research has shifted from Artificial Neural Networks to Spiking Neural Networks (SNN). SNN was inspired by the idea that neuronal signals are short electrical pulses and the frequency of those electrical pulses carries biological meanings on brain activity [2]. Based on this idea, various models have been developed from simplified forms to complex computational ones. It is intuitive to say that the complex models which carry more biological meanings often requires large computational resources, hence simulating activities of thousands of nerve cells in real time is inefficient even with the help of supercomputers. In addition to the fact that a large-scale and real-time neural network simulation is necessary to bring out useful biology information, the huge supercomputer system has enormous power consumption, while the whole brain activity seems to consume a very small amount of power. This reveals the need of suitable platforms to carry out simulation on neuron models.

Due to the computational intensiveness and high-speed interconnectivity of neural networks, simulations on platforms such as supercomputers or FPGA have been carried out successfully but for a limited number of neurons. Experiments performed on models such as Hodgkin-Huxley model have shown that with only tens of coupled spiking neurons in real time, some significant results were able to be drawn on neuron behavior [1]. However, the simulation on the full operation of brain activity, which has not yet been performed successfully, and promises to clear out much more on brain activities, triggers the interest of researchers in the field.

In the mean time, the recent trend is performing complex scientific applications on Graphics Processing Units (GPU). In a system with GPUs, the operations which are computationally expensive can be processed by GPUs instead of CPUs. GPUs have been developed to utilize cores in order to run multiple threads in parallel while these threads can perform operations on streaming data. Hence, GPUs are able to execute computationally intensive programs much faster than CPUs. Along with GPU, Open Computing Language (OpenCL) and the CUDA framework have been introduced to fully support GPU programming. Those frameworks allow general application programmers to exploit GPU without the concern of graphical operations. Both Nvidia's GPU with CUDA and AMD's GPU with OpenCL [3] have been proposed to suit the computationally demanding needs of high performance systems. With the portability, OpenCL is preferably used in comparison of different implementations on different platforms [4].

Since SNN simulation is considered as a complex computational application, the combination of GPU and neural network simulation will be promising to produce an efficient simulation. Several projects have been carried out following this idea with the purpose of evaluating GPU performance on computationally expensive applications in general and on SNN simulation in particular. Even though the SNN models used in previous projects are single neuron models without a network setting, the results showed that implementations on GPU have significant improvements in performance over the other platforms [5], [6]. The achieved performance came up to 857.3 times for Nvidia's Tesla C2050 over the serial implementation on Intel CPU Core 2 Quad [4]. These results encourage similar research on a more complex SNN model, for example an SNN model in a network setting.

In this thesis, simulation of a neuron model in a network setting is performed in order to generate an efficient implementation on a GPU platform which provides significant biological meanings. The purpose of this thesis is to find out the answers for the following questions:

- How much speed-up can be achieved when implementing a particular SNN model on a GPU platform?
- Does the implementation on different platforms provide different performance results? What is the reason for these differences?
- Is it more beneficial to implement IO model on a particular platform?
- Does the application performance depend on the precision accuracy of the simulation, eg. single precision and double precision?
- What is the upper bound of the number of neurons which can be simulated on a particular GPU platform?

1.2 Thesis objectives

This thesis is aimed to achieve the following objectives:

- Neural model selection: A number of models should be studied and evaluated with respect to their computational complexity, biological significance and suitability to map on a GPU platform. Then one model is chosen to be implemented.
- Model implementation on GPU platform: A selected model should be implemented using CUDA on Nvidia's Tesla C2075 and Nvidia's GeForce GT640. The implementation is supposed to have a considerable number of neural cells and a fast simulation to approach real time functions of the SNN.
- Comparison with the corresponding implementation on CPU: The results will be evaluated in comparison with similar model implementations on other platforms such as CPU using the C language.
- Comparison between two GPU platforms: The performance of the implementation on the two chosen platforms is compared to evaluate the difference between the two latest architectures of Nvidia's GPUs.

1.3 Thesis outline

This thesis is organized as follows:

- Chapter 2 presents the theoretical neuron models to simulate SNN.
- Chapter 3 performs analysis on the GPU platforms as well as the programming language CUDA to suit the selected model.
- In Chapter 4, the detailed neuron model of the Inferior Olive cell in a network setting, investigation of the implementation on CPU and its implementation on the GPU platform will be discussed.
- The simulation setup, achieved results and discussions on those results will be covered in Chapter 5.
- Finally, Chapter 6 concludes the thesis and proposes further research on the topic.

Model for brain simulation 2

A biological neuron model is the properties of nerve cell in the form of mathematical representation. To understand the importance and behaviors of a neuron model in general, fundamental knowledge on the human brain, central nervous system, neuron dynamics and various neuron models are introduced in this chapter. The comparison among different neuron models at the end of this chapter shows the importance of Hodgkin-Huxley model in the field.

2.1 Brain, neural networks and neurons

Human brain is considered as the most important and sophisticated organ of the human body because it is the irreplaceable physical structure to generate mind. It is capable of processing information at a very high speed, low power consumption and control different peripheries simultaneously. Inspired by its special abilities, artificial intelligent and robotic control concepts have been developing endlessly. However, a system that is able to perform the same functions is still a challenge to scientists in the field. Therefore, more and more research is being carried out to further understand the human brain. To explain the operations of the human brain, many ideas were proposed and tested using various experiments. Franz Joseph Gall - a German physician and neuron-anatomist proposed that different regions in the brain are responsible for different function. His idea is currently approved as a cornerstone of modern brain science.

By performing various experiments to monitor the activity of the brain in while it is engaging a specific task, three main regions of the brain can be distinguished as the hindbrain, midbrain, and forebrain. The brain together with the spinal cord are called the central nervous system. The central nervous system is a broader concept which contains seven parts: the spinal cord, medulla oblongata, pons, cerebellum, midbrain, diencephalon and the cerebral hemispheres, as illustrated in Figurge 2.1. The central nervous system has bilateral symmetry which means that it consists of two symmetrical parts. Each of these parts is working together or individually to construct human behaviors and perceptions. The spinal cord is a system spreading out to get information from skin, joints and muscles to control movements of limbs and trunk. The spinal cord and the brain are connected through the brain stem to regulate levels of arousal and awareness. The brain stem consists of the medulla, pons and midbrain. The medulla oblongata takes over vital autonomic functions such as digestion, breathing and heart beating. The pons is the information transmitter of movement from the cerebral hemisphere to the cerebellum. The midbrain is responsible for sensory and motor functions such as eye movements and the cooperation of visual and auditory reflexes. The cerebellum regulates the force and range of movement and is related to the learning of motor skills. The diencephalon contains the thalamus (which processes information reaching the cerebral cortex from the other part of the central nervous system) and the hypothalamus (which controls autonomic, endocrine and visceral functions). The largest part of the brain is represented by the cerebral hemispheres which are a wrinkled outer layer (the cerebral cortex and the basal ganglia), the hippocampus and the amygdaloid nuclei. The basal ganglia controls motor performance, the hippocampus is responsible for memory storage, and the amygdaloid nuclei coordinate the autonomic and endocrine responses of emotional states.

Along with the central nervous system, there is a peripheral nervous system which contains receptors and effectors. Neurons communicate intimately with receptors and effectors to form a working network. Receptors provide inputs for the network of neurons by continuously moni-

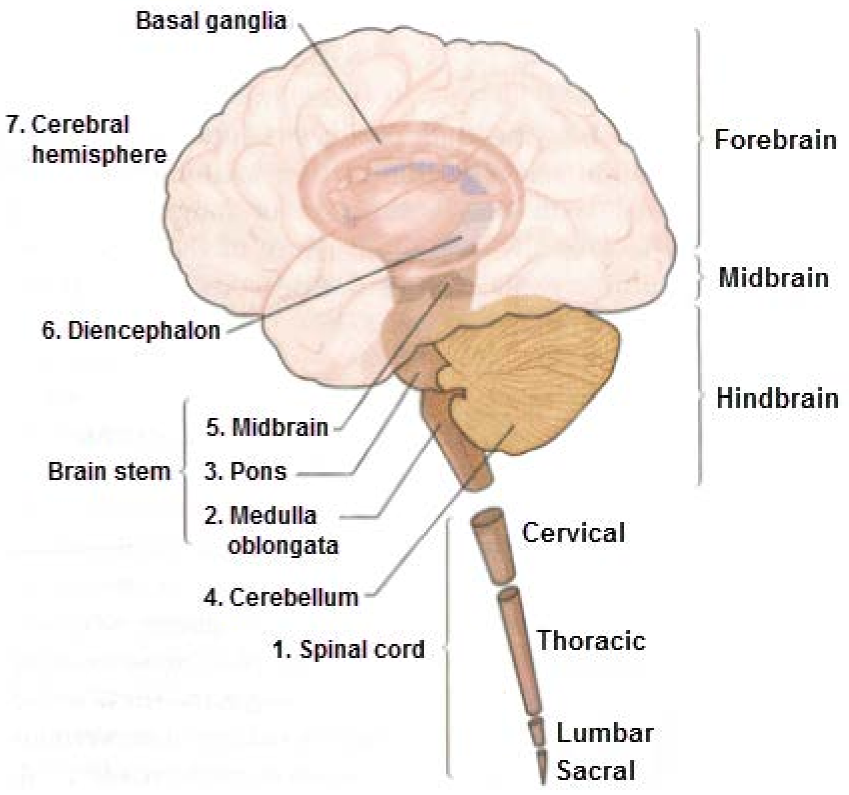


Figure 2.1: The central nervous system can be divided into seven main parts [7]

toring the external and internal environment. Neurons combine those signals from the receptors with signal encoding experiences to barrage the neurons with signals that will create adaptive interactions with the environment. Effectors receive control signals from neurons and produce appropriate activity by comparing the current and target state of the system.

The largest part of the central nervous system is the brain. The human brain contains more than 100 billion individual nerve cells which are interconnected in a system called the neural network. The neural network is a network of cells, but it is not only a chain to carry signals from input to output. It is an enormous network of interconnected cells in loops and tangled skeins. Those connections provide interactions among the input signals in addition to interactions with residues of billions of signals which already occurred in the system in the past. Therefore, not only the output signals are generated but the properties of the network are changed. Hence the prior experience will be reflected in the future behavior.

The cell in the nervous system is called the neuron. It is important to mention that there are quite a lot variations of neuron types. However, the structure and properties of different neuron types are similar.

A neuron has four elements: the cell body (soma), dendrites, axon and presynaptic terminals as shown in Figure 2.2. All those elements contribute to the generation and communication of signals among neurons. The center of the cell is called the soma, which contains the genes of the cell and the endoplasmic reticulum (an extension of the nucleus). From the soma, two types of branches reach out to create a set of short dendrites and one long axon. Dendrites have a form of tree and are responsible of receiving incoming signals from other cells, while the axon conducts signals for various distances to other neurons. The signals mentioned here are electrical signals which are defined as action potentials. A special part of the axon is the axon hillock which is located at the initial segment of the axon or at the connection between the soma and the axon. The axon hillock handles the signal initiation and transmission without loss or distortion at rates of 1-100 m per second; hence the action potential is conducted robustly to a presynaptic terminal regardless of the length of the axon cable. A synapse is a connection point between two neurons. There are presynapses and postsynapses, which are related to the transmitting neurons and receiving neurons, respectively.

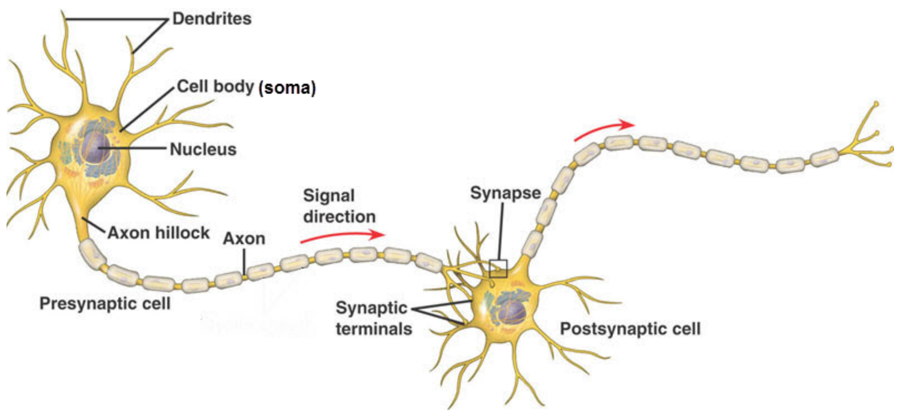


Figure 2.2: Structure of a neuron [7]

In addition to neurons, there is another class of cell in the nervous system called glial cells to support neuron's functions. Although glial cells have been shown not to be involved in information processing in neural network, they have other vital roles to neurons. In a vertebrates nervous system, the number of glial cells is between 10 and 50 times more than neurons number. Glials surround neurons but do not hold neurons together or play the role of transmitting information.

2.2 Modeling neuron behavior

Neurons are enclosed by a liquid called the membrane. The membrane has a concentration of ions which might be different from that in the surrounding environment. This difference generates the electrical potential which is the dominant idea of neuronal dynamics.

As mentioned in the previous section, input signals of a neuron are collected by neuron's dendrites. The potential of the dendrites and the soma is combined to generate the neuron's potential. The difference between the neuron's potential and membrane's potential yields an electrical potential at the axon hillock. If the potential at the axon hillock exceeds a certain threshold, a regenerative process takes place. The result of this process is a spike action potential propagating in the axon. After this process, there occurs a short refractory period in which no new impulse can be created at axon hillock.

The change in potential propagation along the axon could be described using the following equation [8]:

$$\frac{\partial V}{\partial t} = \frac{\partial^2 V}{\partial x^2} \tag{2.1}$$

In which, the starting voltage at a point on axon is V_0, and in this case, the potential will decay exponentially, which means the voltage at distance x from the starting point is described as: $V(x) = V_0 e^{-x}$

Each axon has its own specific length constant, which is defined as the distance at which the potential is reduced by a factor of 1/e. However, the cable equation seems to raise a problem. For a short axon, the propagating signal might be large enough to travel from one end to the other, but it does not apply correctly to a long axon, as the signal might fade off before reaching the other end. Therefore, most models have an assumption that in case the change in potential exceeds the threshold, a pulse can be generated will preserve full amplitude when propagating in axon. Hence, action potential is defined as an undiminished impulse of potential difference.

The propagation of action potentials is caused by flows of ions, which are mostly sodium and potassium in the membrane. For example, Hodgkin and Huxley assumed that the conductance of the membrane to sodium and potassium ions depends on the transmembrane voltage which is the potential difference between the interior and exterior of a neuron. Hence the realistic cellular equation is the one in which the conductance of sodium and potassium are varied by voltage and time.

Although the signal exchange happens between two neurons, there is still a space called the synaptic cleft between the presynapse of one cell to the postsynapse of another cell. The synaptic cleft acts like a capacitor. The transmission of action potential in this area is not electrical transfer but a chemical reaction. At the presynaptic terminal, an incoming impulse causes the release of transmitter molecules, which are stored in vesicles. The transmitter will bind to receptors on the postsynaptic membrane after traveling through a very small synaptic cleft. The transmitter might cause two types of effect which are excitatory and inhibitory. Excitatory is moving the potential difference across the postsynaptic membrane higher, while inhibitory is moving the potential below the threshold. The excitatory or inhibitory effect of the transmitters causes subthreshold changes in the postsynaptic membrane. This change may result in a generation of a new pulse in an axon of another cell if the two following conditions are fulfilled: The potential change at the axon hillock exceeds the threshold; The axon has passed the refractory period of its preceding firing.

Most neural modeling nowadays is built on excitatory and inhibitory interactions on a fast time scale. The assumption mostly used in those models is that the average rate of pulses carries most of the information. A simple neuron model is a mathematical function which takes single or many real-valued inputs and produces single or many real-value outputs. Other properties of neuron models such as linear or non-linear, static or adaptive are different from models to models. In the following section, we will present several single-cell models of neuron.

2.2.1 Formal models

A formal model [8] is a model that is the least similar to real neurons. In this model, the excitatory and inhibitory effects are combined into a single input. The neuron itself has one or more state variables which will be added to the input to produce output. The common point among formal model is that it only takes into account the excitatory, inhibitory and state variable of real neurons.

2.2.1.1 McCulloch-Pitts Model

McCulloch-Pitts Model [8] is also called binary model as it uses binary pulses to represent the value of output. Any of the active neuron's inhibitory inputs cause the output to shut off, while all the active excitatory inputs x_i are multiplied by their synaptic weights w_i and then added up. The output is set active when the total exceeds a threshold θ of the neuron.

$$Y = \begin{cases} 1 & \text{if } \sum_i w_i x_i > \theta \text{ and no inhibition} \\ 0 & \text{otherwise} \end{cases} \tag{2.2}$$

2.2.1.2 Perceptron Model

Perceptron Model [8] has an improvement in comparison with the McCulloch-Pitts model as the model can produce the real-valued output. This real-valued output represents the average firing rate of the cell. The output is calculated by the function g of the subtraction value of threshold θ from V. The function g is sigmoidal: it asymptotes zero at V<<θ and saturates at 1 for V>>θ. The advantages of this model are: the output is non-negative real-valued, and the firing rate has an upper bound.

$$V = \sum_i w_i x i \tag{2.3}$$

$$Y = g(V - \theta) \tag{2.4}$$

2.2.1.3 Hopfield neurons

There are two version of the Hopfield model: binary model and continuous-valued model [8]. In the binary model, the output of a neuron is the comparison of V and θ. However, the update of one neuron's state is carried out at a random time and independently of other neurons.

$$Y = \begin{cases} 1 & \text{if } V_i < \theta \\ 0 & \text{if } V_i > \theta \end{cases} \tag{2.5}$$

2.2.1.4 Polynomial neurons

Polynomial neurons [8] is a model which provides information on individual output contribution of inputs. Therefore, each input is correlated to other input in pairs or groups by including a multiplicative term. The output is still a sum of those products. The state variable V could be used in a usual nonlinear function g afterward.

$$V = a_1 + b_1 x_1 + b_2 x_2 + c_1 x_1^2 + c_2 x_1 x_2 + \dots \tag{2.6}$$

2.2.2 Biophysical models

Biophysical models take neuron properties into consideration and produce spikes instead of continuous-value outputs; hence they are more similar to real neurons.

2.2.2.1 Integrate-and-fire models

Integrate-and-fire models [8] is a family of models that is based on the concept of dividing membrane behavior into 2 distinct phases. The first phase is integration which performs addition on the inputs. Then, a sudden firing is carried out in the second phase. The cell voltage is assumed to be zero at the beginning, and raised or lowered by the input signals. If the voltage exceeds a certain threshold θ, the cell immediately fires an output and resets the voltage. After firing, the cell goes into the refractory period.

The simplest form of this model is a leak-free capacitance model as shown in the Figure 2.3. DC current is the input of the capacitance, which acts like a relaxation oscillator or a current-to-frequency converter. The output is produced as periodic pulses at a rate determined by the input current.

Another form of this model is leaky integrate-and-fire model which adds the leaky resistance in parallel to the capacitance as shown in Figure 2.4. In this model, the firing only occurs when the excitatory input is strong enough to overcome the leak. The time constant τ=RC divides the model's operations into two qualitatively distinct regimes: temporal integration and fluctuation

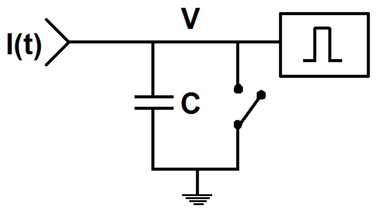


Figure 2.3: An integrate-and-fire unit [8]

detection. When τ is larger than the mean time between output spikes, the model is temporally integrating the inputs. When τ is smaller than the average intervals of output spikes, the output voltage is brought toward threshold by the rare fluctuation of the input. Therefore, the binary output is a function of the timed threshold-crossing computation. In the following section, a number of equations for several types of integrate-and-fire model are explained.

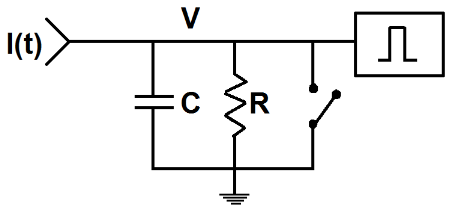


Figure 2.4: Leaky integrate-and-fire model [8]

- Leaky integrate-and-fire

$$\begin{cases} \frac{dv}{dt} = I(t) + a - bv & \text{if v } < v_{thresh} \\ v = c & \text{if v } \geq v_{thresh} \end{cases} \tag{2.7}$$

  where v is the membrane potential, I is the input current, and a, b, c and v_{thresh} are the parameters [1].

- Integrate-and-Fire with Adaptation

$$v' = I + a - bv + g(d - v) \tag{2.8}$$

$$g = \frac{e\delta(t) - g)}{\tau}$$

  where v is the membrane potential, I is the input current, g is activation gate, and a, b are the parameters [1].

- Integrate-and-Fire-or-Burst

$$v' = I + a - bv + gH(v - v_h)h(v_T - v) \quad if \quad v = v_{thresh}, \quad then \quad v \leftarrow c \tag{2.9}$$

$$h' = \begin{cases} \frac{-h}{\tau^-} & \text{if v } > v_h \\ \frac{(1-h)}{\tau^+} & \text{if v } < v_h \end{cases}$$

where v is the membrane potential, I is the input current, h is inactivation of the calcium T-current, g, v_h, v_T, τ^-, τ^+ are the parameters describing dynamics of the T-current, and H is the Heaviside step function [1].

- Reasonate-and-Fire

$$z = I + (b + \imath\omega)z \quad if \quad Imz = a_{thresh}, \quad then \quad z \leftarrow z_0(z) \tag{2.10}$$

where z is the membrane potential, $z_0(z)$ is an arbitrary function describing activity-dependent after spike reset, and b, ω and a_{thresh} are the parameters [1].

- Quadratic Integrate-and-Fire

$$v' = I + a(v - v_{rest})(v - v_{thresh}) \quad if \quad v = v_{peak}, \quad then \quad v \leftarrow v_{reset} \tag{2.11}$$

where v is the membrane potential, I is the input current, v_{rest} and v_thresh are the resting and threshold values of the membrane potential [1].

2.2.2.2 The Hodgkin-Huxley Model [9]

Hodgkin-Huxley model is based on the idea of ion flow from the membrane to the extra-cellular fluid. The concentration gradient gives rise to a tendency for sodium ions, which have a higher concentration in the extra-cellular fluid, to flow into the cell. If there is electrical gradient which is large enough to cancel the concentration gradient, a reversal potential effect occurs, where the net flow of sodium ions will be zero.

Hodgkin-Huxley used an equivalent electrical circuit to model the membrane. At the quiescent state, a negative voltage is maintained inside the neuron. The cell membrane acts like a capacitor. Ionic channels of electrical charge carriers such as Na^+, K^+, Cl^- and Ca^{2+} are described by continuous, deterministic equations. There are two models of ion channel in these equations. The nonlinear channel is described by a conductor. The ohmic channel is represented as a resistor coupled with a capacitor creating a time constant τ to represent the model. An action potential is produced when the membrane is depolarized enough to open the sodium channels. This process triggers the fast positive feedback event of a spike. After that, the refractory period occurs.

In the circuit shown in Figure 2.5, the membrane potential depends on 3 conductances: a voltage-independent (passive) leak conductance g_L, a voltage-dependent (active) sodium conductance g_{Na}, and an active potassium conductance g_K. Each conductance is connected in serial with the respective reversal potentials of the ionic currents E_L, E_{Na} and E_K. As the resistivity of the external medium is assumed to be negligible, the circuit is connected to ground. Based on Figure 2.5, the Hodgkin-Huxley model can be represented by the following equations:

$$-C\frac{dE}{dt} = m^3 h g_{Na}^{-}(E - E_{Na}) + n^4 g_K^{-}(E - E_K) + g_L^{-}(E - E_L) - I \tag{2.12}$$

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

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

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

where I is the total ionic current across the membrane, m is the probability that one of the three required activation particles contributed to the activation of the Na gate (m^3 is the

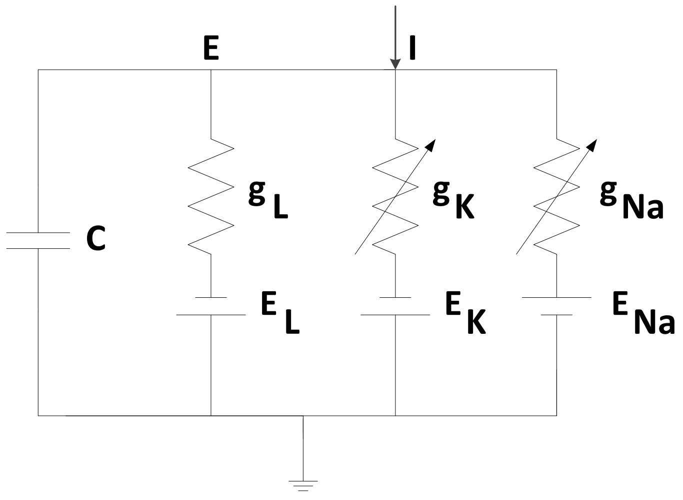


Figure 2.5: Schematic of ionic channel and neuronal membrane of Hodgkin-Huxley Model [8]

probability that all three activation particles have produced an open channel), h is the probability that the one inactivation particle has not caused the Na gate to close, g_{Na} is the maximum possible Na conductance, E is the total membrane potential, E_{Na} is the Na membrane potential, n is the probability that one of four activation particles has influenced the state of the K gate, g_K is the maximum possible K conductance, E_K is the K membrane potential, G_L is the maximum possible leakage conductance, E_L is the leakage membrane potential, α_n, α_m, α_h are the constant rate for particle not activating a gate, β_n, β_m, β_h is the constant rate for particle activating gate [10].

2.2.2.3 FitzHugh-Nagunmo

The model is a simplification of Hodgkin-Huxley model. It describes the same kind of sub-threshold behavior and limit cycle oscillations but uses only two equations instead of four equations. The reduction is done by isolating the mathematical properties of excitation and propagation from the electrochemical properties of Sodium and Potassium ion flow.

$$v' = a + bv + cv^2 + dv^3 - u \tag{2.13}$$

$$u' = \epsilon(ev - u)$$

where v is the membrane potential, u is the membrane recovery variable, and a, b, c, d, and e are parameters which can be tuned so that the model describes spiking dynamics of many resonator neurons.

2.2.2.4 Hindmarsh-Rose [10]

Hindmarsh-Rose is another simplification of the Hodgkin-Huxley model. They figured out that there are some variables which could be replaced by constants. This led to the two first equations in a simpler form. Then they added the third equation in order to represent the movements of neurons more accurately.

$$v' = u - F(v) + I - w \tag{2.14}$$

$$u' = G(v) - u$$

$$u' = \frac{(H(v) - w)}{\tau}$$

where v is the membrane potential, u is the membrane recovery variable, F, G, H are some functions, and a, b, c, d, e are parameters [1].

2.2.2.5 Morris-Lecar [11]

Morris-Lecar is an efficient model for systems which have two non-inactivating voltage-sensitive conductances. It is a two-equation model:

$$C\dot{V} = I - g_L(V - V_L) - g_{Ca}m_\infty(V)(V - V_{Ca}) - g_K n(V - V_K) \tag{2.15}$$

$$\dot{n} = \lambda(V)(n_\infty(V) - n)$$

where

$$m_\infty(V) = \frac{1}{2}(1 + \tanh\frac{(V - V1)}{V_2})$$

$$n_\infty(V) = \frac{1}{2}(1 + \tanh\frac{(V - V3)}{V_4})$$

$$\lambda(V) = \bar{\lambda}\cosh\frac{(V - V3)}{2V_4}$$

with parameters C = 20 μF/cm^2, g_L = 2 mmho/cm^2, V_L = -50 mV, g_{Ca} = 4 mmho/cm^2, V_{Ca} = 100 mV, g_K = 8 mmho/cm^2, V_K = -70 mV, V_1 = 0 mV, V_2 = 15 mV, V_3 = 10 mV, V_1 = 10 mV, λ = 0.1 s^{-1}, and applied current I (μA/cm^2) [1].

2.2.2.6 Wilson Polymonial Neurons [12], [13]

Wilson Polynomial neuron is a model whose equation simplification is not based on biological properties but mathematical methods. Efficient numerical algorithm was used to calculate the voltage and current in the Hodgkin-Huxley equations. By some approximations, Wilson acquired the following equations [12]:

$$C\frac{dV}{dt} = -m_\infty(V - 0.5) - 26R(V + 0.95) - g_T T(V - 1.2) - g_H(V + 0.95) + I \tag{2.16}$$

$$\frac{dR}{dt} = \frac{1}{\tau_R}(-R + R_\infty(V))$$

$$\frac{dT}{dt} = \frac{1}{14}(-T + T_\infty(V))$$

$$\frac{dH}{dt} = \frac{1}{45}(-H + 3T)$$

where V is membrane potential, R is recovery variable, T and H are the model conductance variables, m is Sodium activation, g_T, g_H, and τ are parameters.

2.2.2.7 Spiking Model by Izhikevich [14]

Izhikevich used a bifurcation methodologies to reduce Hodgkin-Huxley model to a two dimensional models:

$$v' = 0.04v^2 + 5v + 140 - u + I \tag{2.17}$$

$$u' = a(bv - u)$$

with the auxiliary after-spike resetting

$$\text{if v} \geq + \text{ 30 mV then } \begin{cases} v \leftarrow c \\ u \leftarrow u + d \end{cases}$$

where v is the membrane potential, u is the membrane recovery variable, and a, b, c, d are parameters given in [1].

2.2.3 Extended models

Extended models take more information on the properties of axon and dendrite into consideration. Those additional elements complicate the model and calculations. Besides, the information of dendrite, axon and even neuron properties have not been investigated fully. Therefore, those models are not employed frequently.

2.2.3.1 Modified Single-Point Models

In order to get a more realistic model, additional variables and ionic currents should be taken into account. Those additional parameters could be the concentration of free, intra-cellular calcium and slow positive feedback currents, respectively.

2.2.3.2 Compartmental Models

In the preceding models, the spatial extent of a neuron is not taken into consideration. However, the complex dendrite and axon branches have certain effects on the cell properties. For example, signal propagation in axon is described by the cable equation 2.1, which is not accurate in the case of attenuating signals. In the compartmental model, those elements are used as a parameter in computation. To be more detailed, an equation of voltage along a passive cable is used with an assumption that the geometrical and electrical properties of the cable is uniform. In fact, by neglecting the active conductance, the non-linear ion channels are only locally uniform, but not uniform for a long distance of cable. In order to make this analytic solution more accurate, the dendritic tree is split into small cylindrical compartments which can be considered as an approximately uniform membrane potential. Adjacent compartments can be coupled by the longitudinal resistance which is determined by the compartment's geometrical properties [2].

For instance, Figure 2.6 is a model of dendritic compartment μ with membrane capacitance C^μ and transversal resistance R^μ. A longitudinal resistance $r^{v\mu}$ which has value of $\dfrac{R^v{}_L + R^\mu{}_L}{2}$ is coupled with these elements. I^μ is the external input to the compartment. If the compartment is a non-linear ion channel, an extra variable resistor is used to represent these characteristics as in the leftmost compartment.

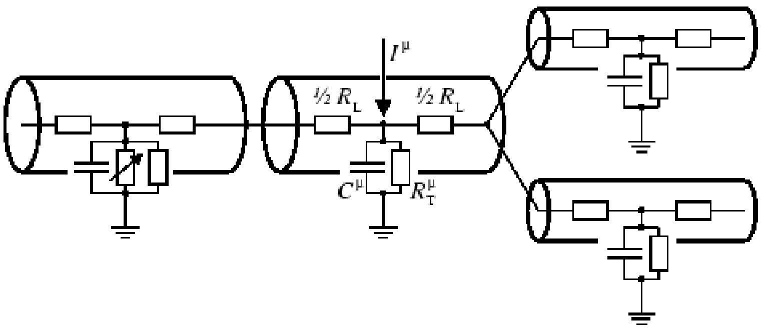


Figure 2.6: Multi-compartment neuron model [2]

2.3 Comparison of models

The comparison of neuron models can be viewed from different points of view. In this thesis, it is limited to several main factors which are related to biological significance and implementation cost of biophysical models. Based on those factors, the effectiveness of GPU implementation is evaluated in comparison with other implementations such as CPU or FPGA.

The first factor is the spiking rate under sustained currents. The real neuron's spiking rate is around 10-120 Hz, hence the ability to produce spiking frequencies in that range is very important to simulate biologically plausible neuron model. The simulation results from Figure 2.7a shows that Hodgkin-Huxley model can only produce spiking rate larger than 50 Hz and the frequency graph is non-linear. In the other words, there are some specific frequencies which Hodgkin-Huxley cannot produce. Wilson Polynomial Neurons model (Figure 2.7d) produces the same non-linear spiking rate as Hodgkin-Huxley model but in a smaller scale and range, which means that more frequencies in the range 70-120 Hz can be produced by Wilson Polynomial Neurons model as compared to the Hodgkin-Huxley model. Frequency range of the regular Spiking Model by Izhikevich (Figure 2.7b) and Leaky Integrate-and-Fire (Figure 2.7f) are similar. They can produce spikes in a frequency range of 1-120 Hz but the frequency of the Spiking Model by Izhikevich follows linear pattern while Leaky Integrate-and-Fire is non-linear. FritzHugh-Nagumo model (Figure 2.7e) is considered as the weakest model in this aspect as its highest possible frequency is less than 9 Hz.

In term of implementation cost, formal models are the most simple. Although they have little few biological significance for real neuron activities, they are useful to build artificial neural networks and also help to understand the fundamental neuron's functions without spending the complex hardware implementation. Biophysical models have a higher cost as the number of parameters and variables used in mathematical equations increases. According to Table 2.1, the number of register to store variables of the Hodgkin-Huxley model and Wilson Polynomial Neurons are two times bigger than that of other models. The storage requirement of the integrate-and-fire model family is the most modest as each model has only one variable.

In term of complexity, Table 2.1 is included here to summarize the complexity of all the models which are represented in the form of an Ordinary Differential Equation (ODE). Those equations are solved by a fixed-step first order Euler method x(t+τ) = x(t) + τf(x(t)) with the integration time step τ so that a reasonable numerical accuracy is achieved. All models are simulated in the time step τ of 1ms. The result number of Floating point Operations (FLOPs) is referenced from [1]. The Hodgkin-Huxley model is again the most complex as it requires 1200 FLOPs for 1ms of simulation. In comparison with Integrate-and-Fire which needs only 5 FLOPs for the same simulation, there is a large gap between them in both biological significance and

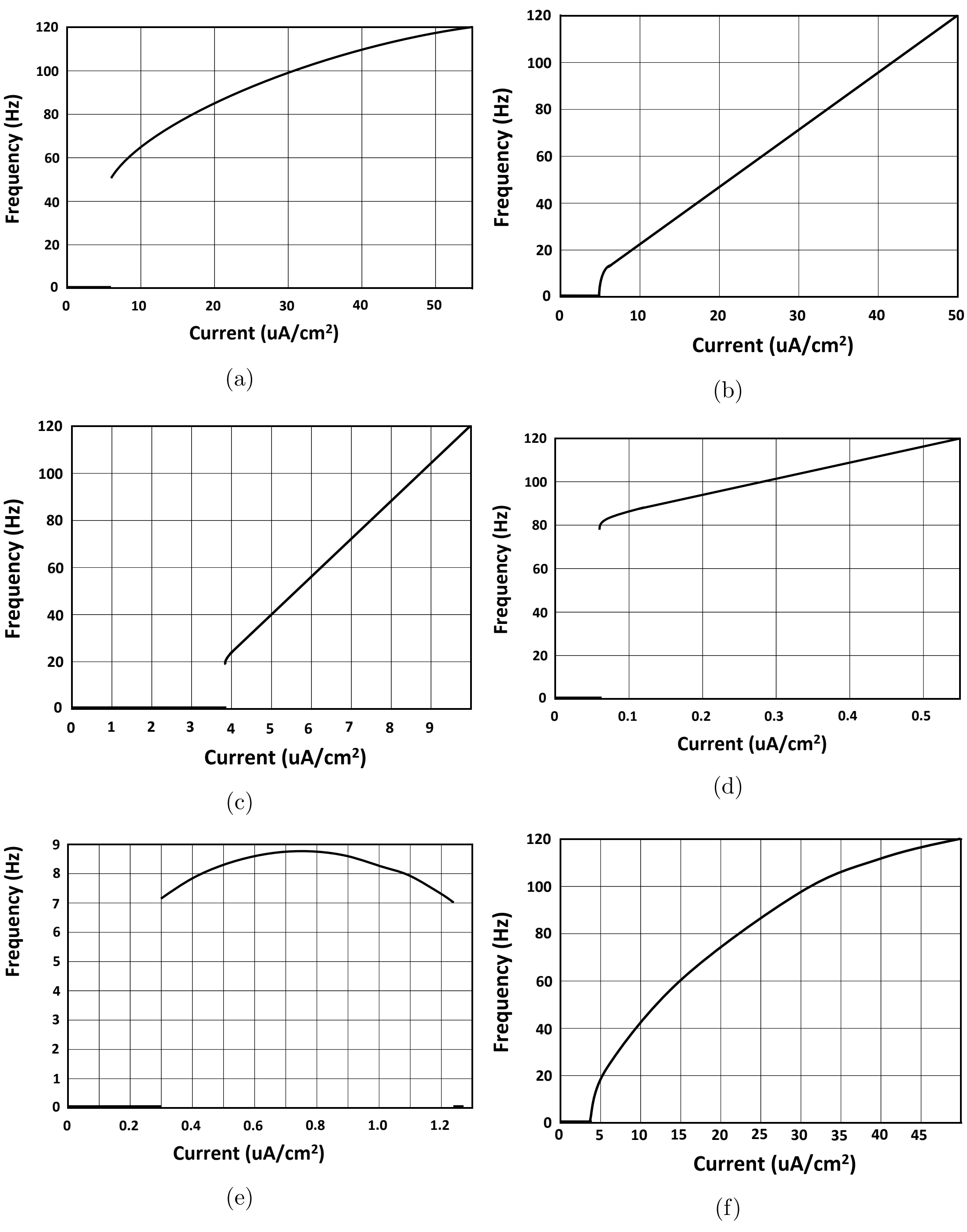


Figure 2.7: Spiking rate of neuron models [15]

complexity. The Morris-Lecar model is also very expensive with 600 FLOPs for 1ms of simulation as it is related to hyperbolic tangents and exponents.

The paper [1] also evaluated the computational cost of different models over the ability of

Model	No. of variable	Complexity (FLOPs)
Integrate-and-Fire	1	5
Integrate-and-Fire with Adaptation	1	10
Integrate-and-Fire-or-Burst	1	between 9 and 13
Resonate-and-Fire	1	10
Quadratic Integrate-and-Fire	1	7
Spiking Model by Izhikevich	2	13
FritzHugh-Nagumo	2	72
Hindmarsh-Rose	2	120
Morris-Lecar	2	600
Wilson Polynomial Neurons	4	180
Hodgkin-Huxley	4	1200

Table 2.1: Model comparison

the model in representing different properties of neurons. All the models are evaluated on the number of neuronal properties that they can express. Empty box means that the model can express those properties in theory, however the author failed to tune the parameters to achieve the properties in the limited amount of time. Some properties are mutually exclusive such as resonator and integrator, hence it is impossible for a model to have them both. According to Table 2.1, Hodgkin-Huxley and Wilson can express almost all biological features. In the meantime, the integrated-and-fire model can only exhibits 3/22 properties.

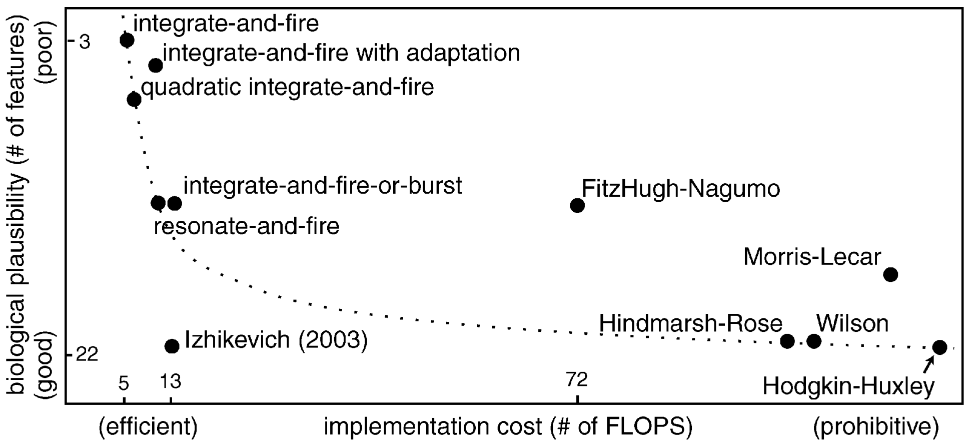
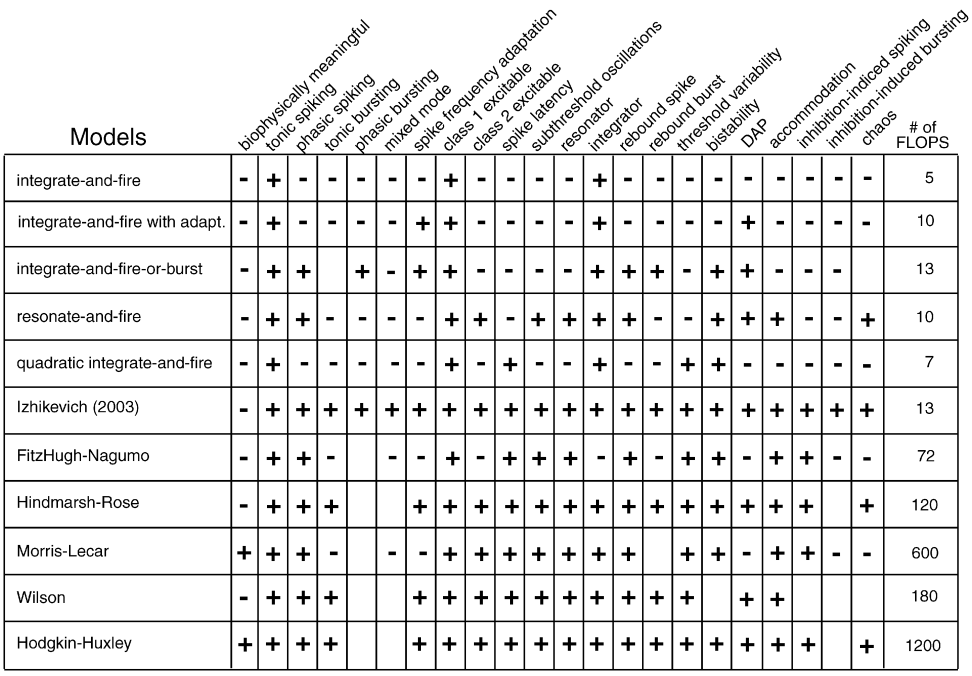


Figure 2.8: The approximate number of floating point operations needed to simulate the model during 1ms time span [1]

The overall comparison of all models in term of complexity and biological significance is described in Figure 2.8. The model of Izhikevich is the most efficient model with the lowest implementation cost and high biological plausibility. One conclusion could be drawn for other models on the curve in Figure 2.8: The more a model costs, the more biologically plausible it is. This observation is understandable as the more neuron properties are considered, the more operations of the model are needed.



Models	biophysically meaningful	tonic spiking	phasic spiking	tonic bursting	phasic bursting	mixed mode	spike frequency adaptation	class 1 excitable	class 2 excitable	spike latency	subthreshold oscillations	resonator	integrator	rebound spike	rebound burst	threshold variability	bistability	DAP	accommodation	inhibition-indiced spiking	inhibition-induced bursting	chaos	# of FLOPS
integrate-and-fire	-	+	-	-	-	-	-	+	-	-	-	-	+	-	-	-	-	-	-	-	-	-	5
integrate-and-fire with adapt.	-	+	-	-	-	-	+	+	-	-	-	-	+	-	-	-	-	+	-	-	-	-	10
integrate-and-fire-or-burst	-	+	+		+	-	+	+	-	-	-	-	+	+	+	-	+	+	-	-	-		13
resonate-and-fire	-	+	+	-	-	-	-	+	+	-	+	+	+	+	-	-	+	+	+	-	-	+	10
quadratic integrate-and-fire	-	+	-	-	-	-	-	+	-	+	-	-	+	-	-	+	+	-	-	-	-	-	7
Izhikevich (2003)	-	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	13
FitzHugh-Nagumo	-	+	+	-		-	-	+	-	+	+	+	-	+	-	+	+	-	+	+	-	-	72
Hindmarsh-Rose	-	+	+	+			+	+	+	+	+	+	+	+	+	+	+	+	+	+		+	120
Morris-Lecar	+	+	+	-		-	-	+	+	+	+	+	+	+		+	+	-	+	+	-	-	600
Wilson	-	+	+	+			+	+	+	+	+	+	+	+	+	+		+	+				180
Hodgkin-Huxley	+	+	+	+			+	+	+	+	+	+	+	+	+	+	+	+	+	+		+	1200

Figure 2.9: The biological significance of biophysical models [1]

In conclusion, it is very expensive to simulate complex models with as many neurons and interconnection as in the brain. Axons and dendrites also play important roles in neuron modeling. None of the aforementioned models, except for extended models, considers the shape and properties of axons and dendrites. For this thesis, GPUs are used to evaluate the performance of a biophysical model on a high performance computing platform. Hence, the chosen model is allowed to be quite complex and has a large biological significance. Besides, Hodgkin-Huxley is considered as the most successful model of all of computational neuron-science models [8]. However, it is extremely expensive to simulate the model; hence only a small number of neurons could be simulated in real time. This encourages the GPU implementation of the neural network with a large number of neurons to get detailed information on neuron activities and their interactions. Therefore, the Hodgkin-Huxley model is the model of choice to simulate in an increasing number of cells to figure out the upper bound number of cell while still ensuring real-time execution.

Platform analysis 3

Graphic Processing Units (GPUs) were introduced initially to support the graphical function of a computer. In the previous years, increasing the number of cores and clock frequency have been the main solution to build faster computing systems, however it is becoming harder and harder to enhance CPU's performance using this technique. In the meantime, GPUs have been developing dramatically to be able to fulfill the requirements of high performance graphic applications as well as intensive computational applications. The usage of GPU in non-graphics applications started in 2003. With its high performance on floating point operations and programmability, GPUs emerged as an ideal processor to accelerate data parallel applications. However, the GPU architecture also had many drawbacks in parallel programming. OpenGL and DirectX, which were the only programming languages for GPU at the time, were not used widely by general applications' programmers. Moreover, mapping computational programs onto GPUs with the awareness of graphics calculations increases program complexity significantly. To solve these problems and utilize GPU performance, firstly, the CUDA framework and afterward, the OpenCL framework were introduced in 2006 to blur the difference between GPU and CPU programming and abstract the graphics processing from programmers. The following NVIDIA CUDA architecture (code-named Fermi) was developed with a better double precision performance, parallel data cache technology, a GigaThread engine and full Error Checking and Correction (ECC) support. The newest architecture (Kepler) was released in 2012 with a completely new streaming processor design SMX, dynamic parallelism and an idle time utilization Hyper-Q. For each architecture, NVIDIA developed three families of product to target different application areas: GeForce, Quadro and Tesla. While GeForce and Quadro products target consumer graphics and professional visualization, the Tesla family is designed for parallel computing and programming, and offers exclusive high performance computing features.

3.1 GPU architecture

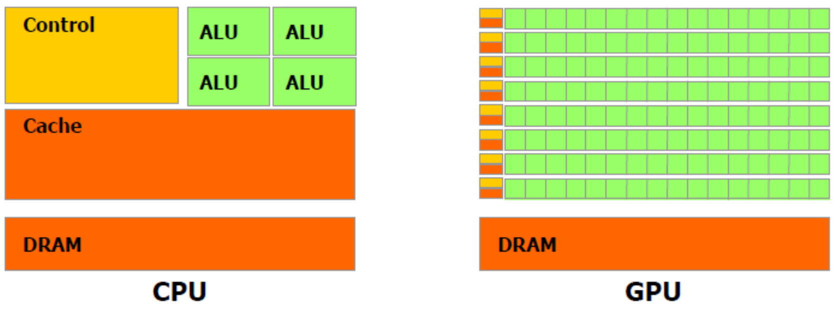


Figure 3.1: The GPU devotes more transistors to data processing [16]

A CPU exploits more and more complex control logic to achieve a high performance in executing a sequential program. Even though the program could be executed in parallel, the code appears to the programmer as sequential. On the other hand, GPU utilizes most of its transistors to do calculations of an already parallelized program. In other words, complex control management hardware is sacrificed for more cores to be able to do more computations at the same time as illustrated in Figure 3.1.

The general idea in GPU architecture is that multiple cores are organized into multiple streaming multiprocessors (SMs). An SM has a number of computing cores, control logic and a memory system. The number of cores per SM depends on each generation of GPU. Each core in an SM is called a streaming processor (SP). GPUs employ Graphics Double Data Rate (GDDR) DRAM which has a high bandwidth because if the high bandwidth requirements of graphics applications. Modern GPUs are also connected to the front-side bus to communicate directly with the CPU.

The parallel model which is used in GPUs is single-instruction, multiple-thread (SIMT). The scheduling is carried out by a scheme of multiple threads called a warp. A warp collects multiple individual threads which are of the same type and able to start together by a single instruction. The parallel threads in a warp should be free to branch and execute independently. The scheduling scheme in the SIMT model is done by firstly selecting a ready warp to execute and then executing the instructions of that warp's active threads. Each warp thread is mapped to an SP and executed independently with its own instruction address and register state. Since warps are independent from each other, the model is considered more flexible and efficient than the SIMD model because SIMD has to execute every single instruction on different data.

3.1.1 Fermi architecture

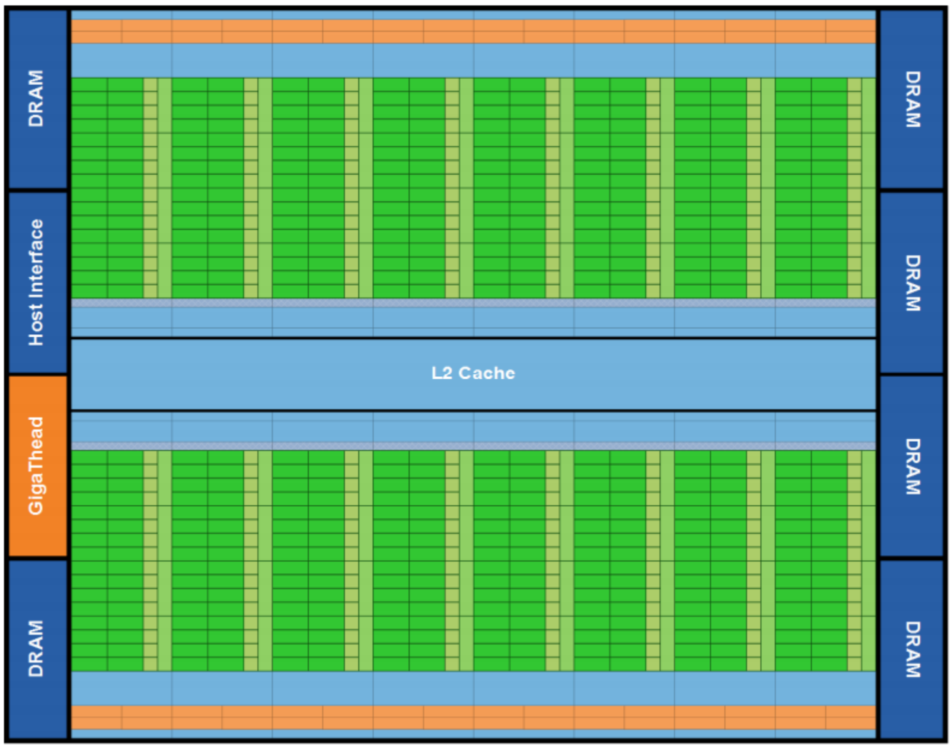


Figure 3.2: Architecture of Fermis 16 SM [17]

Fermi architecture has up to 512 CUDA cores which are organized into 16 SMs. These 16

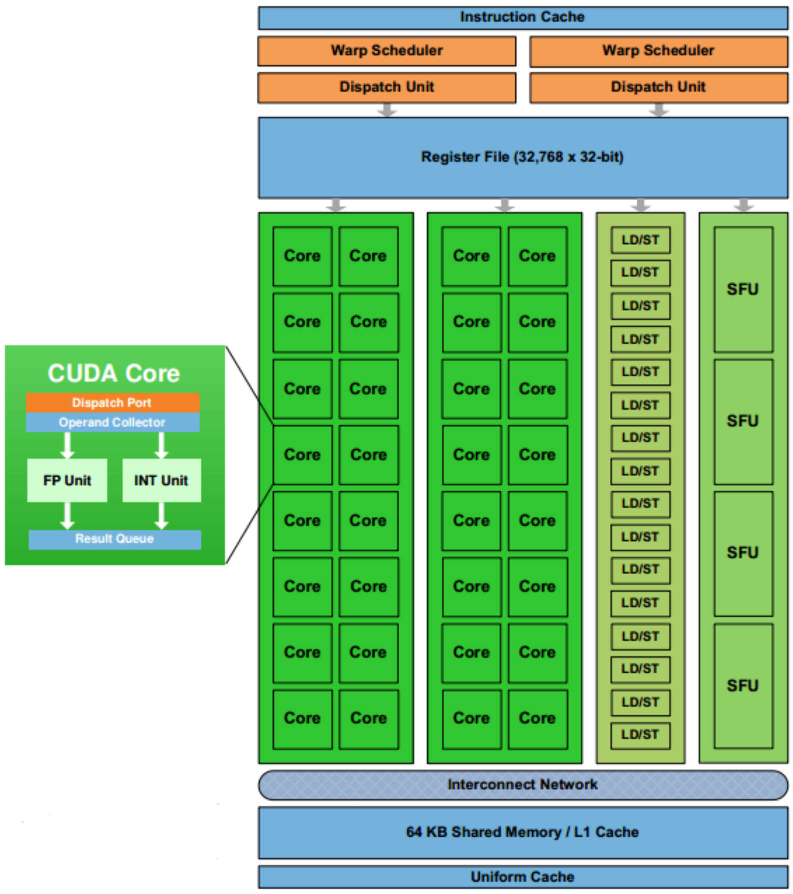


Figure 3.3: Fermi streaming multiprocessor (SM) [17]

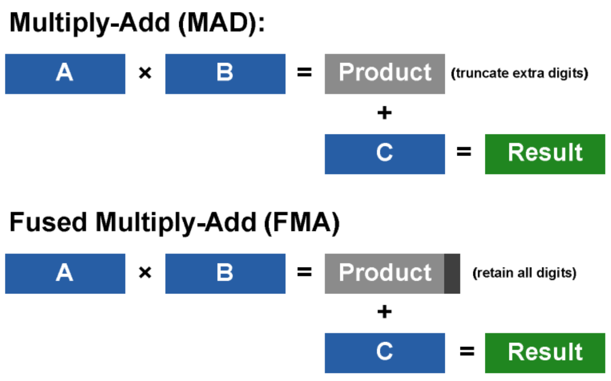


Figure 3.4: Fermi FMA [17]

SMs are located surrounding a L2 cache. Each SM has its own scheduler and dispatch, execution units, register files and a L1 cache (as shown in Figure 3.2). Fermi architecture can hold up to 6 GB of GDDR5 DRAM which is divided into six 64-bit memory interfaces or a 384-bit memory

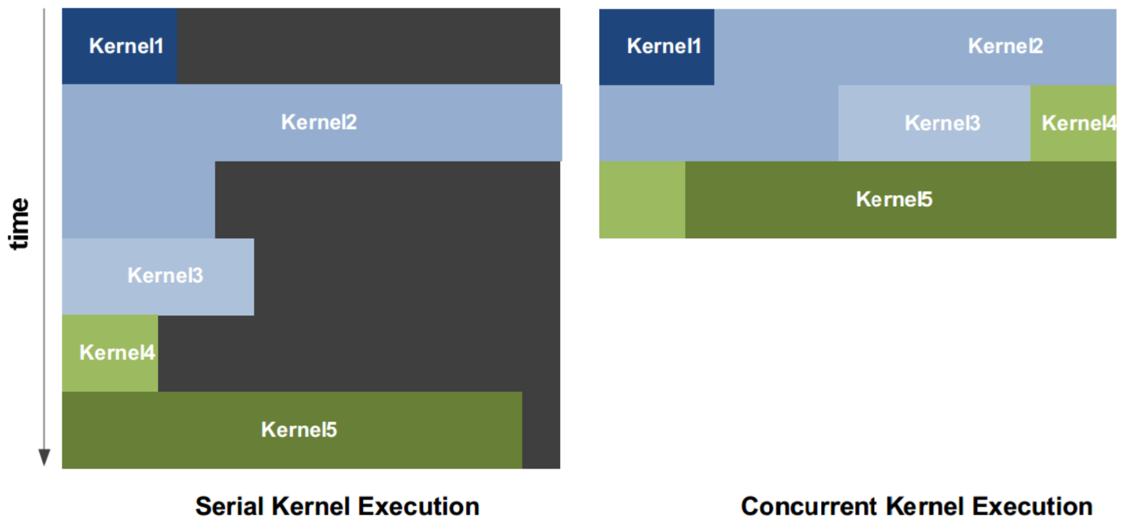


Figure 3.5: NVIDIA GigaThread engine [17]

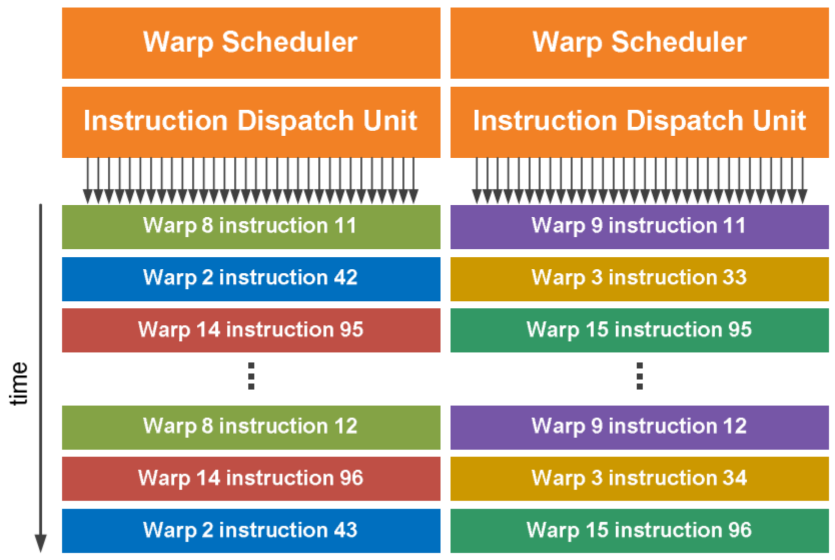


Figure 3.6: Two warp scheduler in Fermi architecture [17]

interface.

Fermi architecture shown in Figure 3.3 uses the third generation of NVIDIA SM which is more programmable and efficient than previous architectures.

The computing units of Fermi architecture include the following:

- Each SM has 32 SPs which consist of a fully pipelined ALU and a floating point unit (FPU). The ALU supports 32-bit precision for all instructions. 64-bit precision is supported for several instructions such as shift, move, convert, compare, etc. The FPU is implemented following the new IEEE 754-2008 floating point standard. It uses the fused multiply-add (FMA) instruction for both single and double precision arithmetic. The FMA is more

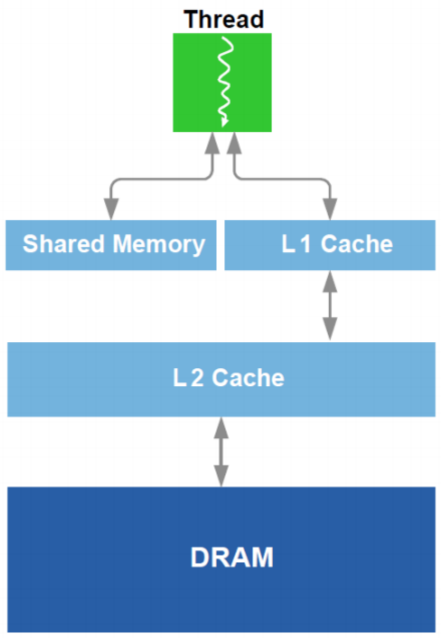


Figure 3.7: Memory hierarchy in Fermi architecture [17]

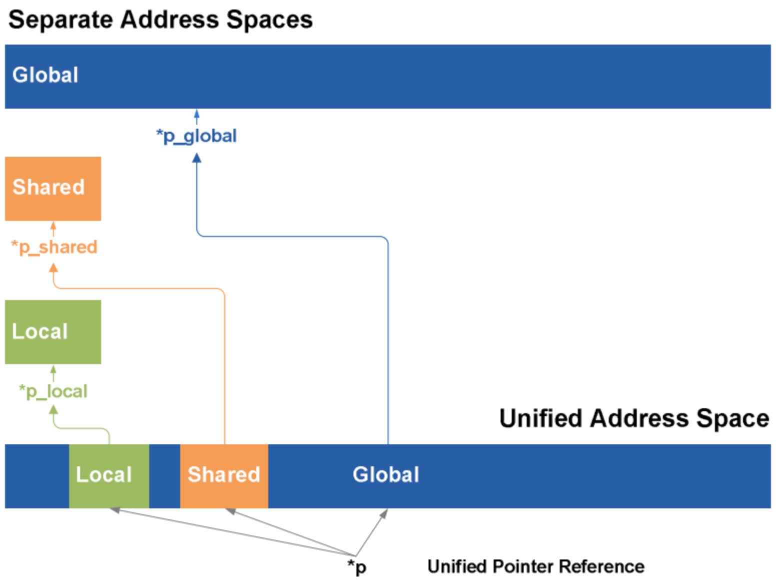


Figure 3.8: Unified Address Space in Fermi architecture [17]

accurate than mutiply-add (MAD) instruction because the multiplication and addition has no loss of precision in the addition. In the FMA, the multiplication result is not truncated its ending digits before performing addition as explained in Figure 3.4. One improvement of Fermi architecture is that 16 double precision FMAs can be executed per clock, hence the performance of the double precision calculation is 4.2 times faster than the preceding generation of GPUs. 32 cores in a SM share 16 load/store (LD/ST) units, and four special function units (SFUs).

- 16 LD/ST units permit data from 16 threads to be loaded or stored from a cache or DRAM at the same time.
- The SFU performs one of transcendental instructions such as sin, cosine, reciprocal and square root in one clock cycle. It is decoupled from the dispatch unit so that the dispatch unit can still be used even when the SFU is occupied.

About control logic, Fermi architecture has a novel technology called GigaThread Thread Scheduler as shown in Figure 3.5. It is a two-level, distributed scheduler which supports faster context switching and concurrent execution. At the chip level, a global work distribution engine forwards thread blocks to different SMs, while at the SM level, each warp scheduler does scheduling again to increase the speed of scheduling. The Fermi context switching was optimized to reduce the cost of context switching to below 25 microsecond. Besides, multiple kernels of the same application can be executed in parallel to utilize fully resources in Fermi architecture. Each SM has two warp schedulers and two instruction dispatch units, which means two warps can be issued and executed in parallel as illustrated in Figure 3.6. Therefore, Fermi scheduler is called dual warp scheduler which controls 32 threads per warp.

Fermi architecture implements a memory hierarchy as shown in Figure 3.7 with a register file, a L1 cache, a L2 cache, a shared memory and a global DRAM memory. The register file consists of 32768 32-bit registers. Fermi architecture is the first GPU architecture with true memory hierarchy and a unified load/store path. This feature ensures the coherency of data in the memory hierarchy and also eases difficulties of programmer in dealing with memory. Besides, the memory accessing time decrease dramatically if the data is cached efficiently. Furthermore, the size of shared memory and L1 cache in the total 64 KB on-chip memory can be configured depending on the requirement of an application. If the application needs a large shared memory, up to 48 KB can be used for shared data. On the other hand, when the amount of shared memory is unknown, up to 48 KB can be used as L1 cache to enhance the speed of memory access. All Fermi memories, from cache to DRAM, are equipped with Error Correcting Code (ECC) which is able to correct single-bit soft error in every data access. Besides, Single-Error Correct Double Error Detect (SECDED) ECC is also provided to detect and report all double bit errors and many multi-bit errors early to increase its reliability. Together with ECC, Fermi architecture also unifies the address space of these memories as shown in Figure 3.8. This feature makes Fermi architecture be able to support true C++ program which has variables passing via pointers.

3.1.2 Kepler architecture

Kepler architecture is the newest NVIDIA GPU architecture which targets optimal performance per watt. It was designed with three important features SMX, HyperQ and dynamic parallelism. These innovations improve performance with higher energy efficiency.

The SMs in Kepler architecture are called SMXs which aim to achieve power efficiency. The new SMX unit contains 192 CUDA cores, 32 Special Function Units (SFU), 32 load/store units. The SMX has more cores than the SM of Fermi (Figure 3.9). Each core can support more threads. Each thread has more registers. All of those characteristics result in more powerful computing units. Although more cores are employed, the new SMX consumes three times less energy than the SM of Fermi architecture. This improvement is achieved thanks to the energy-efficient design of the micro-architecture, software and system level.

To further improve the performance of Kepler architecture, HyperQ is designed to have a ”smarter” queuing scheme than that of the Fermi architecture. The difference between those two architectures is illustrated in Figure 3.10. If the Fermi architecture has a single work queue which might not able to keep the GPU busy all the time, the Kepler architecture has 32 concurrent

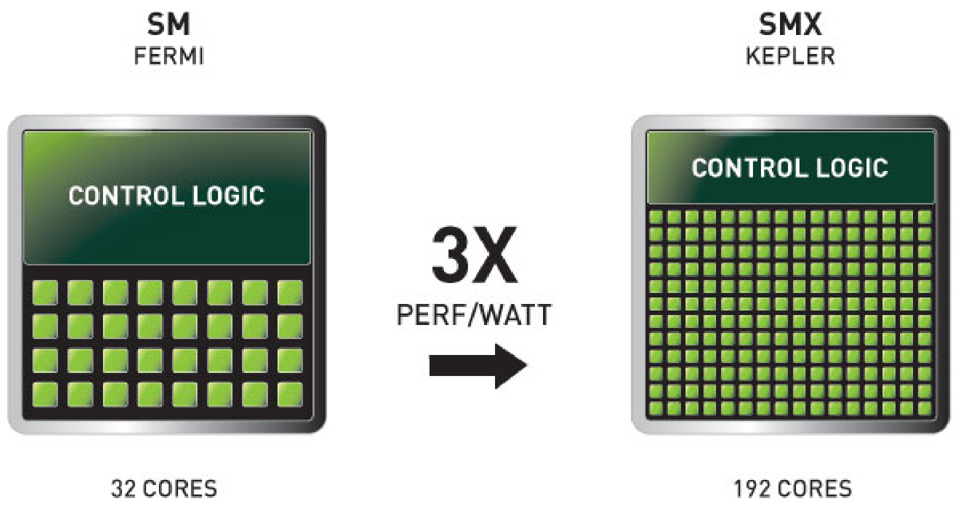


Figure 3.9: The novel SMX design of Kepler architecture [18]

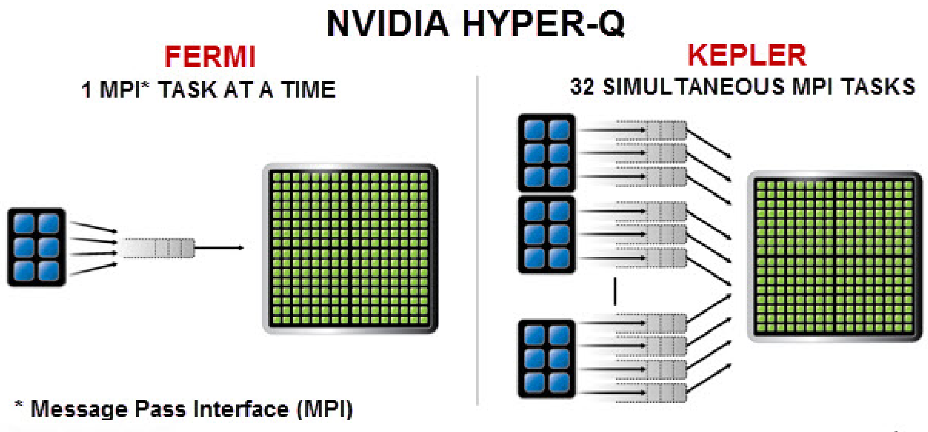


Figure 3.10: The HyperQ scheduling scheme in Kepler architecture [18]

work queues to fulfill 32 processors on GPU at the same time. This improvement decreases the idle time of the GPU cores, hence increases the throughput of executing instructions.

Another innovation technology of Kepler architecture is the dynamic parallelism. Figure 3.11 explains how this makes the Kepler architecture more efficient. In the dynamic parallelism protocol, kernels can be launched directly from the GPU side, which means nested kernels on the GPU side are allowed. This eliminates the unnecessary switch between CPU and GPU to execute kernels. Furthermore, various applications can be mapped on GPU.

3.2 CUDA framework

As GPU is designed to target data parallel applications such as graphical applications, parallelism is handled by GPUs with less effort than by CPU. However, CPU has more complex control logic,

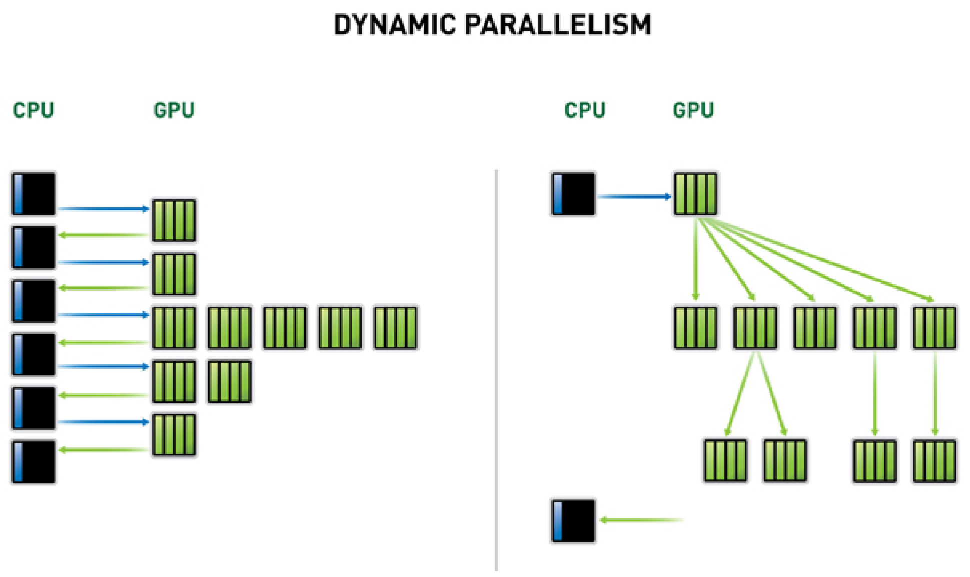


Figure 3.11: Dynamic parallelism in Kepler architecture [19]

hence the program with conditional commands is handled much better by CPU. In other words, there are tasks which are more suitable to run on CPU and vice versa. Hence, it would be beneficial to combine GPU and CPU into an unique platform where they can cooperate for better performance. The CUDA framework was introduced with this purpose.

3.2.1 CUDA program

In a CUDA framework, CPU is considered as a host which will cooperate with one or more devices called CUDA devices. CUDA devices consist of numerous arithmetic execution units which could produce massively parallel processing ability.

A CUDA program contains one or more parts which can be mapped on a host or a device. The tasks which have little or no data parallelism will be executed on a host, while parallel tasks will be performed on devices as shown in Figure 3.12. A CUDA program is a unified source code which combine both host and device code. The program is executed from host side and invoke kernels which are device code. The kernel should be able to operated in parallel on a large number of threads which are assigned to multiple cores.

CUDA device is equipped with its own memories which are separate from CPU memory in the same platform. Therefore, devices have to allocate memory and transfer data from the host memory to the device memory. After executing a kernel, device transfers the data back to the host memory and frees up the memory for further usage.

A kernel is a function which specifies the program to execute simultaneously by all threads. The kernel program should be the same for all threads but operates on different data to produce massively results for a particular computation.

The execution of kernel on GPU is assigned to multiple threads. Threads are organized into multiple equal size blocks. Multiple blocks create a grid. Figure 3.13 is an example of how threads are organized in a grid. Blocks and threads live in a 3-dimensional space, hence they are identified by an x, y, z coordinated index.

The programmer specifies the number of block per grid and the number of thread per block by using a predefined variable in a kernel call <<<...>>>. Those parameters affects the ef-

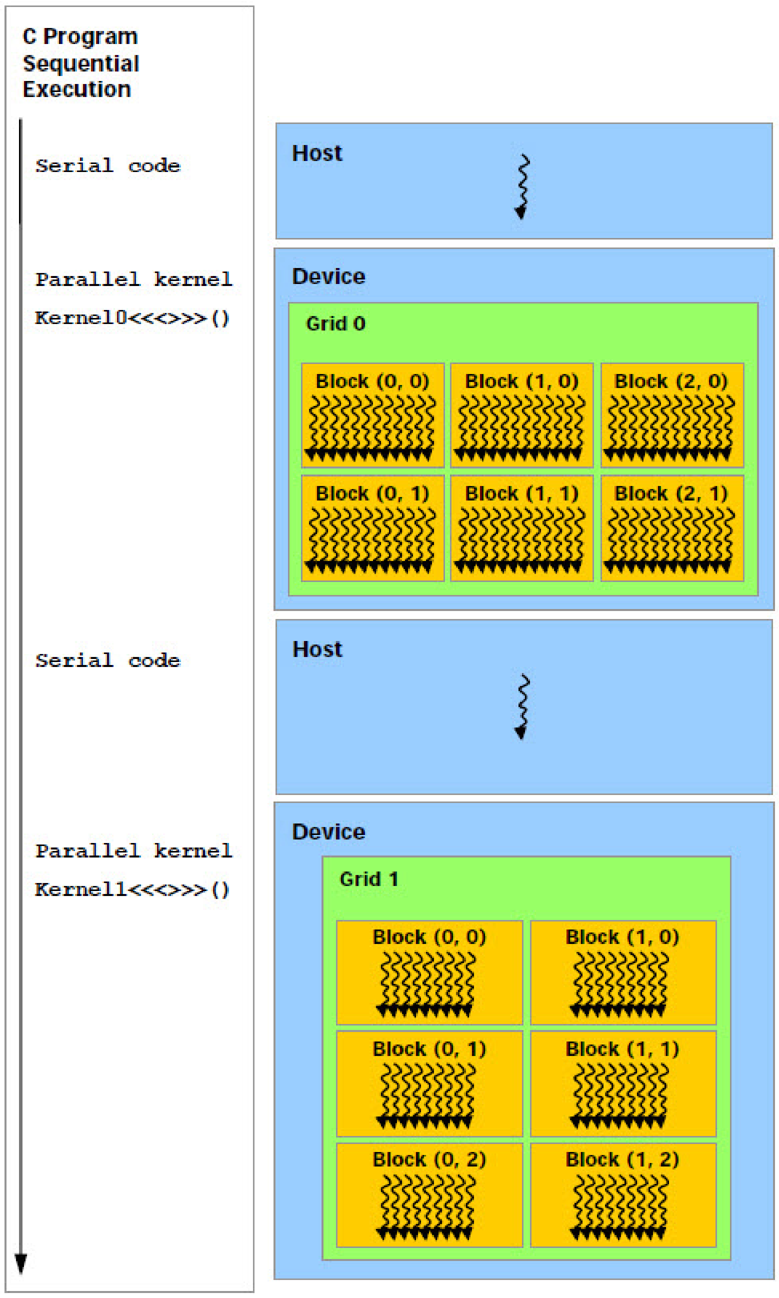


Figure 3.12: The sequence of a CUDA program in host side and device side [20]

fectiveness of the CUDA program. The block ID and thread ID are also accessible by built-in variables.

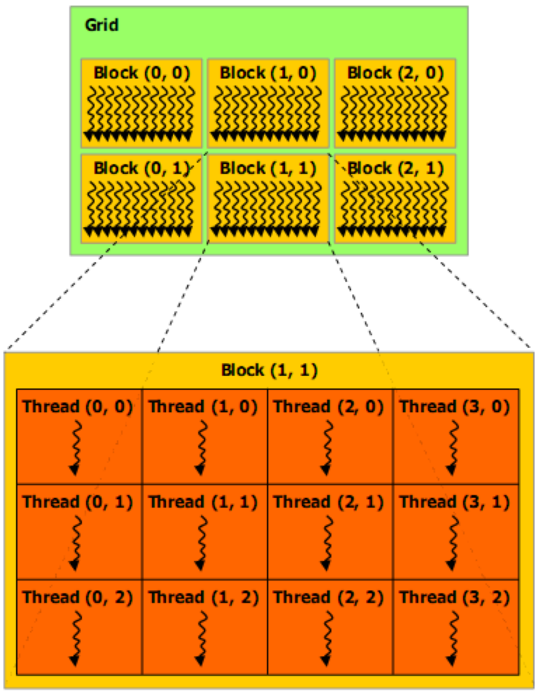


Figure 3.13: A 2D division of a CUDA grid [20]

3.2.2 CUDA memory hierarchy and manipulation

Memory hierarchy

The device memory hierarchy includes constant memory, global memory, registers and shared memory as described in Figure 3.14. The global and constant memory are used to communicate with the host memory and among devices. Each thread has its own registers (local memory) which are inaccessible by other threads. The shared memory is used by several threads within a block. It is expected to be accessed much faster than the global memory, therefore as much data as possible should be allocated on the shared memory first. However, the shared memory size is limited to only maximum 48KB per block, not all the data could be allocated on shared memory. Those memory is classified as linear memory, to be distinguished with other memory types such as texture memory or surface memory.

Memory manipulation

The memory access in a CUDA program is handled by API functions. Apart from those APIs, the new version of CUDA is equipped with more flexible operations with the device memory.

In order to control the L1 and L2 cache which can affect seriously application performance, CUDA functions are provided to allow programmers control the usage of those caches. A cache line is 128 byte long. If the data is cached both in L1 and L2 cache, the memory access is transferred with 128 byte per transaction. Whereas, memory access which is cached in L2 only is transferred with 32 bytes per transaction. By default, all data is cached through both L1 and L2 cache. For some cases, L1 cache is helpful to increase program performance. However, if the access pattern is scatter, L1 cache becomes a bottleneck for data accessing time. To disable L1

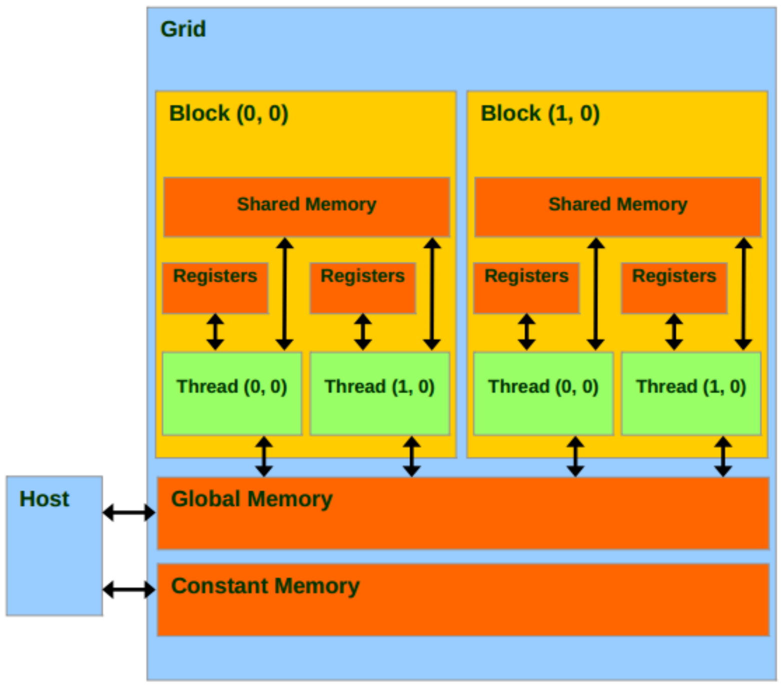


Figure 3.14: Overview of CUDA memories [20]

cache, a programmer can use the compiler option $Xptxasdlcm = cg$ [21].

Access to shared memory is much faster than global/local memory, hence more shared memory per block can be favored in some cases. In addition to the default shared memory size, the amount of extra shared memory per block can be determined inside the kernel call. Besides, the L1 cache which shares the same on-chip memory space with shared memory.

Texture memory

As GPU is designed for graphical applications which involves operations on image pixels. The environment of a pixel is highly relevant to its properties. This characteristic is also necessary for many computing applications. To exploit this ability, the texture memory accessibility is preserved in the CUDA framework. Texture memory is a read-only memory which could be considered as a read-only pointer to a global memory location. Texture memory is useful in some cases where caching principles limits the memory access pattern and bounds performance enhancement. The cache line is considered as a set of consecutive memory locations. In order to load neighboring memory locations as shown in Figure 3.15, a normal cache line needs at least three loads. However, the texture memory could use the two-dimensional coordination system to load those memory locations in one load. In CUDA framework, texture memory is provided with 1D and 2D management. The supported data type includes integer and float (single-precision floating point). Double precision floating point is also possible by using a type int2 together with the function $\_\_hiloint2double()$ which converts a variable of type int2 to type double.

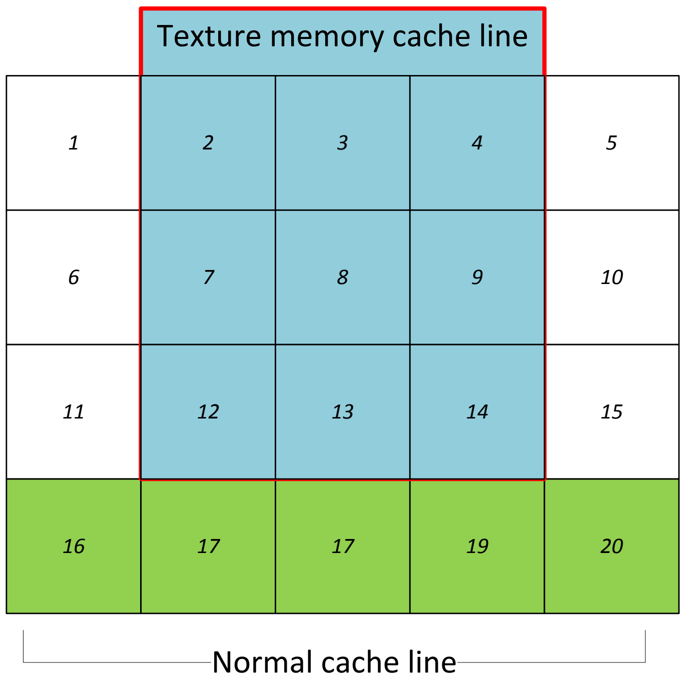


Figure 3.15: Loading pattern of texture memory

3.2.3 Exploit parallelism using CUDA

Concurrent execution

In a CUDA program, there are several methods to execute kernels concurrently. At device level, CUDA provides syntax to invoke different kernels from one host program. If the platform has multiple GPU devices, multiple kernels could run on those devices in parallel and combine results in the host program at the end.

At the task level, different CUDA kernel with different stream ID can execute on the same device in parallel. This configuration is done intentionally by programmer. This configuration increases the utilization of GPU resources if there is enough parallelism in an application.

Overlapped execution

In addition to concurrent execution, CUDA also provides overlapped execution. Overlapped execution is done between memory transactions and kernel execution using asynchronous operations.

3.2.4 Synchronization

The synchronization in a computing application program is very essential. However, CUDA is designed to target high throughput in execution, which means that as many tasks as possible are mapped to run on CUDA devices regardless of the program robustness. Although CUDA has some synchronizations among threads in a block, it has no specific method for global synchronization of all running threads. Some new APIs have been recently introduced to bring more powerful synchronization to kernel execution. However, those APIs only ensure that all the global memory allocations and shared variables are consistently written or prevent RAW (Read After Write) hazard. Up to the newest release, there is no provided global synchronization for all the threads in kernel execution.

At kernel level, an execution is synchronized when the kernel stops executing, which means no command of the kernel needs progressing anymore. This is called implicitly synchronization which might costs the kernel initialization time for one synchronization, hence it may or may not suitable to some specific applications. However, with the new ability to execute two kernels at the same time, this synchronization is not always reliable. In addition to that, the explicit synchronization using APIs such as *cudaDeviceSynchronize*(), *cuCtxSynchronize*(), *cudaStreamSynchronize*(), *cuStreamSynchronize*(), *cudaEventSynchronize*(), etc. can be used in some cases. This synchronization forces all the commands of the previous kernel execution to finish before launching a new kernel. It costs more synchronization time than the implicit synchronization because of the APIs execution time.

In many computing applications, the above synchronizations are indeed not enough. Hence, some research have been carried to figure out more synchronization method in a CUDA program [22], [23]. Those methods exploited some properties of special operations in CUDA such as *atomicAdd*() or the principle that synchronization is done by exhausting all the CUDA resources. However, those synchronization methods are very costly to many real time applications.

3.3 Model mapping on GPU

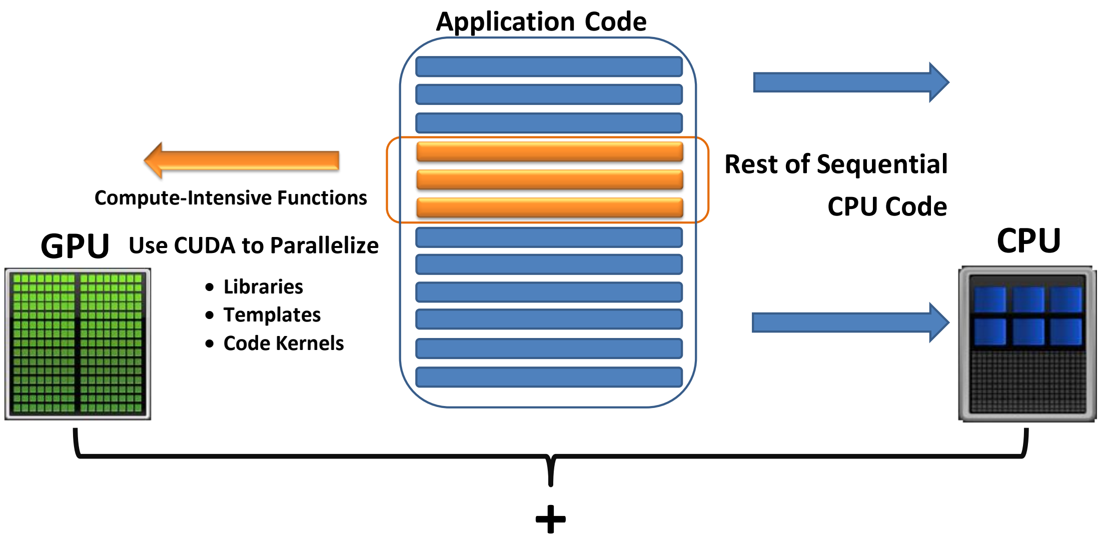


Figure 3.16: Mapping kernel to GPU while the rest of program is still executed on CPU

A neuron model is a set of equations which represents the internal dynamics of each neuron. The neuron model in a network setting represents activities among multiple neurons whose properties change according to their inputs and interactions among each other. Simulation of neural network is computationally expensive because of the number of neurons at one simulation step is up to thousands of neuron. Besides, if we target real-time simulation the output is required to be available after a short period of time which is about several microseconds. Moreover, the number of steps to simulate a neural network is enormous, up to hundreds thousand steps depending on the scale of each time step.

Fortunately, the simulation of neural network is an embarrassingly parallel problem because each neuron could be considered as a single computing element which does its own computation and communicates to others by memory accesses. Hence, the application is suitable to be mapped on a GPU platform. In comparison with simulation on costly server machine, a GPU platform can offer a cheaper and simpler platform for such problem with high efficiency.

In order to do that, a kernel which is an intrinsic calculation of neuronal properties should be extracted as illustrated in Figure 3.16. This kernel would be mapped on multiple individual CUDA threads to be executed in parallel. The kernel is spotted at the part of code which contains big iterative loops. The code in the kernel should avoid any conditional instruction so that the execution is straightforward for all kernels and avoids any delay to read updated values among kernels.

The kernel extraction should also take care of the amount of data needed to transfer to device side. If a large amount of data needs transferring to the device, the limited memory size and long memory accessing time on the device side could be the bottleneck to increase parallel program performance. Moreover, the overlapped execution between memory transactions and kernel execution could be considered as an option to reduce the execution time. In case the memory transactions is not the bottleneck of the application performance, optimizing memory usage is not helpful.

The efficiency of mapping a model on the GPU platform also includes optimizing the bandwidth of memory access. All the data on the device should be analyzed to be assigned correctly to registers, the shared or global memory. As global memory access cost is 300-400 times slower than shared memory access, the device data pattern should be allocated efficiently to utilize efficiently caching. Fortunately, the GPU platform offers multiple read and exclusive write protocol, which allows multiple kernels to read from the same memory location with high bandwidth. This eases the difficulty of implementation partly and also reveals a promising performance result. Texture memory is also suitable to reduce memory accessing time. This memory is a read-only memory, however this drawback can be overcome by synchronizing explicitly. Every time a kernel is loaded, the texture memory will be cached again.

The Tesla C2075 platform has 4GBs of global memory and 48KBs of shared memory per block. Hence, for a double precision calculation (which means that a variable of type double - 8 bytes is used), the shared memory and global memory could hold up to 6144 variables and 524288 variables, respectively. For single precision, this number increases twice. The global data used by both host and device, and the shared data used among threads per block should be less than this amount. The optimum solution for memory bandwidth can be only achieved when the transaction segment is aligned of 1, 2, 4, 8 or 16 bytes. Moreover, a L1 cache line is 128 bytes which determined the memory pitch to allocate data in global memory. Otherwise, a single access will be compiled into multiple instructions with the interleaved access pattern without coalescing. Besides, the usage of L1 and L2 cache could be also considered in the case that the data is scattered and cannot be cached in a more efficient way. In some cases, L1 cache could be disable to decrease memory accessing time. The platform has a warp of 32 threads with a dual warp scheduler, therefore the number of threads in a block should be a multiple of 32 to exploit fully the resources of a SM. Some researches show that the optimum block size in Fermi architecture is 2x128 or 2x256 where provides the minimum cache misses and the maximum occupancy [24], [25].

The GeForce GT640 platform has 2GBs of global memory and 48KBs of shared memory per block. Hence, the data mapped to global memory should be more compact in comparison with the Tesla platform. However, the number of register per block is 65536, which is twice bigger than the Tesla platform. Therefore, more local memory could be used, or more threads could be initialized in parallel.

Implementation 4

This chapter represents the GPU implementation of Inferior Olive cell in a network setting. The implementation in C language is investigated to spot its critical part which directly impacts the simulation on a computer platform. After that, the parallel implementation in CUDA is discussed step by step from constructing the implementation to optimizing its performance.

4.1 Inferior Olive model in a network setting

4.1.1 Inferior Olive cell

The cerebellum, as mentioned in Chapter 2, is the brain region which ensures the force and timing of motor activities, and related to the learning of motor skills.

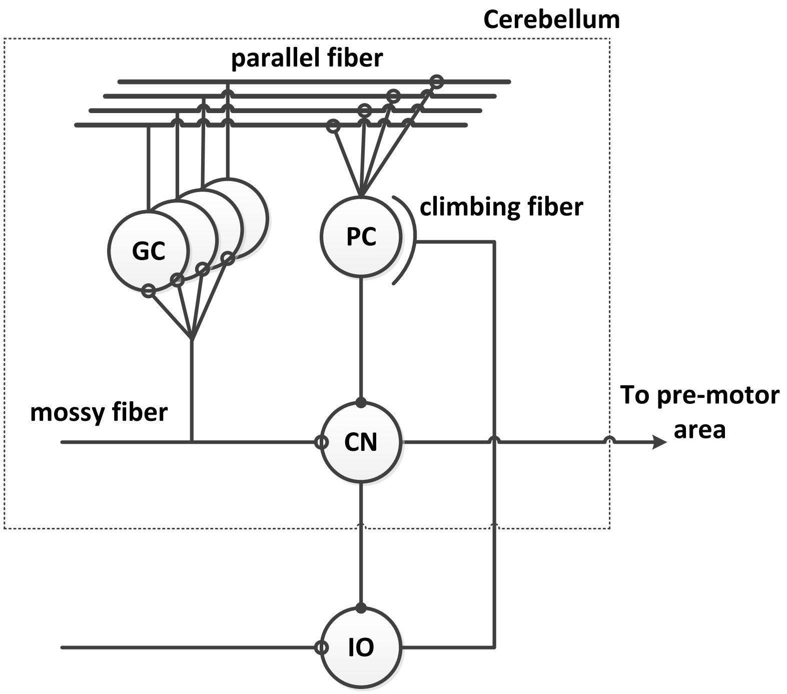


Figure 4.1: Diagram of the cerebellar circuit (GC: Granule Cells; PC: Purkinje Cells; CN: deep Cerebellar Nuclei; IO: Inferior Olive)

The cerebellum [26] has two main input channels: the Mossy Fiber (MF) and the Climbing Fiber (CF) (as shown in Figure 4.1). The MF collects all the contextual information from other regions of the brain. The MF, through the Parallel Fiber (PF), together with CF feed signals into Purkinje Cell (PC). The PC is the only cell which inhibits unwanted activities in the deep cerebellar nucleus neuron so that the appropriate output could be produced. The action of "turning off" the deep cerebellar nucleus is akin to sculpture, where unwanted pieces should be removed to create wanted pattern. That's how the motor learning process happens in order to perform more accurate movements [27]. The Inferior Olive (IO) is the only neural cell which provides input to the CF, making it vital to the cerebellum's function. As the CF is responsible for timing of motor commands and motor learning, IO lesions (disconnection between IO and the cerebellum) lead to motor problems such as nystagmus, ataxia and dystonia.

4.1.2 IO model

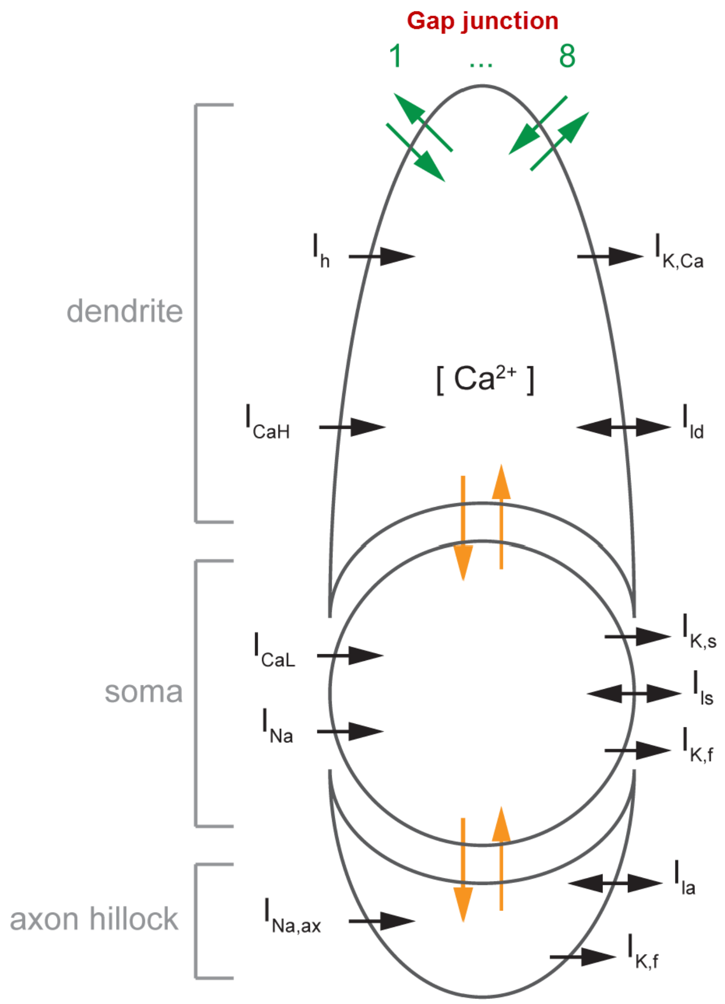


Figure 4.2: Three-compartment dynamics of the IO cell [28]

The model simulated in this thesis is the IO model, which has been developed by Jornt R. de Gruijl from earlier work [28]. In this model, the Hodgkin-Huxley model is applied on three compartments: a dendrite, a soma and an axon hillock. Each compartment is modeled based on three parameters of conductance: leak conductance, sodium conductance and potassium conductance. Each conductance depends on a number of current parameters. The dendrite conduc-

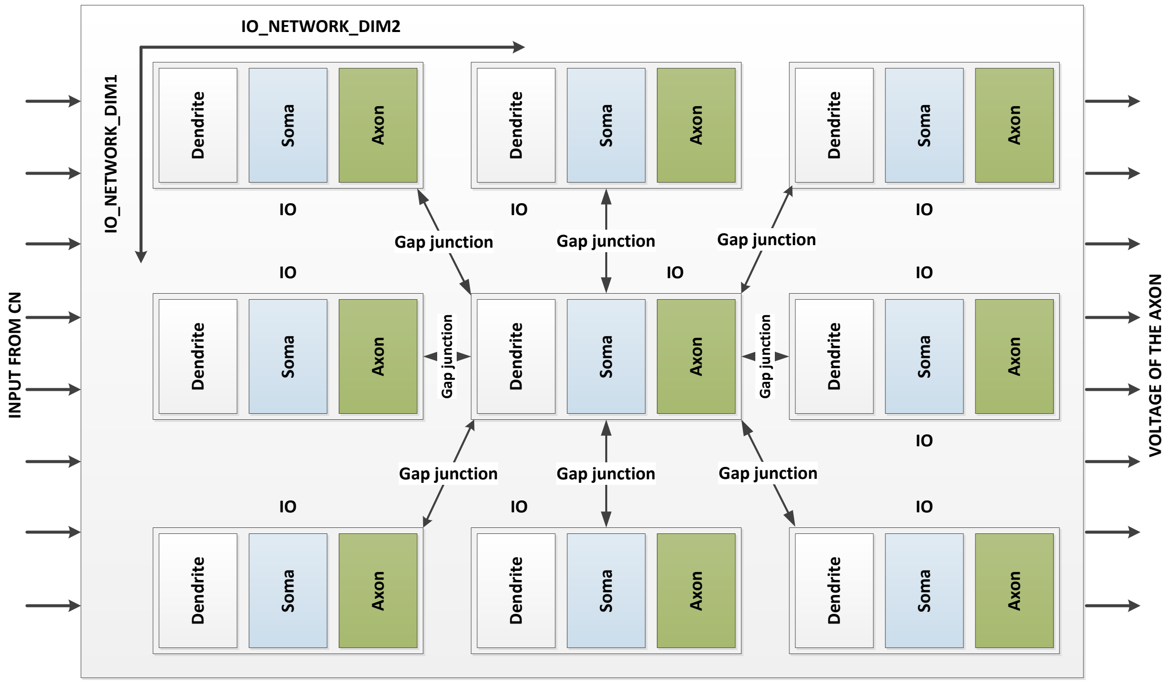


Figure 4.3: The network of IO cell

tance depends on a high-threshold calcium current (I$_{CaH}$), h current (I$_h$), and calcium-dependent potassium current (I$_{K,Ca}$). The soma conductance depends on a low-threshold calcium current (I$_{CaL}$), potassium currents (I$_{K,s}$, I$_{K,f}$) and sodium current (I$_{Na}$). The axon hillock conductance depends on a sodium current (I$_{Na,ax}$) and a potassium current (I$_{K,f}$). Each compartment also include a passive leak current, I$_{ld}$, I$_{ls}$, I$_{la}$, respectively. The above dynamics of the IO cell model are shown in Figure 4.2. Inside an IO model, the three compartments interact with each other.

Multiple IO cells are connected to create an IO network. In order to simplify the calculation on the IO network, the network is considered as a 2D mesh with two dimensions IO_NETWORK_DIM1 and IO_NETWORK_DIM2. The calculation of each IO is involved in computing the three compartments' properties such as currents and voltages. Figure 4.3 is an illustration of the IO network. An external input voltage is fed into the dendrite compartment of every IO, which represents the input from the CN. The interconnection among cells in an IO network is via a so-called the gap junction where each cell is connected to 8 neighbors.

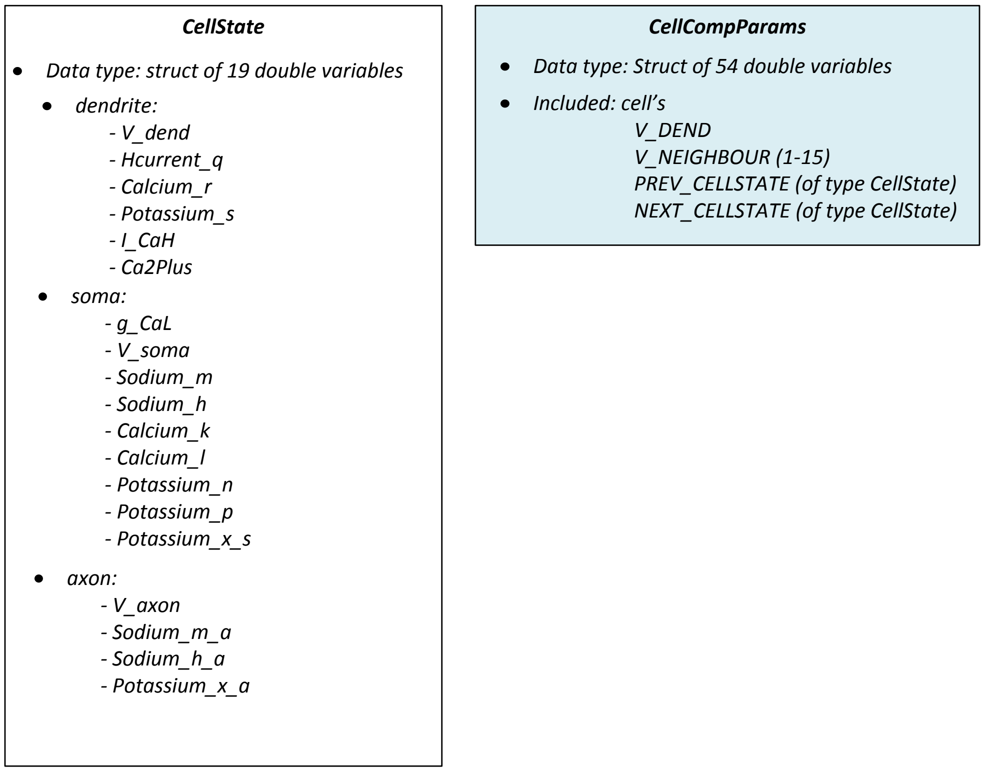
The interconnection among cells is represented by getting all the dendrite's voltages of neighbor cells to compute the dendritic voltage. The equation for the gap junction is discussed in [28]. As the IO network is an 2D cell mesh, cells at the corner have only 3 real neighbors, and other cells at the border have 5 real neighbors.

4.1.3 Model implementation in C programming language

In order to simulate the model on computer, the model is originally implemented in Matlab, then converted into C programming language by Sebastian Isaza.

In the C source code, the data flow of the "main" function is shown in Figure 4.6, which allocates essential memories, initializes the state of the cell network, and executes different parameters calculations.

The program employs "struct" to implement two main data structures in the code, as shown



CellState

- *Data type: struct of 19 double variables*
  - *dendrite:*
    - *V_dend*
    - *Hcurrent_q*
    - *Calcium_r*
    - *Potassium_s*
    - *I_CaH*
    - *Ca2Plus*
  - *soma:*
    - *g_CaL*
    - *V_soma*
    - *Sodium_m*
    - *Sodium_h*
    - *Calcium_k*
    - *Calcium_l*
    - *Potassium_n*
    - *Potassium_p*
    - *Potassium_x_s*
  - *axon:*
    - *V_axon*
    - *Sodium_m_a*
    - *Sodium_h_a*
    - *Potassium_x_a*

CellCompParams

- *Data type: Struct of 54 double variables*
- *Included: cell's*
  V_DEND
  V_NEIGHBOUR (1-15)
  PREV_CELLSTATE (of type CellState)
  NEXT_CELLSTATE (of type CellState)

Figure 4.4: Data structures used in the implementation

in Figure 4.4. The structure called CellState contains 19 variables of type double. These variables are the currents and voltages of three compartments which are explained in [28]. Another structure type is the CellCompParams which contains 54 variables. The CellCompParams type is used for arguments which are passed to functions to compute repeatedly updated values of the three compartments. Since we need to store both the previous and next state of each cell, the amount of data stored for type CellState is double the size of the input size.

In the C implementation, there are two main variables in the form of an array (CellStatePtr and CellCompParamsPtr as shown in Figure 4.5). As the CellStatePtr variable should be large enough to store one previous cell state and one next state, the variable has size of double the IO network size. Each cell needs one variable of type struct CellCompParams to calculate properties of cell at each time step, hence the number of the variables of type CellCompParamsPtr needed is the same as the size of the IO network.

The main function in Figure 4.6 starts with assigning predefined values to all cell states, which are the array variable CellStatePtr. As explained earlier, the external current is fed into the dendrite compartment at the beginning of every time step. Those values are sampling values of a spike which is equal to 6 in a specific range and 0 at the rest. There are 120000 values of external input which is corresponding to 120000 time steps. Therefore, the external input is defined as an array of 120000 elements. Therefore, the time step index is needed to locate the correct value of the external input. After initializing all cell's states and feeding inputs, every IO is computed one by one and gets updated new states after each time-step.

As explained above, each cell is connected to eight neighbors. In this implementation, the

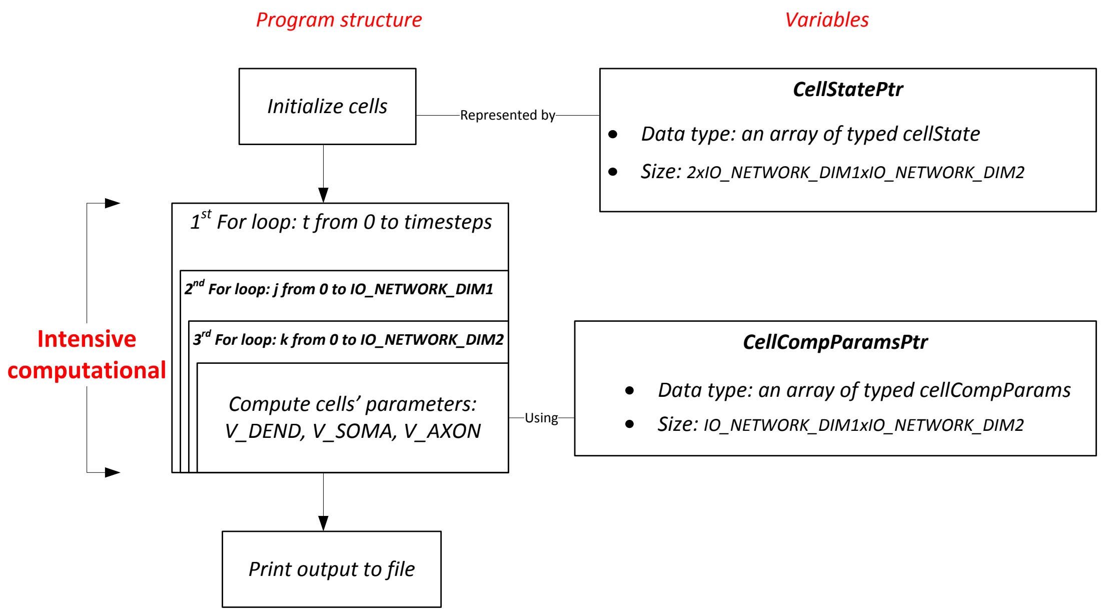

Program structure
Variables
Initialize cells
Represented by
CellStatePtr
Data type: an array of typed cellState
Size: 2xIO_NETWORK_DIM1xIO_NETWORK_DIM2
1st For loop: t from 0 to timesteps
2nd For loop: j from 0 to IO_NETWORK_DIM1
3rd For loop: k from 0 to IO_NETWORK_DIM2
Compute cells' parameters: V_DEND, V_SOMA, V_AXON
Using
CellCompParamsPtr
Data type: an array of typed cellCompParams
Size: IO_NETWORK_DIM1xIO_NETWORK_DIM2
Intensive computational
Print output to file

Figure 4.5: The C implementation of the IO model

connection is represented by getting dendrite's voltage of neighbors to store in a specific array. However, to ease the task of filling those values in an array, the number of neighbors should be all equal to each other. Therefore, each cell is considered as always having eight neighbors. Missing neighbors of cells at the corner and border is filled with the values of the cell itself.

The C implementation in Figure 4.5 includes one loop of 120000 time steps and two loops of visiting two dimensions of the IO network (IO_NETWORK_DIM1 and IO_NETWORK_DIM2). The compute intensive part of the program is located at these three loops of simulation. This information is necessary to parallelize the program later.

The function ComputeOneCell (as shown in Figure 4.7) computes all parameters of dendrite, soma and axon. This function calls three sub-functions to do arithmetic calculation using equations in [28]. Three functions operate on the previous states of IO and the external input, hence they can be executed in parallel. Details on these three functions, which are not related to parallelizing the implementation, are not discussed thoroughly in this thesis.

4.2 CUDA implementation

As the compute intensive part is located at the three loops where the calculation and update of the three compartments are carried out, the CUDA implementation (as shown in Figure 4.8) focuses on resolving this critical part. Firstly, the two inner loops are mapped onto a 2-dimensional grid of CUDA threads. With this setting, every CUDA thread corresponds to an IO cell. Each thread computes parameters of the dendrite, soma and axon for every time steps. In other words, the kernel is actually the function ComputeOneCell. Each time step is one iteration of the outermost loop. As the three compartments are highly associated with each other, the output of one compartment's function is the input for other compartment's functions for the next iteration, hence the three compartments can be computed in parallel.

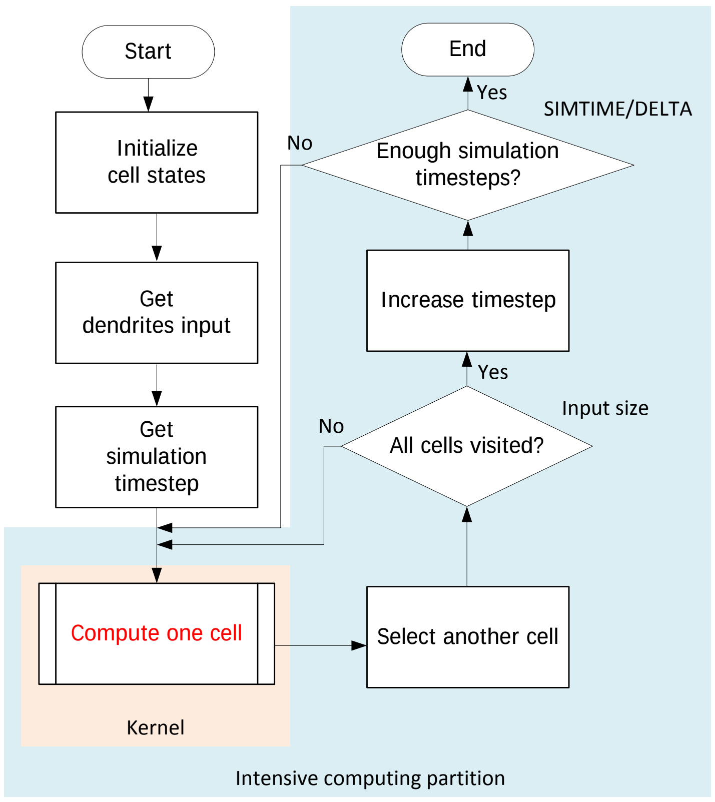


Figure 4.6: Data flow of the "main" function of the C code of the model

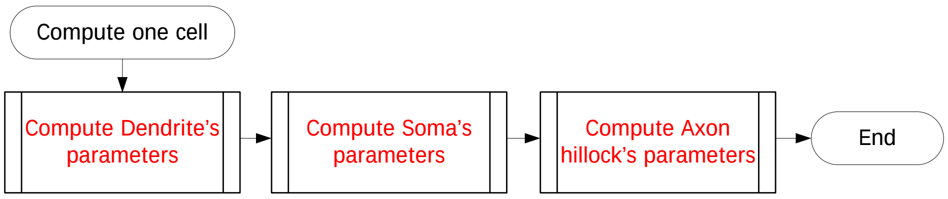

Compute one cell
Compute Dendrite's parameters
Compute Soma's parameters
Compute Axon hillock's parameters
End

Figure 4.7: Data flow of the subprogram to compute single cell's parameters

Needed code synchronization

In each iteration, the new cell state is computed based on the external input and the previous cell state, hence the results of every iteration must be preserved for the next calculation step. In order to ensure the synchronization requirement of every time step, the outermost loop, which controls the time step index, is still carried out on the host side. The synchronization in this case is explicit because it is forced to happen by initializing new kernels at the beginning of every iteration. However, the data on the GPU is preserved for all the iterations and avoid unnecessary host-device communication.

Another synchronization is required at the updating stage of the dendrite voltage. As different

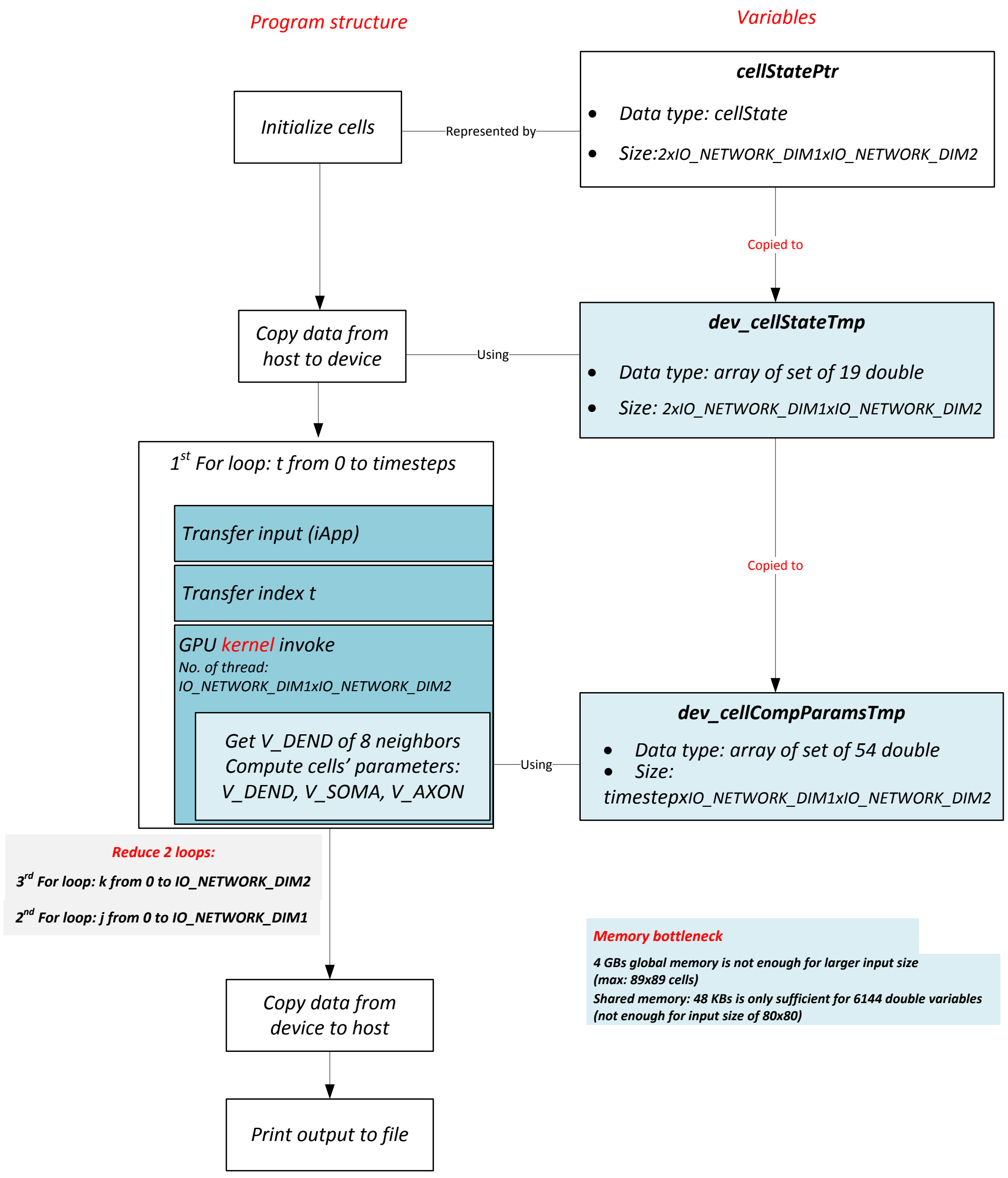

Program structure
Variables
Initialize cells
Represented by
cellStatePtr
Data type: cellState
Size:2xIO_NETWORK_DIM1xIO_NETWORK_DIM2
Copied to
Copy data from host to device
Using
dev_cellStateTmp
Data type: array of set of 19 double
Size: 2xIO_NETWORK_DIM1xIO_NETWORK_DIM2
1st For loop: t from 0 to timesteps
Transfer input (iApp)
Transfer index t
Copied to
GPU kernel invoke
No. of thread: IO_NETWORK_DIM1xIO_NETWORK_DIM2
Get V_DEND of 8 neighbors
Compute cells' parameters: V_DEND, V_SOMA, V_AXON
Using
dev_cellCompParamsTmp
Data type: array of set of 54 double
Size: timestepxIO_NETWORK_DIM1xIO_NETWORK_DIM2
Reduce 2 loops:
3rd For loop: k from 0 to IO_NETWORK_DIM2
2nd For loop: j from 0 to IO_NETWORK_DIM1
Memory bottleneck
4 GBs global memory is not enough for larger input size (max: 89x89 cells)
Shared memory: 48 KBs is only sufficient for 6144 double variables (not enough for input size of 80x80)
Copy data from device to host
Print output to file

Figure 4.8: Original CUDA implementation

threads perform calculation in parallel, a thread might overwrite its dendrite voltage before another thread gets that value. Hence, the variable dev_cellCompParamsPtr has to be defined separately for every time step and for each different IO. However, this leads to large memory consumption for this variable.

Needed data communication

The initialized data in dev_cellStatePtr, which is a copy of CellStatePtr from the CPU side, is transferred once from memory of the host (CPU) to the device (GPU) before kernel starts. Besides, there are two type of data need transferring at every time step. The first data is the external input of dendrite (iApp) which is one variable of typed double. The second one is the index of time step t which is required to locate which variable to update the newly calculating data. After finishing calculation, the final state of every IO cell is transferred back to the host memory and output to a file for further usage.

Needed CUDA data structures

About data structure, as CUDA does not support the data type "struct", an organization of data based on array and index by constants is used. Instead of using a data struct of 19 variables, an array of many sets of 19 variables is used instead and indexed by predefined constants. An initialized array of IO network is allocated on host side, then copied to a temporary array to be transferred to the device memory. In short, an array cellStatePtr of typed cellState is located with the size of inputs on the host memory. On the device memory, an array dev_cellStateTmp and dev_cellCompParamTmp are used with the size of the input size.

Limitations

This implementation reveals a memory bottleneck of the model as the device's memory is not sufficient for the large input size which might be up to billion neuron cells. Besides, the bigger the amount of memory used, the more consuming the memory access time is. Hence, various optimization should be applied on the implementation to reduce memory usage and access time.

4.3 Optimization

Optimization is implemented in such a way which ensures the robustness of computation, and reduces execution time of each time step. As the number of time steps is fixed, the total execution time of the program depends mostly on the execution time of each time step. In the following, we discuss the optimization steps used to reduce execution time.

Combining data transmission

The dendrite's input voltage for each time step is transferred as the whole array whose size equals to the number of time step. As the whole array is transferred, this host-device communication can be moved outside of the first loop as shown in Figure 4.9, hence reduce vastly the time of transferring, avoid unnecessary initializing transmission every time step and also help increase the global memory coalesce.

Using global variable

The index of time step is replaced by a global variable which is controlled by a specific thread on the device side. This global variable is updated by only one thread at the end of each iteration to make sure that the index is increased by one according to each iteration. The correctness of this index is used to determine the proper access to the external input array.

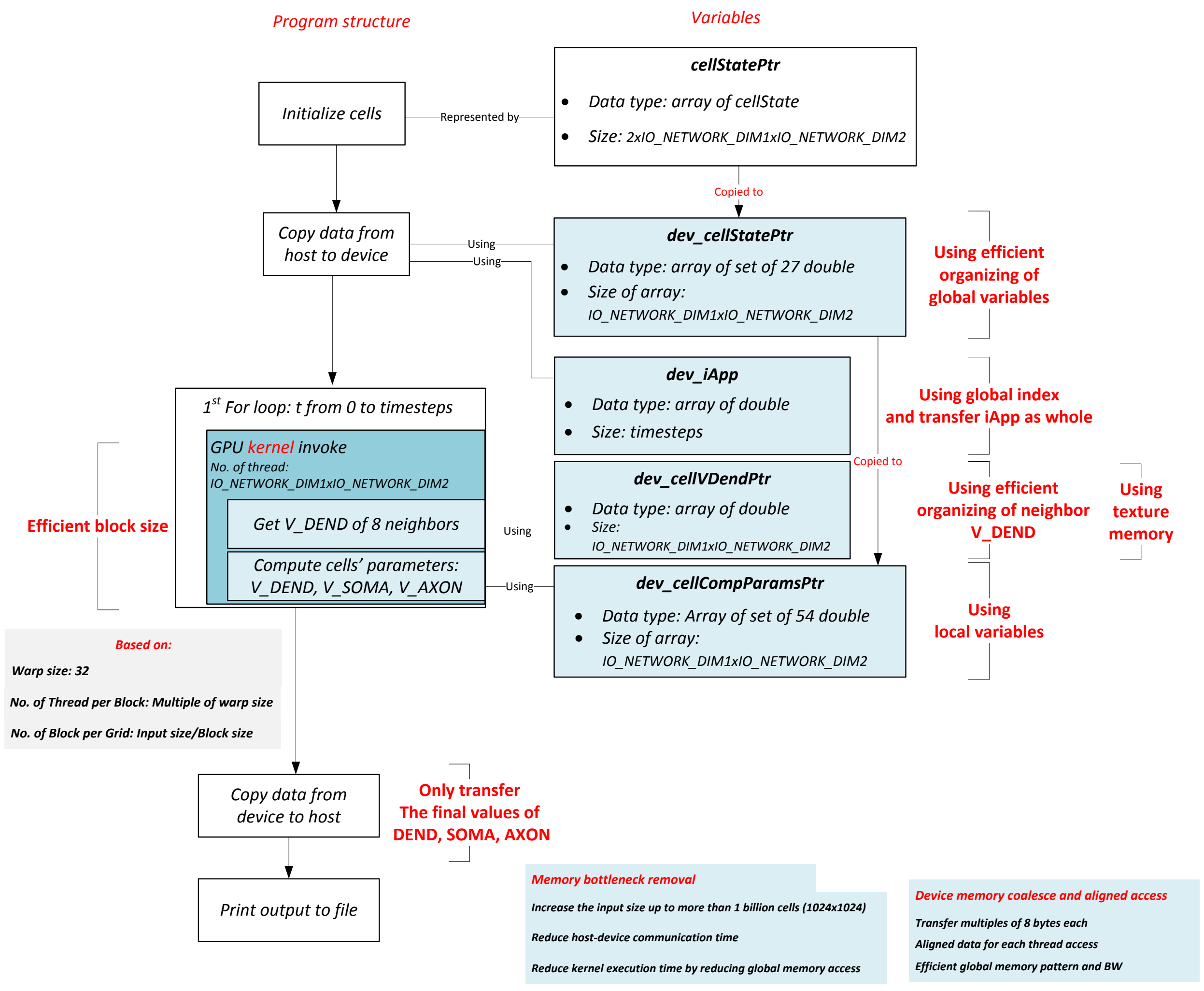

Program structure
Variables
Initialize cells
Represented by
cellStatePtr
Data type: array of cellState
Size: 2xIO_NETWORK_DIM1xIO_NETWORK_DIM2
Copied to
Copy data from host to device
Using
Using
dev_cellStatePtr
Data type: array of set of 27 double
Size of array: IO_NETWORK_DIM1xIO_NETWORK_DIM2
Using efficient organizing of global variables
dev_iApp
Data type: array of double
Size: timesteps
Using global index and transfer iApp as whole
1st For loop: t from 0 to timesteps
GPU kernel invoke
No. of thread: IO_NETWORK_DIM1xIO_NETWORK_DIM2
Get V_DEND of 8 neighbors
Compute cells' parameters: V_DEND, V_SOMA, V_AXON
Efficient block size
Copied to
dev_cellVDendPtr
Data type: array of double
Size: IO_NETWORK_DIM1xIO_NETWORK_DIM2
Using efficient organizing of neighbor V_DEND
Using texture memory
Using
Using
dev_cellCompParamsPtr
Data type: Array of set of 54 double
Size of array: IO_NETWORK_DIM1xIO_NETWORK_DIM2
Using local variables
Based on:
Warp size: 32
No. of Thread per Block: Multiple of warp size
No. of Block per Grid: Input size/Block size
Copy data from device to host
Only transfer The final values of DEND, SOMA, AXON
Print output to file
Memory bottleneck removal
Increase the input size up to more than 1 billion cells (1024x1024)
Reduce host-device communication time
Reduce kernel execution time by reducing global memory access
Device memory coalesce and aligned access
Transfer multiples of 8 bytes each
Aligned data for each thread access
Efficient global memory pattern and BW

Figure 4.9: Optimized CUDA implementation

Reorganizing shared-data set

The value of dendrite's voltage is stored in a separate array which contains only one reserved variable per thread. This optimization reduces the memory accessing time because more useful data is loaded within a cache line. Furthermore, this optimization allows texture memory to be exploited later.

Reorganizing CUDA data structure

The global memory which is used to stored cell's states is reorganized into an array of multiple sets of 27 double variables. A set of 27 double variables includes 19 variables corresponding to the struct cellState, together with 8 variables reserved for 8 values of neighbors. With this organization of data, the kernel can be split into two separate kernels and allows explicit synchronization without affecting the robustness of the program. This optimization also permits the array variable dev_cellCompParamPtr to be defined as local variables with the size of one array element per thread. Furthermore, the local memory is handled by the compiler, hence more systematic optimization is done automatically by the compiler such as coalescing, dynamic allocating, etc. One disadvantage of CUDA memory is that the indexed array is allocated on global memory, therefore, the accessing time does not decrease as wanted. However, this optimization make the memory size bottleneck no longer problematic.

Using texture memory

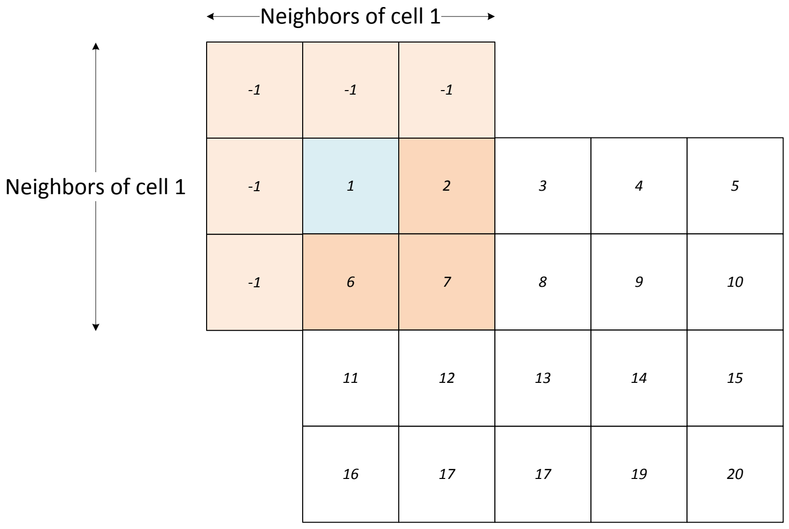


Figure 4.10: Texture memory help eliminate border conditions

Another optimization is to use the texture memory. As the array of neighbor dendrite voltage only changes across time steps. In the other words, the array is considered as read-only in one time step. Besides, the kernel is reloaded every time step, hence the texture memory is also reloaded every kernel. These conditions allow the texture memory to be used to load the neighbor dendrite voltage. The texture memory is useful in this situation because it is cached by the geographic allocation of memory instead of by standard cache line. For the implementation of this IO network, every cell needs its 8 neighbor voltages which fit the caching style of the texture memory. Therefore, the texture memory is effective to reduce the execution time.

Besides, the texture memory help remove the branch divergence. In the original source code, all cells need to check if it is at the border. In that case, some neighbor values might be missing, hence it has to take its voltage value instead of the neighbor voltage values.For example, in Figure 4.10, the cell number 1 has to take 3 neighbors voltage values from cell 2, 5, and 6. As it is at the border, the voltage value of the neighbors marked -1 are invalid. Therefore, the cell uses its own voltage value to fill in those missing values. Using this scheme, all the cells has eight neighbor voltage values and avoid the complexity of dealing with different number of neighbors.

Using efficient block size

The last optimization is an efficient block size. According to the specification of the Tesla platform, the warp size is 32 threads. Hence, the number of threads per block should be a multiple of this warp size to exploit the most the efficient transferring of memory and fine-grain execution of all the threads. Hence the size of the grid is defined based on the input size and the block size.

Implementation variations

The above mentioned optimization is applied on the original GPU implementation. To adapt to different platforms and targets, three variations of the implementation are used. The details of those implementation variations are discussed in Appendix A.

Limitations

The implementation cannot exploit efficiently overlapped of the communication and kernel execution since all the data must be available before the kernel starts. Besides, the shared memory cannot be used as it is not suitable for any type of application data, since we either need to be broadcast the data to all the threads in the grid or since the data is too large to be allocated on the shared memory. Lastly, the application only allows for spacial parallelism but not for temporal parallelism across multiple iterations, which also limits parallelizing the application.

Results and discussion 5

This chapter discusses the achieved results of the implementation in the previous chapter. In order to verify the final results, characteristics of simulation platforms and application implementation, together with the prediction on its performance are included to provide solid base for the simulation result. Finally, the performance and bottlenecks of the application implementation is discussed in detail to evaluate the success of the project.

5.1 Simulation setup

5.1.1 Platforms

The simulation model implementation and the optimization discussed in Chapter 4 are evaluated in various ways. There are three platforms used for this evaluation. Properties of these platforms are discussed below.

Baseline CPU platform

A CPU platform is used as the baseline platform to run the simulation of the sequential implementation. The performance of the implementation is used to compare with that of the parallel implementation on GPU platforms. The baseline platform is an Intel Core i5-2450M (2.5 GHz) with 4 GBs of RAM.

GPU platforms

The parallel implementation is executed on two GPU platforms: Tesla C2075 and GeForce GT640 (referred as Tesla and GeForce, respectively). The characteristics of the two GPU platforms are described in Table 5.1.

5.1.2 Simulation characteristics

Simulation requirements

The implementation is simulated with two versions in single and double precision floating point operations (referred as single and double precision simulation, respectively). The single and double precision are evaluated to identify the impact on performance and accuracy for neuron science applications. Even though most of simulations in the field require double precision accuracy, single precision is useful in determining overall behavior of a neuron model in some cases. Hence, the simulation is benchmarked to determine:

- whether it is more effective to use a particular platform for single or double precision implementation, and
- in which cases, it is necessary to use a particular platform for double precision, and
- what is the cost efficiency of both platforms Tesla C2075 and GeForce GT640 for the implementation.

	Tesla C2075	GeForce GT640
GPU Clock rate	1147 MHz	928 MHz
CUDA Cores	448	384
No. of SM	14	2
No. of cores per SM	32	192
Memory Clock rate	1566 Mhz	891 Mhz
Global memory	4096 MBytes	2047 MBytes
Shared memory per block	49152 bytes	49152 bytes
L2 Cache Size	786432 bytes	262144 bytes
Max Layered Texture Size (dim) x layers	1D=(65536), 2D=(65536,65535), 3D=(2048,2048,2048)	1D=(16384) x 2048, 2D=(16384,16384) x 2048
Texture alignment	512 bytes	512 bytes
Registers available per block	32768	65536
Registers available per thread	63	256
Warp size	32	32
Maximum number of threads per SM	1536	2048
Maximum number of threads per block	1024	1024
Maximum number of blocks per SM	8	16
Maximum sizes of each dimension of a grid	65535 x 65535 x 65535	2147483647 x 65535 x 65535
CUDA Capability	2.0	3.0

Table 5.1: Properties of GPU platforms

Besides, multiple impacts on the performance such as L1 cache usage and GPU thread block size (also referred as block size) are evaluated. The results are required to answer the following questions:

- Does the usage of L1 cache improve the performance of the implementation?
- How does the thread block size affect the performance of the implementation? What is the best block size?
- How is the block size related to the usage of other resources, for example the number of registers?

A square thread block is preferably used with equal x and y block dimension. In the same way, a simulation model with the same x and y dimension input size is used for code simplicity. The simulation is carried out with the block sizes of 16 (4x4), 32 (4x8), 64 (8x8), 256 (16x16), and 1024 (32x32) threads per block. Those numbers are chosen so that the block size is a multiple of 32 as explained in Chapter 3. Block size of 16 is chosen to prove the inefficient block size. Moreover, the block is also limited by architecture, 1024 threads per block for both Tesla C2050 and GeForce GT640.

Theoretical parameters

- Occupancy

Occupancy or multiprocessor occupancy is the ratio of active warps to the maximum number of warps supported on a multiprocessor of the GPU. The theoretical occupancy is needed to pre-evaluate the platform performance before simulation. With the above block size configurations, the GPU implementation characteristics and theoretical occupancy are described in Table 5.2.

The theoretical occupancy shows that the performance of the simulation with a block size 64 is limited by the number of blocks per SM for both platforms (8 blocks and 16 blocks for Tesla C2075 and GeForce GT640, respectively, as shown in Table 5.1). While the performance of the block size 256 is restrained by the number of registers per SM as 51 registers per thread, or 13056 registers per block are required. Hence, for Tesla C2075, two blocks per SM require 26112 registers per SM, while three blocks per SM require 39168 registers per SM (larger than the maximum number of register per SM which is 32768 registers). The same explanation is applied for GeForce GT640, but the maximum number of register per SM of Kepler architecture is 65536. Therefore, GeForce GT640 is still able to increase the block size up to 1024 threads.

The maximum threads per SM of Tesla C2075 is 1536, which means the simulation with the block size of 64 threads needs 1536/64=24 blocks in a SM to reach the maximum occupancy. However, the architectural limit of 8 blocks per SM prevents us from reaching the maximum occupancy. For the block size 256, the simulation needs 1536/256=6 blocks in a SM to reach the maximum occupancy. However, the number of register per block prevents this. In short, for both block sizes, the theoretical occupancy shows that the Tesla C2075 platform is not utilized fully. With the same explanation, the occupancy of GeForce GT640 only reaches 63%, which means only 63% computing resources of the platform is utilized. The maximum thread per SM of GeForce GT640 is 2048 threads. Hence, the block size of 64 or 256 or 1024 threads reaches the maximum occupancy at 2048/64=32 or 2048/256=8 or 2048/1024=2 blocks per SM, respectively. Those number of blocks per SM are also restricted by the number of register, which leads to less efficient occupancy.

The performance reaches saturation point when the occupancy reaches its maximum. The larger the thread block size is, the fewer thread block per SM it needs to fill in.

- Bandwidth

  The theoretical memory bandwidth of a GPU platform is calculated using the hardware specifications with the equation 5.1. This parameter is compared with the effective memory bandwidth to evaluate how much the hardware (memory) is utilized by the application.

$$Bandwidth = (mem\_speed * (bus\_width/8) * 2)/1024^3[GB/s] \tag{5.1}$$

  Hence, the theoretical memory bandwidth of the Tesla C2075 and GeForce GT640 are 140 and 27 GB/s, respectively.

Target metrics

For each simulation, the total execution time and the execution-time-per-time-step are monitored.

The total execution time is the most important metric as it indicates whether the implementation is feasible to serve as a neuron simulator or not. This metric is used to evaluate the speed-up of the implementation over the simulation on CPU platform. The speed-up is achieved by comparing the execution time on GPU platform with the corresponding execution time on CPU platform.

The execution-time-per-time-step is another metric to evaluate the feasibility of the neuron model in real time. As the output from the computation for each time step is required to be

	Tesla C2075		GeForce GT640	
	Single	Double	Single	Double
No. of registers per thread	51	63	48	62
Block size	16			
Active thread blocks per SM	8	8	16	16
Active warps per SM	8	8	16	16
Occupancy	17%	17%	25%	25%
Occupancy limited by	blocks/SM	blocks/SM	blocks/SM	blocks/SM
Block size	32			
Active thread blocks per SM	8	8	16	16
Active warps per SM	8	8	16	16
Occupancy	17%	17%	25%	25%
Occupancy limited by	blocks/SM	blocks/SM	blocks/SM	blocks/SM
Block size	64			
Active thread blocks per SM	8	8	16	16
Active warps per SM	16	16	32	32
Occupancy	33%	33%	50%	50%
Occupancy limited by	blocks/SM	blocks/SM	blocks/SM	blocks/SM
Block size	128			
Active thread blocks per SM	4	4	10	8
Active warps per SM	16	16	40	32
Occupancy	33%	33%	63%	50%
Occupancy limited by	registers/SM	registers/SM	registers/SM	registers/SM
Block size	256			
Active thread blocks per SM	2	2	5	4
Active warps per SM	16	16	40	32
Occupancy	33%	33%	63%	50%
Occupancy limited by	registers/SM	registers/SM	registers/SM	registers/SM
Block size	512			
Active thread blocks per SM	1	1	3	2
Active warps per SM	16	16	32	32
Occupancy	33%	33%	50%	50%
Occupancy limited by	registers/SM	registers/SM	registers/SM	registers/SM
Block size	1024			
Active thread blocks per SM	x	x	1	1
Active warps per SM	x	x	32	32
Occupancy	x	x	50%	50%
Occupancy limited by	registers/SM	registers/SM	registers/SM	registers/SM

Table 5.2: Theoretical characteristics of the GPU implementation based on platform analysis

available after 50us, the metric execution-time-per-time-step is used to evaluate this characteristic of the implementation. The execution time per time step is evaluated with the double precision as real time execution is valid for double precision only.

The communication between host and device is not the critical part in this implementation since the amount and number of time of data transfer are limited. However, this metric is included for completeness in Appendix A.

Measuring methods

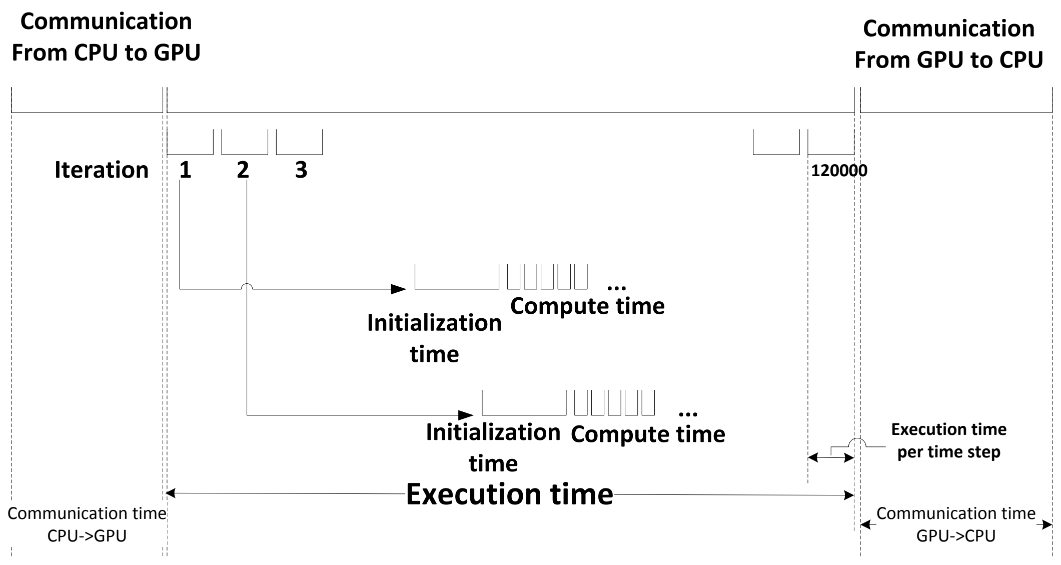


Figure 5.1: Execution flow of the GPU implementation

Most of the measurements in this simulation are time measurements. The function gettimeofday() is used to determine those metrics. Although this function is accurate on the CPU side, it should be used in companion with a synchronization APIs of GPU to ensure the robustness of time measurement.

Another method in GPU time measurement is using the CUDA event record. In this simulation, both methods are used to verify the measurements robustness.

The measurement can only be performed on the CPU side. Hence, the measuring functions are placed before and after the loop of controlling time step to measure the total execution time, inside the loop to measure the execution time per time step and in between the two main communications to measure the communication time (as shown in Figure 5.1).

5.2 Evaluation of platform configuration

In this section, the impact of thread block size and L1 cache on application performance is evaluated by comparing performance of different block size and with/out the usage of L1 cache. The results are recorded from various simulations of selected block size. The results help prove the theoretical anaylis which is made in Section 5.1.2.

5.2.1 Thread block size

Optimizing block size is one of tuning techniques on GPU platform to achieve the higher performance of an application [29]. The choice of thread block size affects significantly application performance. The idea of choosing thread block size is to maximize the SM occupancy when executing the application. The performance of different thread block sizes is measured to figure out the best thread block size which suits our application.

To fulfill that goal, we measure execution time of the application with double precision accuracy and 48 KB L1 cache on the Tesla platform. Each measurement is carried out with different thread block sizes and input sizes. The chosen thread block size is 16 (4x4), 32 (4x8), 64 (8x8), 128 (8x16), and 256 (16x16) for Tesla platform.

Table 5.3 shows the performance of the double precision simulation. The graphical form is shown in Figure 5.2.

Input size (cells)	Block size (threads/block) (s)				
	16	32	64	128	256
64	9	9	9		
256	9	9	9,	9	10
1,024	10	9	10	10	12
4,096	32	26	19	19	21
9,216	64	45	61	52	54
16,384	106	73	91	84	96
36,864	224	158	189	179	203
65,536	393	253	303	287	354
102,400	611	393	463	463	587
262,144	1,558	999	1,184	1,222	1,495
409,600	2,431	1,560	1,904	2,089	2,309
802,816	4,821	3,185	3,897	4,227	4,843
1,048,576	6,195	3,966	4,849	5,483	5,927

Table 5.3: Execution time varies by different thread block sizes (double precision simulation on Tesla C2075)

The result shows that the block size of 32 has the lowest execution time, which implies the highest performance. With the same input size, the execution time increases with the increasing of the block size. Particularly, the block size 16 also has the highest execution time, which is nearly equal to the execution time of the thread block size of 256.

At the small input size, the difference among the results of different thread block size is small. The difference increases with the increasing input size. This happens because the occupancy of the small input sizes is low for all the block sizes since the number of inputs is not large enough to fill in all SMs.

The results of the larger input sizes reflect the theoretical occupancy in Section 5.1.2. All the thread block size performance is bounded by the inefficient block size, which is restricted by the number of registers per SM. However, the bigger block size is, the higher occupancy the application should get. This expectation is not shown in the results, which means that there should be a limitation in the platform resource.

For the double precision, the memory throughput is very low at about 1% of the available bandwidth of the Tesla platform.

To evaluate the application performance with single precision on the Tesla platform, we

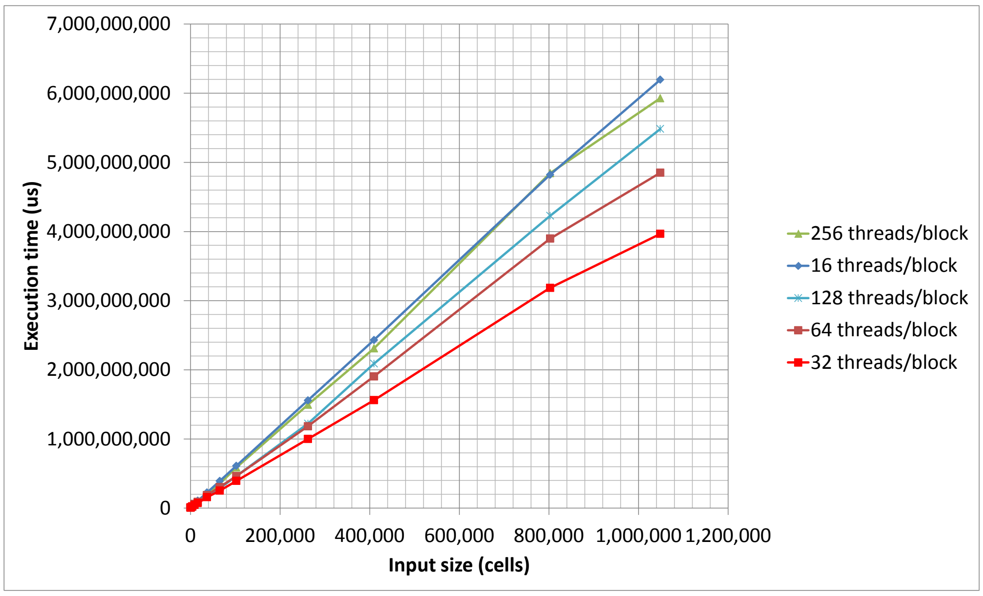


Figure 5.2: Comparison of execution time of different thread block sizes (double precision simulation on Tesla C2075)

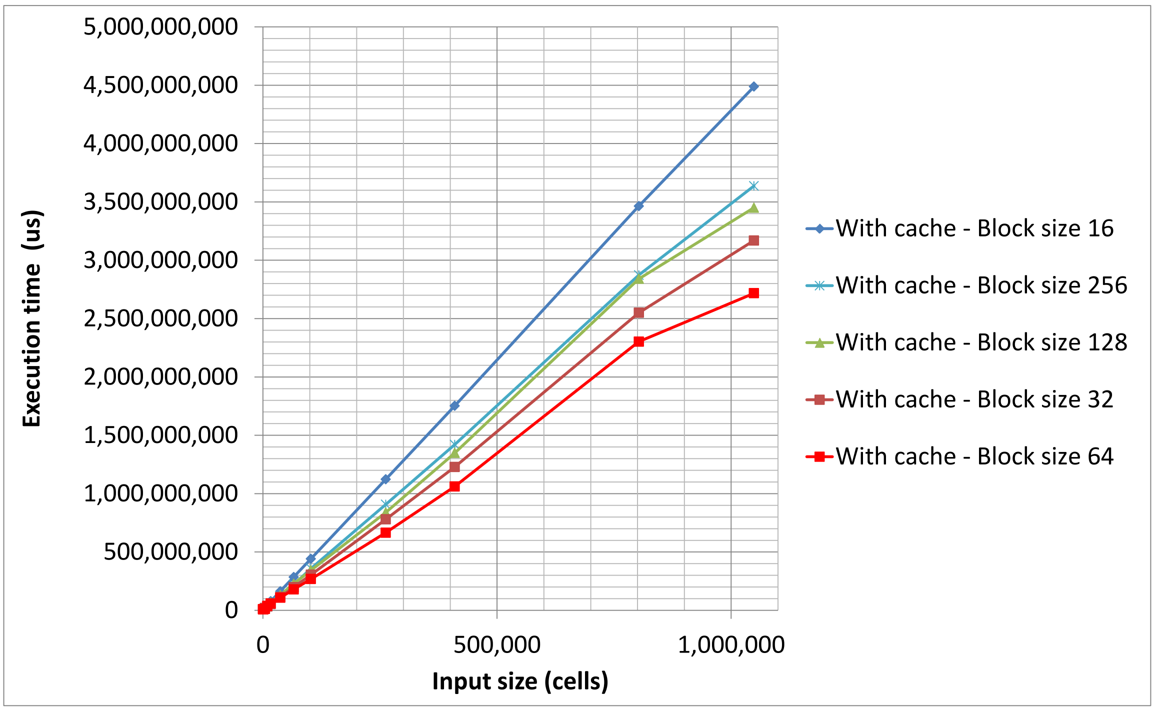


Figure 5.3: Comparison of execution time of different thread block sizes (single precision simulation on Tesla C2075)

Input size (cells)	Block size (threads/block) (s)				
	16	32	64	128	256
64	7	9	9		
256	7	9	9	9	10
1,024	7	9	9	9	10
4,096	22	19	11	13	15
9,216	46	31	37	37	31
16,384	77	52	57	66	57
36,864	162	114	107	130	123
65,536	284	198	179	227	220
102,400	441	305	268	343	358
262,144	1,122	779	664	837	906
409,600	1,752	1,228	1,061	1,348	1,420
802,816	3,463	2,549	2,302	2,840	2,872
1,048,576	4,487	3,168	2,716	3,450	3,636

Table 5.4: Execution time varies by different thread block sizes (single precision simulation on Tesla C2075)

measure the execution time of the application with single precision for various block sizes. The simulation is carried out with the support of L1 cache. Table 5.4 shows the performance of the application with single precision. The graphical representation of this table is shown in Figure 5.3. With the single precision, the same memory throughput can facilitate double amount of data because one single precision variable (of type float) needs only 4 bytes of data instead of 8 bytes of data like one double precision variable (of type double). Besides, the instruction throughput for single precision operations is also increased since it requires only half of computing units of the double precision. Hence, the block size can increase up to 64 to get the best acquired performance of the application with single precision.

In order to evaluate the performance variation on GeForce platform, the double precision simulation with different block sizes and with L1 cache support is carried out on the platform. The best achieved speed-up on GeForce platform belongs to the block size of 64 as shown in Table 5.5 and Figure 5.4. For single precision simulation, the same block size of 64 has the best performance among the four selected block sizes. The result for single precision simulation on GeForce platform is shown in Table 5.6 and Figure 5.5.

5.2.2 L1 cache usage

The usage of cache is useful for application which has a data organization pattern which is suitable for caching. Hence, some applications which have scatter location of data will not benefit from cache. In that case, the application performance might be improved when the cache is removed. To verify this assumption, we measure the execution time of the application with the double precision and without L1 cache usage to compare with the same simulation with the maximum size of L1 cache (48KB). The Table 5.7 shows the execution time of the application with different block sizes and without L1 cache usage. The graphical form of this Table compared with the corresponding results from Table 5.3 is included in Figure 5.6.

It is observed that all the simulations without L1 cache usage have the execution time much higher than those with L1 cache. Even though, the simulation without L1 cache has performance which increases with the increasing of the thread block size. The distance among those results are

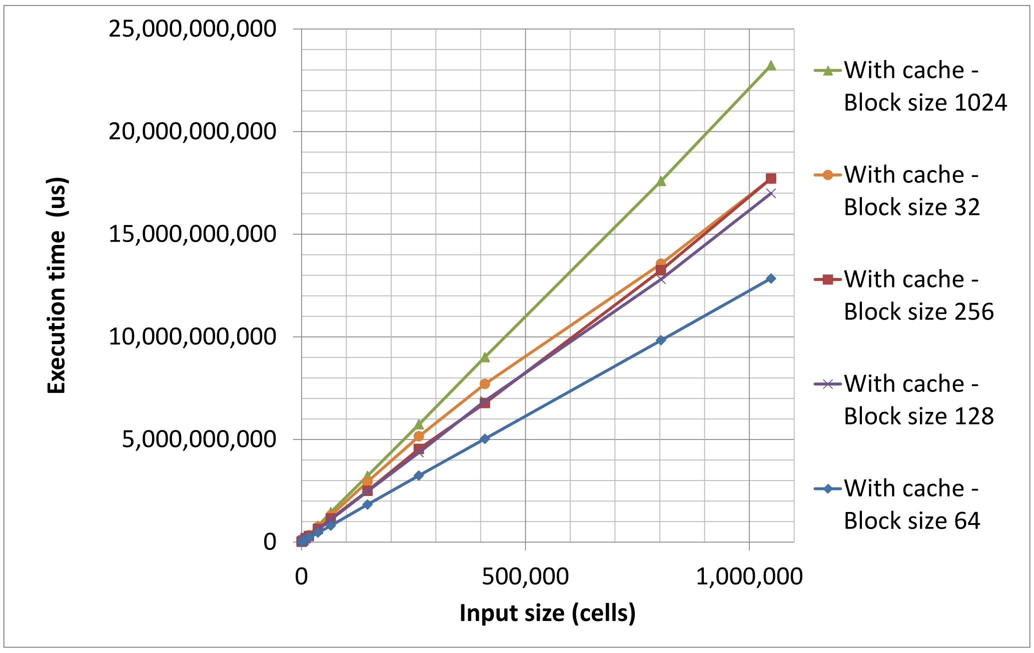


Figure 5.4: Comparison of execution time of different thread block sizes (double precision simulation on Tesla GT640)

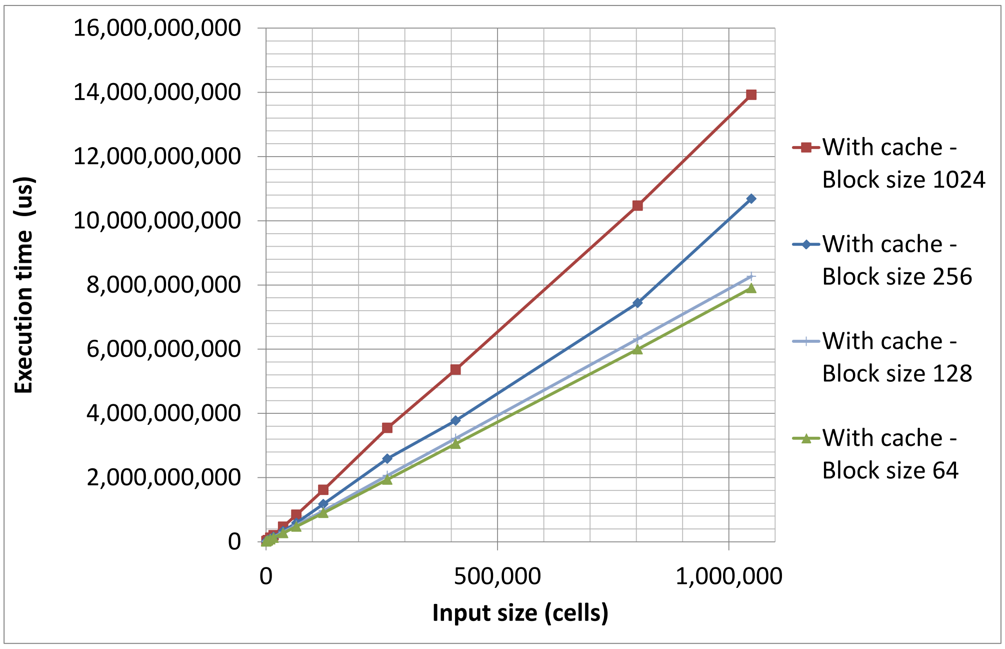


Figure 5.5: Comparison of execution time of different thread block sizes (single precision simulation on Tesla GT640)

Input size (cells)	Block size (threads/block) (s)				
	32	64	128	256	1024
64	19	16			
256	19	16	16	17	
1,024	26	22	22	21	35
4,096	88	66	80	81	82
9,216	191	127	175	185	202
16,384	331	211	286	293	345
36,864	736	458	608	633	787
65,536	1,302	800	1,099	1,143	1,463
147,456	2,935	1,826	2,474	2,490	3,225
262,144	5,149	3,240	4,366	4,536	5,734
409,600	7,703	5,027	6,876	6,771	9,013
802,816	13,559	9,829	13,245	2,840	17,590
1,048,576	17,710	12,842	17,708	3,450	23,224

Table 5.5: Execution time varies by different thread block sizes (double precision simulation on GeForce GT640)

Input size (cells)	Block size (threads/block) (s)			
	64	128	256	1024
64	13			
256	13	13	14	
1,024	16	16	16	24
4,096	42	50	51	53
9,216	78	95	95	127
16,384	128	148	155	205
36,864	272	300	336	470
65,536	476	518	590	842
123,904	900	966	1,176	1,617
262,144	1,939	2,067	2,589	3,564
409,600	3,053	3,221	3,774	5,364
802,816	5,997	6,318	7,437	10,469
1,048,576	7,904	8,228	10,688	13,925

Table 5.6: Execution time varies by different thread block sizes (single precision simulation on GeForce GT640)

less than that among the result of the simulations with L1 cache. Hence, L1 cache has significant impact on the application performance. Moreover, the tuning technique related to the thread block size is also strongly supported by L1 cache.

Besides, the improvement in performance of the application simulation with L1 cache also partly shows that the global memory pattern is suitable for caching. If the data in the global memory is scatter, L1 cache cannot helps increase the application performance.

The performance of the application with single precision without L1 cache has the same

Input size (cells)	Block size (threads/block) (s)				
	16	32	64	128	256
64	9	9	9		
256	9	9	9	10	14
1,024	11	10	10	10	14
4,096	38	30	21	21	24
9,216	76	54	74	65	62
16,384	126	94	124	112	116
36,864	267	202	254	24	252
65,536	469	339	441	432	435
102,400	731	538	684	673	695
262,144	1,875	1,439	1,757	1,741	1,785
409,600	2,919	2,260	2,731	2,774,	2,852
802,816	5,956	4,868	5,527	5,667	6,023
1,048,576	7,525	5,884	7,047	7,103	7,412

Table 5.7: Execution time without L1 cache usage varies by different thread block sizes (double precision simulation on Tesla C2075)

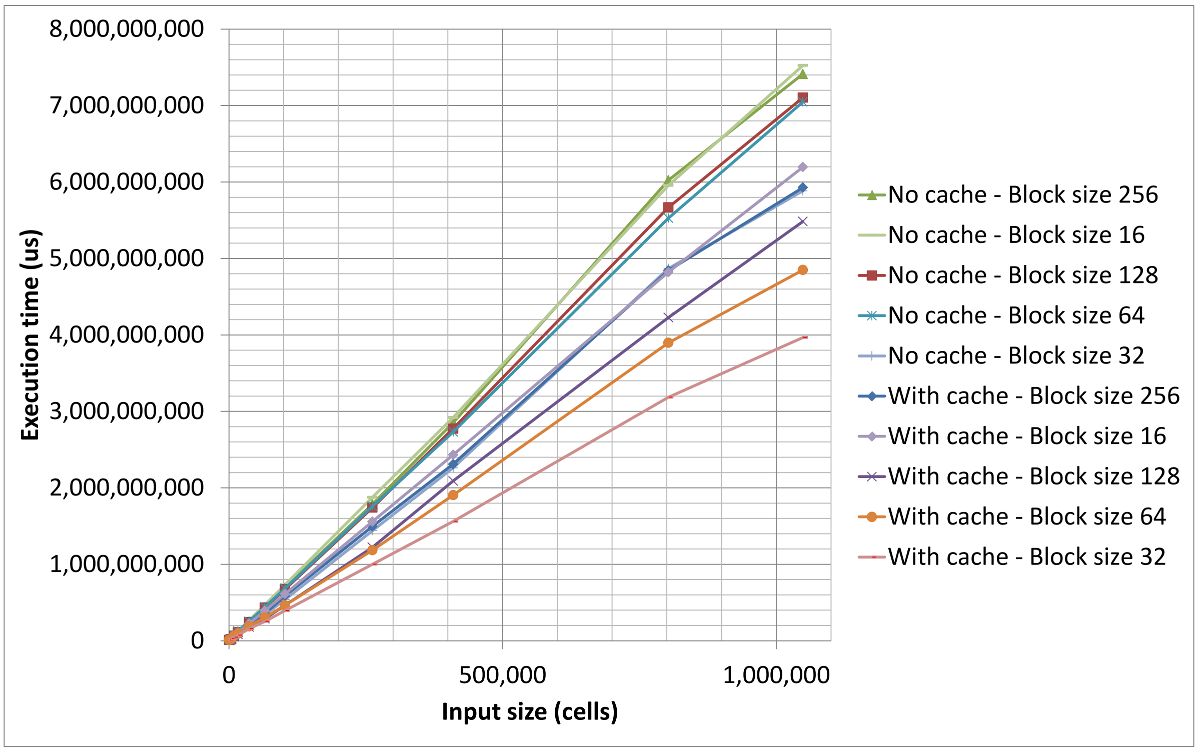


Figure 5.6: Comparison of execution time with/out L1 cache usage (double precision simulation on Tesla C2075)

Input size (cells)	Block size (s)		
	64	256	1024
64	13		
256	13	14	
1,024	16	16	24
4,096	42	51	54
9,216	78	95	128
16,384	128	154	207
36,864	272	329	474
65,536	476	575	851
123,904	900	1,134	1,649
262,144	1,940	2,482	3,637
409,600	3,055	3,663	5,438
802,816	6,000	7,217	10,647
1,048,576	7,909	10,266	14,281

Table 5.8: Execution time without L1 cache usage varies by different thread block sizes (single precision simulation on GeForce GT640)

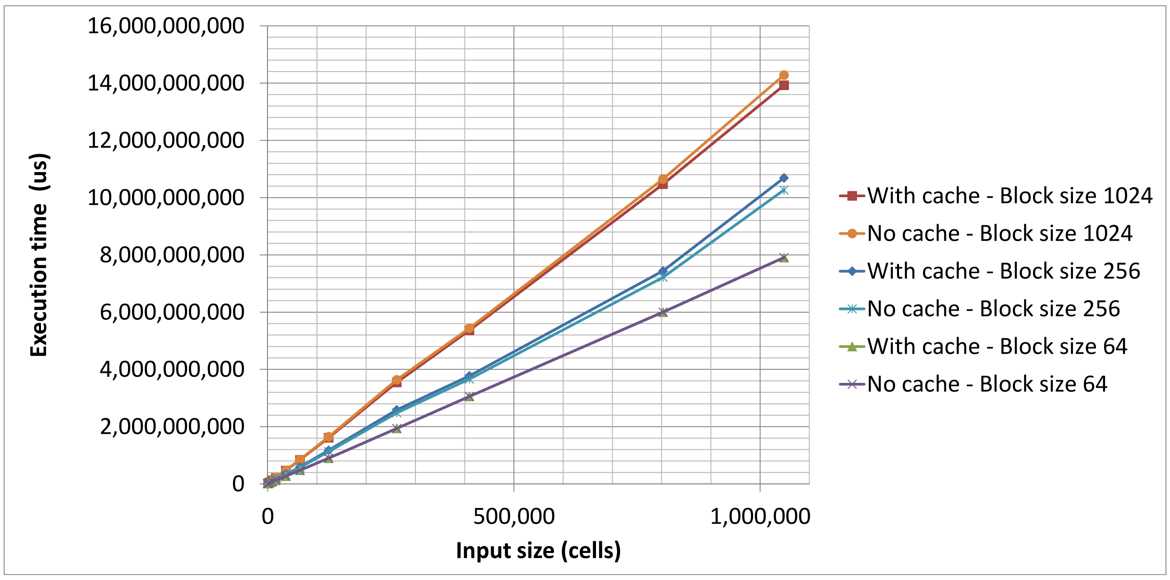


Figure 5.7: Comparison of execution time with/out L1 cache usage (single precision simulation on GeForce GT640)

characteristics as that of the double precision.

For GeForce platform, an interesting phenomenon is observed. The performance of the application with and without L1 cache is not very different from each other. Table 5.8 and Figure 5.7 represent this observation. The behavior of cache is hard to predict. However, the explanation might be that the L1 cache in the latest architecture Kepler is specialized for local memory accesses such as register spills and stack data. The global memory are cached in L2 only. The simulated application depends on global memory considerably, hence L1 cache is only useful, in case it is used for reducing global memory accessing time.

Conclusion

In this section, the two tuning techniques which include the thread block size and cache usage are evaluated thoroughly. Using both techniques, the best achieved performance of double precision simulation on Tesla platform is the execution with the block size of 32 and with L1 cache usage. While GeForce platform and single precision simulation on Tesla platform achieve the best performance with the block size of 64 and with L1 cache usage.

5.3 Performance on Tesla C2075 platform

In this section, the performance of the application on Tesla platform is discussed in the following targets: speed-up and execution time per time step. In each part, the best corresponding performance on Tesla platform is selected to observe its characteristics.

5.3.1 Speed-up

- Single precision

Input size (cells)	CPU (us)	GPU - Block size 64 (us)	Speed-up
64	15	9	1.7
256	61	9	6.6
1,024	242	9	25.6
4,096	973	11	81.9
9,216	2,2415	37	59.1
16,384	3,921	57	67.6
36,864	8,822	107	81.9
65,536	16,064	179	89.6
10,400	26,269	268	98.0
262,144	66,046	664	99.4
409,600	103,321	106	97.4
802,816	202,391	2,302	87.9
1,048,576	268,915	2,716	99.0

Table 5.9: Speed-up of single precision simulation on Tesla C2075

Table 5.9 and Figure 5.8 show the accurate number and graphical representation for the highest achieved performance of the application with single precision. The best achieved speed-up which is obtained with the block size of 64 is 99.4 with the input size of 262,144.

- Double precision

  Table 5.10 and Figure 5.9 show the performance of the application with double precision on Tesla platform. The best achieved speed-up is 68.1 at the input size 1,048,576 cells. This performance is obtained with the block size of 32 as explained in Section 5.2.1.

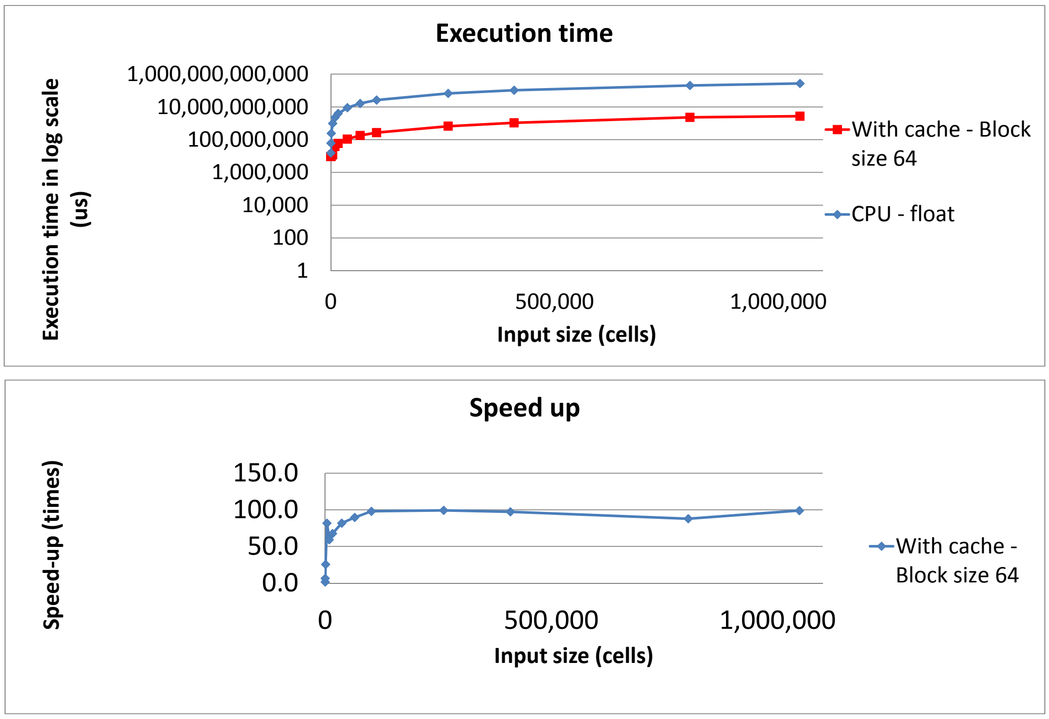


Figure 5.8: Representation of speed-up (single precision simulation on Tesla C2075

Input size (cells)	CPU (us)	GPU - Block size 32 (us)	Speed-up
64	16	9	1.7
256	65	9	6.9
1,024	263	9	27.2
4,096	1,054	26	40.0
9,216	2,372	45	52.5
16,384	4,217	73	57.7
36,864	9,489	158	59.8
65,536	16,870	253	66.9
10,400	26,359	393	67.5
262,144	67,480	999	67.6
409,600	105,438	1,560	67.6
802,816	205,861	3,185	64.6
1,048,576	269,922	3,966	68.1

Table 5.10: Speed-up of double precision simulation on Tesla C2075

It is observed that the speed-up is low at the small input sizes because the computing units are not fully occupied. The result from the profiler also proves this assumption. With the increasing input size, the speed-up is also increased by logarithmic scale. Up to the input size of 65,536 cells, the utilization rate of the application on the platform does not reach 100% (according

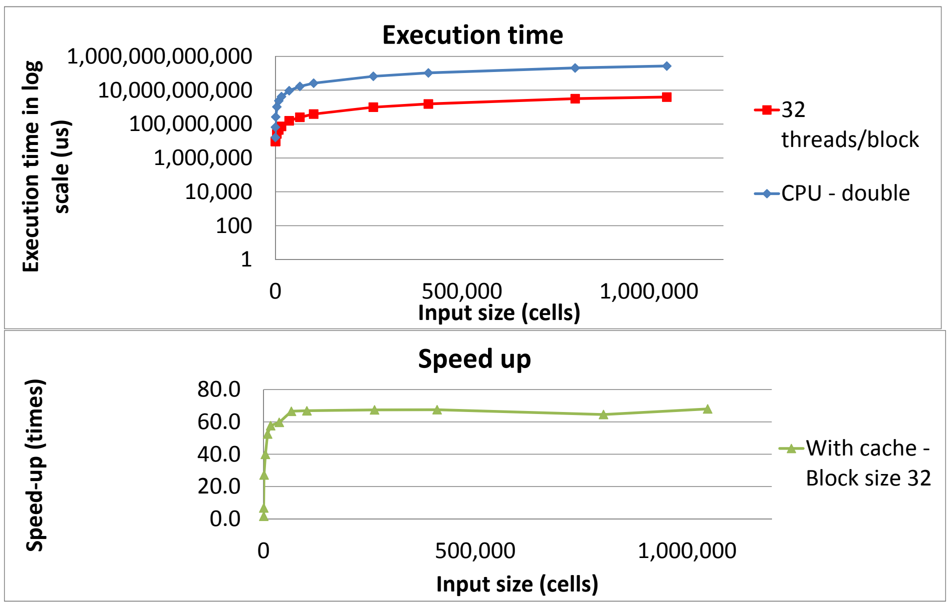


Figure 5.9: Representation of speed-up (double precision simulation on Tesla C2075)

to profiler results). However, from the input size of 262,144 cells, the utilization rate reaches maximum. This means that the computing resources in the GPU platform is fully occupied, or the saturation point is reached. From this saturation point, the execution time increases linearly with increasing input size. Besides, the execution time on CPU also increases linearly. In Figure 5.9 and Figure 5.8, it is observed that the two line in logarithmic scale is parallel to each other from the input size of about 100,000 cells. Therefore, the speed-up stays constantly.

The single precision has nearly two times higher speed-up than the double precision because of several reasons. Firstly, Fermi architecture uses two single precision computing units to do one calculation of a double precision operation. Secondly, the CPU performance on single and double precision is nearly the same. Thirdly, the single precision simulation has higher memory throughput because of a variable involved in the single precision required half of the size of that in the double precision.

5.3.2 Execution time per time step

As mentioned before, the execution time is necessary to evaluate the feasibility of the model simulation in real time. In order to estimate the real time performance, the fastest double precision simulation (with the block size of 32 and cache support) is taken into consideration. With this configuration, the initialization time for kernel launch which is measured by removing the computational part is 14us per time step. Table 5.11 shows the execution time per time step of several input size. The result indicates that the real time simulation can be achieved with the input size of 256 cells because the initialized time can be considered as the necessary setup time of the system and can be deducted from the kernel execution time.

Figure 5.10 shows that even though the real time execution is only achieved with input size, the Tesla platform still performs better than the CPU platform for all input sizes. With the input size smaller than 1000 cells, the execution time per time step on Tesla platform only increase slightly. Moreover, the larger input sizes on Tesla platform also have the execution time per time step which is much lower than that on CPU platform.

Input size (cells)	CPU (us)	GPU - Block size 32 (us)
64	137	57*
256	549	65*
1,024	2,197	80
4,096	8,787	229
9,216	19,770	383
16,384	35,146	601
36,864	79,079	1,263
65,536	140,585	2,078
10,400	219,664	3,320
262,144	562,339	8,347
409,600	878,655	13,009
802,816	1,715,512	26,613
1,048,576	2,249,358	33,146

Table 5.11: Execution time per time step of double precision simulation on Tesla C2075. The (*) is the execution time achieved by another implementation which is only robust for small input sizes (64 and 256 cells).

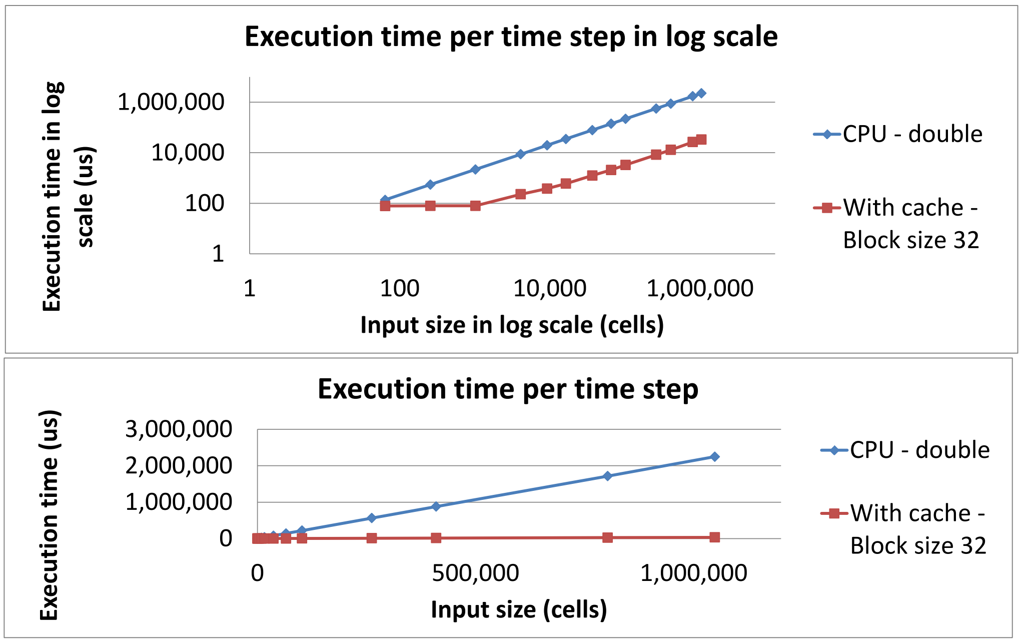


Figure 5.10: Representation of execution time per time step on Tesla C2075

5.4 Performance on GeForce platform

5.4.1 Speed-up

- Single precision

Input size (cells)	CPU (s)	GPU - Block size 64 (s)	Speed-up
64	15	13	1.1
256	61	13	4.5
1,024	242	16	14.4
4,096	973	42	22.8
9,216	2,241	78	28.5
16,384	3,921	128	30.6
36,864	8,822	272	32.4
65,536	16,064	476	33.7
10,400	26,269	900	33.7
262,144	66,046	1,939	34.1
409,600	103,321	3,053	33.8
802,816	202,391	5,997	33.7
1,048,576	268,915	7,904	34.0

Table 5.12: Speed-up of single precision simulation on GeForce GT640

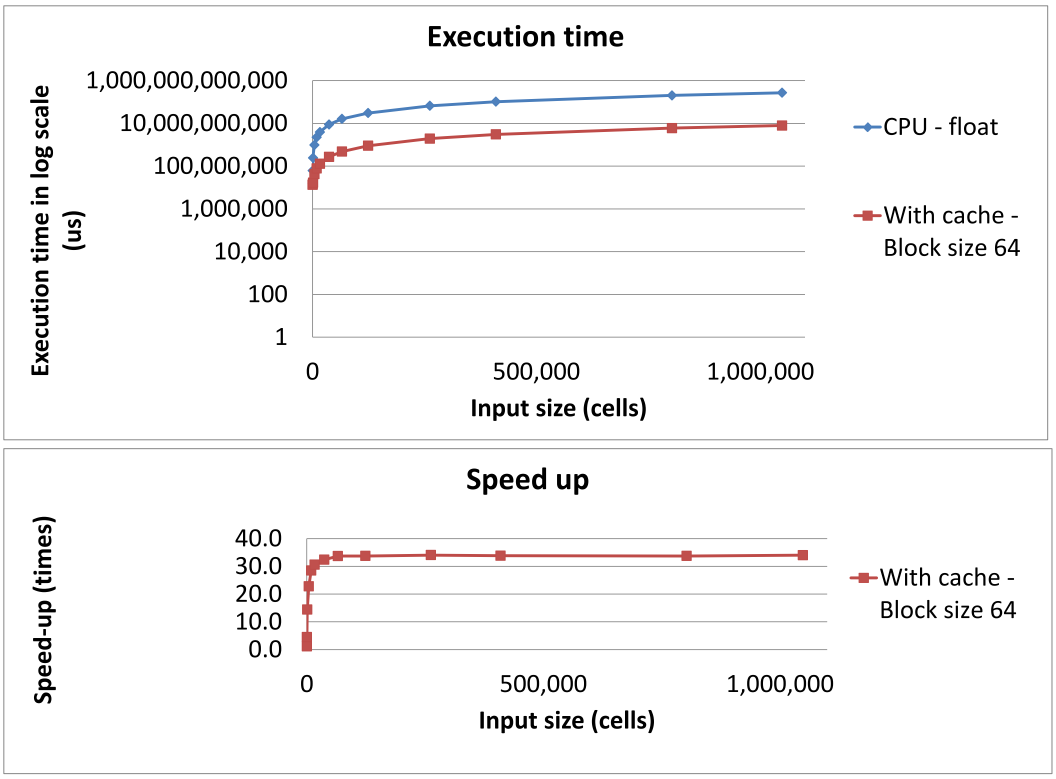


Figure 5.11: Representation of speed-up (single precision simulation on GeForce GT640)

The chosen speed-up for single precision simulation obtained on GeForce platform is shown in Table 5.9 and represents in graphical form in Figure 5.11. The result is achieved with the block size of 64, with L1 cache support. The best speed-up is 34.1, which belongs to the input size 262,144 cells.

- Double precision

Input size (cells)	CPU (us)	GPU - Block size 64 (us)	Speed-up
64	16	16	1.0
256	65	16	4.0
1,024	263	22	12.0
4,096	1,054	66	15.9
9,216	2,372	127	18.6
16,384	4,217	211	20.0
36,864	9,489	458	20.7
65,536	16,870	800	21.1
10,400	26,359	1,826	20.7
262,144	67,480	3,240	20.8
409,600	105,438	5,027	21.0
802,816	205,861	9,829	20.9
1,048,576	269,922	12,842	21.0

Table 5.13: Speed-up of double precision simulation on GeForce GT640

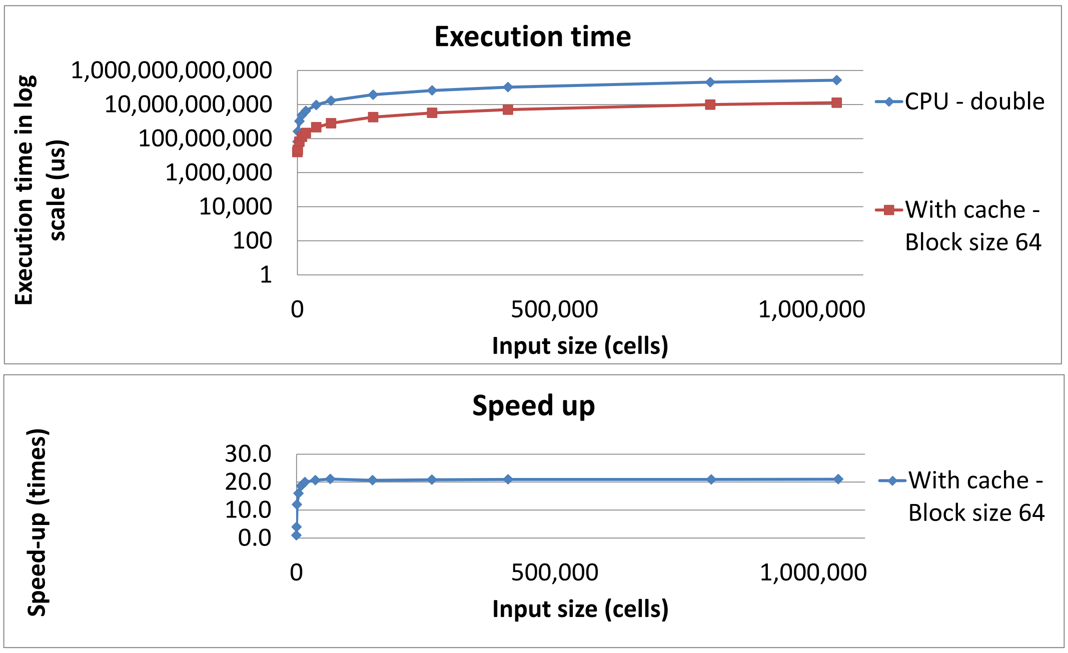


Figure 5.12: Representation of speed-up (double precision simulation on GeForce GT640)

The speed-up of double precision simulation on GeForce platform is shown in Table 5.13. The graphical form of the Table is illustrated in Figure 5.12. The best speed-up for double precision simulation is 21.1, which is 40% less than that for single precision simulation.

5.4.2 Execution time per time step

Input size (cells)	CPU (us)	GPU - Block size 64 (us)
64	128	136
256	511	136
1,024	2,025	181
4,096	8,111	546
9,216	18,675	1,054
16,384	32,677	1,763
36,864	73,524	3,810
65,536	133,871	6,636
10,400	218,912	15,228
262,144	550,384	26,905
409,600	816,017	41,899
802,816	1,686,593	81,895
1,048,576	2,240,962	107,022

Table 5.14: Execution time per time step of double precision simulation on GeForce GT640

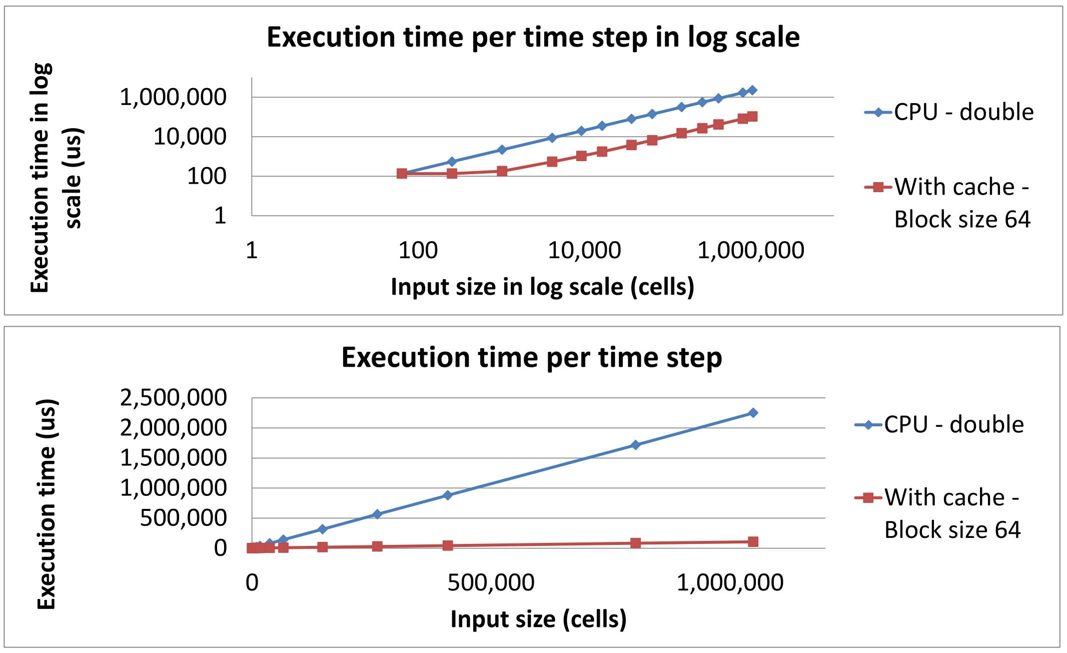


Figure 5.13: Representation of execution time per time step on GeForce GT640

As from the Table 5.14, the best performance of the simulation on GeForce has the smallest execution time per step as 136us. Whereas, the initialization time of the kernel on this platform is 27us per time step. Hence, even with the smallest input size of 64 cells, the real time performance cannot be achieved.

5.5 Discussion of results

5.5.1 Speed-up comparison

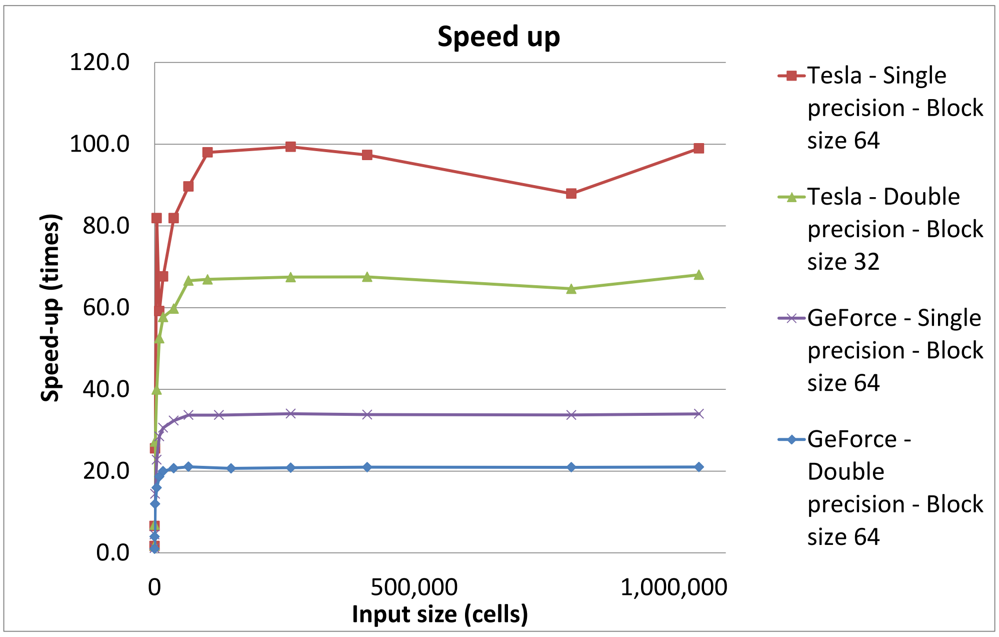


Figure 5.14: Performance comparison between Tesla C2075 and GeForce GT640

From the results of Section 5.3 and 5.4, it can be observed that the speed-up of the simulation on Tesla C2075 platform is approximately three times higher than that on the GeForce GT640. Figure 5.14 shows the comparison between the maximum speed up on the two platforms. This result is achieved by comparing both the best simulation results on the two platforms with the simulation results on the same CPU platform. The figure shows that speed-up is low for small input size and increases with increasing input size until it reaches a saturation point beyond input size of 50,000 cells. The performance of the application reaches a saturation point and stays constant for all the bigger input sizes. The performance stays high with the very large input size, hence is considered suitable to simulate large networks of neurons.

In theory, the performance on a Kepler architecture platform should be higher than that on a Fermi architecture platform if the application has enough parallelism. In this case, the application performance is bounded by one of the resource limits, which leads to the fact that both platforms have not fully utilized their other resources. Even though the application has more parallelism, the performance cannot be improved on either platform. Besides, GeForce has lower processor frequency and lower memory frequency, which also restrict its performance partially.

Real time execution per step

As explained in Section 5.3 and 5.4, the execution time per time step is discussed for some input sizes on both platforms. The performance of GeForce is not considered as a possible real time execution per time step because even with the smallest input size of 64 cells, the simulation on this platform takes much more time than 50us. Hence, the real time execution per step is achieved only on Tesla platform.

5.5.2 Cost efficiency

The purpose of executing simulation on the GeForce is to evaluate the possibility of using a low cost graphics card for a high performance computing application. The Tesla card costs 1789 US dollars, while the GeForce costs only 79 US dollars.

Single precision

Consider that the maximum speed-up on the Tesla platform and GeForce platform is 99 and 34, respectively. The cost-per-unit of the speed-up on the two platforms is 1789/99.4=18 and 79/34.1=2.3 (US dollars/unit), respectively. Hence, the cost efficiency of the Tesla platform is 7.8 times less than the GeForce platform.

Double precision

Consider that the maximum speed-up on the Tesla platform and GeForce platform is 68 and 21 times, respectively. The cost-per-unit of the speed-up on the two platforms is 1789/68.1=26.3 and 79/21.1=3.7 (US dollars/unit), respectively. Hence, the cost efficiency of the Tesla platform is 7.1 times less than the GeForce platform.

5.5.3 Platform comparison

On the one hand, the Tesla platform has the better application performance on both single and double precision simulation. On the other hand, the cost efficiency of the Tesla platform is much smaller than that of the GeForce platform. As both platforms have not been utilized fully, it is hard to conclude which platform is better in absolute value. The real time execution cannot be obtained on GeForce platform. Besides, the application is difficult to be split to map on different platform because of the correlation among the application data at every iteration. Hence, due to the high overhead of splitting the problem, even with multiple separate GeForce platforms, we might not achieve the performance that can be achieved on Tesla platform. However, the GeForce platform might be more suitable for the application in single precision. In case, the real time execution is not required, GeForce platform offers a cheap, compact and flexible host in order to reduce the simulation time in neural science.

5.5.4 Application bottlenecks

A GPU kernel might have the following bottlenecks: memory bandwidth, instruction throughput and latency (the execution time of the slowest thread). With this application, the parallelism is huge as each cell could be considered as an independent computing unit to map onto a thread. Besides, the branching divergence in the application (due to “if“ condition) is nearly zero as a result of using the texture memory. Besides, the memory bandwidth usage is rather low (about 2.4GB/s or 1% of available bandwidth), hence, it cannot be the bottleneck of this application.

Indeed, the NVIDIA visual profiler (nvvp [30]) shows that both platforms have maximum instruction throughput of 0.86 for double precision simulation even with the large input sizes.

For single precision simulation, the maximum instruction throughput is reaching 1.74. For both platforms, the double precision and single precision can execute up to 1.0 and 2.0 instructions per cycle, respectively [31]. Hence, instruction throughput is not the bottleneck of this application.

Moreover, the occupancy is low and limited by the lack of registers per SM, hence, both platforms theoretically cannot reach 100% occupancy. The overlap between memory access and arithmetic operations is low. Hence, we can conclude that the latency is the bottleneck of this application. Theoretically, latency bottleneck might be caused by two main reasons. Firstly, the occupancy is low due to the limitation in the number of register per SM. Hence, there are too little concurrent threads to hide latency. Secondly, the synchronization method is used in every kernel execution, which reduces the overlap between math and memory within the same thread block. For this application, the reason is the latter one since the synchronization is absolutely required to make sure that all the cells update correctly their neighbor values at each time step.

Unfortunately, this bottleneck can only be solved by increasing the occupancy, increasing the parallelism, or removing the synchronization barriers. All of those solutions are mainly related to the application characteristics, which are defined by the neural science problem. Therefore, few optimization methods can help improve the application performance.

Conclusions and recommendations 6

This thesis aimed to implement a neuron model in a network setting on a GPU platform. The GPU implementation was expected to have improved performance over the CPU platform. The selected model is the Inferior Olive model which is originally implemented in Matlab. The implementation in C language is available for investigation. The model is quite complex in comparison with other neuron model implementations which were simulated successfully on GPU platforms before. The complexity stems from the three compartment model in which each compartment uses the Hodgkin Huxley model. Furthermore, the model is considered in a network setting, which means multiple neurons are connected to each other and exchange their information at every simulation iteration.

6.1 Conclusions

The selected model was implemented successfully to simulate on a GPU platform. Our implementation was simulated on two platforms Tesla C2075 and GeForce GT640. On Tesla platform, our double precision and single precision speed-up is 68.1 and 99.4 times faster than that on CPU platform, respectively. The speed-up of double precision simulation and single precision simulation on GeForce platform is 21.1 and 34.1, respectively. On GeForce platform which is 20 times less expensive than the Tesla platform, the performance reduces to 67% of the performance on Tesla platform. Therefore, the GeForce platform has higher cost efficiency than the Tesla platform.

The execution time per time step of the model is fast enough to be considered for real time simulation with the simulation of up to 256 cells. However, without the requirement of real time execution, the simulation on Tesla platform can afford up to 18,939,904 cells with double precision simulation. Single precision simulation doubles this number. The upper bound of the number of cells is defined by the global memory size of the platform. Hence, with a platform which has bigger global memory, the number of simulated cells would increase.

It is observed that the performance of the GPU implementation depends significantly on the thread block size and L1 cache support. The best performance regarding the thread block size is of 32 threads for the Tesla platform and of 64 threads for the GeForce platform. Besides, the performance is higher with the availability of a larger L1 cache.

The occupancy is low mostly because of the limitation on the number of registers per SM. Furthermore, the performance results also show that the implementation has a latency bottleneck due to low occupancy. Unfortunately, the number of registers required for the application calculation cannot be optimized further due to the large number of parameters which represents detailed neural cell's properties.

6.2 Contribution of the results

6.2.1 To neural science

Although a lot of neuron models have been implemented on a GPU platform, the IO model in a network setting is considered as a more complex model with much more details on biophysical properties of neuron cell. The successful implementation of such a complex neuron model would bring us closer to construct the digital brain.

Moreover, the implementation is helpful in decreasing significantly the total time to run simulation in neural science. With the Tesla platform, the double precision simulation can be reduced by nearly 70 times. With the low cost platform such as GeForce platform, the simulation time can also be reduce by 20 times. In comparison with a costly platform such as a supercomputer platform, GPU platform is more portable, affordable and flexible. This inspires and provides the technical support for neural scientists to put more efforts into detailed neuron models reflecting insight activities of the human brain.

6.2.2 To high performance computing

The result shows the variation of the application performance depending on the thread block size and L1 cache of the two platforms. The simulation was executed on the two latest Nvidia GPU architecture Fermi and Kepler, which offer the characteristic comparison between the two platforms. Furthermore, it shows the feasibility and efficiency of implementing an intensive computational application on a GPU platforms.

6.3 Limitations

The implementation is still a fresh start on a very complex neuron model. Two limitations emerges from the implementation are:

- The number of registers: Due to the large amount of properties needed to represent in the IO model, the large number of registers are required to use in the implementation. This requirement exposes the difficulty in efficiently increasing the thread block size as the number of registers per SM is limited (63 for Tesla platform and 255 for GeForce platform).
- Synchronization: The IO model in a network setting requires to transfer updated results from one cell to its neighbor at every iterations. Hence, it is difficult to split the IO network into multiple partitions to be mapped on multiple platforms. Besides, the synchronization causes problems in utilizing resources among multiple threads.

6.4 Recommendation for further research

As the computing resource and memory bandwidth of the two platforms have not been used completely, more optimization to increase occupancy could be applied to enhance the model simulation performance.

- The main problem of limited occupancy is due to the limit number of registers per SM. In detail, the number of variables used as register/local memory could be reallocated on global or shared memory to reduce the number of needed registers [32]. Besides, the model itself could be reworked to limit the number of variables required. Together with the development of GPU technology, a new GPU generation with a large register file also helps increase the application performance. Otherwise, an explicitly compiling option which limits the register number can also be exploited, however, this optimization might lead to other problems such as memory bandwidth bottlenecks.
- As the synchronization among parallel threads is the main bottleneck of the application, reducing the number of registers used and re-organizing the data distribution will not help eliminate this bottleneck. However, increasing the occupancy will also increase the application performance in general. The synchronization bottleneck can be lifted only when

the synchronization among threads is removed. This requires a better parallel algorithm which can suggest a ”smarter” scheme to communicate among cells.

Bibliography

[1] E. Izhikevich, “Which model to use for cortical spiking neurons,” *Neural Networks, IEEE Transactions on*, vol. 15, no. 5, pp. 1063 –1070, sept. 2004.

[2] W. Gerstner and W. M. Kistler, *Spiking Neuron Models: Single Neurons, Populations, Plasticity.* Cambridge University Press.

[3] Khronos, “Opencl-open standard for parallel programming of heterugeneous systems,” December 2012. [Online]. Available: http://www.khronos.org/opencl/

[4] V. Pallipuram, M. Bhuiyan, and M. Smith, “Evaluation of gpu architectures using spiking neural networks,” in *Application Accelerators in High-Performance Computing (SAAHPC), 2011 Symposium on*, july 2011, pp. 93 –102.

[5] A. Fidjeland and M. Shanahan, “Accelerated simulation of spiking neural networks using gpus,” in *Neural Networks (IJCNN), The 2010 International Joint Conference on*, july 2010, pp. 1 –8.

[6] D. Yudanov, M. Shaaban, R. Melton, and L. Reznik, “Gpu-based simulation of spiking neural networks with real-time performance amp; high accuracy,” in *Neural Networks (IJCNN), The 2010 International Joint Conference on*, july 2010, pp. 1 –8.

[7] T. J. Eric Kandel, James Schwartz, *Principles of Neural Science.* McGraw-Hill Companies, Incorporated.

[8] M. A.Arbib, *The Handbook of Brain Theory and Neural Networks.* The MIT Press.

[9] A. L. HODGKIN and A. F. HUXLEY, “A quantitative description of membrane current and its application to conduction and excitation in nerve.” *The Journal of physiology*, vol. 117, no. 4, pp. 500–544, Aug. 1952. [Online]. Available: http://jp.physoc.org/content/117/4/500.abstract

[10] N. Corson and M. Aziz-Alaoui, “Dynamics and complexity of hindmarsh-rose neuronal systems.”

[11] H. Lecar, “Morris-lecar model,” February 2013. [Online]. Available: http://www.scholarpedia.org/article/Morris-Lecar_model

[12] H. R. Wilson, “Simplified dynamics of human and mammalian neocortical neurons,” *J. Theo: Biol*, vol. 200, pp. 375 – 388, 1999.

[13] R. B. Wells, “The wilson model of cortical neurons.”

[14] E. Izhikevich, “Simple model of spiking neurons,” *Neural Networks, IEEE Transactions on*, vol. 14, no. 6, pp. 1569 – 1572, nov. 2003.

[15] G. F. Lyle N.Long, “A review of biologically plausible neuron models for spiking neural networks,” *Aerospace Conference*, Apr. 2010.

[16] Nvidia, *OpenCL Programming Guide for CUDA Architecture.* Nvidia.

[17] *NVIDIA Fermi Compute Architecture Whitepaper.*

[18] NVIDIA, “Kepler white paper,” 2012. [Online]. Available: http://www.nvidia.com/object/nvidia-kepler.html

[19] ——, “Dynamic parallelism white paper,” 2012. [Online]. Available: http://developer.download.nvidia.com/assets/cuda/docs/TechBrief_Dynamic_Parallelism_in_CUDA_v2.pdf

[20] D. B. Kirk and W. mei W. Hwu, *Programming Massively Parallel Processors: A Hands on Approach.* Morgan Kaufmann.

[21] *NVIDIA CUDA Compiler Driver NVCC.*

[22] W. chun Feng and S. Xiao, "To gpu synchronize or not gpu synchronize?" in *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, 2010, pp. 3801–3804.

[23] S. Xiao and W. chun Feng, "Inter-block gpu communication via fast barrier synchronization," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, 2010, pp. 1–12.

[24] D.-K. Lee and S.-J. Oh, "Variable block size motion estimation implementation on compute unified device architecture (cuda)," in *Consumer Electronics (ICCE), 2013 IEEE International Conference on*, 2013, pp. 633–634.

[25] Y. Torres, A. Gonzalez-Escribano, and D. Llanos, "Understanding the impact of cuda tuning techniques for fermi," in *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, 2011, pp. 631–639.

[26] Wikipedia, "Cerebellum," November 2011. [Online]. Available: http://en.wikipedia.org/wiki/Cerebellum

[27] R. Swenson, "Cerebellar systems," 2006. [Online]. Available: http://www.dartmouth.edu/~rswenson/NeuroSci/chapter_8B.html

[28] J. R. De Gruijl, P. Bazzigaluppi, M. T. G. de Jeu, and C. I. De Zeeuw, "Climbing fiber burst size and olivary sub-threshold oscillations in a network setting," *PLoS Comput Biol*, vol. 8, no. 12, p. e1002814, 12 2012. [Online]. Available: http://dx.doi.org/10.1371%2Fjournal.pcbi.1002814

[29] NVIDIA, "Tuning cuda applications for kepler," July 2013. [Online]. Available: http://docs.nvidia.com/cuda/kepler-tuning-guide/index.html

[30] ——, "Profiler user's guide," July 2013. [Online]. Available: http://docs.nvidia.com/cuda/profiler-users-guide/index.html

[31] *Instruction Limited Kernels - CUDA Optimization Webinar.*

[32] *Local Memory and Register Spilling.*

A Implementation variations

In this appendix, three GPU implementations are discussed. The reason for having multiple GPU implementations is that the performance of an application on a GPU platform is varied by configurations.

A.1 GPU implementation for small thread block sizes

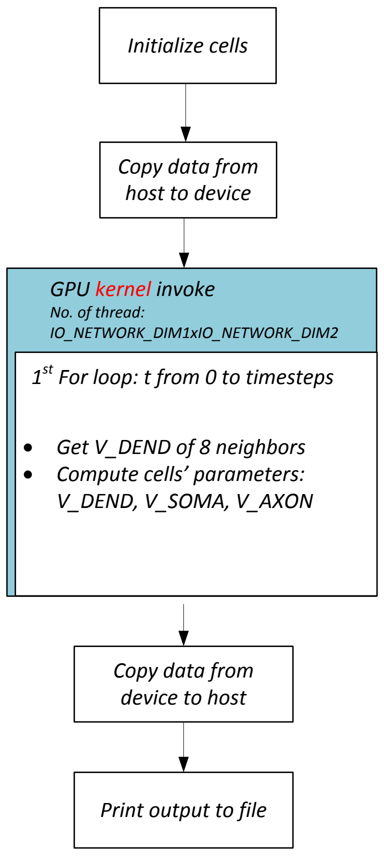

Initialize cells
Copy data from host to device
GPU kernel invoke
No. of thread: IO_NETWORK_DIM1xIO_NETWORK_DIM2
1st For loop: t from 0 to timesteps
Get V_DEND of 8 neighbors
Compute cells' parameters: V_DEND, V_SOMA, V_AXON
Copy data from device to host
Print output to file

Figure A.1: GPU implementation for small thread block sizes

The first implementation is shown in Figure A.1. This implementation works correctly for small input sizes which are equal to the thread block size.

A.2 GPU implementation on Tesla C2075 platform

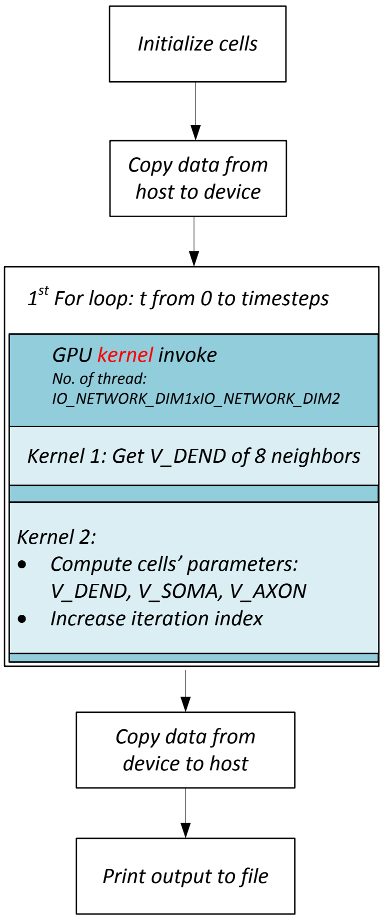

Initialize cells
Copy data from host to device
1st For loop: t from 0 to timesteps
GPU kernel invoke
No. of thread: IO_NETWORK_DIM1xIO_NETWORK_DIM2
Kernel 1: Get V_DEND of 8 neighbors
Kernel 2:
Compute cells' parameters: V_DEND, V_SOMA, V_AXON
Increase iteration index
Copy data from device to host
Print output to file

Figure A.2: GPU implementation on Tesla C2075 platform

The second implementation (as shown in Figure A.2) is an improved version of the first implementation for all input sizes. As the IO model needs synchronization among neural cells during execution, the input sizes which are equal to the thread block size use the API __syncthread() to do synchronization. Whereas, the input sizes larger than the thread block sizes need explicit synchronization because the API __syncthread() is not applied on different thread blocks. This implementation is used for simulation on Tesla C2075 platform.

A.3 GPU implementation on GeForce GT640 platform

The third implementation is used for simulation on GeForce GT640 platform (as shown in Figure A.3. GeForce GT640 platform belongs to the Kepler architecture which has a better scheduler. The scheduling scheme on Kepler architecture allows inter-warp scheduling, where the compiler can determine when instruction will be ready to issue. Hence the increasing of the iteration index might occur Read-After-Write hazards because one thread is used to update a variable in the

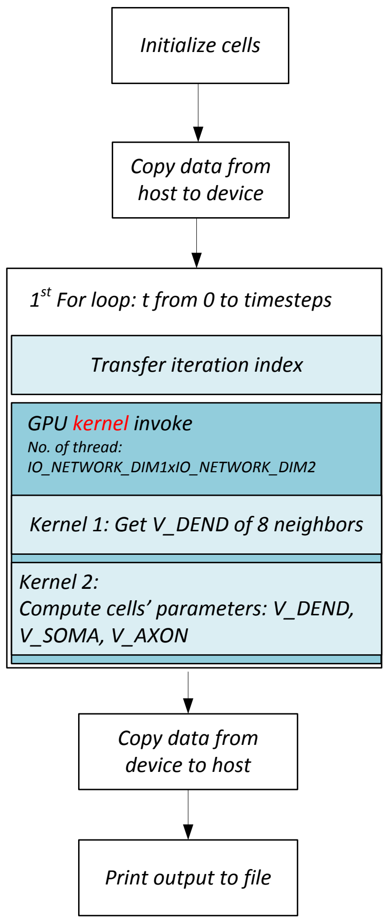

Initialize cells
Copy data from host to device
1st For loop: t from 0 to timesteps
Transfer iteration index
GPU kernel invoke
No. of thread: IO_NETWORK_DIM1xIO_NETWORK_DIM2
Kernel 1: Get V_DEND of 8 neighbors
Kernel 2: Compute cells' parameters: V_DEND, V_SOMA, V_AXON
Copy data from device to host
Print output to file

Figure A.3: GPU implementation on GeForce GT640 platform

global memory. Hence, this update is replaced by a transfer of one index variable from host to device at every iterations.