5-2010

ACCELERATION OF SPIKING NEURAL NETWORKS ON SINGLE-GPU AND MULTI-GPU SYSTEMS

Venkittaraman vivek Pallipuram krishnamani
Clemson University, nitt.vivek@gmail.com

ACCELERATION OF SPIKING NEURAL NETWORKS ON SINGLE-GPU AND MULTI-GPU SYSTEMS

A Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
Computer Engineering

by
Venkittaraman Vivek, Pallipuram Krishnamani
May 2010

Accepted by:
Dr. Melissa C. Smith, Committee Chair
Dr. Stanley Birchfield
Dr. John Gowdy

ABSTRACT

There has been a strong interest in modeling a mammalian brain in order to study the architectural and functional principles of the brain and offer tools to neuroscientists and medical researchers for related studies. Artificial Neural Networks (ANNs) are compute models that try to simulate the structure and/or the functional behavior of neurons and process information using the connectionist approach to computation. Hence, the ANNs are the viable options for such studies.

Of many classes of ANNs, Spiking Neuron Network models (SNNs) have been employed to simulate mammalian brain, capturing its functionality and inference capabilities. In this class of neuron models, some of the biologically accurate models are the Hodgkin Huxley (HH) model, Morris Lecar (ML) model, Wilson model, and the Izhikevich model. The HH model is the oldest, most biologically accurate and the most compute intensive of the listed models. The Izhikevich model, a more recent development, is sufficiently accurate and involves the least computations. Accurate modeling of the neurons calls for compute intensive models and hence single core processors are not suitable for large scale SNN simulations due to their serial computation and low memory bandwidth. Graphical Processing Units have been used for general purpose computing as they offer raw computing power, with a majority of logic solely dedicated for computing purpose.

The work presented in this thesis implements two-level character recognition networks using the four previously mentioned SNN models in Nvidia’s Tesla C870 card and investigates performance improvements over the equivalent software implementation on a 2.66 GHz Intel Core 2 Quad. The work probes some of the important parameters such as the kernel time, memory transfer time and flops offered by the GPU device for the implementations. In this work, we report speed-ups as high as 576x on a single GPU device for the most compute-intensive, highly biologically realistic Hodgkin Huxley model. These results demonstrate the potential of GPUs for large-scale, accurate modeling of the mammalian brain. The research in this thesis also presents several optimization techniques and strategies, and discusses the major bottlenecks that must be avoided in order to achieve maximum performance benefits for applications involving complex computations. The research also investigates an initial multi-GPU implementation to study the problem partitioning for simulating biological-scale neuron networks on a cluster of GPU devices.

DEDICATION

I dedicate this thesis to my parents, my academic advisor and all the elders; who have been guiding me throughout; be it in my work or in walks of life. I also dedicate this thesis to my younger brother, who has always stood by my side.

This thesis is also dedicated to Kameswarie, whose instant messages would make my GPUs work.

ACKNOWLEDGMENTS

This thesis was made possible due to the help and support of the following faculty members, friends and colleagues.

I must begin by thanking my advisor Dr. Melissa C. Smith, who graciously took me as her pupil. Dr. Smith's encouragement, words of wisdom, and insightful suggestions helped me in carrying research and writing this thesis.

I would also like to thank Dr. Stanley Birchfield and Dr. John Gowdy for being in my committee and teaching me various aspects of Signal Processing.

I am indebted to my fellow colleagues from FCTL: Ashraf, Sudha, Randy, Ananth and many others whom I fail to mention, for providing me a stimulating and joyful environment at work.

I am also thankful to my roommates: Abhishek and Koushik, who never hesitated to help me out, when this manuscript was in progress.

TABLE OF CONTENTS

Table of Contents (Continued)

LIST OF TABLES

List of Tables (Continued)

List of Tables (Continued)

LIST OF FIGURES

List of Figures (Continued)

List of Figures (Continued)

Figure Page

Chapter 1

Introduction

The complexity of the mammalian brain has continued to interest the scientific community. A mammalian brain has the ability to perform cognitive tasks reliably and much faster than a silicon-based processor. The complex brain mechanisms of the mammalian brain, its ability to make decisions, remember, think and make inferences to its surroundings are of interest to neuroscientists and computing community alike. This interest has motivated several researchers to accurately model mammalian brain activity revealing several implications. The modeling research will help neuroscientists better understand human brain activity, hence assisting in the diagnosis and treatment of nervous disorders. Modeling the brain has also encouraged the domain of Artificial Intelligence, giving hope to creating a machine with human-like capabilities.

The brain is populated with neurons that are connected to one another with dendrites and axons, which act like “biological wiring”. The brain processes information by sending electrical signals across neurons through these connections. Tens of thousands of neurons are grouped together in a microcircuit called “the neocortical column.” These microcircuits are repeated several millions of times in the cortex and are responsible for cognitive processes such as thinking, remembering, and making decisions and inferences. Mammals differ only in the neuron density of neocortical column, for instance, 11 billion in humans and about 4 million in rats.

The brain structure, with the neuron connections as described, is amenable to a systematic modeling approach and has given rise to Artificial Neural Networks (ANNs), an important area of study in Digital Signal Processing. ANNs use neuron models to process signals and have been used in a multitude of applications such as Pattern Recognition, Computer Vision, Robotic control, etc. Since ANNs were inspired by the structure of neurons in the brain, they are practical options for brain modeling and related studies. However, to keep pace with the brain's superior processing capabilities, dense neuron models are needed, and given their ability to accurately model the brain, these models are highly compute-intensive. For instance, the highly accurate Hodgkin Huxley model for a neuronal simulation presented in this thesis requires about 246 flops to update the dynamics of a single neuron. If this model is used to simulate a rat size cortex with 4 million neurons, it will require approximately 10 billion flops to update the complete network. Further, to closely model the operation of said cortex, the calculations must be accomplished in real time. Hence specialized architectures such as GPUs and clusters for high-performance computing are intriguing platforms for implementing these large scale simulations which can enable highly accurate real time simulations of mammalian brain.

Our Work

Of several available Neural Networks, Spiking Neural Networks (SNNs) are of particular interest for modeling a mammalian brain because their functionality is highly biologically realistic. This research will investigate the acceleration of a two-level character recognition network that can recognize 48 alpha-numeric characters: English characters (A-Z), 10 numerals (0-9), 8 Greek letters and 4 symbols as used in [1]. The two-level network was implemented on an Nvidia Tesla C870 GPU with four SNN models, namely, the Izhikevich model, Wilson model, Morris Lecar model and lastly the Hodgkin Huxley model. A detailed discussion of the computation requirements for each of these models is presented in Chapter 3.

Our single computing node implementation performs the most compute-intensive portion of the algorithm in the GPU device and the remaining calculations are executed on the host processor. As we will discuss in Chapter 3, of the two levels, the input level-1 is the most compute-intensive level since it involves the neuron dynamics computation for N^2 neurons, where each neuron corresponds to a pixel in the input image. The output level-2, has significantly fewer total computations because it involves the neuron dynamics computation for only 48 neurons. The neurons for level-1 are implemented in the GPU and the level-2 neurons are simulated by the host processor, an Intel Core 2 Quad. The networks were written using CUDA for C version 2.1. The implementation details are discussed in Chapter 5. We have analyzed runtime performance on the GPU for each of the neuron models, evaluated each of the individual optimization techniques in detail, and investigated important device performance metrics such as kernel execution time, memory transfer time and flops offered for each of the implementations. The primary contribution of this thesis is the analysis of algorithmically different neuron models with their mappings on the compute device and implementing effective optimization strategies for optimal performance. Additionally, we investigate multiple GPUs in a single workstation to provide additional computing performance that will allow the accommodation of larger networks and provide higher application throughput. It also serves as an initial study for a cluster based implementation of the algorithms used in this research and similar applications.

The chapters hereafter are organized as follows: Chapter 2 surveys related work. Chapter 3 discusses Spiking Neural Networks in detail. Chapter 4 explores the GPU architecture, Nvidia's CUDA and the single node multi-GPU model. Experimental Setup and Implementation are presented in Chapter 5. Chapter 6 discusses the results of our implementations. Finally, conclusions and future work are discussed in Chapter 7.

Chapter 2

Related Work

Izhikevich in his classic paper [2] has described the features of biologically realistic neurons and proposed various models that describe the spiking and bursting properties of neurons. The simplest model to implement is the “Integrate & Fire” (I&F) model, which involves only 4 flops and one additional comparison with the threshold voltage. Many models for neural network simulations have used this I&F approach and this chapter will summarize some of the prominent works in this area and how they differ from the research presented in this thesis.

In [3], Delorme et al. attempted to capture the biophysical activity of neurons in their SpikeNET implementation, a simulator for modeling large networks of asynchronous spiking neurons using the I&F approach. The authors claim their implementation is efficient due to the small number of neuron updates required with each neuron occupying merely 16 bytes of RAM. The authors also claim that this implementation is fast as it can update roughly 20 million connections per second, which is sufficient to model 400,000 neurons in real time with a 1 ms time-step. The authors have applied this system to face and finger print recognition, demonstrating that the implementation is not only fast, but computationally efficient.

I&F based SNNs have been implemented to identify online cursive handwriting by Baig [4]. Baig was successful in recognizing 72% of the individual characters written digitally by the same writer with sample size of 1000 characters. Other implementations of spiking neural networks in the literature have been for sequence learning and detection [5]. In [6], Panchev et al. have used a temporal sequence detection scheme with spiking neurons for robot navigation and grasping tasks.

Several research activities have been motivated by the idea of modeling the neocortex. In [7], the authors have studied the mammalian neocortex in detail and implemented an abstract and scalable neural network model of the neocortex. The authors investigated its computation, memory and communication requirements while running on a parallel computer. The authors were successful in simulating a rat size cortex in 42% of real time and a cat size cortex in 23% of real time. The algorithms were written in MPI for C and executed on a Dell Xeon Cluster with 442 nodes. The authors found that the execution time is dependent on the computation rather than communication. The authors have also implemented a two-layer version of the model to detect 128x128 pixel images from the COIL-100 database [8]. They have instantiated their model with sparse recurrently connected neural networks that have spiking leaky integrator units and continuous Hebbian learning. These models also fit in the I&F category.

According to Izhikevich [2], the I&F based method used by the previous authors, is not biophysically meaningful and it does not model some of the prominent features of biological neurons. In his discussion, Izhikevich asserts that despite I&F's simplicity, they are not viable models for use in cortical spiking neuron simulations unless one wants to prove analytical results. In his study, Izhikevich proposed the use of biologically accurate models if the goal is to study true neuronal behavior.

In [9], the authors successfully used a biologically accurate Izhikevich model based neuronal simulation with 10^9 neurons and 10^{13} synapses (equivalent to a cat-scale cortical model) using the state-of-the-art Blue Gene/P machine with 147,456 processors and 144 TB of main memory. The authors claim that this simulation scale is roughly 1-2 orders smaller than the human cortex and 2-3 orders slower than real-time. EPFL's blue brain project is trying to reverse engineer the brain using highly biologically realistic models. They have used Hodgkin Huxley [10] and Wilfred Rall [11] models for simulating 100,000 neurons on the IBM BlueGene/L supercomputer.

The previous two research activities have focused on large scale supercomputers, which imply significant investment in machine procurement, maintenance and operation. Alternative architectures such as the Graphical Processing Units (GPUs) are now being investigated for biologically realistic simulations. In [12], the authors have implemented Izhikevich's random network on Nvidia's GTX-280 GPU with 1GB memory. They achieved a speedup of 26x over an equivalent software implementation for a network simulation of 100K neurons with 50 million synaptic connections. The authors have written their models in C++ similar to PyNN programming interface for neuronal simulations. They have discussed efficient mapping strategies on the GPU such that memory bandwidth and parallelism are efficiently utilized and have proposed the use of a GPU cluster for larger simulations.

The research presented in this thesis makes use of four different highly biologically realistic spiking neuron models to implement the image recognition system described in [1], whereas the research work in [3], [4], [5], [6], and [7] rely on I&F

approach, which is not biologically meaningful, hence cannot be used for the purpose of modeling studies. This effort to model accurate neuronal behavior in a complete network on a GPU differs from the work in [12], which focuses solely on implementing Izhikevich’s random network on a single GPU. The research in this thesis also analyzes multiple combinations of optimization strategies available with the CUDA programming model and investigates the appropriateness of an optimization technique for a particular application characteristic. Neither of these analyses is found in the related work. Further we analyze device performance metrics such as kernel time, memory transfer time, and flops in order to draw relationships between the achieved speed-up and network scalability and model complexity. We further initiate a first study of a GPU cluster implementation of the four networks using a single host multi-GPU system and study their feasibility for neuronal simulations.

In the next chapter we introduce the SNN models explored in this research and develop the two-level character recognition system based on these models.

Chapter 3

Background

This chapter presents background on the Spiking Neural Network (SNN) models used in this research, namely, the Izhikevich model, Wilson model, Morris Lecar model and the Hodgkin Huxley model. We present neuronal update equations for each of the models and compare them based on their Flop requirements. The chapter is concluded with the discussion of the two-level network implemented using the above mentioned SNN models.

Spiking Neural Networks

Simon Haykin in his book [14] defines a neural network as *"A massively parallel distributed processor made up of simple processing units, which has a natural propensity for storing experiential knowledge and making it available to use. It resembles brain in two aspects: 1. Knowledge is acquired through a learning process; and 2. Interneuron connection strengths, called synaptic weights, are used to store acquired knowledge."* Artificial Neural Networks (ANNs) are an attempt to mimic certain functionalities of the brain such as pattern recognition, decision-making, learning etc. The most fundamental component of the brain, the "neuron", takes electrochemical signals from other neurons and determines how to act based on those signals: fire or not. Accumulation and passing of these signals by collection of neurons describes the most basic functionality of the mammalian brain. In essence, neural networks are simulations of these neurons, typically 10^3 neurons for small applications. But in order to understand brain functionality, network sizes of more than a million neurons are required. Present day neural network simulations involve a kind of "clock timing" where time frames are specified for inter-neuron firing and the amplitude of these firings are considered to be constant, "Binary High" for instance. These models do not accurately replicate the behavior of the mammalian brain where signals are sent as brief spikes, typically 1 ms, and of small amplitude, typically 0.1 V.

The third generation of Neural Networks, the Spiking Neural Networks (SNNs), is highly biologically accurate. A "spiking" neuron fires an electric pulse at certain points in time which is commonly referred to as action potential or spike. The amplitude of the spike is irrespective of the input but the time of the spike is governed by the input received and hence this form of time encoding is used for processing information. Hence the SNNs are more suited for those applications where timing carries significant information such as speech recognition and signal processing.

Several models have been proposed for SNNs, some of them are compute efficient and some are more compute intensive and complex. As cited by Izhikevich in [2], biologically realistic models can describe the spike dynamics of each neuron and neuron connectivity. [2] describes the 20 most prominent features of biological neurons and ranks different models based on the number of neuro-computational features they can produce, their computational efficiency and the number of floating point operations per second each require. Four models namely the Izhikevich Model, Wilson Model, Morris Lecar Model and the Hodgkin Huxley Model were found to satisfy the requirements of accurately modeling neuron dynamics and hence were used in this research. In the following sections, we briefly describe each model as under and later describe the two level network based on [15].

Izhikevich Model

Izhikevich [16] developed a simple spiking model of neurons that is almost as biologically plausible as the Hodgkin-Huxley (HH) models and as computationally efficient as the I&F models for SNN. Using bifurcation methodologies, Izhikevich in [16] was successful in reducing the complex HH type models to 2-D system of ordinary equations as follows:

$v' = 0.04*v^2 + 5*v + 140 - u + I$ EQ. 3.1

$u' = a*(b*v - u)$ EQ. 3.2

After Spike resetting is done as:

If $v \geq +30\ mV$, $\quad$ then $v \leftarrow c$ and $u \leftarrow u + d$ EQ. 3.3

The variable *v* represents membrane potential of the neuron and *u* represents the membrane recovery variable which accounts for K^+ ionic current activations and Na^+ ionic current inactivation. Once the membrane potential reaches a peak of +30 mV, the membrane voltage and recovery variables are reset as given by equation 3. This model is reported to mimic firing patterns of all types of cortical neurons with appropriate selection of variables *a, b, c* and *d.* The model requires only 13 Flops to simulate 1 ms time-step of the model and hence highly plausible for large-scale simulations. Selection of parameters in this research is based on [1].

Wilson Model

In [17], the author attempted to model cortical neurons with system of polynomial equations. This model introduces additional Ca^{2+} and K^{+} channels to the Hodgkin Huxley model. These are "T" channel, "H" channel and the "R" channel. It has been reported in [2], that on proper tuning of its parameters, the Wilson model can mimic all the characteristics of spiking neurons. Four differential equations which govern the Wilson Model are:

$H' = (1.0 / 45.0) * (- H + 3.0 * T);$ EQ. 3.4

$T' = (1.0 / 14.0) * (- T + T\_infty);$ EQ. 3.5

$R' = \mathrm{dt} * (1/\mathrm{tau\_R}) * (- R + R\_infty) ;$ EQ. 3.6

$v' = (C) * (- m\_infty * (v - E\_Na) - 26.0 * R * (v + E\_K) - g\_T * T * (v - E\_Ca) - g\_H * H * (v+ E\_K) + I) ;$ EQ. 3.7

where,

$T\_infty = 8.0 *(v + 0.725) * (v + 0.725);$ EQ. 3.8

$R\_infty = 1.24 + 3.7 * v + 3.2 * v * v;$ EQ. 3.9

$m\_infty = 17.8 + 47.6 * v + 33.8 * v * v;$ EQ. 3.10

The model takes 45 Flops for 0.25 ms time-step and hence takes 180 Flops/ 1 ms. The parameters used in this thesis for the Wilson model are given in [1].

Morris Lecar Model

The Morris Lecar model is another biophysically accurate model replicating almost all the spiking neuron properties. As given in [18], the following equations describe the membrane potential with instantaneous Ca current activation and slow K current activation:

$Cv' = I - g_L*(V-V_L) - g_{Ca}*m_\infty(V)*(V-V_{Ca}) - g_K n(V-V_K)$ EQ. 3.11

$n' = \lambda(V)*(n_\infty(V) - n)$ EQ. 3.12

where

$m_\infty(V) = 0.5*(1 + tanh[(V-V_1)/V_2]$ EQ. 3.13

$n_\infty(V) = 0.5*(1 + tanh[(V-V_3)/V_4]$ EQ. 3.14

$\lambda(V) = \lambda' cosh[(V-V_3)/2V_4]$ EQ. 3.15

Parameters used in this thesis for these equations are used as mentioned in [1]. Since the model involves the evaluation of hyperbolic tangents and cosines, the model takes 60 Flops per 0.1ms time-step and hence 600 Flops/ 1 ms.

Hodgkin Huxley Model

The Hodgkin Huxley model [10] is considered to be the most accurate and the most important model in the neuroscience community to date. As mentioned in [2], the model involves 4 equations and ten parameters describing the Na and K current activation and Na current inactivation. The model is very computationally expensive to implement. Relevant equations are given as under:

$v' = (1/C)*\{I - g_K*n^4*(V-E_k) - g_{Na}*m^3*h*(V-E_{Na}) - g_L*(V - E_L)\}$ EQ. 3.16

$n' = (n_\infty(V) - n)/\Gamma_n(V)$ EQ. 3.17

$m' = (m_\infty(V) - m)/\Gamma_m(V)$ EQ. 3.18

$h' = (h_\infty(V) - h)/\Gamma_h(V)$ EQ. 3.19

The model takes 120 Flops for 0.1 ms time-step and hence 1200 Flops/ 1 ms. Table 3.1 shows the comparison of each of the models based on their Flop requirements.

Table 3.1: Flop requirements for all the models

SNN Model	**Flops Required**
Izhikevich Model	13 Flops/ms
Wilson Model	180 Flops/ms
Morris Lecar Model	600 Flops/ms
Hodgkin Huxley Model	1200 Flops/ms

The Two-Level Network

The SNN used for this research is based on [15] and the network used to test the models is shown in Figure 3.1. The task of the network is to detect images from a training set and its detailed operations is discussed in Chapter 5. The first level of neurons acts as an input collection layer and the second level of neurons acts as an output collection layer.

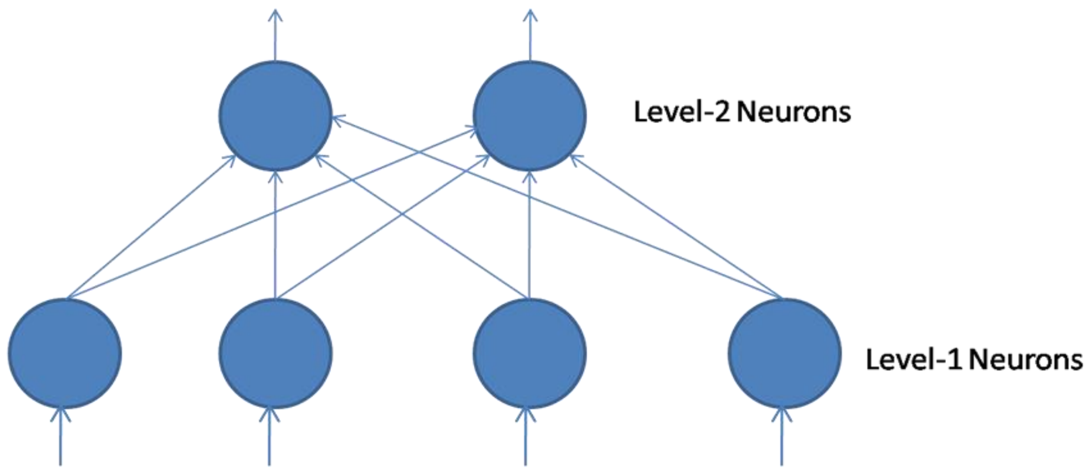

Figure 3.1: Two-Level Character Recognition Network

Each neuron in the input layer corresponds to a single pixel in the test images; hence the input layer level-1 has number of neurons equal to the total number of pixels in the image. The output layer, level-2, has number of neurons equal to the number of images in the database. Each level-1 neuron is fully connected to the level-2 neurons. A binary input image is presented to the input layer level-1 neurons and each neuron generates its membrane potential based on the pixel level presented to it. If a pixel is "on", a constant current is supplied to the input neuron for evaluating its membrane potential. A neuron is said to have "fired" if its membrane potential crosses a threshold value that is determined based on the model chosen. The input current for level-2 neuron is determined as:

$$I_j = \sum w(i,j)f(i) \qquad \text{EQ. 3.20}$$

Where, I_j is net input current for neuron j in level-2, $w(i,j)$ is the element (i,j) of the weight matrix representing the weight of the synaptic connection from neuron i of level-1 to neuron j in level-2. The weight matrix w is determined as described in [1]. A neuron fire in level-2 implies an image detected in a particular time step. The networks can be scaled to accommodate any images of larger size. The research presented in this thesis accelerates the recognition phase of each network by implementing all of the level-1 neurons on the single node and multi GPU systems.

In this chapter, we have introduced the four types of spiking neural network models which according to [2], are highly biologically accurate. We have presented neuronal update equations for each of the models and compared them based on their Flop requirements. The Hodgkin Huxley model was identified as the most compute intensive of all, with a Flop requirement of 1200 Flops/ 1 ms time-step. Izhikevich's model is the least compute intensive requiring only 13 Flops/ 1 ms time-step. The chapter has further discussed the two-level character recognition system, which uses the SNN models discussed. In the next chapter, we explore the single-GPU and multi-GPU architectures and the programming model used in this research, the Compute Unified Device Architecture (CUDA).

Chapter 4

GPU Architecture, Multi-GPU Systems and CUDA

This chapter will introduce the GPU architecture and multi-GPU systems, cover the main features of the Compute Unified Device Architecture (CUDA) by Nvidia, and discuss several of the optimization techniques available within the CUDA framework. The CUDA environment and optimizations covered are used in several implementations discussed in later chapters.

GPU Architecture Overview

A Graphical Processing Unit (GPU) also called a Visual Processing Unit (VPU), off-loads the computations for 3D graphics rendering from the microprocessor. Since their inception, GPUs have been employed in a variety of domains such as personal computer graphics rendering, gaming consoles, embedded systems such as mobile phones etc. Highly parallel structure of the GPUs makes them even more useful than the commodity processors for implementing complex algorithms.

A GPU is a fixed many-core processor dedicated to transforming 3-D scenes to a 2-D image composed of pixels. Figure 4.1 provides an outline of a traditional GPU pipeline, which is based on the purpose for which they were originally designed. As described in [19], the geometry stage, also called “Transform and lighting” stage, transforms objects to various spaces, each with its own co-ordinate system before transforming the object from 3-D to 2-D. Transformations are applied on a vertex-to-vertex basis. Lighting, which is another major component of this pipeline stage, computes the lighting properties of the vertex based on the camera and source positions.

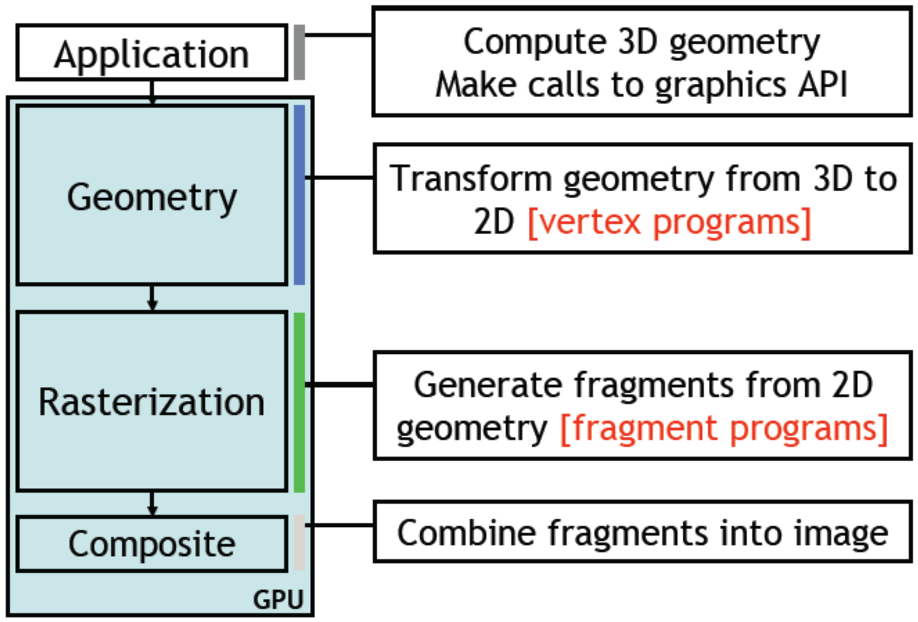
Application
Compute 3D geometry
Make calls to graphics API
Geometry
Transform geometry from 3D to 2D [vertex programs]
Rasterization
Generate fragments from 2D geometry [fragment programs]
Composite
Combine fragments into image
GPU

Figure 4.1: Traditional GPU Pipeline

The rasterization stage involves traversal of the generated 2-D image and conversion of the data into pixel candidates called *fragments*. A fragment is a data structure with data properties such as color, depth, pixel coordinates, etc. and is generated by checking which parts of the primitive intersect with the pixels in the scene. If a fragment intersects with the primitive but not with any of its vertices, the fragment attributes are calculated by interpolation. In Figure 4.2, it is seen that vertices of the fragment intersecting with the primitive take on the primitive's vertex colors. Pixels in the region inside the rectangle need to be evaluated by interpolation. Additional methods such as "alpha-bending" and "optional fog" can also be applied to obtain the colors.

Additional processes such as anti-aliasing can be applied to obtain the final result: declaring fragments as pixels.

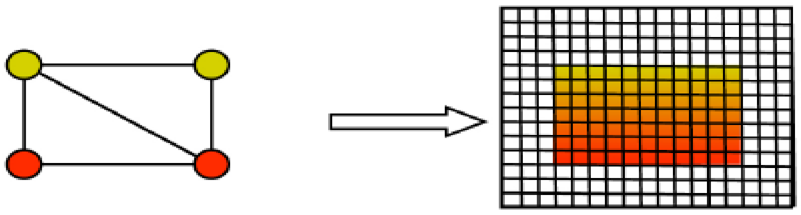

Figure 4.2: Rasterization of a triangle and interpolation of its color values.

The composite stage is responsible for combining fragments to produce an output image. The above description provides an outline of a traditional Graphics Pipeline which according to [20], has certain limitations:

1. Limited data reuse in subsequent stage
2. High state change overhead
3. Excessive variations in hardware usage, different code path for different hardware
4. Lack of integer instructions and weak floating point precision
5. Inability to write to the memory in the middle of the pipeline and read back on the top of the pipeline, limited resource utilization such as textures, shaders and registers, etc.

In November 2006, NVIDIA introduced the GeForce 8800 GTX which was designed to overcome these limitations with its unified pipeline and shader architecture. Figure 4.3 [20] shows the block diagram of the GeForce 8800 GTX GPU. The major processing stages of the traditional pipeline follow a fairly linear sequence, starting from vertex shading and proceeding to pixel shading, raster operations and writing pixels to the buffer. The shading stage in the traditional pipeline, for instance in the GeForce 7, has about 200 pipeline stages in the shader. The GeForce 8800 has a unified shader stage, such that the inputs are fed to the shader cores, outputs are written to the registers and again fed back to the cores forming a loop, hence significantly reducing the number of pipeline stages.

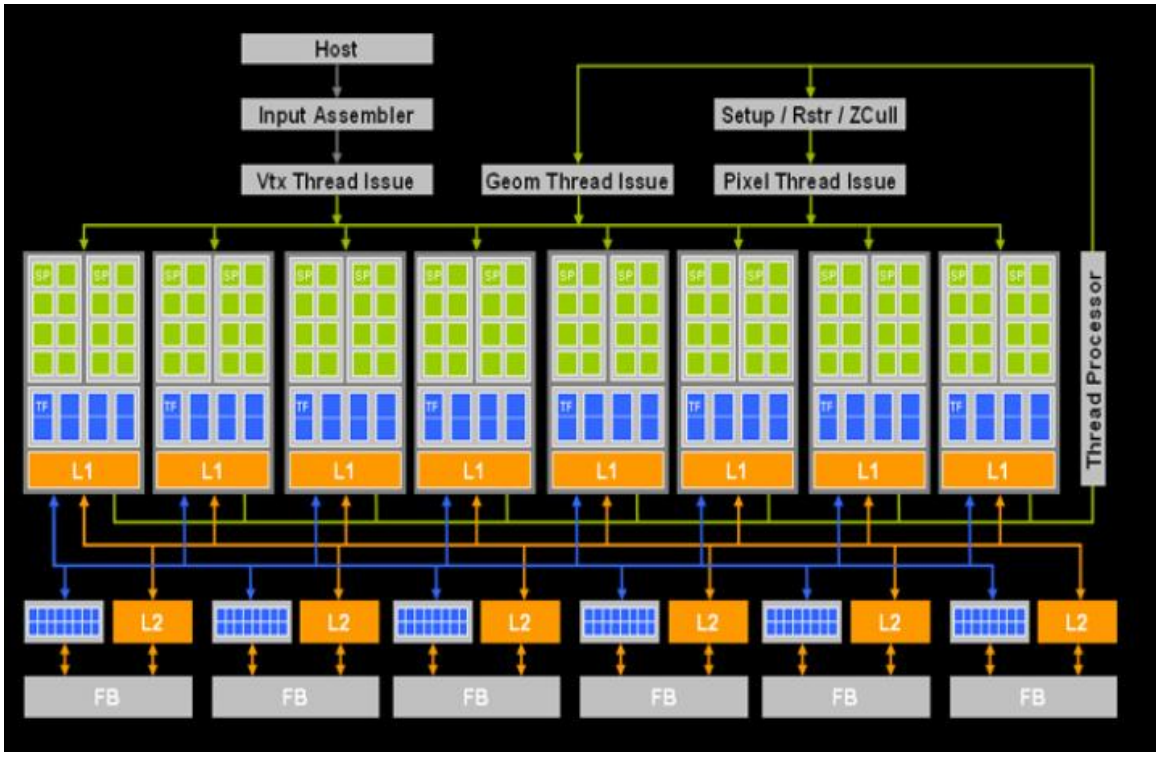

Figure 4.3: GeForce 8800 GTX block diagram [20]

GPUs perform vertex and pixel shading operations on the images and the workload on these shaders may vary from scene to scene. The traditional pipeline would simply have one of these units underutilized if the workload for that unit is less, hence not an optimal performance. GeForce 8800's unified shader architecture better utilizes the hardware such that it is not idle, irrespective of the vertex and pixel shader workload.

Unified stream processors in the GeForce 8800 can support vertex, pixel, geometry, and physics workloads and different workloads can be mapped on different processors.

Of the many features of the GeForce 8800, the streaming processor (SPs) architecture is the most pertinent to GP-GPU programming. Stream processors can work together in close proximity with extremely high parallel processing power. The outputs produced can be stored in fast cache and can be used by other streaming processors. SPs have instruction decoder units and execution logic performing similar operations on the data. This architecture allows SIMD instructions to be efficiently ported across groups of streaming processors. Figure 4.4 provides a layout of the SPs with relevant units.

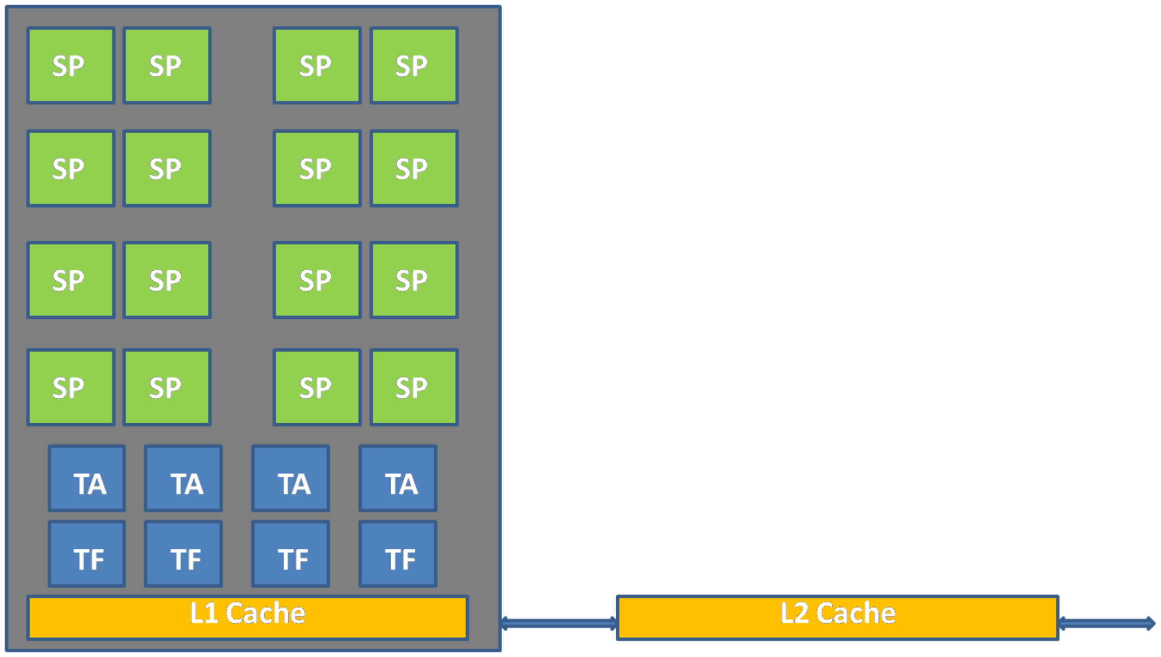

Figure 4.4: Streaming Processors and Texture Units

The streaming processors are accompanied by units for texture fetch (TF), texture addressing (TA), and caches. The structure is maintained and scaled upto 128 SPs. The SPs are fully decoupled, generalized, and scalar. They can dual issue MAD and MUL operations and support IEEE floating point precision. The SPs operate at 2.35 GHz in the GeForce 8800, which is separate from core clock operating at 575 MHz .

Multi-GPU Systems

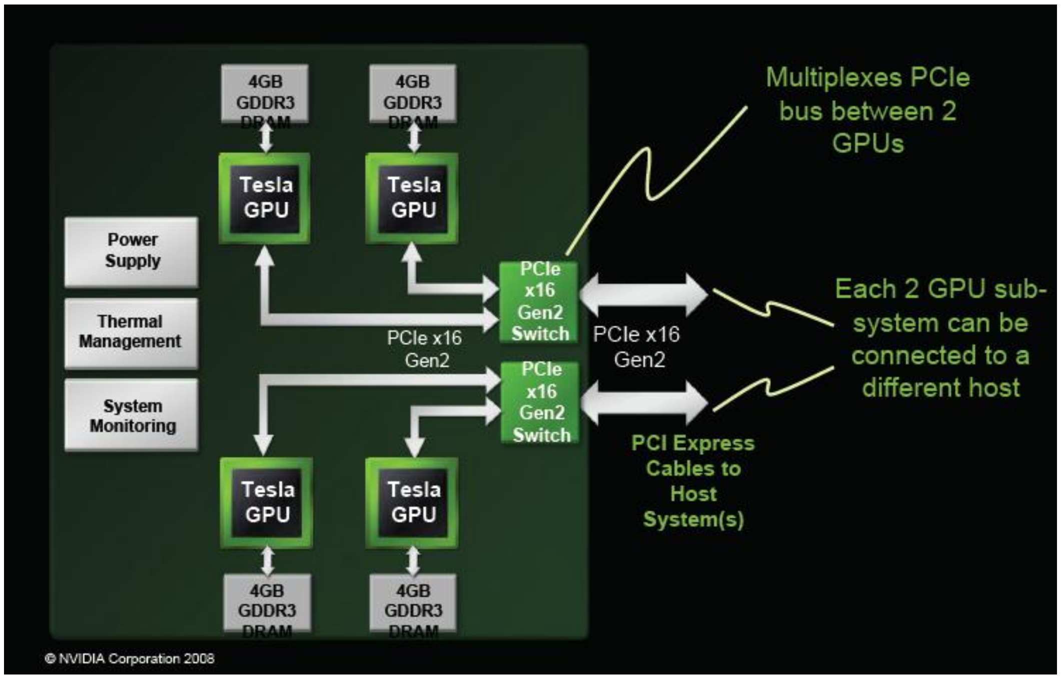

Figure 4.5: A Tesla S1070 Multi-GPU system

Figure 4.5 shows an example of a multi-GPU system; this figure specifically depicts the Tesla S1070 architecture. Many systems contain multiple GPUs, for instance many servers have Quadro/Tesla multi-GPU systems and often desktops are equipped with multiple GeForce/Tesla GPUs. Many applications consume more memory than provided in current GPUs and require investigation in multi-GPU systems. As shown in the above figure, each of the GPUs has its own bank of global memory. In order to transfer data from the global memory of one device to the global memory of another requires the host to act as a mediator. Typically a GPU transfers its data to the host and the host establishes context with the destination GPU to complete the transfer. Data transfers are accomplished through the PCIe bus although the GPUs may be connected to the same switch. Multiple GPUs in a single host requires explicit context creation between the host and the GPU device and only one such context can be created at a time. Multiple CPU threads can be created to establish contexts with multiple GPUs. A CPU thread can have context with a single GPU, although single GPU can establish context with multiple CPU threads. Device API management will be discussed in detail in Chapter 5.

Compute Unified Device Architecture (CUDA)

In this sub-section, we introduce the CUDA architecture framework and describe some of its prominent features. We introduce *kernels* and *thread hierarchy*, and then conclude this sub-section with the *memory hierarchy.*

In CUDA, the GPU functionality is defined by writing device functions in C, which are called in the literature, as *kernels.* Only one kernel can be executed in the GPU device at a time. A *thread* is a fundamental entity which is a sequence of instructions and is instantiated several millions of times. Typically, when a kernel is called, *N* threads execute the same kernel in parallel. The *thread hierarchy* is described as follows. A thread, in a CUDA program, is accessed inside kernels using in-built variable *threadIdx.* The *threadIdx* is a 3-D vector used to access 1-D, 2-D or a 3-D thread. Threads are further collected into 1-D, 2-D or 3-D *blocks.* Blocks are further divided into SIMD groups of 32 threads called *warps.* Warps are further divided into groups of 16 threads called the *half-warps*. The thread blocks can further be arranged in 1-D or 2-D *grids*. Figure 4.6 shows the thread hierarchy described above.

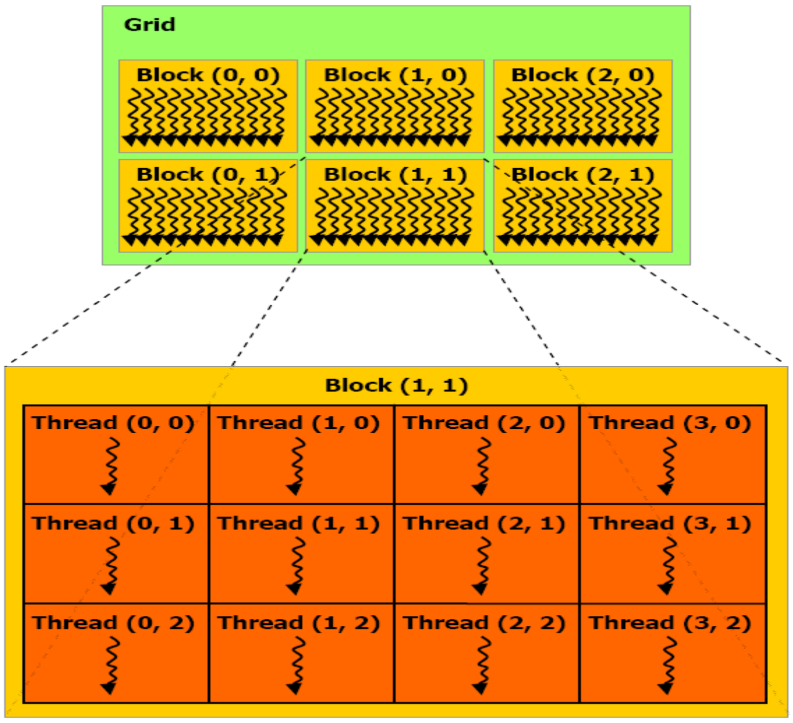

Figure 4.6: Thread Hierarchy in CUDA framework

Before we delve into the memory hierarchy, it is worth mentioning that the GPU device memory and the host memory reside in separate spaces. Host controls the data transfer flow between the GPU device memory and the host memory.

We now describe the *memory hierarchy*. At the fundamental level, each of threads have their own set of *local memory* and a set of *registers*. The local memory is off-chip and resides in the external device memory space. The term "local" refers to the scope of the variable. Threads in a block have collective access to *shared memory* and the shared memory is local to that block. Threads need to synchronize with each other in order to communicate via shared memory. All the threads have access to a *global memory* which resides off-chip in a DRAM. Off-chip, cached, read-only memory spaces, namely, the *constant memory* and the *texture memory* are also available. Texture cache is usually bound to either, *pitch memory* to satisfy padding requirements, *CUDA arrays* to provide specialized functions such as interpolation and filtering, or to the global memory itself. Figure 4.7 summarizes pictorally the memory hierarchy described above.

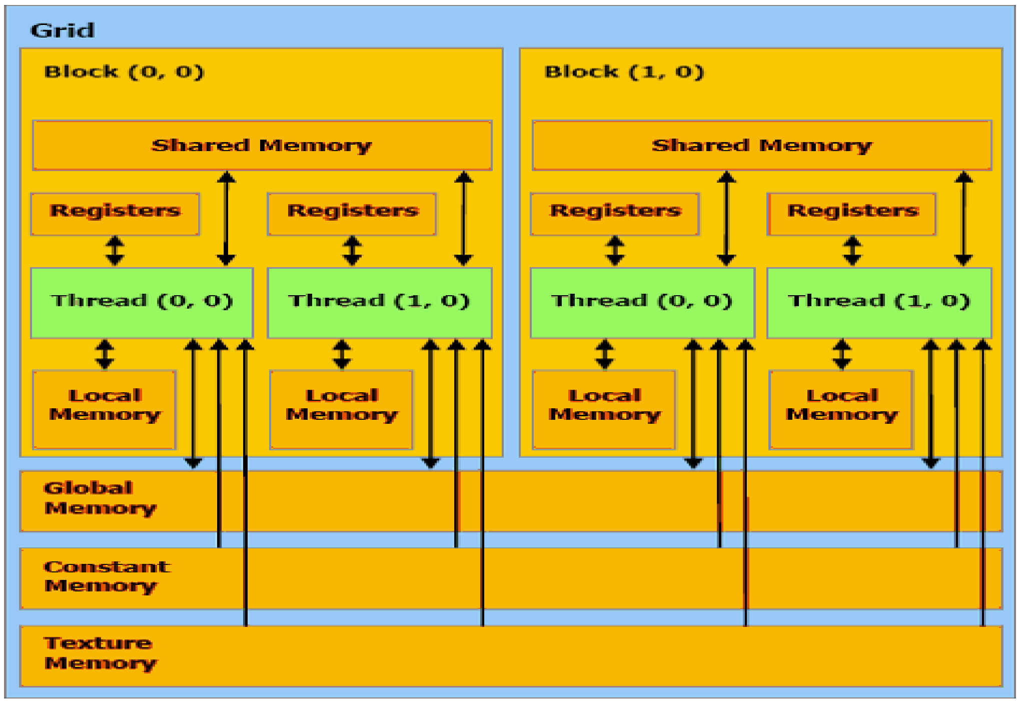

Figure 4.7: Memory Hierarchy in CUDA framework

The following sub-section will introduce some of the prominent optimization techniques made available by the CUDA programming model.

Optimization Techniques

CUDA programming model provides optimization strategies that can be used by the programmer for writing optimal codes. They are as follows. *Memory Optimization* involves the use of *coalesced global memory access*, use of *registers*, use of *shared* *memory*, and the use of *texture cache* and *constant memory* spaces. *Execution configuration* optimization will deal with the effects of varying the number of threads per block, and finally, this sub-section will be concluded with the discussion of *instruction optimization*.

Coalesced global memory accesses are discussed as the first memory optimization technique as the global memory accesses form the bottleneck for several applications. Global memory is off-chip and is not cached. This implies that each global memory access is an explicit memory access consuming 4 cycles for issuing reads and writes, and as many as 400-500 cycles to get the data from the global memory. Hence it is required to keep the use the global memory to a minimum and use registers and/or shared memory instead. Hence an application would typically fetch the data from the global memory space and load them into the registers or the shared memory space, where they are further operated upon. Once all the operations are completed, the data is sent back to the global memory. This process further involves the problem of *uncoalesced accesses to global memory*. According to compute capability 1.0, accesses to the global memory are *coalesced* if threads in a half-warp complete access to 32-bit word in a single 64-byte transaction, 64-bit word in a single 128-byte transaction, or 128-bit word in a series of 128-byte transactions. This is achievable if:

1) All 16 words lie in the same memory segment equal to memory transaction size
2) Threads access words in sequence, i.e., k^{th} thread accesses k^{th} word in the segment

If these conditions are not satisfied, it results in *uncoalesced global memory accesses* which are serialized, thus hurting performance. Hence in order to make use of maximum global memory bandwidth, it is required to keep the accesses coalesced. CUDA profiler signals *gld_incoherent* and *gst_incoherent* inform about uncoalesced loads and stores.

Registers are the fastest form of memory available on-chip, taking only 1 clock cycle for access. As discussed previously, in order to save clock cycles incurred by accessing global memory, operating on registers can improve performance. Registers are limited resources, typically 8192 registers per multiprocessor is made available for Tesla C870. Using too many registers leads to *register spilling*, a condition where off-chip local memory is used if the compiler cannot satisfy the programmer's request for register usage. This can lead to performance degradation as the local memory is off-chip. Hence in order to reduce the pressure on registers, *shared memory* can be utilized.

Shared memory is on-chip and is cached. 16 KB shared memory is made available for use per multiprocessor. Shared memory is as fast as registers if there are no *bank conflicts.* Shared memory is divided into *banks* and successive words are stored in successive banks. Threads in a half-warp are required to fetch data from these banks. If threads in a half-warp access the same bank, it leads to serialized accesses, and this is referred to as a *bank conflict.* The CUDA profiler signal *warp_serialize* informs about any bank conflicts. A broadcast mechanism is invoked if all threads try to access the same bank, hence avoiding serialization. Hence the key to maximum performance benefits is to avoid these bank conflicts.

Texture memory space is slow since it is off-chip but unlike the global memory, it is cached. Only on a cache miss will it incur a read from device memory; else, it leads to one read from texture cache. As mentioned previously, a texture cache can be bound to linear pitch memory, or to CUDA arrays, or to global memory. Since our application does not require specialized functions accompanying CUDA arrays, we have used texture binding to global memory. In our research, we have used texture bound to global memory as a fast look-up of the input image. *Constant memory* is again a limited, 8 KB per multiprocessor, read-only space, which is usually used to denote constants to reduce excessive register usage. It is also off-chip and incurs several clock cycles only if threads read different constant address spaces.

We will now discuss the *Execution Configuration* optimization. This technique involves changing number of threads per block, which leads to varying register and shared memory usage, and subsequently changing the *multiprocessor occupancy.* This value is provided by the CUDA profiler. Multiprocessor occupancy is defined as the ratio of number of active warps running on a multiprocessor and the maximum number of warps that can physically run on a multiprocessor. It is advisable to keep multiprocessor occupancy high. It is usually accomplished by keeping a large number (typically greater than or equal to 192) of threads per block and keeping this number a multiple of 32. This facilitates coalescing and hence hiding latency on memory bound kernels. A high multiprocessor occupancy does not always imply performance. A block configuration sufficiently higher with lower occupancy may still perform better than the one with lower block configuration and higher occupancy. This will become more evident in chapter 6.

Instruction optimization is discussed next. To execute an instruction, a multiprocessor must issue an operand read, evaluate the arithmetic instruction, and then write the result back. Hence the instruction throughput will depend on memory latency and bandwidth, and the nominal instruction throughput. All of these components can be optimized. Memory latency and bandwidth can be optimized by the use of fast memory spaces discussed previously. Nominal instruction throughput can be optimized by using *fast math* functions, for instance, replacing *funcf()* with *__funcf()*, replacing integer multiplication by *__mul24()*, floating point division by *__fdividef()*, etc. The fast math instructions can significantly reduce the number of clock cycles required for an operation thereby improving performance. One consequence of using the fast math instructions is that the accuracy may drop. *Control flow instructions* may require modification. For example, threads in a half-warp following different execution paths will lead to *divergent branches* whose execution is serialized. Programmer's ingenuity is required to avoid such divergent branches, although the compiler may choose to optimize the control flow instructions such that there are no divergent branches.

In summary, this chapter has served to introduce the GPU architecture and multi-GPU systems. We have also discussed the CUDA programming model used to program Nvidia's Tesla C870 card, several optimization techniques made available to the programmer, and how to use the CUDA profiler for optimizing performance. In the next chapter, we describe the experimental setup and implementation of the two-level character recognition networks based on the four SNN models introduced in Chapter 3.

Chapter 5

Experimental Setup and Implementation

In this chapter, we describe the single-GPU and multi-GPU setup used to implement the two-level character recognition system based on the four neuron models described in Chapter 3. We discuss the features of the two available GPU cards, and compare them in terms of compute and memory resources available to the programmer. This chapter concludes with a detailed section on parallelization and mapping of the models on single-GPU and multi-GPU systems.

Single-GPU and Multi-GPU Setup

The single-GPU experimental setup consists of a single Tesla C870 and a 2.66 GHz Intel Core 2 Quad host processor. The multi-GPU system consists of a Tesla C870 as the primary GPU accelerator and a GeForce 8400 GS as the secondary GPU accelerator. The host processor was the same 2.66 GHz Intel core 2 Quad. The SNN networks were developed using CUDA 2.1 installed on the host system running a 32 bit Fedora 8. The CUDA SDK provides the *deviceQuery* utility, which enables the developer to check various device properties. Table 5.1 lists some of the features offered by the two accelerators. CUDA, by default, sets Tesla C870 as Device 0. A single CPU thread can only establish context with a single device. Multiple threads are needed to establish multiple contexts across more than one GPU device on the same host and the details are discussed later in this chapter.

Table 5.1: Features available on the Tesla C870 and GeForce 8400 devices

Features	**Tesla C870**	**GeForce 8400 GS**
Compute Capability	1.0	1.1
Total Global Memory	1.5 GB	0.25 GB
Number of Multiprocessors	16	2
Number of Cores	128	16
Total amount of Constant Memory	64 KB	64 KB
Total Shared Memory Per Block	16 KB	16 KB
Total Registers available per block	8192	8192
Warp Size	32	32
Maximum number of Threads per block	512	512
Maximum Dimensions of each block	512 x 512 x 64	512 x 512 x 64
Maximum Dimension of Grid	65535 x 65535 x 1	65535 x 65535 x 1
Maximum Memory Pitch	256 KB	256 KB
Clock Rate	1.19 GHz	0.92 GHz
Concurrent Copy and Execution	No	Yes

The theoretical bandwidth of a device is an important factor to consider which is not provided by the *deviceQuery* but can be calculated from the data provided in the device specification sheet and as given in EQ. 5.1. The theoretical bandwidth of a device can be computed as:

Theoritical BW in GB/s

$$= (\text{Memory Clock})*(\text{Memory Interface})*(\text{DDR})/1024^3 \qquad \text{EQ. 5.1}$$

Figure 5.2: Tesla C870 Device Memory features

Memory Size	1536 MB
Memory Clock	800 MHz
Memory Interface	384-bit GDDR3

From EQ. 5.1 and the data in Table 5.2, the theoretical Bandwidth of the Tesla C870 card is 71.53 GB/s.

The Compiler and Profiling tools

Source files with the extension .cu should be compiled with the *nvcc* compiler. The *nvcc* compiler invokes all the necessary intermediate compilers and tools such as *cudacc*, *g++* and *cl* to create the final executable. The *nvcc* can either create C code that will be compiled with other tools, or *ptx* codes or object code. An executable with CUDA code requires the CUDA core library (*cuda*) and CUDA runtime library (*cudart*). More information on *nvcc* can be found in [21].

The CUDA SDK offers both textual and visual profilers to profile codes written in CUDA. In this thesis, the visual profiler is used to produce important information regarding the number of coalesced and uncoalesced global loads and stores, divergent branches, and other information such as total number of branches taken, microprocessor occupancy, GPU and CPU times, and the time taken for memory transfers. The CUDA occupancy calculator is yet another utility that can be used to obtain the microprocessor occupancy.

Parallelization and Mapping of Models

Table 5.3 provides the number of neurons at each level for the two-level networks. The initial neuron models with 576 neurons in level-1 and 48 neurons in level-2 were developed, trained and tested in MATLAB before converting them to C [1]. The research published in [13] used this model but could simulate only up to 5.7 million neurons. In this thesis, the above mentioned implementations were further improved, allowing the models to be scaled from 5.7 million neurons to 9.7 million neurons in level-1 as shown in the Table 5.3.

Table 5.3: Network Configurations for different Images

Input Image size	**Level 1 neurons**	**Level 2 neurons**	**Total Neurons**
96 x 96	9216	48	9264
192 x 192	36864	48	36912
240 x 240	57600	48	57648
384 x 384	147456	48	147504
480 x 480	230400	48	230448
720 x 720	518400	48	518448
960 x 960	921600	48	921648
1200 x 1200	1440000	48	1440048
1680 x 1680	2822400	48	2822448
2160 x 2160	4665600	48	4665648
2400 x 2400	5760000	48	5760048
2592 x 2592	6718464	48	6718512
2640 x 2640	6969600	48	6969648
2784 x 2784	7750656	48	7750608
2880 x 2880	8294400	48	8294448
3120 x 3120	9734400	48	9734448

An initial implementation described in [13], which was improved to be scaled up to 9.7 million neurons, involved acceleration of the level-1 neurons on the GPU accelerator and level-2 neurons on the host. The GPU computes the level-1 neuron firing information and sends this firing vector to the host processor, which in turn uses this information to evaluate the level-2 neuron dynamics. Single neuron dynamics in level-1 are computed by a single thread in the GPU; hence the number of threads is equal to the number of level-1 neurons, which is equal to the number of pixels in the input image.

Optimization techniques such as memory optimizations involving the use of *coalesced global memory access (G), use of shared memory (GS), coalesced global memory access with texture cache look-up (GT),* and a combination of these *(GST)* will be explored. Execution configuration optimization and instruction optimizations as described in the previous chapter will also be applied. The implementation described involves a single host to device transfer of all the parameters pertaining to the level-1. These parameters, from the equations discussed in the Chapter 3, can be viewed as vectors of neurons. Prior to computing the neuron dynamics, this set of data is transferred to the GPU device RAM in a single large transfer. Once the simulation begins, the firing vector, which is the neuron fire status vector is transferred from the GPU device to host at each time-step. This implementation uses the integer data-type for vector representation. Since each integer occupies 4 bytes, an image of size 3120 x 3120 would involve the transfer of about 37 MB firing vector. Since more accurate models such as the Hodgkin Huxley require several time-steps, this transfer from the host to device will constitute a significant amount of overhead. The performance achieved for this implementation is discussed in Chapter 6. We will refer the above implementation as *Implementation 1.*

Studying *Implementation 1*, it is apparent that the data transfers will cause significant overhead. Since the firing vector is merely a collection of flags, one per neuron, the data transfer overheads can be reduced significantly by changing the data-type for the firing vector from integer to character. This change will reduce the amount of data transfer by 1/4th. Discussed in detail in Chapter 6, this step although it saves transfer time introduces new uncoalesced global memory accesses. The Tesla C870 has Compute

Capability 1.0 that has stricter rules regarding coalesced accesses. Character data-types will involve uncoalesced accesses if successive threads access successive character data elements. Uncoalescing can be avoided if thread k^{th} accesses $(k+4)^{th}$ character. The speedup achieved and details of the uncoalesced accesses are discussed in Chapter 6. The approach described in this paragraph will be referred to as *Implementation 2* in future chapters.

Careful study of the models reveals that the level-1 neurons only fire every few time steps. In *Implementation 2*, the device kernel updates the vector at each kernel call leading to unnecessary global memory accesses and in turn uncoalesced accesses that lead to poor performance. To improve performance, the previous method was extended to perform vector updates inside the kernel only when a neuron fires instead of at each time step. This method removes a significant number uncoalesced access. The algorithm was also inspected for redundant computations and key mathematical operations such as division and exponentiation, which were replaced with fast math functions. Fast user-defined device functions were also written to evaluate the hyperbolic functions such as hyperbolic cosine, sine, and tangent. Additionally, registers were used to store the global memory data elements during operation to avoid repeated accesses to the high-latency un-cached global memory. This technique provides a savings of several clock cycles since a global memory read takes approximately 600 cycles and registers only require 1 clock cycle. A complete discussion of the implementation, *Implementation 3*, and performance is provided in Chapter 6.

Implementation 3, however, still involves the device to host transfer of the firing vector at each time step. By introducing a *block vector* to collect the flags per block, the frequency of transfer can be reduced. This block vector is similar to the firing vector, but instead of acting as collection of flags for neurons, it acts as a collection of flags for blocks. As discussed in previous chapters, a collection of threads called a thread block is defined by the user. The block vector is *blocksize* magnitude smaller than the complete firing vector and therefore reduces the data to be transferred to the host. If at any time-step, the block vector contains information of a firing event, only then will the entire firing vector be transferred from the device to the host. For instance, the HH model involves 373 time-step evaluations and the level-1 fires just once. Hence a nominal transfer of the block vector would mean merely a single transfer of the firing vector instead of 373 transfers. Results obtained using this implementation are discussed in Chapter 6. The technique introduced in this paragraph will be referred to as *Implementation 4.*

For the multi-GPU implementation, two POSIX-threads (p-threads) were created from the main CPU thread, each thread establishing a context with a single GPU. Both level-1 and level-2 parameter vectors were equally divided among the threads. Each thread was responsible for obtaining a partial firing vector from their respective GPU accelerator and synchronizing with each other to combine their partial firing vectors into a single global firing vector. The global firing vector is then used by the two CPU threads to evaluate the partial level-2 dynamics. Figure 5.1 illustrates this method. Figure shows two POSIX threads being spawned from the master CPU thread. POSIX thread-1 establishes context with the Tesla C870 and POSIX thread-2 establishes context with GeForce 8400. Each of the devices computes their designated partial firing vectors and passes it to their respective CPU threads. CPU threads then synchronize to obtain a global firing vector, and use the global firing vector to evaluate partial level-2 dynamics. The Implementations discussed for single node are also applied for Multi-GPU experiments. The multi-GPU system is studied for performance of different ratios of data division among the threads.

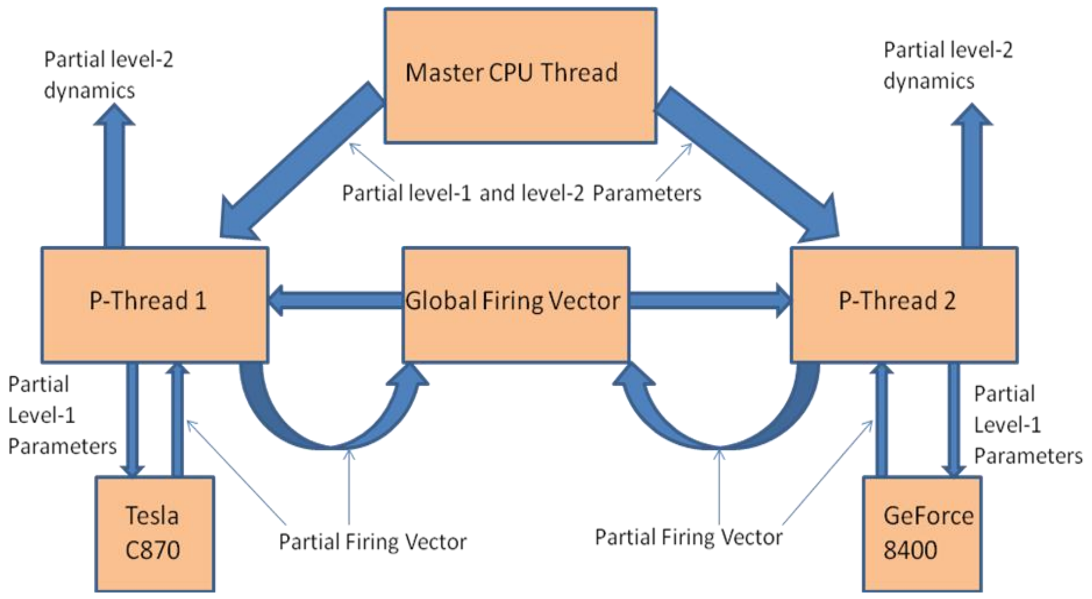

Figure 5.1: SNN models implementation on the Multi-GPU system

In this chapter we have introduced the single-GPU and multi-GPU setups for implementing the two-level character recognition systems based on the four models introduced in Chapter 3. Parallelization and mapping of the models has been described for both single-GPU and multi-GPU setups. Chapter 6 will present a detailed discussion of the results obtained.

Chapter 6

Results and Discussion

Although the models were introduced in order of increasing complexity, we will discuss the results obtained with highest complexity HH model first and then proceed to the lowest complexity Izhikevich model, as most of the features connecting the GPU architecture to algorithm are more apparent in the higher compute density models. We will discuss the single node experimental results first, analyze the Flop rate achieved using single node, and finally review multi-GPU results.

Single Node Results

The Hodgkin Huxley Model

Figure 6.1 shows the speed-up performance with the Hodgkin Huxley (HH) model for all four implementations discussed in Chapter 5. Table 6.1 provides the performance results for the intermediate network sizes. Clearly in the figure, *Implementation 4* outperforms the other implementations. For the largest network size, a speed-up of 576.9x was observed for *Implementation 4*, 213.81x for *Implementation 3*, 145.79x for *Implementation 2,* and 119.72x for *Implementation 1*. Such high speed-up values are expected for the HH model since the model has a high Flop/Byte ratio of 9.84. Calculation of the Flop/Byte ratio is discussed later in this chapter. Minimizing the communication and maintaining high arithmetic intensity enables the GPU to perform even better for this model. Next we will discuss the results for each of the implementations with the HH model.

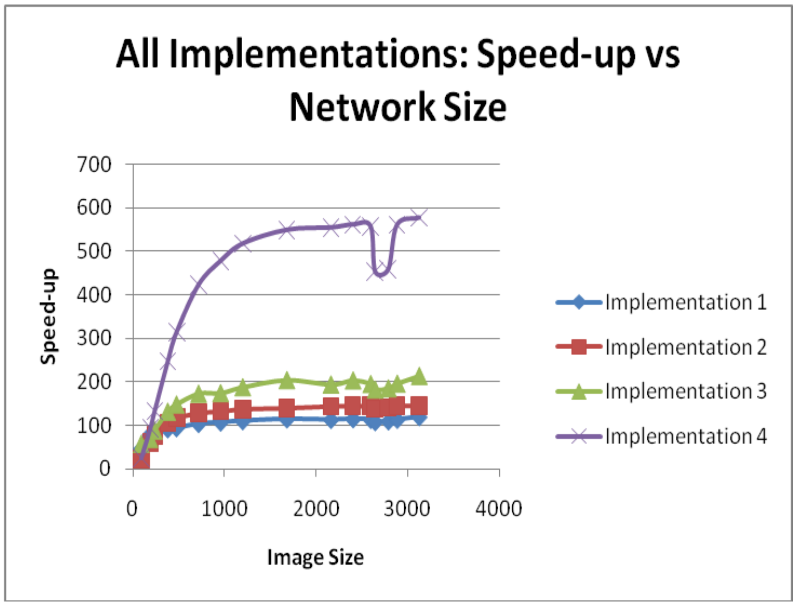

Figure 6.1: HH model: Speed-up vs Network Size; All Implementations

Table 6.1: HH model: Speed-up values for intermediate network sizes; All Implementations

Image Size	**Implementation 1**	**Implementation 2**	**Implementation 3**	**Implementation 4**
3120x3120	119.72	145.79	213.81	576.89
2400x2400	115.21	145.07	203.72	561.363
1680x1680	114.69	140.59	204.39	548.97
960x960	107.42	133	174.47	477.95
480x480	94.26	117.58	148.5	315.043

Implementation 1 consists of 4 designs corresponding to the optimizations introduced in Chapter 5: *G, GT, GS,* and *GST*. Figure 6.2 compares each of these memory optimization techniques with a common block configuration of 192. As discussed in Chapter 4, *blocksize* refers to the number of threads per block and is defined by the user.

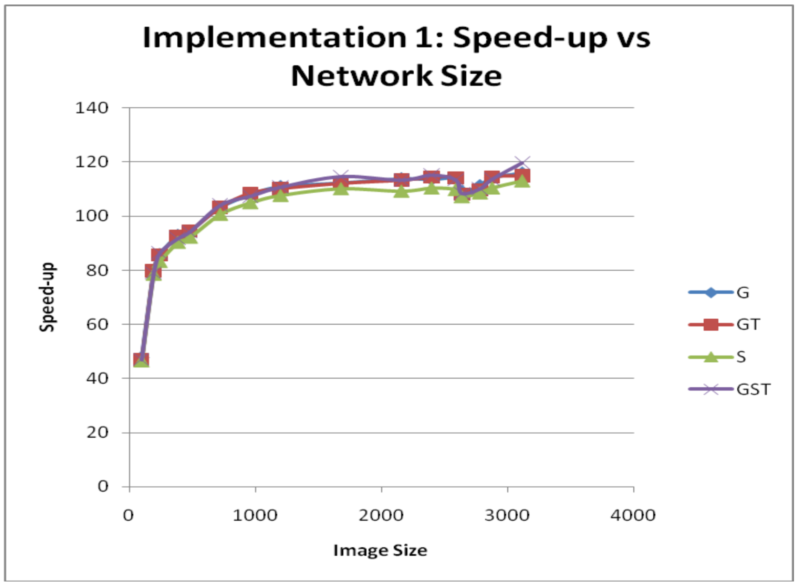

Figure 6.2: HH model: Speed-up vs. Network Size; Implementation 1

It has been observed that memory technique *GST* performs better than the others with a maximum speed-up of 119.7x for the largest network. Technique *S* has the lowest performance. Table 6.2 shows performance results for some of the intermediate network sizes across each of the optimizations.

Table 6.2: HH model: Speed-up values for intermediate network sizes; Implementation 1

Image Size	**G**	**GT**	**GS**	**GST**
3120x3120	115.95	114.93	113.11	119.73
2400x2400	113.85	114.37	110.44	115.22
1680x1680	112.334	112.12	110.2	114.7
960x960	103.18	103.28	100.7	107.42
480x480	94.53	94.46	92.21	94.25

The *CUDA Visual Profiler* is a useful tool for analyzing performance results. Table 6.3 shows the relevant profiler information that could potentially limit the performance. The data shown in the table corresponds to an image size of 3120. This network size will be used in similar studies throughout the thesis for consistency.

Table 6.3: HH model: CUDA Profiler results; Implementation 1

Parameter	**G**	**GT**	**GS**	**GST**
Block Configuration	192	192	192	192
Execution time (ms)	16876	16779	17094	16217
# of kernel calls	373	373	373	373
# of memcpy	379	379	379	379
Occupancy	1	1	0.5	0.5
Uncoalesced load	0	0	0	0
Uncoalesced Store	0	0	0	0
Divergent Branch	79430	79430	79430	79430
Serialized warps	0	0	0	0
Registers used per thread	11	12	18	18
Shared memory used per block	52	52	4660	3892
GPU time (sec) kernel	4.702	4.747	5.07	4.123
GPU time(sec) memcpy	10.13	10.125	10.025	10.0377

As seen in Table 6.3, *GST* optimization has the lowest kernel execution time while G*S* has the maximum kernel execution time of all designs. Overall for this implementation, *GST* has provided optimum performance with the use of fast shared memory to store parameters from global memory and texture cache as look-up for the image. *GS* and *GST* both have the minimum multiprocessor occupancy of 0.5, but *GST* was able to utilize texture cache resulting in lower kernel execution time since texture cache can provide better performance as discussed in Chapter 4. Memory transfer size was uniform across all the designs for *implementation 1*. The firing vector was transferred to the host at each time-step with its size equal to number of neurons in level-1, 3120^2 in this case. The last row of Table 6.3 also suggests the need to minimize the time spent in memory transfers for better performance.

Implementation 2, as discussed in Chapter 5, changes the firing vector from integer to character data-type to reduce the communication time. This implementation only uses *GT* memory optimization since the subsequent implementations will use faster and preferred registers instead of shared memory. A maximum speed-up of 145.8x was observed for this implementation for the largest network size.

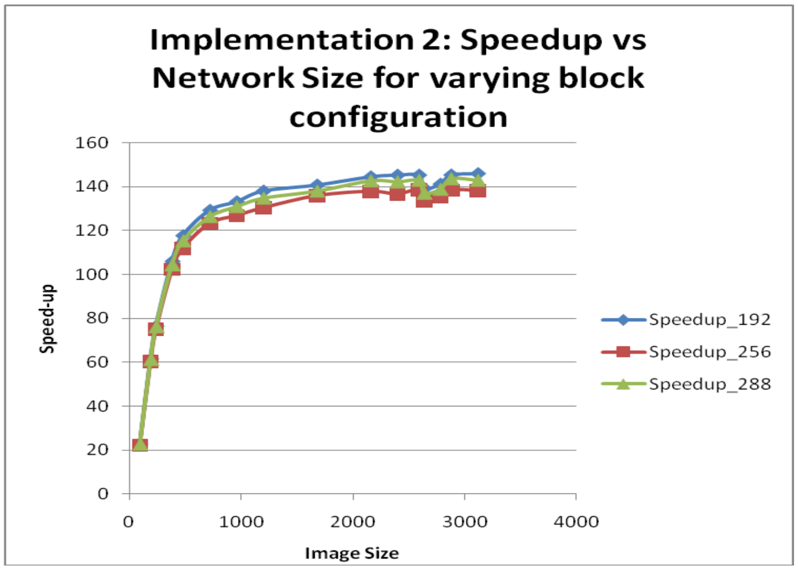

Figure 6.3: HH model: Speed-up vs Network Size; Implementation 2

Figure 6.3 compares the speed-up achieved for various block sizes and shows that optimal performance occurs at block size 192. Table 6.4 shows the intermediate performance values for the network sizes in this implementation. Occupancy equal to unity for blocksize of 192 confirms that it is the preferred size, for this implementation.

Table 6.4: HH Model: Speed-up values for intermediate network sizes; Implementation 2

Image Size	**Performance for BLOCKSIZE**			**Occupancy**		
	192	**256**	**288**	**192**	**256**	**288**
3120x3120	145.8	138.15	143.03	1	0.667	0.75
2400x2400	145.07	136.53	142.3	1	0.667	0.75
1680x1680	140.6	135.77	138.05	1	0.667	0.75
960x960	133	126.7	130.97	1	0.667	0.75
480x480	117.58	111.91	115.38	1	0.667	0.75

Implementation 3 is an extension of *Implementation 2,* using fast registers to minimize redundant accesses to global memory. Since the global memory is not cached, these accesses can lead to additional clock cycles. In this implementation, arithmetic intensity is further increased by maximizing utilization of math functions and removing redundant computation. The *GT* memory optimization was used in this implementation since it employs the use of registers instead of shared memory. A speed-up of 213.81x was observed for this implementation for the largest network size. Figure 6.4 shows the performance for different block configurations and Table 6.5 shows the performance for intermediate network sizes. A block configuration of 256 is most suited for this implementation given its high multiprocessor occupancy. Block configuration size of 288 has the least occupancy of 0.375 so it is unlikely that it will be capable of hiding global memory latencies.

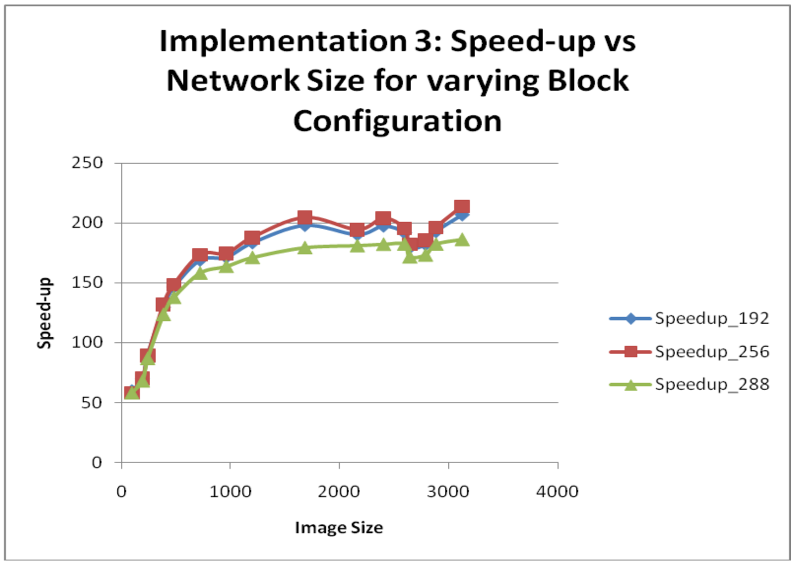

Figure 6.4: HH model: Speed-up vs Network Size; Implementation 3

Table 6.5: HH model: Speed-up values for intermediate network sizes; Implementation 3

Image Size	**Performance for BLOCKSIZE**			**Occupancy**		
	192	**256**	**288**	**192**	**256**	**288**
3120x3120	206.67	213.81	186.32	0.75	0.67	0.375
2400x2400	197.35	203.72	182.26	0.75	0.67	0.375
1680x1680	197.89	204.38	181.17	0.75	0.67	0.375
960x960	171.234	174.5	163.8	0.75	0.67	0.375
480x480	146.54	148.5	137.92	0.75	0.67	0.375

Implementation 4, a further extension to *Implementation 3*, minimizes the transfer frequency of the global firing vector with the introduction of the block vector as discussed in Chapter 5. Similar to *Implementation 3*, the *GT* optimization has been used as the memory optimization technique. A maximum speed-up of 576.89x was observed for this model. Figure 6.5 provides a graphical view of the performance for all the configurations and Table 6.6 shows the performance for different block configurations for this implementation. A block configuration of 256 has performed the best with occupancy of 0.67. Although it is slightly lower than that of block configuration 192, adding more threads per block for the configuration of 256 has lead to better performance. Clearly a lower occupancy of 0.375 for configuration of 288 has the least performance and hence is the least suited. This implementation shows clear difference in performance for configurations with different occupancy values. It also suggests maintaining a high multiprocessor occupancy, while maintaining larger block configurations. An unexpected drop in the performance was seen for image sizes 2640 and 2784. Inspection with the visual profiler, reveals that these network sizes had higher instruction counts than the other network sizes and thus an increase in execution time.

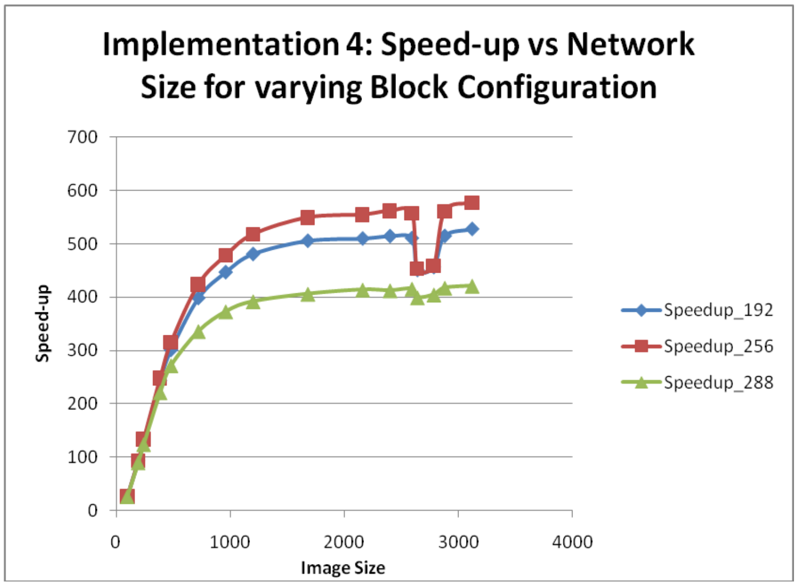

Figure 6.5: HH model: Speed-up vs Network Size; Implementation 4

Table 6.6: HH model: Speed-up values for intermediate network sizes; Implementation 4

Image Size	**Performance for BLOCKSIZE**			**Occupancy**		
	192	**256**	**288**	**192**	**256**	**288**
3120x3120	527.44	576.89	421.31	0.75	0.67	0.375
2400x2400	514.1	561.36	412.5	0.75	0.67	0.375
1680x1680	505.33	548.97	414	0.75	0.67	0.375
960x960	446.48	477.95	373.13	0.75	0.67	0.375
480x480	300.85	315.04	272	0.75	0.67	0.375

Now we present the CUDA visual profiler output in Table 6.7 for each of the 4 implementations with their best block configurations to provide a better insight of their relative performance. All implementations are compared with the same image size, 3120.

Table 6.7: HH model: CUDA Profiler results; All Implementations

Parameter	**Implementation 1**	**Implementation 2**	**Implementation 3**	**Implementation 4**
Block Configuration	192	192	256	256
Execution time (ms)	16217	13266.56	8998.98	3327.87
# of kernel calls	373	373	373	373
# of memcpy	379	379	375	376
Occupancy	0.5	1	0.67	0.67
Uncoalesced load	0	0	0	0
Uncoalesced Store	0	907732992	97920	3643432
Divergent Branch	79430	79430	51489	937954
Serialized warps	0	0	0	60861901
Registers used per thread	18	12	14	14
Shared memory used per block	3892	52	52	1084
GPU time (sec) kernel	4.123	6.74	2.65	3.02
GPU time(sec) memcpy	10.0377	2.5	2.42	0.384

Implementation 1 has a large memory transfer time due to the large firing vector that is transferred in each time-step. Introducing the use of character data-type for the firing vector in *Implementation 2,* reduces the firing vector byte count by ¼ and is reflected in an almost ¼ reduction in transfer time. The use of character firing vectors introduces uncoalesced accesses in *Implementations 2, 3,* and *4*. *Implementation 2* has the maximum uncoalesced accesses due to the firing vector update in each kernel call, resulting in significantly higher kernel time due to uncoalesced stores. *Implementation 3* minimizes uncoalesced accesses as discussed in Chapter 5, reducing the kernel time, but *Implementation 4* will have additional uncoalesced accesses due to the introduction of the block vector. Serialized warps in *Implementation 4* are primarily a result of synchronized access to and update of the block vector element by multiple threads in an active block. Table 6.7 also shows the importance of minimizing the host to device communication to gain better performance, as was accomplished in *Implementation 4.* Although a memory transfer is involved in each time-step, as discussed in Chapter 5, transfer of a relatively smaller block vector is more affordable than transferring the large firing vector in each time-step. It is seen that the memory transfer time is the primary reason for performance degradation in this application, dominating the other factors. The next hazard to performance is the uncoalesced accesses to global memory. Reducing the serialized warps, although not removed for *Implementation 4*, could further improve the performance. Since the access to shared memory is sufficiently fast in this case, this improvement is not expected to be as significant as removing the uncoalesced accesses to the global memory.

Morris Lecar Model

Figure 6.6 shows the performance results for all optimization techniques applied to the Morris-Lecar (ML) Model. Morris Lecar is the second most compute intensive algorithm after the HH model discussed in this thesis. Table 6.8 shows the intermediate performance results for each of the models. As seen in Figure 6.6, *Implementation 4* outperforms all the others. For an image size of 3120, *Implementation 4* gives a maximum speed-up of 105.86x, *Implementation 3* follows with a speed-up performance of 75.56x, *Implementation 2* has a speed-up of 65.7x, and finally *Implementation 1* is last with a speed-up performance of 55.4x. The Flop/Byte ratio for this model is slightly less than the HH model but is still sufficiently high to hide the data transfer overhead and utilize the computing performance of the GPU. We will now discuss each of the implementations in detail.

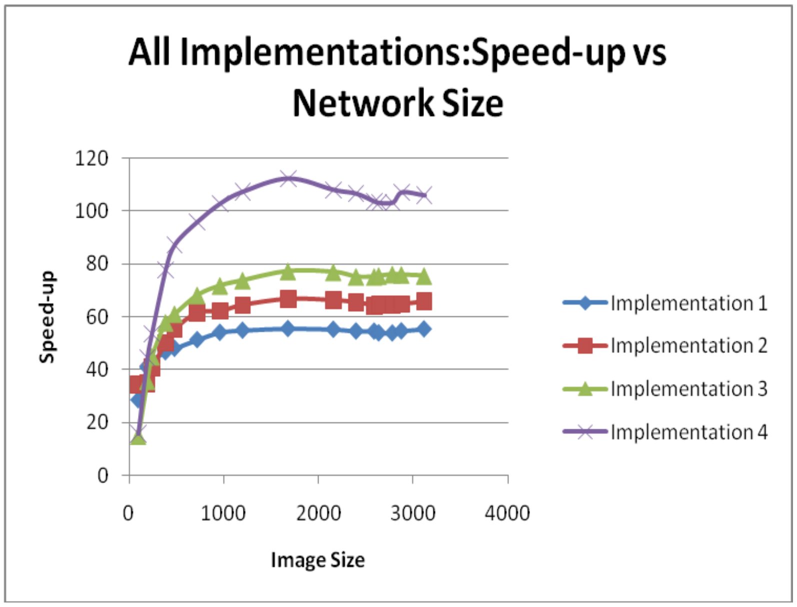

Figure 6.6: ML model: Speed-up vs Network Size; All Implementations

Table 6.8: ML model: Speed-up values for intermediate network sizes; All Implementations

Image Size	**Implementation 1**	**Implementation 2**	**Implementation 3**	**Implementation 4**
3120x3120	55.4	65.7	75.6	105.86
2400x2400	54.6	65.3	75.17	103.51
1680x1680	55.55	66.66	76.94	107.96
960x960	54.1	62.08	71.8	102.77
480x480	48.03	55.33	61.02	87.17

Implementation 1 involves the use of the memory optimization techniques discussed in Chapter 5 with the best block configuration for each. The performance graph is shown in Figure 6.7 and performance results for intermediate networks are given in Table 6.9.

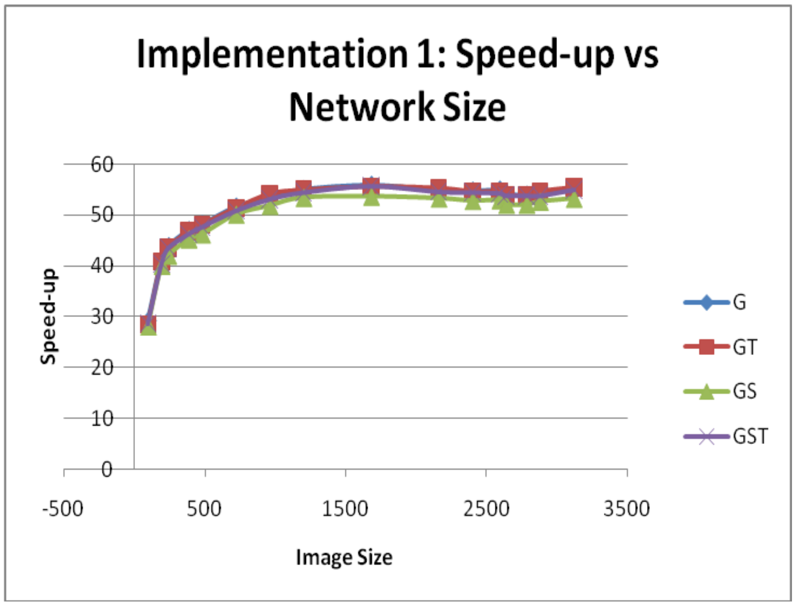

Figure 6.7: ML model: Speed-up vs Network Size; Implementation 1

Table 6.9: ML model: Speed-up values for intermediate network sizes; Implementation 1

Image Size	**G**	**GT**	**GS**	**GST**
3120x3120	54.8	55.4	53.2	54.9
2400x2400	54.72	54.59	52.89	54.44
1680x1680	55.85	55.55	53.6	55.65
960x960	53.53	54.13	51.70	53.09
480x480	48.17	48.03	46.17	47.64

It has been observed that memory optimization techniques *G*, *GT* and *GST* have near similar performance with *G* being marginally ahead of the others. The CUDA visual profiler results for image size 3120 are provided in Table 6.10 for each of the memory optimization techniques with their respective block configurations. For *G*, *GT* and *GST*, the optimal block configuration was found to be 192 whereas for *GS*, the configuration of 256 performed the best. The kernel execution time is maximum for the *GS* memory technique, which also has the least multiprocessor occupancy of 0.667. Optimization *G*, due to its maximum multiprocessor occupancy of 1, is capable of hiding latency, and results in the best choice for the implementation.

Table 6.10: ML model: CUDA Profiler results; Implementation 1

Parameter	**G**	**GT**	**GS**	**GST**
Block Configuration	192	192	256	192
Execution time (ms)	906.47	902.4	934.25	907.934
# of kernel calls	16	16	16	16
# of memcpy	21	21	21	21
Occupancy	1	1	0.667	0.75
Uncoalesced load	0	0	0	0
Uncoalesced Store	0	0	0	0
Divergent Branch	3055	3055	3048	3055
Serialized warps	0	0	0	0
Registers used per thread	10	10	15	12
Shared memory used per block	44	44	5164	3116
GPU time (sec) kernel	0.124782	0.12485	0.156337	0.129281
GPU time(sec) memcpy	0.497625	0.496658	0.497551	0.501096

Implementation 2, as discussed in Chapter 5, reduces the size of the firing vector and hence is expected to perform better than *Implementation 1.* Optimization *G* was the chosen memory optimization technique given its performance in *Implementation 1*. Figure 6.8 shows the speed-up performance for different block configurations and Table 6.11 shows the performance of the implementation for intermediate network sizes.

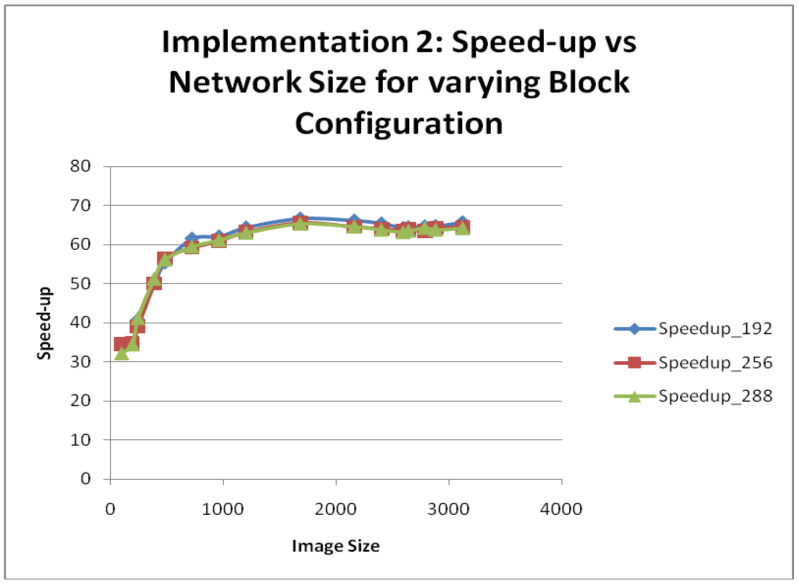

Figure 6.8: ML model: Speed-up vs Network Size; Implementation 2

A block configuration of 192 was found to be optimal for this implementation with highest occupancy of 1. Configuration 256 with occupancy identical to 192 performed similar to block configuration of 192, whereas a configuration of 288 with least occupancy of 0.75 could not perform better than the others.

Table 6.11: ML model: Speed-up values for intermediate network sizes; Implementation 2

Image Size	**Performance for BLOCKSIZE**			**Occupancy**		
	192	**256**	**288**	**192**	**256**	**288**
3120x3120	65.7	64.5	64.2	1	1	0.75
2400x2400	65.39	64.1	63.83	1	1	0.75
1680x1680	66.66	65.61	65.38	1	1	0.75
960x960	62.1	60.96	61.22	1	1	0.75
480x480	55.34	56.46	56.23	1	1	0.75

Implementation 3 further enhances the previous implementation with the introduction of the fast math functions as discussed in Chapter 5. Optimization *G* was again chosen as the memory optimization technique. Figure 6.9 shows the performance of different block configurations chosen and Table 6.12 provides performance results for intermediate network sizes.

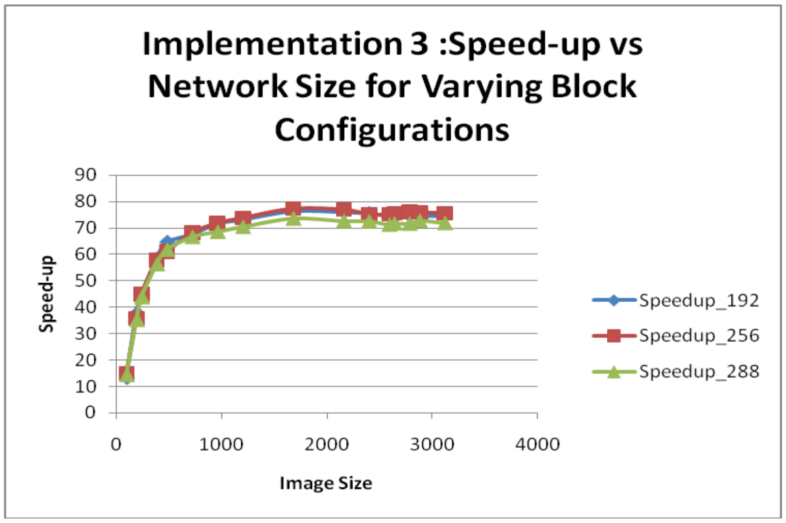

Figure 6.9: ML Model: Speed-up vs Network Size; Implementation 3

Table 6.12: ML model: Speed-up values for intermediate network sizes; Implementation 3

Image Size	**Performance for BLOCKSIZE**			**Occupancy**		
	192	**256**	**288**	**192**	**256**	**288**
3120x3120	74.6	75.56	72.03	0.75	0.667	0.375
2400x2400	75.53	75.17	71.36	0.75	0.667	0.375
1680x1680	76.46	77.3	73.74	0.75	0.667	0.375
960x960	71.77	71.8	68.69	0.75	0.667	0.375
480x480	64.67	61.02	61.92	0.75	0.667	0.375

This study also shows that high multiprocessor occupancy is important for performance, but is not the only factor. Clearly, although with a very large block configuration of 288, the implementation did not perform well due to its lower processor occupancy. The block configuration of 256, with similar occupancy to block size 192 performed marginally better. In this implementation, both the block configuration and occupancy need to be sufficiently high in order to provide optimal performance.

Results for *Implementation 4* of the ML model with the *G* memory optimization are given in Figure 6.10. Table 6.13 shows the results for intermediate network sizes. Block configuration of 192 performs the best, given its occupancy of 0.75.

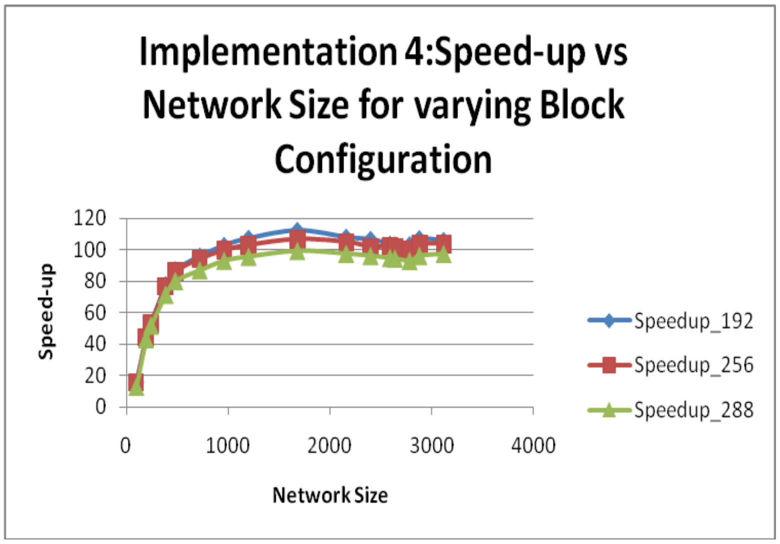

Figure 6.10: ML model: Speed-up vs Network Size; Implementation 4

Table 6.13: ML model: Speed-up values for intermediate network sizes; Implementation 4

Image Size	**Performance for BLOCKSIZE**			**Occupancy**		
	192	**256**	**288**	**192**	**256**	**288**
3120x3120	105.86	103.63	97.54	0.75	0.667	0.375
2400x2400	106.51	101.76	96.28	0.75	0.667	0.375
1680x1680	112.3	106.66	99.43	0.75	0.667	0.375
960x960	102.77	99.63	93.2	0.75	0.667	0.375
480x480	87.17	86.12	80.2	0.75	0.667	0.375

Details of the CUDA profiler are presented now to analyze the difference between the four implementations. The best block configuration from each of the implementations is chosen. Table 6.14 shows the relevant parameter details for all the implementations.

Table 6.14: ML model: CUDA Profiler results; All Implementations

Parameter	**Implementation 1**	**Implementation 2**	**Implementation 3**	**Implementation 4**
Block Configuration	192	192	256	192
Execution time (ms)	906.47	739.35	632.627	441.74
# of kernel calls	16	16	16	16
# of memcpy	21	21	20	21
Occupancy	1	1	0.667	0.75
Uncoalesced load	0	0	0	0
Uncoalesced Store	0	38937600	97564	300104
Divergent Branch	3055	3055	1521	50700
Serialized warps	0	0	0	1977096
Registers used per thread	10	10	14	13
Shared memory used per block	44	44	44	820
GPU time (sec) kernel	0.124782	0.215743	0.108979	0.157908
GPU time(sec) memcpy	0.497625	0.160376	0.157359	0.0599104

Implementation 1 does not involve any uncoalesced accesses or serial warps but memory transfer is a bottleneck (see last row of Table 6.14). This bottleneck is overcome in *Implementation 2* with the introduction of the character data-type firing vector, but it also introduces a large number of uncoalesced stores. While these uncoalesced accesses increase the kernel execution time, the reduction of memory transfer time dominates. *Implementation 3* reduces the number of uncoalesced accesses and introduces fast math functions, which together reduces kernel execution time. *Implementation 4* reduces the transfer frequency of the global firing data using the block vector approach and reduces the overall execution time, even though it introduces uncoalesced stores and serial warps. Since all threads of a warp access a particular block vector element, it leads to bank conflicts that result in serialized warps. Still, the significant reduction in data transfer time provides the best performance.

Wilson Model

The performance results for all implementations with the Wilson model are given in Figure 6.11 and Table 6.15. Again *Implementation 4* provides the best performance with a speed-up of 13.34x for the maximum image size 3120, followed by *Implementation 3* with 7.79x, *Implementation 2* with 5.72x, and lastly *Implementation 1* with a speed-up of 4.856x.

The best performance for *Implementation 1* was achieved with the *GS* memory optimization technique and a block configuration of 192. The speed-up performance for *Implementation 1* with different memory optimization techniques and their respective optimal block configuration are given in Figure 6.12 and intermediate results for the network sizes are given in Table 6.16.

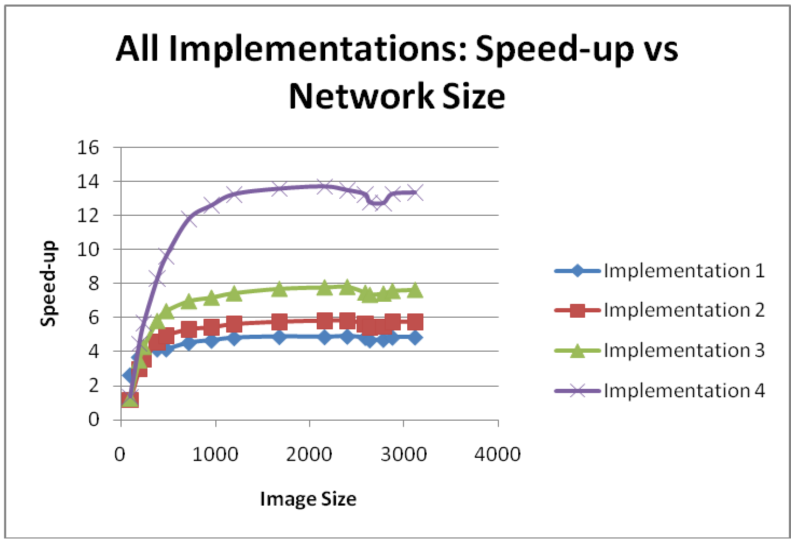

Figure 6.11: Wilson model: Speed-up vs Network Sizes; All Implementations

Table 6.15: Wilson model: Speed-up values for intermediate network sizes; All Implementations

Image Size	Implementation 1	Implementation 2	Implementation 3	Implementation 4
3120x3120	4.856	5.72	7.79	13.34
2400x2400	4.90	5.82	7.79	13.23
1680x1680	4.88	5.75	7.79	13.34
960x960	4.67	5.42	7.23	12.49
480x480	4.16	4.95	6.5	9.6

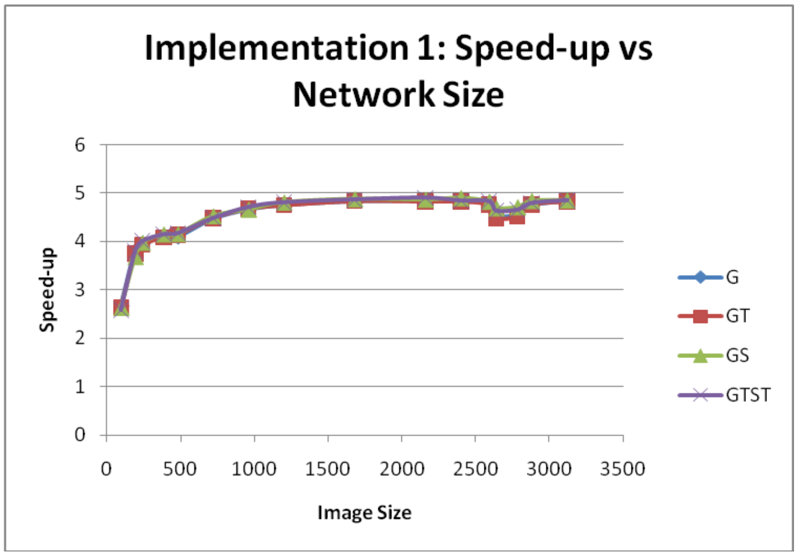

Figure 6.12:Wilson model: Speed-up vs Network Size; Implementation 1

Table 6.16: Wilson model: Speed-up values for intermediate network sizes; Implementation 1

Image Size	G	GT	GS	GST
3120x3120	4.839	4.826	4.856	4.847
2400x2400	4.836	4.823	4.9	4.84
1680x1680	4.832	4.831	4.88	4.86
960x960	4.66	4.672	4.667	4.7
480x480	4.1	4.136	4.158	4.168

In this implementation, all of the optimizations have provided similar results. The CUDA visual profiler results are given in Table 6.17. As is evident from the Table 6.17, *GS* although with lower occupancy spends less time in the kernel than the other optimizations. This behavior was not observed for *Implementation 1* with the HH and ML Models which are significantly more compute intensive than the Wilson Model.

Table 6.17: Wilson model: CUDA Profiler results; Implementation 1

Parameter	**G**	**GT**	**GS**	**GST**
Block Configuration	192	192	192	192
Execution time (ms)	1644.1	1645.81	1638.86	1639.99
# of kernel calls	30	30	30	30
# of memcpy	37	37	37	37
Occupancy	0.75	0.75	0.5	0.5
Uncoalesced load	0	0	0	0
Uncoalesced Store	0	0	0	0
Divergent Branch	1833	1833	1833	1833
Serialized warps	0	0	0	0
Registers used per thread	11	12	18	17
Shared memory used per block	52	52	5428	4660
GPU time (sec) kernel	0.334729	0.335067	0.315549	0.325544
GPU time(sec) memcpy	0.893263	0.897693	0.906602	0.891313

Figure 6.13 shows the performance results for *Implementation 2* with the *G* memory optimization for different block configurations. Table 6.18 shows the performance of the block configurations for intermediate network sizes. A block configuration of 192 has provided the best performance with a multiprocessor occupancy

of 0.75. The block configuration of 288 with occupancy of 0.75 is marginally better than the block configuration of 256 with occupancy 0.667.

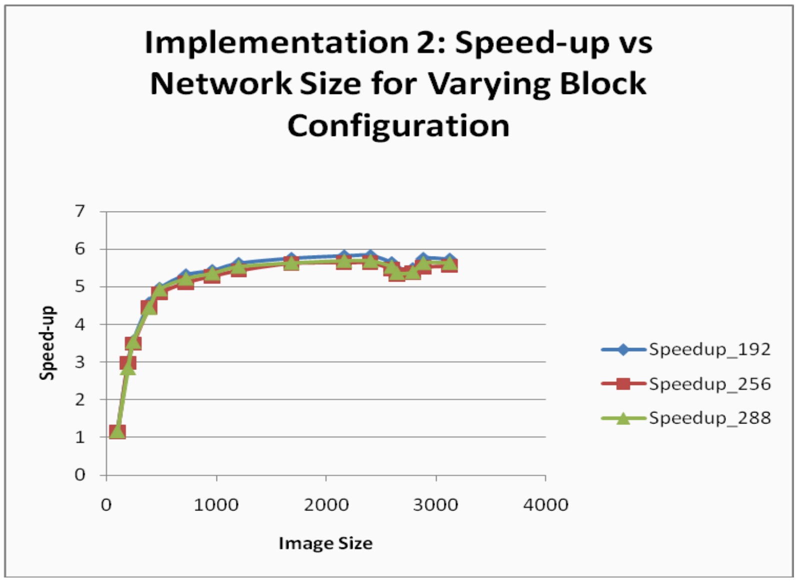

Figure 6.13: Wilson model: Speed-up vs Network Size; Implementation 2

Table 6.18: Wilson model: Speed-up values for intermediate network sizes; Implementation 2

Image Size	**Performance for BLOCKSIZE**			**Occupancy**		
	192	**256**	**288**	**192**	**256**	**288**
3120x3120	5.72	5.55	5.63	0.75	0.667	0.75
2400x2400	5.82	5.63	5.68	0.75	0.667	0.75
1680x1680	5.747	5.62	5.63	0.75	0.667	0.75
960x960	5.42	5.28	5.35	0.75	0.667	0.75
480x480	4.95	4.82	4.92	0.75	0.667	0.75

Implementation 3 results are shown in Figure 6.14 and Table 6.19. For this implementation, the block configuration of 288 provided the best performance with relatively higher occupancy of 0.75. Again this model indicates that the multiprocessor occupancy and the block configuration together impact performance; Occupancy alone is not an indicator of performance.

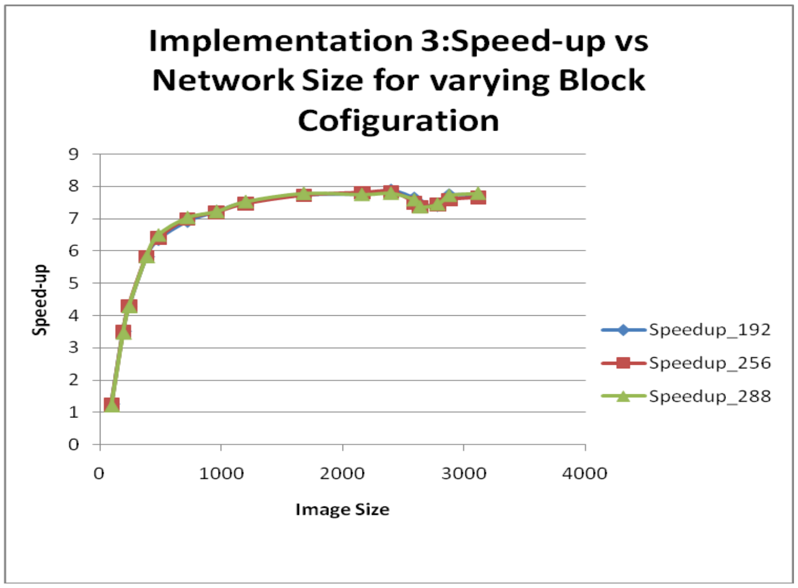

Figure 6.14: Wilson Model: Speed-up vs Network Size; Implementation 3

Table 6.19: Wilson model: Speed-up values for intermediate network sizes; Implementation 3

Image Size	**Performance for BLOCKSIZE**			**Occupancy**		
	192	**256**	**288**	**192**	**256**	**288**
3120x3120	7.71	7.66	7.79	0.75	0.667	0.75
2400x2400	7.86	7.83	7.79	0.75	0.667	0.75
1680x1680	7.73	7.72	7.79	0.75	0.667	0.75
960x960	7.21	7.2	7.23	0.75	0.667	0.75
480x480	6.35	6.4	6.5	0.75	0.667	0.75

Implementation 4 performs the best with a block configuration of 192 yielding a multiprocessor occupancy of 0.75. The block configuration of 288 with occupancy of 0.375 performed significantly slower than the others as shown in Figure 6.15 and Table 6.20. This implementation also shows that significantly lower multiprocessor occupancy is likely to perform worse than one with lower block configuration but much higher occupancy.

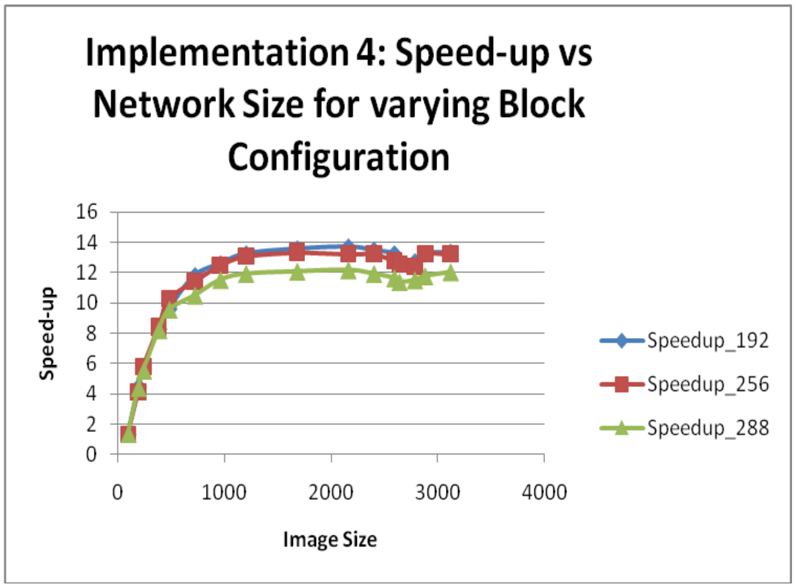

Figure 6.15: Wilson Model : Speed-up vs Network Size; Implementation 4

Table 6.20: Wilson model: Speed-up values for intermediate network sizes; Implementation 4

Image Size	**Performance for BLOCKSIZE**			**Occupancy**		
	192	**256**	**288**	**192**	**256**	**288**
3120x3120	13.25	13.25	11.75	0.75	0.67	0.375
2400x2400	13.47	13.23	11.91	0.75	0.67	0.375
1680x1680	13.56	13.33	12.06	0.75	0.67	0.375
960x960	12.58	12.49	11.9	0.75	0.67	0.375
480x480	9.6	10.26	0.5	0.75	0.67	0.375

The CUDA profiler results contrasting the different implementations for the Wilson model follow in Table 6.21. As shown in the table and discussed in the previous models, the relative performance of each of the implementations is as expected. The memory transfer time is reduced almost by ¼ when the firing vector size is reduced by a forth. Uncoalesced accesses clearly affect the kernel execution time but since the memory transfer time dominates, *Implementation 2* performs better than *Implementation 1*. Reducing the uncoalesced accesses in *Implementation 3* reduces the kernel execution time. The serial warps in *Implementation 4* do not significantly affect the kernel execution time. The order of impact on performance starting from the most important, memory transfer time, uncoalesced accesses and serial warps, is followed by Wilson Model.

Table 6.21: Wilson Model: CUDA profiler results; All Implementations

Parameter	**Implementation 1**	**Implementation 2**	**Implementation 3**	**Implementation 4**
Block Configuration	192	192	288	192
Execution time (ms)	1638.86	1376.99	993.35	565.924
# of kernel calls	30	30	30	30
# of memcpy	37	36	36	37
Occupancy	0.5	0.75	0.75	0.75
Uncoalesced load	0	0	0	0
Uncoalesced Store	0	73008000	97412	477554
Divergent Branch	1833	1822	570372	95062
Serialized warps	0	0	0	5375455
Registers used per thread	18	12	12	13
Shared memory used per block	5428	52	52	828
GPU time (sec) kernel	0.315549	0.516097	0.146388	0.182418
GPU time(sec) memcpy	0.906602	0.274549	0.274465	0.0870184

Izhikevich Model

Performance results for all the implementations of the Izhikevich model are shown in Figure 6.16 and Table 6.22 shows the performance results for intermediate network sizes. Similar to the previously discussed models, *Implementation 4* outperforms the rest by minimizing communication and maintaining high arithmetic intensity. For the largest network size, a maximum speed-up of 11.82x over the equivalent software implementation was observed.

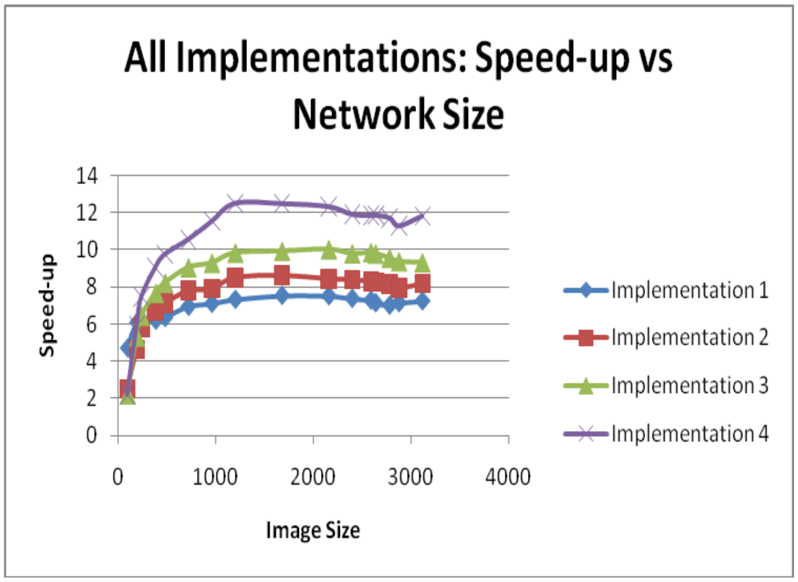

Figure 6.16: Izhikevich Model : Speed-up vs Network Size; All Implementations

Table 6.22: Izhikevich model: Speed-up values for intermediate network sizes; All Implementations

Image Size	**Implementation 1**	**Implementation 2**	**Implementation 3**	**Implementation 4**
3120x3120	7.23	8.16	9.33	11.82
2400x2400	7.36	8.38	9.8	11.91
1680x1680	7.51	8.6	9.94	12.48
960x960	7.1	7.9	9.3	11.5
480x480	6.33	7.11	8.22	9.75

Implementation 1 results are shown in Figure 6.17 and intermediate network size performance is given in Table 6.23. Similar to the Wilson Model, the *GS* optimization performed better than the other memory optimizations. It has been observed that *GS* has performed better for models having a lower flop/byte ratio (Wilson and Izhikevich); models with higher a flop/byte ratio (HH and ML) perform better than *GST or G optimization*.

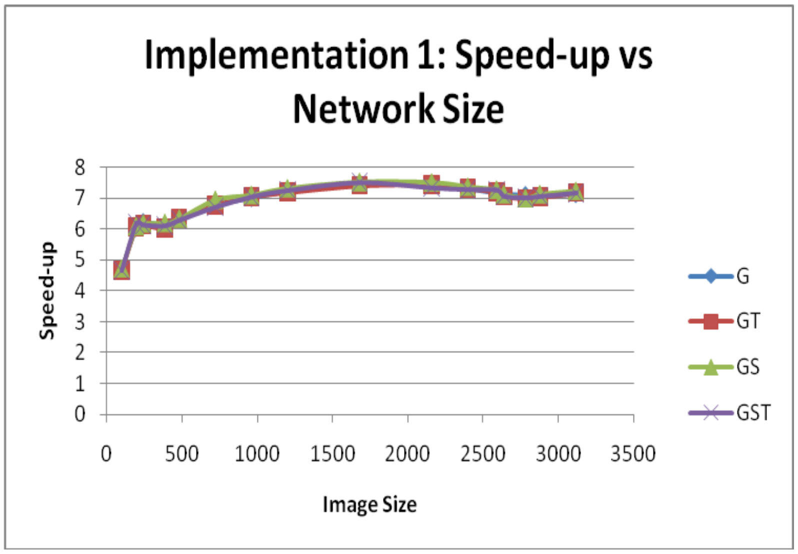

Figure 6.17: Izhikevich Model: Speed-up vs Network Size; Implementation 1

Table 6.23: Izhikevich model: Speed-up values for intermediate network sizes; Implementation 1

Image Size	**G**	**GT**	**GS**	**GST**
3120x3120	7.13	7.17	7.23	7.17
2400x2400	7.3	7.32	7.36	7.27
1680x1680	7.44	7.4	7.52	7.33
960x960	7	7.04	7.1	7.04
480x480	6.31	6.33	6.33	6.28

The CUDA visual profiler details for *Implementation* 1 with the Izhikevich model are given in Table 6.24. All of the memory techniques have the same multiprocessor occupancy of 1 and *GS* minimum kernel execution time. It is noted that *GS* performs better when the occupancy is high, but since Occupancy is a function of shared memory usage, a balanced usage between registers and shared memory can provide a higher occupancy and better performance.

Table 6.24: Izhikevich Model: CUDA Profiler results; Implementation 1

Parameter	**G**	**GT**	**GS**	**GST**
Block Configuration	192	192	192	192
Execution time (ms)	778.26	779.02	77.39	77.9
# of kernel calls	12	12	12	12
# of memcpy	16	16	16	16
Occupancy	1	1	1	1
Uncoalesced load	0	0	0	0
Uncoalesced Store	0	0	0	0
Divergent Branch	622	622	0	0
Serialized warps	0	0	0	0
Registers used per thread	6	6	8	9
Shared memory used per block	40	40	3112	3112
GPU time (sec) kernel	0.0598476	0.060057	0.0528743	0.056899
GPU time(sec) memcpy	0.369568	0.373658	0.378529	0.375308

Implementation 2 results are given in Figure 6.18. Block configuration 192 has provided optimal performance with occupancy of 0.75. Table 6.25 provides performance results for intermediate network sizes. The performance difference for block configurations 192 and 288 is very small even though these configurations share a high occupancy of 0.75. Additionally, the effects of multiprocessor occupancy and block configuration are somewhat nebulous for lesser compute density models. It can be established that algorithms need higher compute density in order to gain benefits from block configuration optimization.

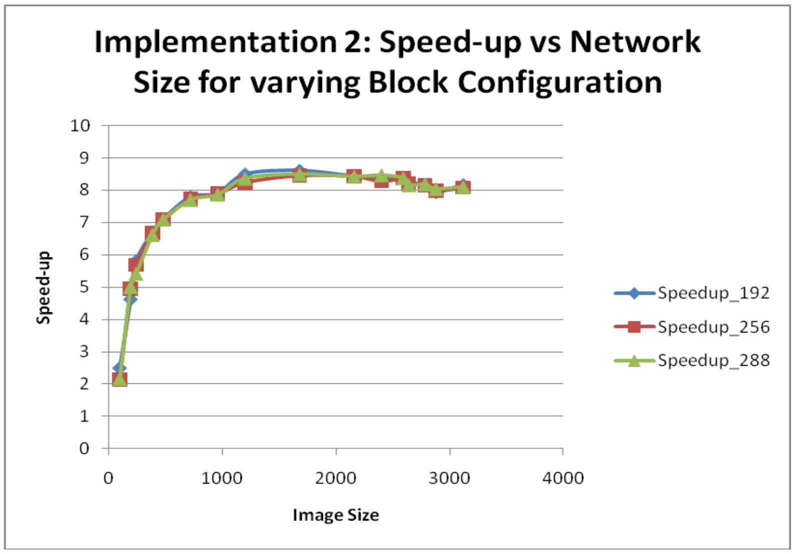

Figure 6.18: Izhikevich Model: Speed-up vs Network Size; Implementation 2

Table 6.25: Izhikevich Model: Speed-up values for intermediate network sizes; Implementation 2

Image Size	**Performance for BLOCKSIZE**			**Occupancy**		
	192	**256**	**288**	**192**	**256**	**288**
3120x3120	8.17	8.07	8.09	0.75	0.667	0.75
2400x2400	8.38	8.27	8.45	0.75	0.667	0.75
1680x1680	8.6	8.45	8.5	0.75	0.667	0.75
960x960	7.9	7.88	7.85	0.75	0.667	0.75
480x480	7.12	7.08	7.08	0.75	0.667	0.75

Implementation 3 results follow in Figure 6.19 and Table 6.26. For *Implementation* 3 with the Izhikevich model, a block configuration of 192 with a higher occupancy of 1 performed better in general although a clear winner was difficult to choose for this implementation. Discussion for this implementation is similar to

Implementation 2. Multiprocessor occupancy and block configuration do not appear to have significant direct impact on the performance.

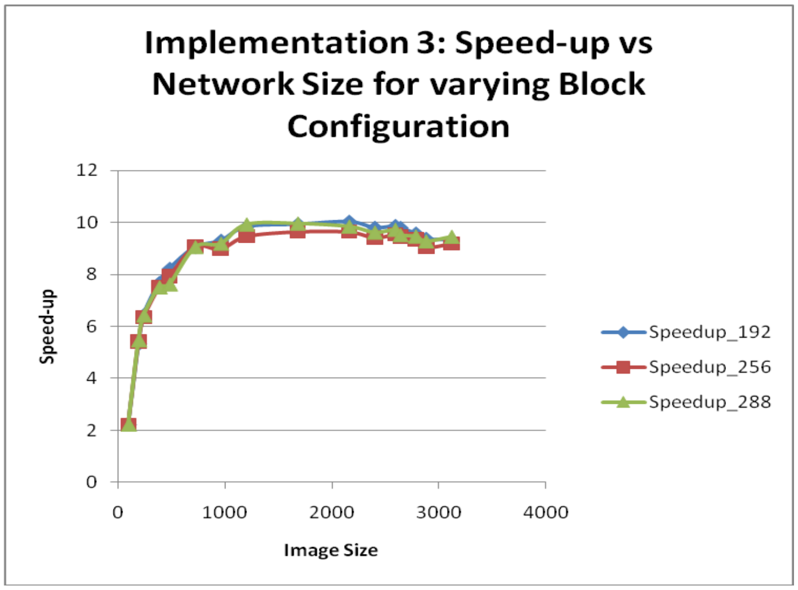

Figure 6.19: Izhikevich Model: Speed-up vs Network Size; Implementation 3

Table 6.26: Izhikevich model: Speed-up values for intermediate network sizes; Implementation 3

Image Size	**Performance for BLOCKSIZE**			**Occupancy**		
	192	**256**	**288**	**192**	**256**	**288**
3120x3120	9.34	9.2	9.46	1	1	0.75
2400x2400	9.8	9.42	9.62	1	1	0.75
1680x1680	9.94	9.65	9.96	1	1	0.75
960x960	9.3	8.98	9.2	1	1	0.75
480x480	8.22	7.93	7.61	1	1	0.75

Implementation 4 results are given in Figure 6.20 and Table 6.27. Following the trend seen in previous models, the block configuration with the maximum multiprocessor occupancy performs best. In this implementation, a configuration of 256 and occupancy 1 provides optimal performance for the largest image size, although a block configuration of 192 appears to perform better for other image sizes.

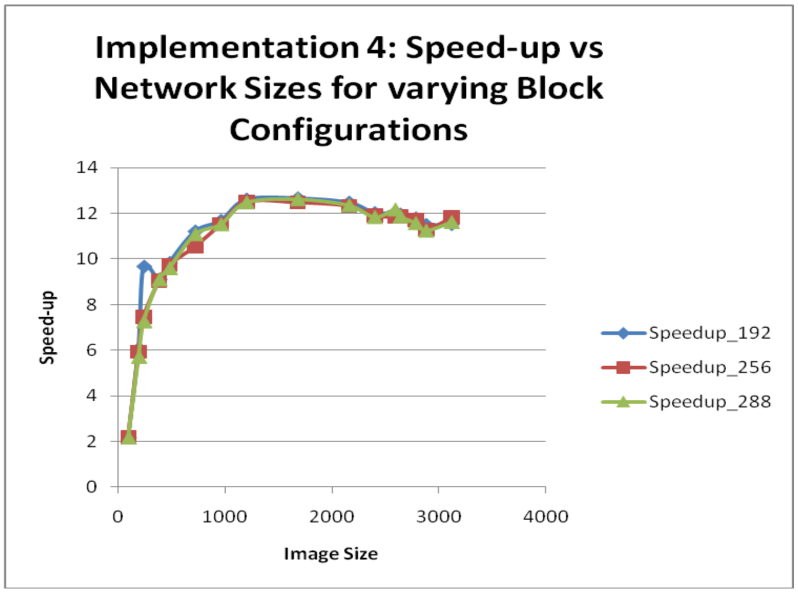

Figure 6.20: Izhikevich Model: Speed-up vs Network Size; Implementation 4

Table 6.27: Izhikevich Model: Speed-up values for intermediate network sizes; Implementation 4

Image Size	**Performance for BLOCKSIZE**			**Occupancy**		
	192	**256**	**288**	**192**	**256**	**288**
3120x3120	11.5	11.82	11.61	1	1	0.75
2400x2400	12.03	11.91	11.82	1	1	0.75
1680x1680	12.68	12.5	12.61	1	1	0.75
960x960	11.69	11.52	11.5	1	1	0.75
480x480	9.84	9.74	9.6	1	1	0.75

The CUDA visual profiler results are provided in Table 6.28. As seen in previous models, change of data-type, reducing the data transfer, reducing uncoalesced accesses and reducing the number of firing vector transfers gives significant performance improvement in each case.

Table 6.28: Izhikevich Model: CUDA Profiler results; All Implementations

Parameter	**Implementation 1**	**Implementation 2**	**Implementation 3**	**Implementation 4**
Block Configuration	192	192	192	192/256
Execution time (ms)	77.39	658.91	564.58	453.25
# of kernel calls	12	12	12	12
# of memcpy	16	16	16	17
Occupancy	1	1	1	1
Uncoalesced load	0	0	0	0
Uncoalesced Store	0	29203200	194500	308006
Divergent Branch	0	622	0	28518
Serialized warps	0	0	0	1589350
Registers used per thread	8	6	6	8
Shared memory used per block	3112	40	40	1072
GPU time (sec) kernel	0.0528743	0.135464	0.0370735	0.1052
GPU time(sec) memcpy	0.378529	0.122646	0.122912	0.0541576

FLOPs Study

We now present a study of Flops and Flop/Byte ratio for the best performing implementation of each of the models, *Implementation 4*. The Flop/Byte ratio is an important parameter for determining the appropriateness of an algorithm for architecture and vice versa. In Table 6.29, we present the Flop requirements for each of the four models. The HH model requires the most, since it is more biologically accurate, followed by Morris Lecar, Wilson, and Izhikevich Models. The memory requested by each of the models is another significant parameter and is used to evaluate the Flops/Byte ratio.

Table 6.29: Flops/Byte Ratio for all models

Model	**FLOPS Required for Neuron Update**	**Flops/Byte Ratio**
HH	246	9.84
Morris Lecar	147	8.65
Wilson	38	1.52
Izhikevich	13	0.9997

The Flops/Byte ratio in our research is viewed as an algorithm specific value and is defined as the ratio of the Flops required by all the neuron updates (level-1 and level-2) to the overall bytes requested for all neuron updates (all the parameters, firing vector and block vector).

Figure 6.21 shows the Flop count vs. image size for each of the models using *Implementation 4*. Table 6.30 shows the Flops achieved for the intermediate network sizes. Figure 6.22 provides a closer look at the Flops achieved for the Izhikevich's model.

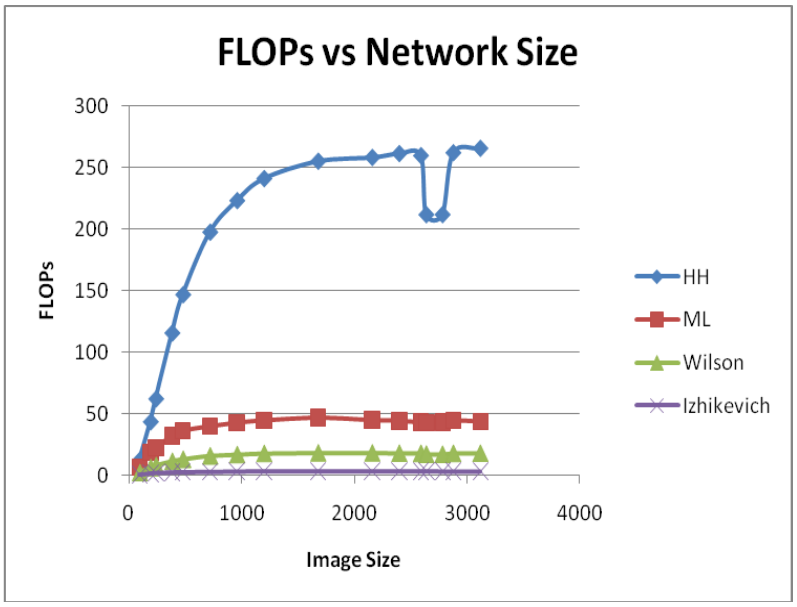

Figure 6.21: FLOPs vs Network Size; All Models

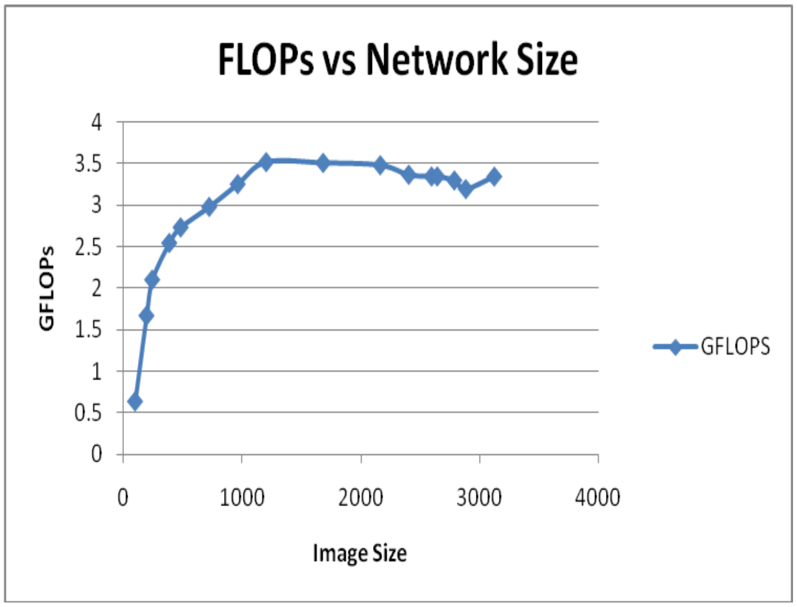

Figure 6.22: FLOPs vs Network Size; Izhikevich Model

Table 6.30: FLOPs achieved for intermediate network sizes; All Models

Image Size	**HH (GFLOPs)**	**Morris Lecar (GLFOPs)**	**Wilson (GFLOPs)**	**Izhikevich(GFLOPs)**
3120	261.97	44.47	17.58	3.19
2640	261.27	44.2	17.68	3.36
1680	255.06	46.68	17.96	3.5
480	146.72	36.21	12.77	2.73
240	62.06	22.26	7.67	2.1

Flops are calculated as:

Total Flops=(flops per neuron update)(number of time-steps)*(total number of neurons)/ (Application time in seconds).* EQ. 6.1

As shown in Figures 6.21 and 6.22, the device saturates beyond a certain network size. For the HH model, the saturation occurs beyond image size 1680, which corresponds to 2.3 million neurons. The ML model saturates beyond the same network size as that of the HH model. The less compute density Wilson model saturates beyond image size 1200, corresponding to 1.44 million and the same was observed for the Izhikevich's model. It is seen that the higher the Flops/Byte ratio, the better the device is expected to perform. In our case the HH model, with highest Flop/Byte ratio, 9.84, performed the best (576.9x). Izhikevich with lowest Flop/Byte ratio of 0.9997 performs the least (11.82x).

Multi-GPU Results

This section investigates the feasibility of multi-GPU systems for large-scale SNN simulations. In this section, we consider implementing the most compute-intensive Hodgkin Huxley model on a multi-GPU system. The best implementation, *Implementation 4*, from the single-GPU investigations will be used here. This section also provides an initial study of partitioning the problem across multi-GPUs. Figure 6.23 shows the performance achieved for different block configurations of *Implementation 4* on the multi-GPU system when the data is divided equally between the GPU devices. Table 6.31 shows the performance results for the network sizes for *Implementation 4*. The block configuration of 256 consistently performed better than 192 and 288 block configurations; for the multi-GPU implementation. Context creation, described in Chapter 5 involves significant overhead. Although not presented in this section, all of the models have similar application time for the lower network sizes. The CUDA visual profiler was not able to determine multiprocessor occupancy for each of the GPU devices.

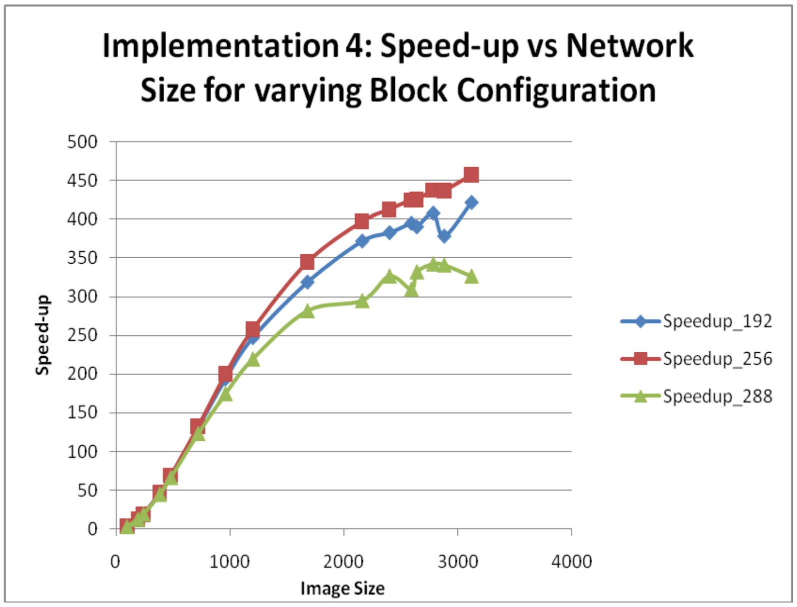

Figure 6.23: HH Model: Multi-GPU: Speed-up vs Network Size; Implementation 4

Table 6.31: HH Model: Multi-GPU: Speed-up values for intermediate network sizes; Implmentation 4

Image Size	**Performance for BLOCKSIZE**		
	192	**256**	**288**
3120x3120	421.26	457.16	327
2400x2400	382.28	412.94	327.17
1680x1680	318.51	344.6	282.26
960x960	193.66	199.97	174.97
480x480	68.31	69.19	66.5

Since the GPU devices are heterogeneous (having different compute capabilities, clock frequency, etc.), we will analyze the results when the data is split between the GPUs in different proportions. When the data is equally divided between the GPUs, the block configuration of 256 performed the best and was therefore chosen for this study.

Figure 6.24 and Table 6.32 summarize the results as the amount of data processed by each of the GPU is varied. The *Ratio_x* in the Figure 6.24 denotes the fraction of data to be processed by the Tesla C870 card. Hence a ratio 0 would mean the entire data is moved to GeForce 8400 device memory and hence entirely processed by the GeForce 8400 GPU device; a ratio 1 would mean the entire data is moved to Tesla C870 device memory and hence solely processed by Tesla C870.

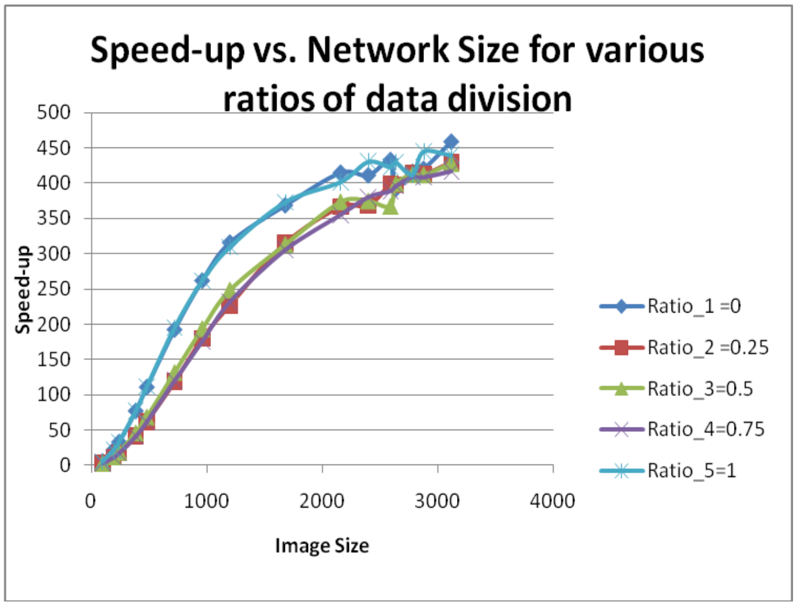

Figure 6.24: HH model: Varying Ratios: Speed-up vs Network Size; Implementation 4

Table 6.32: HH model: Varying Ratios: Speed-up values for intermediate network sizes

Image Size	**Performance for different Ratios**				
	0	**0.25**	**0.5**	**0.75**	**1**
3120x3120	459.11	429.3	428.3	416.9	439.8
2400x2400	411.4	368.1	375.1	380.23	430.41
1680x1680	368.91	314.86	313.5	305.45	373.17
960x960	261.96	179.78	194.01	175.73	261.7
480x480	110.76	61.67	68.86	61.72	111.62

Analysis of the performance for the ratios 0 and 1 in Table 6.32 indicates that the GPU devices have near similar processing power. The multi-GPU system performance for a ratio of 0.25 or 0.75 is also similar. From these results, it is inferred for these applications, multi-GPU systems do not provide additional speed-up, although they can be useful when the problem size is too large for the memory capacity of a single GPU system.

In summary, this chapter has presented and analyzed the single-GPU results for four implementations utilizing the different parallelization techniques introduced in chapter 5. It was observed that the effects of multiprocessor occupancy and block configuration size are more noticeable for highly compute-intensive models. While the optimizations *G* and *GT* are the viable memory optimization techniques for the compute-intensive models, *GS* proved to be a better technique for the lesser compute-intensive models. The best performing implementation, *Implementation 4*, was applied for the most compute-intensive Hodgkin Huxley model on the multi-GPU systems. The problem was partitioned in several ratios and evaluated. Although the devices are heterogeneous (see Table 5.1), the performance of the two devices in the multi-GPU system was found to be similar. The conclusion is multi-GPU systems are more suited for situations where the GPU device memory is insufficient to hold the network data (problem size). Equal division of the data between the GPUs will provide optimal performance. In the next chapter we present the conclusions and future work.

Chapter 7

Conclusions and Future Work

In this thesis, we were successful in implementing the level-1 neurons, the most compute intensive layer of neurons in a two-level SNN character recognition network. The neuron dynamics of the network were based on the four most biologically realistic SNN models, namely, the Izhikevich model, Wilson model, Morris Lecar model and the Hodgkin Huxley (HH) model. In contrast to the implementation in [13] which could scale only upto 5.76 million neurons, the implementations in this thesis were enabled to be scaled upto 9.7 million neurons. For the single-GPU implementation, substantial speed-ups were achieved, 576.9x for the HH model, 105.86x for Morris Lecar model, 13.34x for lesser compute intensive Wilson model, and 11.82x for the least compute intensive Izhikevich model. Speed-ups were reported to increase with network size, except for a few intermediate network sizes. It is important to note that the GPUs perform best when the applications have a significantly high Flop/Byte ratio. Optimization techniques made available by the CUDA programming model, namely, *Memory optimization techniques, Execution configuration optimization,* and *Instruction optimization* were applied to develop four successive mapping methods described in Chapter 5.

Each of the implementations was examined for the speed-up performance and parameters that could lead to performance degradation. Study of *Implementation 1* revealed that for heterogeneous systems involving problem partitioning between the host and the device, memory transfer size and frequency are the most important bottlenecks that should be minimized. For *Implementation 1*, it was also observed that the memory optimization techniques *G* and *GT* performed the best for models with significantly higher Flop/Byte ratios, whereas for the models with lower Flop/Byte ratio, *GS* is best suited. *Implementation 2* has showed significant improvement by reducing the transfer size of the firing vector, but suffered from the problem of uncoalesced accesses. These uncoalesced accesses were avoided in *Implementation 3*, which together with use of registers and fast math operations produced performance improvements over the previous two implementations. *Implementation 4* successfully reduced the global firing vector transfer frequency with the use of block vectors, hence significantly reducing the overall application time. Although a large number of uncoalesced accesses (due to character data-type for firing vector) and serialized warps (due to shared memory) were observed, reducing the memory transfer time overshadows the effects of these hazards. Another important observation is made regarding the *Execution configuration optimization.* Higher occupancy and large number of threads per block have more direct implication on the performance only for models with higher Flop/Byte ratio such as the HH model and the Morris Lecar model. The effect of the execution configuration optimization was not direct in the cases of the two less compute-intensive models, although it is observed in general that one should strive to keep the occupancy and block configuration sufficiently high to avoid memory access latencies.

The multi-GPU system was successfully used to implement the best implementation, *Implementation 4,* for the most compute intensive HH model. Although speed-ups as high as the single-GPU implementation were not observed, the problem partitioning was successful. It is inferred that for applications that may run out of GPU device memory, multi-GPU systems can be of great use and partitioning the data equally between the GPU devices can yield desirable performance. Nonetheless, a need for further studies is required with equivalent GPUs to determine true performance.

The research work in this thesis sufficiently establishes GPU accelerators as a potential candidate for large-scale, accurate neuron model simulations. With the speed-ups achieved in this research, it is inferred that a cluster of GPUs can be more effective in performing these simulations accurately and in near-real time compared to the large clusters of commodity processors. Hence, the next step will be to investigate the use of a cluster of GPU devices and develop parallelization methods for large-scale neural network simulations. Additionally, programming models for GP-GPUs, such as CUDA and OpenCL, are both gaining in popularity and a comparison of the capabilities of these developing programming models to fully exploit GPU computing performance is another interesting investigation. Finally, in this thesis we have only considered Nvidia GPUs and future studies should include a comparison with AMD/ATI GPUs.

References

1. M.A. Bhuiyan, T.M. Taha, and R. Jalasutram, "Character recognition with two spiking neural network models on multi-core architectures," *in Proceedings of IEEE Symposium on CIMSVP,* Nashville, TN, pp. 29-34, March 2009.

2. E. Izhikevich, "Which Model to Use for Cortical Spiking Neurons?" *IEEE Transactions on Neural Networks*, vol. 15(5), pp. 1063-1070, 2004.

3. A. Delorme and S.J. Thorpe, "SpikeNET: an event-driven simulation package for modeling large networks of spiking neurons," *Network-computation in neural systems*, vol. 14(4), pp. 613-627, November 2003.

4. A.R. Baig, "Spatial-temporal artificial neurons applied to online cursive handwritten character recognition," *in Proceedings of the European Symposium on Artificial Neural Networks*, pp. 561-566, April 2004.

5. T. Ichishita, R. Fujii, "Performance evaluation of a temporal sequence learning spiking neural network," *Proceedings of the 7th IEEE International Conference on Computer and Information Technology*, pp. 616-620, October 2007.

6. C. Panchev and S. Wermter, "Temporal sequence detection with spiking neurons: towards recognizing robot language instructions," *Connect. Sci.,* vol. 18, issue 1, pp.1-22, 2006.

7. C. Johansson and A. Lansner, "Towards Cortex Sized Artificial Neural Systems," *Neural Networks*, 20(1), pp. 48-61, January 2007.

8. Nene, S. A., Nayar, S. K., & Murase, H. (1996). Columbia Object Image Library (COIL-100) (No. CUCS-006-96): Columbia Automated Vision Environment.

9. R. Ananthanarayanan, S. K. Esser, H. D. Simon, and D. S. Modha, "The Cat is Out of the Bag: Cortical Simulations with 10^9 Neurons, 10^{13} Synapses," *Proceedings of SC '09*, Portland, Oregon, November 2009.

10. A. L. Hodgkin and A. F. Huxley, "A quantitative description of membrane current and application to conduction and excitation in nerve," *Journal of Physiology*, vol. 117, pp. 500-544, 1952.

11. W. Rall, "Branching dendritic trees and motoneuron membrane resistivity," *Experimental Neurology*, vol. 1, pp. 503-532, 1959.

12. J. M. Nageswaran, N. Dutt, J. L. Krichmar, A. Nicolau, A. V. Veidenbauma, "A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors," *Special issue of Neural Network, Elsevier*, vol. 22(5-6), pp. 791-800, July 2009.

13. M. A. Bhuiyan, Vivek K. Pallipuram, Melissa C. Smith, "Acceleration of Spiking Neural Networks in Emerging Multi-core and GPU architectures", to be presented in *HiCOMB 2010, a workshop in IPDPS 2010*, April 2010.

14. Simon Haykin, "Neural Networks, A Comprehensive Foundation, Second Edition."

15. A. Gupta, L. Long, "Character Recognition using Spiking Neural Networks," *in Proceedings of IJCNN*, pp. 53-58, August 2007.

16. E. M. Izhikevich, "Simple Model to Use for Cortical Spiking Neurons," *IEEE transactions on Neural Networks*, vol. 14, no. 6, pp. 1569-1572, November 2003.

17. H. R. Wilson, "Simplified dynamics of human and mammalian neocortical neurons," *J. Theor. Biol.*, vol. 200, pp. 375-388, 1999.

18. C. Morris and H. Lecar, "Voltage oscillations in the barnacle giant muscle fiber," *Biophys. J.*, vol. 35, pp. 193-213, 1981.

19. Minh Tri Do Dinh, "GPUs – Graphical Processing Units," *Vertiefungsseminar Architektur von Prozessoren*, SS 2008, Inst. of Comp. Sci., Univ. of Innsbruck, July 2008.

20. www.nvidia.com, "Technical Brief: NVIDIA GeForce 8800 GPU Architecture Overview," November 2006.

21. www.nvidia.com, “The CUDA compiler driver: NVCC”, May 2007.