Computational Modeling of Biological Neural Networks on GPUs: Strategies and Performance

Byron Galbraith
Marquette University

COMPUTATIONAL MODELING OF BIOLOGICAL NEURAL NETWORKS ON GPUS: STRATEGIES AND PERFORMANCE

by

Byron V. Galbraith

A Thesis submitted to the Faculty of the Graduate School,
Marquette University,
in Partial Fulfillment of the Requirements for
the Degree of Master of Science

Milwaukee, Wisconsin

August 2010

ABSTRACT

COMPUTATIONAL MODELING OF BIOLOGICAL NEURAL NETWORKS
ON GPUS: STRATEGIES AND PERFORMANCE

Byron V. Galbraith

Marquette University, 2010

Simulating biological neural networks is an important task for computational neuroscientists attempting to model and analyze brain activity and function. As these networks become larger and more complex, the computational power required grows significantly, often requiring the use of supercomputers or compute clusters. An emerging low-cost, highly accessible alternative to many of these resources is the Graphics Processing Unit (GPU) - specialized massively-parallel graphics hardware that has seen increasing use as a general purpose computational accelerator thanks largely due to NVIDIA's CUDA programming interface. We evaluated the relative benefits and limitations of GPU-based tools for large-scale neural network simulation and analysis, first by developing an agent-inspired spiking neural network simulator then by adapting a neural signal decoding algorithm. Under certain network configurations, the simulator was able to outperform an equivalent MPI-based parallel implementation run on a dedicated compute cluster, while the decoding algorithm implementation consistently outperformed its serial counterpart. Additionally, the GPU-based simulator was able to readily visualize network spiking activity in real-time due to the close integration with standard computer graphics APIs. The GPU was shown to provide significant performance benefits under certain circumstances while lagging behind in others. Given the complex nature of these research tasks, a hybrid strategy that combines GPU- and CPU-based approaches provides greater performance than either separately.

ACKNOWLEDGMENTS

Byron V. Galbraith

I would like to thank the following people for their contributions to the completion of this thesis. Craig Struble, my thesis advisor, enabled and supported my research goals while offering invaluable guidance and perspective. Scott Beardsley and Rong Ge, my committee members, provided the opportunities and insights that allowed me to succeed. David Herzfeld provided not only software assistance in the completion of this work, but also friendly competition that encouraged me to work harder.

I would like to extend special thanks to my wife, Karen, for her patience and support during this process.

This research was funded in part by National Science Foundation awards OCI-0923037 and CBET-0521602.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

Introduction

1.1 Problem Statement

Artificial neural networks are a class of computational tools used in such fields as pattern recognition, machine learning, and knowledge discovery. They are inspired from the emergent properties of actual neurons found in the human brain where several small independent units acting in unison can give rise to complex functionality. One classic example is the self-organizing map, a network of artificial neurons that, when trained, can be used to discover novel relationships between multidimensional data sets. The training process can be very slow dependent on the data set which may prohibit use, so developing methods to speed up training can enable larger and more complex data sets to be analyzed.

Modeling actual biological neural networks is also of significant importance in the fields of computational neuroscience and robotics. By simulating networks of biologically-based neurons, researchers hope to gain deeper understanding of brain activity and function as well as develop control devices that can mimic the brain's own fine motor control ability. As these networks become larger and more complex, the computational power required grows significantly, often requiring the use of supercomputers or compute clusters. As such, many researchers are unable to achieve the level of granularity or network complexity desired, thus inhibiting or halting progress.

An emerging low-cost, highly accessible alternative to many of these resources is the Graphics Processing Unit (GPU) - specialized massively-parallel graphics hardware that has seen increasing use as a general purpose computational accelerator thanks largely due to NVIDIA's CUDA programming interface. Scientific and clinical applications that benefit from parallelization,

such as in fluid dynamics, bioinformatics, and medical image analysis, can gain significant speedup just by adapting exiting strategies to incorporate GPU acceleration. As NVIDIA continues to invest heavily in research and marketing for advancing the GPU as a general computational tool, the GPU is an intriguing platform to target development efforts for scientific applications.

Adapting large-scale biologically-inspired neural networks to utilize GPUs is especially attractive as these classes of problems consume significant computational resources. These enhanced simulations could lead to greater understanding of neural function resulting in improved prosthetic and rehabilitation devices for patients. As GPUs are orders of magnitude cheaper than supercomputers or large compute clusters, GPU-based tools for large-scale neural simulations enable a larger pool of researchers to study these problems. However, utilizing GPUs as computational accelerators is not without drawbacks and challenges. There exists a need for solid understanding of what the GPU is able to offer the field of neural simulation as well as sound strategies for incorporating GPUs into existing and future models and tools.

1.2 Thesis Statement

Computational models of biological neural networks can be efficiently implemented on GPUs.

1.3 Contribution

The design and implementation of GPU-enabled neural network modeling tools using the CUDA platform are provided in this thesis. They were developed as follows (Figure 1.1).

- A self-organizing map implementation demonstrates the trade-offs between different arrangements of mapping threads to neurons, the inherent barrier of simulation time step, and a glimpse into concurrent network training and visualization.
- An agent-inspired spiking neural network simulation tool is designed and implemented, taking advantage of the massively parallel nature of the GPU to mimic neural behavior. The beginning of a GPU-enhanced neural network simulation tool focused on biological constraints and modalities is presented in detail.

- A neural signal decoding algorithm is successfully adapted to use the GPU as a computational accelerator for finding effective linear filter decoder weights.
- As part of the neural decoding algorithm implementation, the CUSUMMA algorithm was created to perform general matrix-matrix multiplications portably and dynamically on any CUDA-enabled GPU.

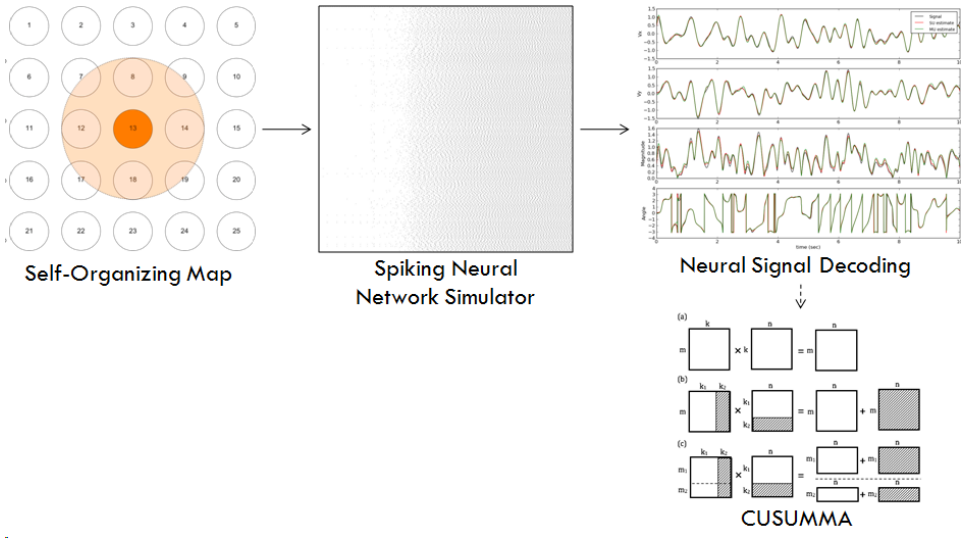

Figure 1.1: Organization of research contributions described in this thesis.

1.4 Organization

The rest of this thesis is arranged as follows. Chapter 2 provides background information relating to NVIDIA's GPU architecture, the CUDA programming model, basic neural physiology, modeling biological neural networks, and neural network simulation. The next three chapters present the creation and evaluation of GPU-based neural network modeling and analysis tools with focus on performance versus non-GPU implementations. Chapter 3 describes the performance of a self-organizing map implementation on a GPU with additional focus on the ability to visualize the network as it trains in real-time. Chapter 4 describes a GPU-based agent-inspired spiking neural network simulator. The design and implementation of this tool are detailed along with performance results compared to an MPI version and another look at visualizing real-time network activity. In Chapter 5, a neural signal decoding algorithm is adapted to use a GPU for acceleration of linear

algebra and convolution operations in order to speed up the decoding process. Concluding remarks are made in Chapter 6, where commentary on the overall success of GPU-adapted neural population simulations is offered and areas for further development are considered. Finally, several appendices provide complete code listings for some of the GPU code developed.

CHAPTER 2

Background

The work contained herein relies heavily on two key technologies: computational acceleration using graphics processing units; and the modeling and simulation of biologically-based neural networks. This chapter provides an overview of these concepts with emphasis on the specific tools employed.

2.1 Graphics Processing Units

A *graphics processing unit (GPU)* is a specialized piece of computer hardware optimized for the processing and rendering of 3D computer graphics. As this process is largely *data-parallel*, modern GPUs have been designed to be massively parallel processors that are highly efficient at performing billions of floating point operations per second (Figure 2.1). In order to accomplish this feat, GPU hardware designers departed from traditional CPU architecture.

2.1.1 GPU Architecture

At the time of writing, modern CPU's contain up to six processing cores. Each core is a general purpose processing element that supports a wide variety of instructions, while memory access is multi-tiered, employing several layers of caching. On the other hand, GPUs — specifically those produced by NVIDIA — dedicate their silicon to several multiprocessors, each containing many single-precision processing elements with limited caching and flow control. Since caching is not as important as arithmetic for graphics operations, emphasis was instead placed on increasing the memory bandwidth to the GPU's DRAM (Figure 2.2). In comparison to the six

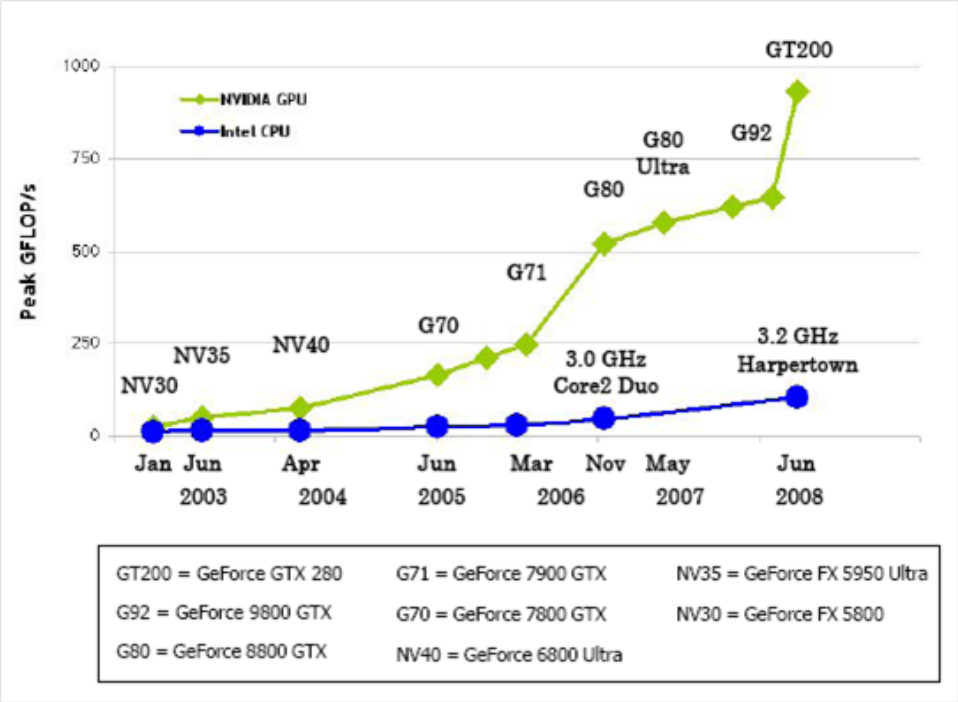

Figure 2.1: Comparison of peak GFLOPS for NVIDIA GPUs and Intel CPUs [46].

cores of a current high-end CPU, a high-end GPU has 240 cores, and, unlike the limited single or double thread concurrency model of current CPUs, the GPU can manage and efficiently schedule hundreds of thousands of light-weight threads across its cores. Figure 2.3 provides a simple schematic representing the differences in general architectures between the two platforms.

Looking closer at the NVIDIA GPU architecture (Figure 2.4), it is seen to be made up of a number of *multiprocessors*. In current architectures, each multiprocessor contains eight single-precision floating point operation processing elements with a shared instruction unit. Not shown here is the single double-precision processing unit also present on newer NVIDIA GPUs. There are registers for each of the processors used to handle per-thread memory requirements, as well as a slightly slower shared memory buffer that allows threads on the same multiprocessor to communicate. Additionally, each multiprocessor has a limited amount of read-only cache for constant parameters and texture data. In all cases but the registers, the programmer must make explicit use of of these other memories. Finally, there is the GPU DRAM or device memory that any thread can access. All in all there are 8kb or 16kb total registers, 16kb shared memory, and 64kb constant memory per multiprocessor, while the amount of DRAM can range from 256Mb in

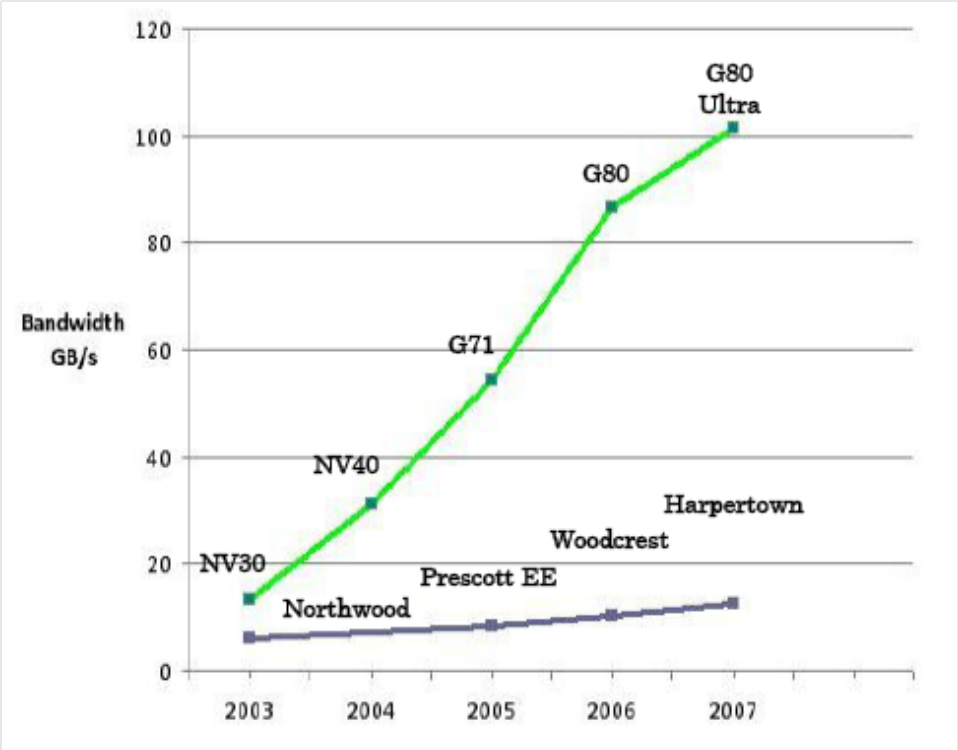

Figure 2.2: Comparison of peak DRAM bandwidth for NVIDIA GPUs and Intel CPUs [46].

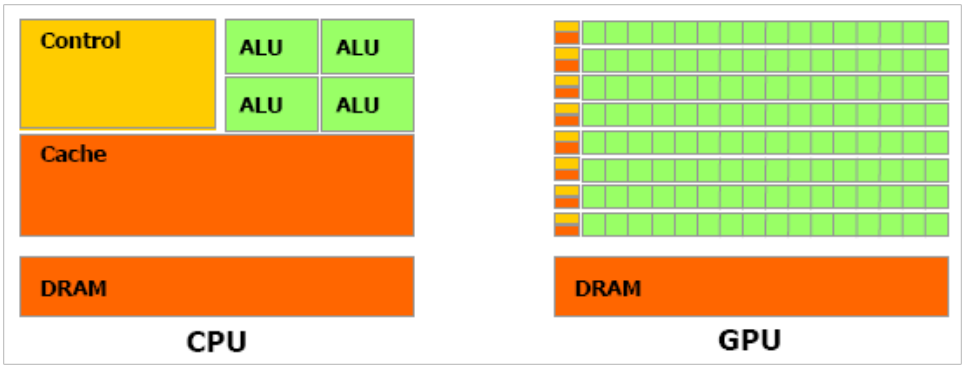

Figure 2.3: Relative devotion of transistors in a 4-core CPU vs. 128-core GPU [46].

low end and laptop GPUs to 4Gb in dedicated compute devices. Texture memory is slightly different, in that the texture data is stored in DRAM, but must be specially defined at compile time in order to achieve the performance benefits due to caching.

2.1.2 General Purpose Computing on GPUs

While parallelization libraries such as MPI [28] and OpenMP [15] allow developers to utilize each core simultaneously, the generation of these threads is computationally expensive.

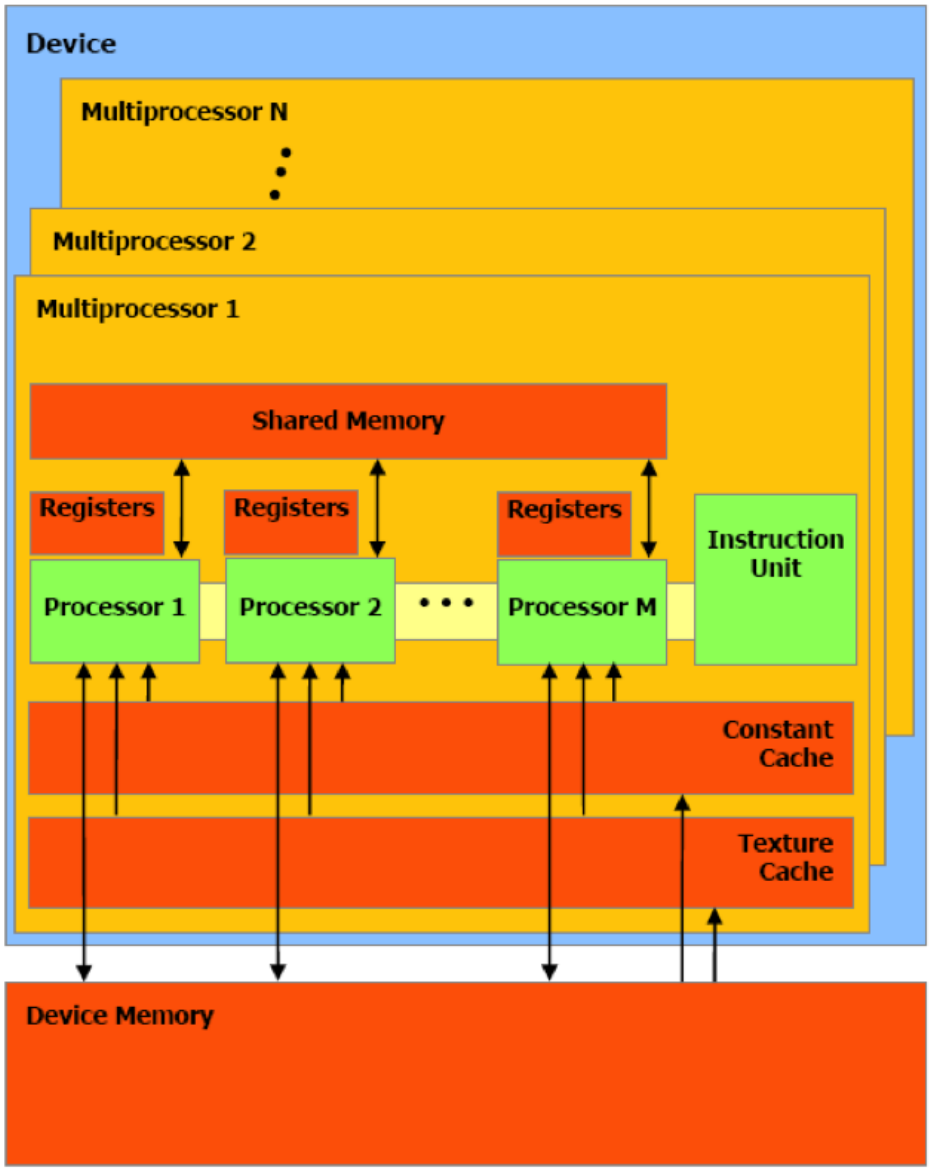

Figure 2.4: Layout schematic for the NVIDIA G80 GPU architecture [46].

Additionally, the number of concurrently active threads is limited to the number of cores available. Attempting to provision more threads than cores can cause performance degradation as threads must wait until cores become available. It seems natural then that researchers would want to harness the highly optimized, massively parallel GPU for non-graphics applications. Early efforts to do so were termed *General Purpose Computing on GPUs (GPGPU)* and were met with significant challenges. Until the introduction of NVIDIA's CUDA (Section 2.1.3) in 2006, the only option for GPGPU was to reformulate the scientific problem into OpenGL or DirectX, low-level

graphics APIs (Application Programming Interface) that did not support many of the common constructs found in general purpose programming languages. As this required both a mastery of the graphics APIs as well as a thorough understanding of the application being ported, very few people were successful.

2.1.3 CUDA

In its G80 class of GPUs, NVIDIA dedicated extra silicon to a programmable interface that allowed developers to use the GPU as a general purpose stream processor. *Stream processing* is related to the SIMD (Single Instruction Multiple Data) programming paradigm where a single kernel function is applied to multiple data elements in parallel. Unlike SIMD, NVIDIA's CUDA (Compute Unified Device Architecture) does not guarantee explicit synchronization or execution order between processing elements. They call their model SIMT (Single Instruction Multiple Thread) as the *thread* is the basic processing element from the programmer's perspective.

Thread Model

The CUDA SIMT paradigm achieves massive parallelism through the scheduling and execution of hundreds to millions of threads. As this number of threads greatly exceeds the number of physical processing elements, the threads are arranged into a two-tiered structure to facilitate scheduling. The basic unit of organization is a *thread block* (Figure 2.5 bottom), a 3D array of no more than 512 threads with the maximum allowable dimensions in the x, y, and z coordinates being 512, 512, and 64, respectively. The thread scheduler will physically keep a thread block together by assigning it to an available multiprocessor. Thread blocks are in turn organized into a 2D array called a *thread grid* (Figure 2.5 top), which has maximum allowable dimensions of 65535 x 65535. All thread blocks in a grid share the same dimensions. As only one thread grid can be specified per kernel launch, the maximum number of threads that can be requested at a time is 2.2 trillion.

Whether requesting 2.2 trillion threads or just one, the actual execution of threads occurs in 32-thread bundles called *warps*. A warp executes in SIMD fashion, so all 32 threads execute simultaneously and finish at the same time. A multiprocessor can execute up to 24 or 32 warps at any given time for a total of 768 or 1024 threads. Since a thread block is limited to 512 threads, it

is possible for multiple blocks to occupy a single multiprocessor. Thread blocks cannot share processing elements, however, so the maximum number of thread blocks per multiprocessor is eight.

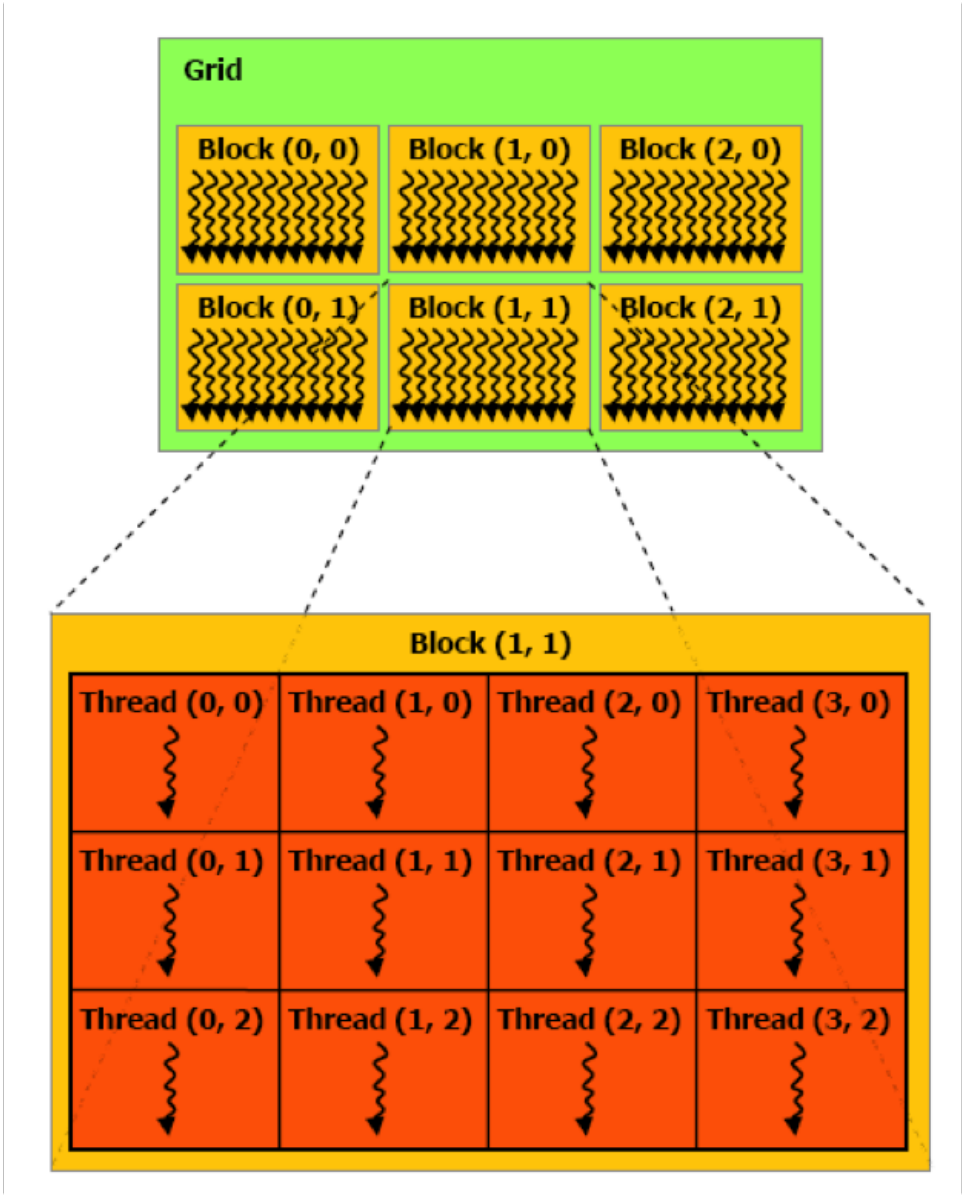

Figure 2.5: Illustration depicting the two-tier CUDA thread hierarchy [46].

Programming Model

One of the attractive features of CUDA for GPGPU is the programming API provided by NVIDIA. Instead of having to deal with low-level graphics APIs, NVIDIA added a few key

extensions to ANSI C, allowing programmers to use a familiar environment to create GPU-based applications which could then be compiled using the provided *nvcc* compiler. This tool is actually a wrapper around GCC on the Linux and Mac OS X platforms and the Visual C++ compiler on the Windows platform. In addition to processing the CUDA-specific extensions, it generates the PTX assembly code used by the GPU for device specific kernels and functions, while passing the standard C/C++ code on to the respective local compiler.

The GPU is incapable of handling general program flow by itself, so every CUDA program is a CPU, or *host*, application that must explicitly invoke kernel execution and data transfer to and from the GPU, or *device*. Typical execution flow is depicted in Figure 2.6, where a serial process sets up the application state and prepares data for processing before launching a kernel with a defined thread grid. It's important to note that kernel launching is asynchronous, so control immediately returns to the host process without waiting for all the kernel threads to finish. Certain operations, such as copying data from the device back to the host, block until all threads have completed execution.

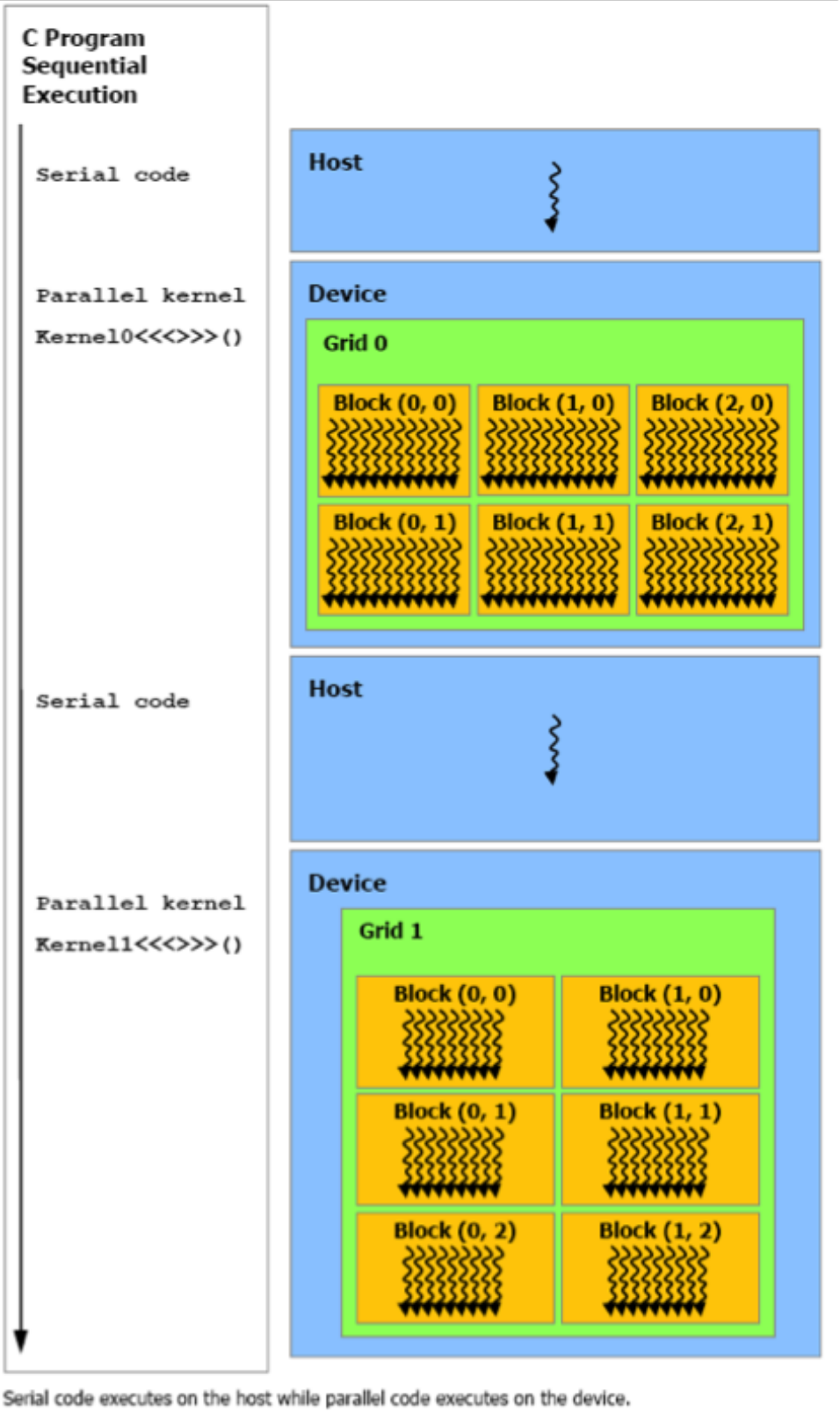

Figure 2.6: Typical CUDA program switches between serial host (CPU) and parallel device (GPU) execution [46].

CUDA supports multiple GPUs if present, though the programmer has to manually switch contexts between GPUs as part of the application. The programmer can also decide which GPU to use at runtime based on the reported compute capabilities. Examples of device query output can be seen for a GeForce GTX 260 (Figure 2.7) and a GeForce 8200 (Fig. 2.8).

```
Device 0: "GeForce GTX 260"
  CUDA Driver Version:                           3.0
  CUDA Runtime Version:                          2.30
  CUDA Capability Major revision number:         1
  CUDA Capability Minor revision number:         3
  Total amount of global memory:                 939327488 bytes
  Number of multiprocessors:                     27
  Number of cores:                               216
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       16384 bytes
  Total number of registers available per block: 16384
  Warp size:                                     32
  Maximum number of threads per block:           512
  Maximum sizes of each dimension of a block:    512 x 512 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             256 bytes
  Clock rate:                                    1.35 GHz
  Concurrent copy and execution:                 Yes
  Run time limit on kernels:                     No
  Integrated:                                    No
  Support host page-locked memory mapping:       Yes
  Compute mode:                                  Default (multiple host threads can use
this device simultaneously)
```

Figure 2.7: Results of querying a GeForce GTX 260 (GT200 series) GPU.

```
Device 1: "GeForce 8200"
  CUDA Driver Version:                           3.0
  CUDA Runtime Version:                          2.30
  CUDA Capability Major revision number:         1
  CUDA Capability Minor revision number:         1
  Total amount of global memory:                 265617408 bytes
  Number of multiprocessors:                     1
  Number of cores:                               8
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       16384 bytes
  Total number of registers available per block: 8192
  Warp size:                                     32
  Maximum number of threads per block:           512
  Maximum sizes of each dimension of a block:    512 x 512 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             256 bytes
  Clock rate:                                    1.20 GHz
  Concurrent copy and execution:                 No
  Run time limit on kernels:                     No
  Integrated:                                    Yes
  Support host page-locked memory mapping:       Yes
  Compute mode:                                  Default (multiple host threads can use
this device simultaneously)
```

Figure 2.8: Results of querying a GeForce 8200 (G80 series) GPU.

For a simple Hello World example CUDA program, see Listing A.1. A more in-depth presentation of CUDA can can be found in NVIDIA's CUDA Programming Guide [46], while the textbook Programming Massively Parallel Processors: A Hands-on Approach, by Kirk and Hwu, provides a solid introduction to CUDA programming with exercises and case studies [37].

Libraries

Along with the standard Runtime API to CUDA, NVIDIA also provides a Driver API. The main difference is that the Driver API requires more context initialization for kernel launches with the potential for increased performance. It also supports Just-In-Time (JIT) compilation of kernels, allowing developers to invoke dynamic kernel functions from other environments, such as Java [1] and Python [38]. At a much higher level, NVIDIA also includes the CUFFT [45] and CUBLAS [44] libraries as standalone tools for GPU-enhanced FFT and linear algebra operations. These require even less expertise on the programmer's part as the interfaces are designed to mimic the FFTW [23] and BLAS [9] routine calls, enabling them to be dropped in-place to existing code with minimal changes required.

2.1.4 OpenCL

Worth briefly mentioning, OpenCL [42] is a standardized language overseen by the Khronos Group (of OpenGL fame) for harnessing a variety of computational accelerators including GPUs, IBM's Cell Broadband Engine, digital signal processors, and other special purpose hardware. The first draft specification was released in 2009, with vendor driver implementations following shortly thereafter. As CUDA is specific to NVIDIA GPUs, OpenCL offers a platform-independent alternative to developing GPU applications. However, OpenCL has adopted many of the extensions introduced by CUDA into its programming model, so transitioning from CUDA to OpenCL should be fairly straightforward if so desired.

2.2 Modeling Neural Networks

The animal nervous system consists of a complex network of intercommunicating cells called *neurons*. The number of neurons in an organism can range from a handful in invertebrates

up to the tens of billions found in the human brain. Attempting to understand and model how neural behavior gives rise to the phenomena of cognition, perception, and memory is a subject that has driven researchers for decades.

2.2.1 The Neuron

A neuron consists of three main components: the cell body, or *soma*, the *dendrites*, and the *axon*. In engineering terms, the dendrites handle the input, the soma integrates the signals coming from the dendrites, and the axon transmits any output signal (Figure 2.9). While certain neurons transduce environmental information or deliver control signals to other tissues such as muscle, the large majority of neurons communicate with other neurons. This neural communication occurs via chemo-electrical signaling in the form of modulated cell membrane potentials. Some neurons use graded changes in potential to communicate information, while most use a series of sharp, rapid changes called *action potentials*, or spikes. These spikes are generated at the soma and travel down the axon to the axon terminal, where the sudden change in potential causes the release of neurotransmitter molecules into the space between neurons called the *synapse*. These molecules bind to receptors in the afferent neuron's dendrites resulting in a *post-synaptic current (PSC)* that represents the input to that neuron. Through adaptive addition and subtraction of the dendrite receptors, effective weighting of the inputs from various neurons is achieved.

For an excellent resource on neurons and neural physiology, see Principles of Neural Science, by Kandel, Schwartz, and Jessell [36].

2.2.2 Neural Models

The level of complexity undertaken in modeling neurons is almost entirely dependent on the type of question the modeler is asking. For instance, if simple spiking behavior is of interest, then all that is needed from a model is a way to aggregate inputs and emit spikes over time thus allowing for the the neuron to be treated largely as a mathematical construct. For more in-depth studies of intra-cell behavior, this simple representation is insufficient, so morphological elements such as ion channels and the 3D structure of the cell are incorporated. In general, the researcher

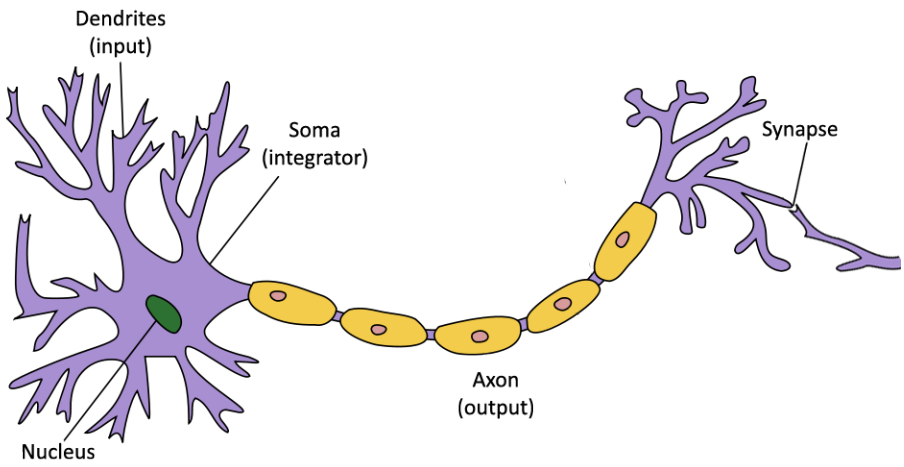

Figure 2.9: The key components of a neuron. Signals in the form of ionic currents travel from the dendrites to the soma, increasing the cell membrane potential. If the soma reaches threshold, an action potential is generated and sent down the axon. The action potential event is relayed to the connecting neuron at the synapse. Image modified from [35]. Licensed under CC-BY-SA-3.0.

typically needs to identify the models for neural morphology, dynamics, and communication, as well as the framework in which to evaluate neural response.

2.2.3 Neural Morphology

The representation of neuron morphology is divided into either single or multiple compartment models. Single compartment models, or *point neurons*, treat the entire neuron as a single entity from a system state perspective. For instance, while cell membrane potential will differ across the body of a real neuron depending on the structure and location, a point neuron assumes the membrane potential is uniform. All the neural models in this work are point neurons. Multi-compartment models, on the other hand, break the neuron into individual sections, with the number and shape of each compartment dependent on the type of neuron being modeled and the level of granularity required.

2.2.4 Neural Dynamics

The way a neuron responds to the change in input current over time is expressed in its dynamical behavior model. There exist several models for neural spiking dynamics, each with their own strengths and weaknesses. Three of the common classes of models are described below.

Leaky Integrate-and-Fire

The *Leaky Integrate-and-Fire (LIF)* neuron model [2] is one of the oldest and most basic representations of neural activity. It is computationally straightforward, consisting of the single differential equation

$$\frac{dV(t)}{dt} = -\frac{1}{\tau^{RC}}(V(t) - J_M(t)R), \tag{2.1}$$

where $V(t)$ is the membrane voltage at time t, τ^{RC} is the RC time constant, R is the leak resistance, C is the capacitance of the cell membrane, and $J_M(t)$ is the membrane current at time t. When $V(t)$ reaches some predetermined threshold value, a δ-function representing the emission of a spike is added to the temporal response of the neuron. After the spike occurs, the membrane voltage is reset followed by a short refractory period, typically on the order of a few milliseconds, where no additional spiking can occur.

LIF neurons can approximate the spiking behavior of neurons only to the level of the temporal occurrence of action potentials. The spikes themselves have no shape as they are represented by a δ-function, and the neurons are stereotyped so that no adaptation to certain input patterns is possible. Extensions to the LIF model exist, such as the Adaptive Exponential LIF model [12], that attempt to retain the simplicity of the original while adding additional representative features.

Hodgkin-Huxley

While the LIF model neuron can represent simple spiking behavior, the spikes do not arise naturally from the model and instead are simply added when needed. The *Hodgkin-Huxley (HH)* model [33], on the other hand, does give rise to the observed rapid depolarization and subsequent hyperpolarization seen with real action potentials. It accomplishes this by modeling at the level of ion channel kinetics. While this model has had great success in characterizing the behavior of a single neuron, it is computationally expensive due to the number of simultaneous differential equations that must be evaluated. It also has a large number of system parameters that need to be specified for the type and environment of the neuron being modeled which must be determined experimentally. As such, it is generally not used when attempting to model large-scale networks.

Phenomenological

Phenomenological neural models try to combine the success of the HH model for representing a large variety of spiking dynamics with the computational efficiency of the LIF model. Several models have been developed, such as the θ-neuron [22] and the Izhikevich model [34]. The latter has seen use in very large-scale simulations [3] as it is able to represent a wide variety of neural spiking behaviors while maintaining a quadratic term as the most computationally expensive element whereas the θ-neuron model requires trigonometric functions while those closer to the HH model still rely on exponentials.

While phenomenological models are able to represent complex spiking behavior cheaply, they do this at the cost of not having any physiological basis for the parameters and terms in the model equations, the former generally obtained analytically through sweeps of the parameter space.

2.2.5 Neural Communication

When modeling connected networks of neurons, synaptic communication must be modeled in addition to neural spiking dynamics. This too can be done at varying degrees of detail, with synaptic strength representations ranging from simple weights represented as scalar values to multiple ion channel models, each with their own dynamics. The particular synaptic model used in the simulations presented later is based on PSCs.

As mentioned in Section 2.2.1, PSCs are generated by neurotransmitters in the synaptic cleft binding to receptors in the membrane of the neuron. The neurotransmitters do not immediately bind and then disappear after an action potential event, rather they linger in the synapse until they are taken back up into the axon terminal of the efferent neuron or diffuse elsewhere. This allows the PSC to rise or decay relative the occurrence of spikes over time, creating a concept of spike history at the dendrite. Computationally, this is achieved by convolving the spikes with a linear filter given by

$$h_{PSC}(t) = t^n e^{-t/\tau_{syn}}, \tag{2.2}$$

where t is time, n is the order of the filter, and τ_{syn} is the synaptic time constant. From a signal processing standpoint this is an example of a causal filter. Non-causal filters, which may be more optimal, do not exist in real-time systems, thus making the choice of a causal filter more appropriate for modeling neural communication in simulation.

2.2.6 Analysis Framework

Not only are there several ways of constructing a model environment for simulating neural activity, there are also a myriad ways to record and analyze that activity. The choice of analysis framework is, like the the physiological model under evaluation, dependent on the research question and the way the neurons are expected to represent information.

Information Representation

Firing Rate Some types of neurons encode information in the frequency of action potentials emitted over a given time window. In these cases, the *firing rate* (spikes per second) of the neuron is more interesting than the individual spike events, so only the rate is determined and stored. A classic example of where this occurs in nature was demonstrated by Georgopoulos et al. [25] in showing that certain neurons fired more rapidly than others given a particular directed arm reach event.

The rate-based framework for neural activity is well established, but does have its weaknesses. Any information encoded in the spacing between spikes or the shape of the action potentials is lost. The communication between neurons is also reduced in sophistication, allowing for only a small range of strategies, such as simple weighting.

Temporal Spiking Instead of averaging spikes over time, the individual spiking events themselves can be recorded and analyzed. This approach gives a much smaller level of granularity and is easier to manage computationally as no additional averaging step is required. However, if the information encoded by the neurons is indeed rate-based, it will be more difficult to identify. Spike-based frameworks are suited to simulations where the shape pr timing of the neural response is important as well as when detailed synaptic communication is required.

Information Decoding

Linear Filtering One method for decoding the information represented by a neural population is to use linear filtering, a common signal processing technique. The linear filter approach scales and combines the time-varying signals of the neural responses using a linear transform defined by a vector of decoding weights. To reconstruct the signal of interest, the decoding weights are simply multiplied against the response signal. Finding the optimal decoding weights can be very difficult, however.

Bayesian Approaches Another popular technique is to use Bayesian analysis to conduct probabilistic approaches to evaluating the neural code. Generally, these methods attempt to determine the information represented by the neurons by evaluating the probability of observing a certain firing rates under certain conditions. Bayesian analysis has proven successful [20], but suffers from computational complexity making in unsuitable for real-time applications.

2.3 Neural Network Simulation

As indicated previously, the type and range of models employed by a researcher is dependent on the question under study. Most of the broad areas of interest have corresponding software tools that exist to enable ease of creation and simulation of models. For instance, NEURON [32] and GENESIS [11] are simulation packages geared toward studying models of complicated neuron morphology while NEST [18] and Brian [27] are more focused on large scale point-neuron network dynamics. All of these tools are CPU-based, and, at present, do not incorporate any form of GPU-based acceleration.

These are only a small sample of the many simulation tools that exist (see [13] for a review), and, while they enable researchers to focus more on modeling and less on technical implementation, there still exists a certain amount of challenge in configuring the tools and expressing the desired model in a way that the tool requires. In an attempt to standardize and simplify model definitions, tools are actively being developed such as PyNN [17], a Python-based framework for interfacing with several popular simulation packages, and NeuroML [26], an XML-based model and network specification language.

2.3.1 Simulation Process

The general process followed by most simulation tools is to evaluate the neural models' required differential equations, usually through the use of an iterative solver such Runge-Kutta. These evaluations happen in either a time- or event-driven fashion [13]. In the time-driven scenario, the neurons (or compartments of the neurons) are evaluated in synchronous lock-step with a single simulation-wide clock. It is possible for somewhat asynchronous evaluation, especially in parallel environments, though the overall simulation process would still need a way to track where in simulation time all the elements were.

For event-driven environments, each neuron's state is only evaluated when an event such as a spike is sent to it. This allows for a much more asynchronous execution of the simulation. Agent-based models [10] are an example of a purely event-driven simulation, where each neuron is acting as an independent process tending to its own internal state and communication is handled in asynchronously. This is the most akin to neurons in the brain, but also the most complicated and resource intensive to simulate for any significantly-sized neural population.

2.3.2 Simulating Neural Networks on the GPU

The GPU provides an intriguing platform for neural network simulation because its massive parallelization and shared memory model allow it to overcome many of the computational bottlenecks of the event-driven model while still maintaining large population sizes. It is no surprise, then, that previous work has been done in this area.

One of the earliest attempts was by Bernhard and Keriven [8], who used the GPGPU approach to demonstrate a simple spiking neural network application on a GPU. Recently, Nageswaran et al. developed a spiking neural network simulator on an NVIDIA GPU that can handle up to 10 million synaptic connections between 100 thousand neurons at near real-time [43]. This simulation tool was developed with an eye toward maximizing computational efficiency in order to maximize the size of the neural populations simulated. It accomplishes this by using Izhikevich model [34] neurons and overlapping neural and synaptic updates within thread blocks. The approach taken in Chapter 4 varies primarily from this model in that it uses a more physiologically-driven agent-inspired model for handling neural updates. While this doesn't

achieve the same level of performance for the simple populations described in [43], it was designed for a greater range of flexibility in customizing the simulation environment.

CHAPTER 3

Self-Organizing Maps

One of the classic neural network models, the Self-Organizing Map (SOM), is used for multi-dimensional data analysis and visualization. Also referred to as a Kohonen network, the SOM was originally described by Teuvo Kohonen in 1982 [39] and is built on two key concepts: a competitive, or *winner-take-all* neuron response to an input and a topological arrangement of neurons. The SOM has found success in clustering, dimensionality reduction, and other data mining applications.

The work presented in this chapter describes an implementation of the SOM training algorithm on a GPU using CUDA. This was conducted as a proof of concept exercise to evaluate how neural networks could be implemented and potentially visualized using the GPU as an accelerator. First the SOM training algorithm is described, followed by the strategy used in implementing the algorithm on the GPU. Finally, performance results comparing the GPU version to a similar, CPU-only implementation are provided and analyzed with additional discussion regarding visualization efforts.

3.1 SOM Algorithm

In order to apply the SOM to a data set, the configuration of the neural network must defined. This involves decisions regarding the number of neurons in the network as well as their topographical layout. Once established, the network is trained on the data set by presenting one data point at a time as input. Once training is complete, the results can be analyzed by searching

for common elements in the data points associated with a particular neuron. These steps are described in more detail in the following sections.

3.1.1 Network Configuration

The first step required is to determine the topological arrangement of neurons. Frequently this is in a 2-dimensional grid, though 3-dimensional arrangements such as cubes, tori, or other custom shapes are also possible. The selection of topology is based on assumptions about the potential relationships within the data as well as convenience in representing and visualizing the resulting network. The number of neurons in the grid should also be chosen to allow enough distinct groupings to appear, as too few causes most data points to lump together, while too many divides each data point into its own group, with both cases failing to produce meaningful or interesting associations.

Each neuron is assigned a *weight*, or value, that represents its relationship to the data set under study. This weight takes the form of a multi-dimensional vector that has as many elements as the data. For instance, if training on genomic sequence data, the neurons would have weight vectors long enough to account for each base pair. These weights are initialized to random values so as not to bias the training process.

3.1.2 Training

To train the network, the SOM uses a machine learning algorithm. There are two primary approaches to machine learning: supervised and unsupervised. In supervised learning, the expected output is provided along with the input to evaluate the success of the network. Learning is accomplished by adjusting weights to minimize the error between the actual and expected output. In unsupervised learning, no expected output is provided. Instead of adjusting weights based on the error, the weights are updated to closer match the input. The SOM uses an unsupervised learning approach.

In addition to being unsupervised, the learning process is also *competitive*. Competitive learning means that only certain neurons in the network actually modify their weights in response to an input. In this particular case, the neuron that most closely matches the input is designated the

winner, or *Best Matching Unit (BMU)* (Figure 3.1). This neuron, and, to some extent, its neighbors, are the only neurons to train their weights against the input.

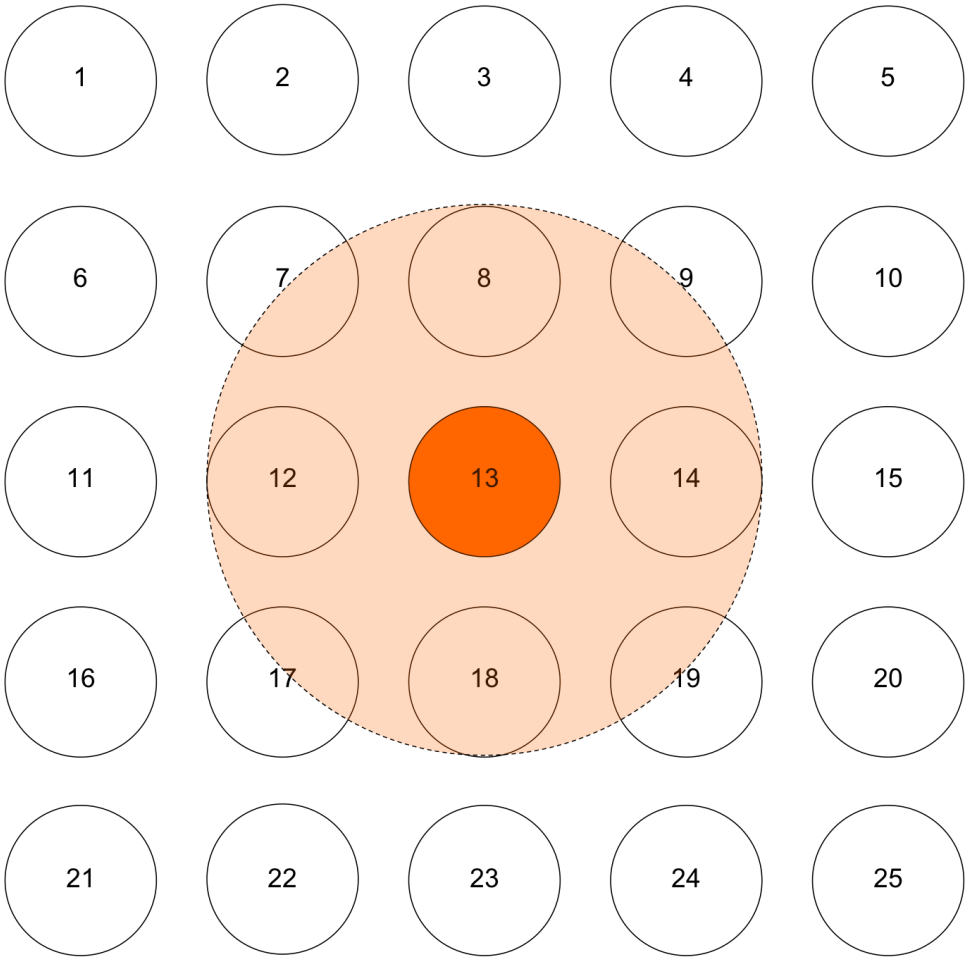

Figure 3.1: A 25 neuron SOM configured in a 5x5 grid. The orange neuron represents the BMU, while the shaded area around it is the neighborhood of effect.

The formal training algorithm is presented in Algorithm 1. In this process, p is one data sample from the data set and α is the *learning rate*. The learning rate is typically expressed as a percentage and controls how strongly the neuron weight is adjusted toward the input value. A high learning rate may result in instability in that neurons will be overly sensitive to each individual input. A low learning rate, on the other hand, may never satisfactorily train the network. A typical value for the learning rate is 5%. The weight of neuron n is represented by $w(n)$, while $d(w(n), p)$ is the distance measure of the input to the neuron's weights. This comparison can be any kind of distance or similarity measurement, though a common choice is Euclidean distance. Finally, $N(n, bmu)$ is the *neighborhood scaling function*, an additional scaling factor applied to the

learning rate of each neuron dependent on how close it is to the BMU in the network topology. Examples of neighborhood functions are rectangular windows (all neurons within a certain distance get full learning rate while the rest have no change), triangular or Guassian curves (the farther away from the BMU, the less influence the input exerts), and the Mexican Hat curve (like the Guassian, only the curve can go negative, so neurons a certain distance away from the BMU have a negative learning rate).

Each data element is presented to the SOM for training one at a time. Often several iterations of training are conducted, with the learning rate or neighborhood size decreasing after each full iteration until the SOM has reached equilibrium or a predefined maximal number of iterations have passed.

Algorithm 1 SOM Algorithm

```
1:  procedure SOM(p, α)
2:      for all n do
3:          d_p = d(w(n), p)
4:          if d_p < d_min then
5:              d_min ← d_p
6:              bmu ← n
7:          end if
8:      end for
9:      for all n do
10:         w(n) ← w(n) + N(n, bmu) * α * (p − w(n))          ▷ Kohonen rule
11:     end for
12: end procedure
```

3.1.3 Analysis

Once the SOM has been sufficiently trained, the researcher can analyze and review the results. This is frequently accomplished through some form of visualization, where it is possible to discern which data elements are aligned with each neuron. In some situations, simple color coding is capable of conveying a clear picture of how certain classes of data elements align, while interactive interfaces or additional processing is needed to extract the groupings from the SOM in order to determine if any new knowledge can be gained.

```
__global__ void
findDistance(float *dist, float3 *som, float3 p)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  float x = som[idx].x - p.x;
  float y = som[idx].y - p.y;
  float z = som[idx].z - p.z;

  dist[idx] = x*x + y*y + z*z;
}
```

Listing 3.1: `findDistance` kernel definition

3.2 GPU Approach

The SOM contains several properties that make it an attractive neural network to implement on a GPU. First, operations required by the training algorithm are data parallel, making it an ideal fit for the massive parallelism of the GPU. Second, the visualization aspect of SOM analysis aligns directly with the GPU's primary purpose. Whereas existing SOM implementations [49] use 2D visualizations, 3D views could be created with the ability to rotate and zoom within the structure for a more interactive experience.

The first step in parallelizing the SOM algorithm with CUDA was to identify where the dependent steps were. As each training step must happen sequentially, the only places where parallelism was possible was a nearest neighbor search used to identify the BMU and the updating of the neuron weights.

3.2.1 BMU Identification

Searching for the nearest neighbor to a point in a given space has been studied extensively [4, 52, 6]. For this implementation, neurons were arranged in a 2D square grid configuration and Euclidean distance was used to determine closeness of the neurons to the input. The neurons were treated as a single 1D array of N-element vectors, with each thread calculated the distance between the input and a neuron then stored that value in an intermediate buffer (Listing 3.1). A parallel minimization reduction operation was performed on the distance buffer to identify the ID of the neuron with the closest distance and a combination of division and modulus operations were used to determine the respective x and y coordinates of the BMU in the grid (Listing 3.2).

```
min = cublasIsamin(n, d_dist, 1);
bmu = make_uint2(min % w, min / w);
```

Listing 3.2: Determining BMU from distance calculations

```
__global__ void
updateWeights(float3 *som, uint2 bmu, float3 p, float radius, float alpha, int
    width)
{
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;
  int idx = y * width + x;

  int u = x - bmu.x;
  int v = y - bmu.y;
  int d = u*u + v*v;

  if(d < radius) {
    som[idx].x += alpha*(p.x - som[idx].x);
    som[idx].y += alpha*(p.y - som[idx].y);
    som[idx].z += alpha*(p.z - som[idx].z);
  }
}
```

Listing 3.3: `updateWeights` kernel definition

3.2.2 Update Neighborhood

Once the BMU has been identified, the bounding box surrounding the affected neural neighborhood was identified and used to quickly reduce the number of neurons that needed to be evaluated. Each thread determined the Euclidean distance of a neuron to the BMU and adjusted the values accordingly using a step function and a fixed learning rate (Listing 3.3).

3.3 Results

A 24-bit bitmap image (Figure 3.2) was chosen as a data source, with each pixel representing a single 3-dimensional vector input. Pixels were presented to the SOM one at a time, starting at the bottom-left of the image and progressing in a left-to-right, bottom-to-top fashion. The learning rate was fixed at 0.05 and a rectangular windows of width 5 was used to determine neighborhood scaling.

The GPU performed worse than an equivalent serial CPU implementation for small grids then eventually achieved up to a 2.5x speedup for very large grids (Figure 3.3). This is not

Figure 3.2: 2400 x 1800 pixel bitmap image [47] used as the input data source. Licensed under CC-BY-NC-2.0.

unexpected, as the GPU incurs additional time overhead in both data transfer to and from the device as well as kernel function invocation. For such a relatively straightforward algorithm, the data set needs to be significantly large for the GPU's parallelism to achieve noticeable gain.

3.4 Discussion

As each input data point has to be presented one at a time, the potential for performance gains is confined to the two steps of the training algorithm. For very large data sets, such as several million pixel images, tiny performance gains in a single iteration translate to significant benefit over the course of a complete run. As there were relatively only small number of neurons that needed to be evaluated during the update phase, the GPU offered little in the way of benefit over the CPU. Where the GPU managed to outperform the CPU was in identifying the BMU, though that gain was small only when compared to a naive CPU implementation of a nearest neighbor search i.e. iterating through all points and keeping track of the one with the shortest computed distance.

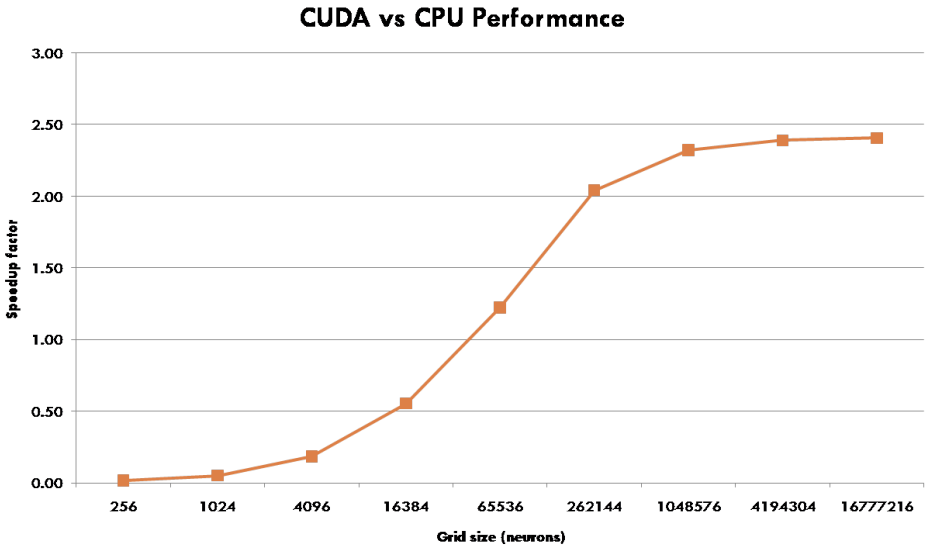

Figure 3.3: Performance of the GPU version versus the CPU version of SOM training for a single iteration. Speedup is defined as GPU/CPU.

There are much more efficient ways to do multi-dimensional searches, such as using kd-trees [7]. The main drawback of the kd-tree data structure is that it is expensive to generate. This is fine for multiple searches against a static data set, but the SOM neurons update every time step and in what would be many disparate locations in the tree until well into the training. Some work had been done relating to ray-tracing on the GPU where kd-trees were generated for each frame of an animation [53], but the times reported, while encouraging for real-time video rendering, were too slow to be useful for SOMs. Splay trees [48] and R-trees [29] also seemed promising, but neither presented an obvious parallel implementation that effectively maintained performance across multiple updates to the tree at each iteration.

3.4.1 Visualization

Because of their topographical nature, SOMs can be visualized in a number of ways. One such approach is to translate the weights of the trained SOM into RGB values and present the SOM as a bitmap. This allows a quick view into how the SOM was clustered and may provide some insight into either the input data or the SOM itself. Bitmap representations of the SOM state at various points during the training (Figure 3.4) as well as at completion (Figure 3.5) were captured. Given that the source data are also pixels, the SOM is simply clustering the palette of the

original image into blobs whose general shape and location are dependent on the psuedorandom number generator seed used to initialize the network. While this particular example does not provide much useful information, it does present an interesting application for computer-generated art that may be worth future exploration.

Additionally, one of the attractive features of GPUs for neural networks is that the integration with standard computer graphics APIs such as OpenGL [51] allows network activity to be visualized in real-time. The SOM implementation was incorporated into a simple OpenGL application (Appendix B) that represented the SOM as an image on screen (Figure 3.6). Updates to the neuron values could be observed as slight changes to the image as the pixel RBG values were updated. While of limited value for the actual purpose of the SOM, it does demonstrate the ability to visualize the evolution of a neural network which may provide additional insight when applied to more realistic neural models.

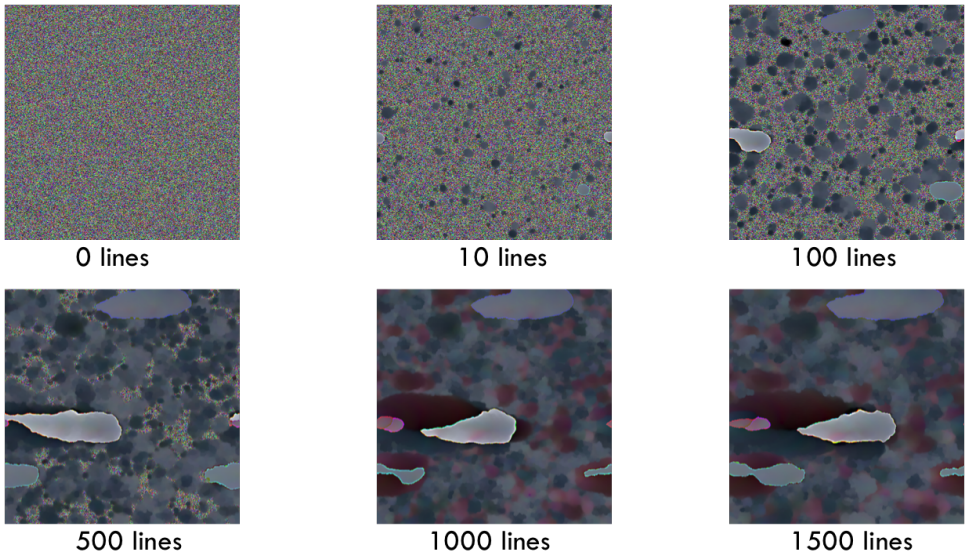

Figure 3.4: State of the SOM as rendered by a bitmap image at various stages in training. One line = 1800 inputs.

3.5 Conclusion

Implementing and visualizing the SOM neural network on the GPU was a beneficial and enlightening exercise. Valuable experience was gained by programming in the CUDA environment, and many of the realities faced in attempting to parallelize applications were

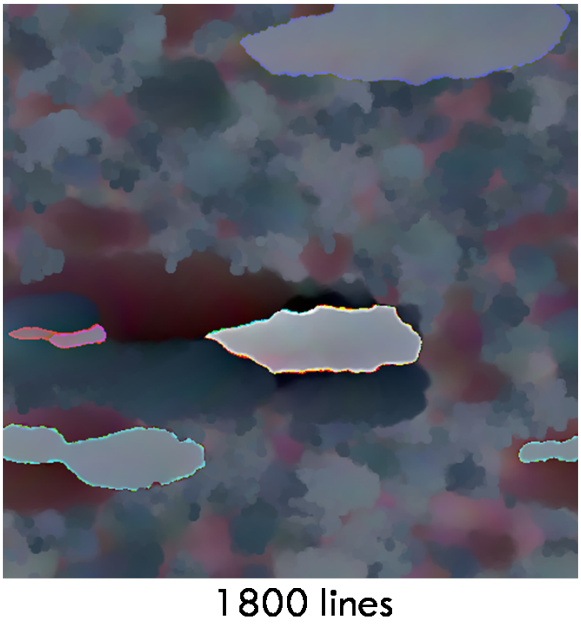

Figure 3.5: Final state of the SOM after all pixels had been presented.

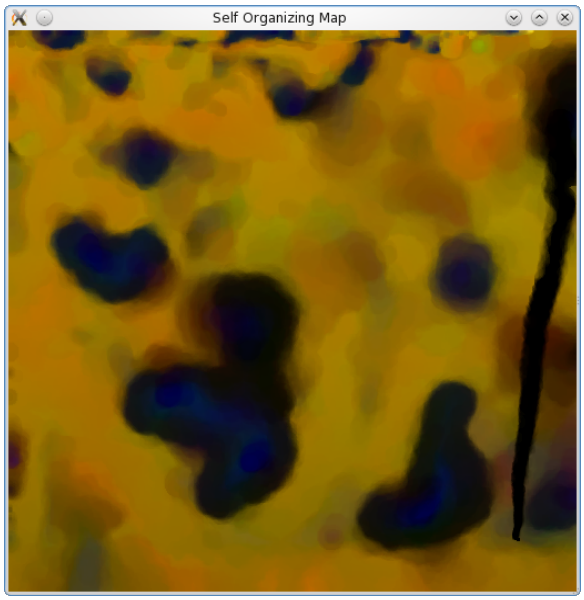

Figure 3.6: A screen capture of an OpenGL application depicting the training of a SOM in real time. The source input used was related to Marquette University, thus the emergence of gold and blue.

encountered. For instance, the limitations in performance gains through parallelization imposed by Amdahl's Law was quite apparent, give the highly serial nature of the training process. Furthermore, overall performance improvement only appeared when the problem size was

increased enough for the speedup gains in parallelization to overcome the overhead costs incurred by the GPU platform.

It is unclear what practical applications exist for SOMs large enough to benefit from GPU acceleration. The data elements used in testing were relatively small in both dimensionality and value, so it is possible that applications with more complicated data such as text mining or bioinformatics would see greater benefit. Exploring these possible applications as well as enhancing the visualization tool are both solid avenues for further study.

CHAPTER 4

Spiking Neural Networks

The SOM from Chapter 3 is a classical neural network that was successfully implemented on a GPU. The SOM does not, however, accurately represent actual biological neural networks as it lacks inter-neuron communication, time-dependent behavior, and several other features. The work presented in this chapter demonstrates the design and implementation of a recurrent biological neural network simulator on a GPU using CUDA. For a connected network, neuron communication becomes an important and limiting factor of the simulation. In a fully connected network — one where each neuron has a directed connection, or synapse, to every other neuron including itself — the number of synapses is equal to the square of the number of neurons. Even at small percentages of full connectivity, the number of synapses is orders of magnitude larger than the number of neurons. Communication quickly dominates the computational time of a simulation. As the neurons and synapses have no direct effect on each other at each time step of the simulation, it is possible to evaluate them in parallel, making the GPU an ideal platform to develop a simulator.

The simulation environment developed models network communication through synaptic transmission of action potentials, while leaky integrate-and-fire neurons were used to handle the spiking dynamics. The key motivating design principle, however, was to map the massively parallel nature of the GPU to the naturally concurrent processing exhibited by neurons. An agent-inspired approach was chosen that maps one neuron per thread. The neuron itself was broken down into its functional components of dendrites, soma, axon, and synapses. Each maintained the one component per thread design with the exception of the synapse which was handled by a thread block. The rest of this chapter details the design decisions and implementation

strategies adopted, concluding with performance results and analysis compared to an MPI-based version of the same general simulation.

4.1 GPU Approach

The recurrent neural population simulation is comprised of three main parts: initialization, execution, and output. In the initialization phase, the population's individual neuron characteristics and network topology are read in from specification files. As defining the exact configuration for each neuron and synapse quickly becomes infeasible as the population size grows, these files are generated by a higher level interface that randomly generates parameters based on specified ranges.

Once the data structures needed by the simulation have been fully initialized and the data to be monitored for later output defined, the execution of the simulation begins. In this phase, the state of each neuron is updated once per time step according to its dynamics based on input from external stimuli and communication from other neurons. The execution lasts until the total prescribed simulation time has been reached, at which point results of any data monitoring are output to a file for off-line processing and review. In this implementation, all initialization and output were handled by the CPU while the GPU was employed to enable massive parallelism in the execution step.

4.1.1 Data Model

The data model chosen to represent the system state of the neural network under simulation was based on neuron morphology. An agent-inspired system [10] was conceived that employed a one-to-one mapping between processing elements and neurological structures. In order to accomplish this, several data structures were defined to represent the various active and passive components of the neurons in the population (Table 4.1).

The Neuron, LIFNeuron, and Synapse structures represent passive information about the simulated neurons, such as number of dendrites, neural dynamic properties, and connectivity. These are constant values — once initialized they do not change throughout the course of a simulation. The active, or dynamic, properties of the neurons are represented by the Dendrite,

Structure	Neural Analog	Data Elements	Description
Neuron	The Cell	dendriteOffset dendriteLength pscOffset pscLength	Address information of dendrites Address information of PSC filter
LIFNeuron	Spiking Dynamics	tauRef tauRC Rleak Vth	Action potential refractory period time constant Neural membrane RC time constant Neural membrane resistance Membrane potential at which an action potential is generated
Dendrite	Dendrites	Jin weight	Input current from connected axon The magnitude and direction of the dendrite's influence
Soma	Cell Body	Jspike Jstim Vm refEnd	Input current from neural communication Input current from an external stimulus Neural membrane potential Time at which the neuron can spike again
Axon	Axon	spike buffer	Indicates a spike has reached the axon terminal Indicates a spike has been initiated at the axon hillock
Synapse	Synapse	sendId recvId pscOffset pscLength	Address of the neuron emitting the spike Address of the neuron receiving the spike Address information of PSC buffer

Table 4.1: Data structures used in the execution of the agent-based GPU model.

Soma, and Axon structures. These take on the dynamic aspect of the neuron, such as the changing input currents, membrane voltage, and action potential events.

Memory Requirements

A major constraint on any GPU-based application is available memory, as the total available device memory is fixed and typically less than system memory. Also, unlike system memory, there is no concept of virtual memory, so if memory requirements exceed physical resources, the application either cannot execute or the application developer needs to implement a custom memory manager.

For a simulation with N neurons, S synapses, and a maximum PSC filter size of $pscFilterSize$, the relative memory requirements (in bytes) are detailed in Table 4.2. As can be

GPU Element	Memory Requirement
dNeurons	$16 * N$
dLIFNeuronS	$16 * N$
dSynapses	$16 * S$
dSoma	$16 * N$
dAxons	$2 * N$
dDendrites	$8 * S$
dPSCFilter	$4 * pscFilterSize * N$
dPSCBuffer	$4 * pscFilterSize * S$
Total	$50 * N + 24 * S + 4 * pscFilterSize * (N + S)$

Table 4.2: Memory requirements (in bytes) for data elements stored on the GPU. As $S >> N$ and $pscFilterSize$ is typically between 50 and 200, the number of synapses dominate the memory requirements.

seen, the PSC-related elements require the most memory and thus are the limiting factor. Additionally, in order to reduce overhead from transferring data off the GPU at every time step, monitoring of simulation state should be done on the GPU resulting in additional memory requirements. This simulation records spike events for each neuron, requiring an additional 4*N bytes of space.

4.1.2 Simulation Flow

The general flow of the recurrent network simulation is depicted in Figure 4.1. First comes the initialization phase, where the neural population and topology data structures are created from externally specified configuration files. All associated GPU data buffers are also allocated and populated with initial values at this time. At this point the simulation enters the main loop which updates the neural population state for every time step required.

For each time step of the simulation, several CUDA kernel functions are called in a set order to approximate the flow of information through the neurons (Listing 4.1). The configuration of the thread grid and thread blocks for neuronal and synaptic kernels are defined at runtime based on the number of neurons and synapses present. Each kernel invocation requests the same number of threads or thread blocks as neural structures defined at that phase, ensuring that each neuron is handled by a single thread, while each synapse is handled by a thread block.

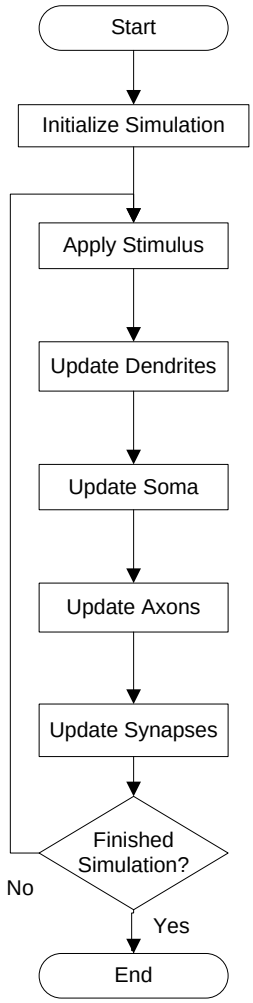

Figure 4.1: The general recurrent network simulation starts with an initialization phase followed by the actual simulation loop and a series of five update processes that together handle a single time step. This loop runs until either all specified time steps have completed or the user ends the simulation.

External Stimulus

The first step in a neural population simulation is to apply any external stimulus present. This step (Listing 4.2) acquires the stimulus data for each neuron at the given time step, transfers the data to the GPU, then performs the required *neural tuning* transformation to convert the input into current based on the particular neuron's *tuning profile* for that stimulus source.

Neural tuning is the concept that the sensitivity of a neuron receiving some form of external stimulus is aligned, or tuned, to a preferred value. The neuron has the greatest response as the input stimulus approaches its preferred stimulus. A neuron's tuning profile is the response of

```
// Soma update kernel invocation
for(int i = 0: i < Nt; ++i) {
  // apply constant external stimulus to neurons
  cudaMemcpy(dStimulus, hStimulus, stimSize, cudaMemcpyHostToDevice);
  applyStimulus<<<neuronGrid, neuronBlock>>>(dSoma, dProfiles, dStimulus);
  cudaThreadSynchronize();

  // sum spike-related input currents
  updateDendrites<<<neuronGrid, neuronBlock>>>(dDendrites, dSoma, dNeurons);
  cudaThreadSynchronize();

  // update neuron membrane voltages and emit spikes
  updateSomas<<<neuronGrid, neuronBlock>>>(dSoma, dAxons, dLIFNeurons, dMonitors
      , i*(sim->dt-FLT_EPSILON));
  cudaThreadSynchronize();

  // handle transmission delays
  updateAxons<<<neuronGrid, neuronBlock>>>(dAxons);
  cudaThreadSynchronize();

  // apply psc filter to spikes, update input currents
  updateSynapses<<<synapseGrid, synapseBlock>>>(dSynapses, dPSCFilter,
      dPSCBuffer, dDendrites, dAxons, i);
  cudaThreadSynchronize();
}
```

Listing 4.1: Simulation execution loop

```
__global__ void
applyStimulus(Soma *dSoma, const TuningProfile *dProfiles, float *dStimulus)
{
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  if(tid >= NEURONS) return;

  dSoma[tid].Jstim = applyTuning(dProfiles[tid], dStimulus[tid]);
}
```

Listing 4.2: `applyStimulus` kernel definition

the neuron across all possible stimulus values, usually expressed through parameters of a continuous function such as a Guassian distribution. This profile is then used to convert the external stimulus values into an associated input current to the neuron. In the current implementation, the stimulus is constant throughout and neurons are stimulated through direct current injection. This means that the result of the tuning transformation is simply unity gain. The generated stimulus current is then placed in the neuron's Soma data structure to be processed.

```
__global__ void
updateDendrites(Dendrite *dDendrites, Soma *dSoma, Neuron *dNeurons)
{
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  if(tid >= NEURONS) return;

  Neuron n = dNeurons[tid];
  Soma   s = dSoma[tid];

  // reset input current
  s.Jspike = 0.0f;

  // sum weighted inputs
  for(int i = 0; i < n.dendriteLength; ++i) {
    Dendrite d = dDendrites[n.dendriteOffset + i];
    s.Jspike  += d.Jin * d.weight;
  }

  dSoma[tid] = s;
}
```

Listing 4.3: `updateDendrites` kernel definition

Dendrites

After the external stimulus is applied, any input current picked up by the the neurons' dendrites are processed and summed (Listing 4.3). At present, each dendrite has a static weight associated with it that is applied to the incoming current.

Soma

Once all the input sources have been updated, the soma processes the input and adjusts the neuron's membrane potential accordingly (Listing 4.4). In this implementation, the soma uses LIF spiking dynamics with a fourth-order Runge-Kutta (RK4) ODE solver to handle the numerical integration. If action potential threshold is reached by the membrane potential, the membrane is reset and the neuron's axon is updated to indicate a spike has been initiated.

As the GPU operates on single-precision floating point values, error can creep into the simulation state variables due to the inherent round-off error introduced by floating-point arithmetic. This can have a significant effect on simulation results, skewing the output. This is most apparent in the check for the end of the neuron's refractory period, which is based on the calculated current simulation time. The simulation clock operates on a resolution of microseconds, but is tracked in terms of seconds. At single-precision, tiny round-off errors that appear in the

```
__global__ void
updateSomas(Soma *dSomas, Axon *dAxons, const LIFNeuron *dLIFNeurons, Monitor *
    dMonitors, const float time)
{
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  if(tid >= NEURONS) return;

  float  Jd;
  float4 rk4;

  Soma      s = dSomas[tid];
  LIFNeuron n = dLIFNeurons[tid];

  // combine input sources
  Jd = s.Jspike + s.Jstim;

  // only integrate if allowed
  if(time > s.refEnd) {
    // RK4
    rk4.x = DT * (Jd * n.Rleak -  s.Vm              ) / n.tauRC;
    rk4.y = DT * (Jd * n.Rleak - (s.Vm + 0.5f*rk4.x)) / n.tauRC;
    rk4.z = DT * (Jd * n.Rleak - (s.Vm + 0.5f*rk4.y)) / n.tauRC;
    rk4.w = DT * (Jd * n.Rleak - (s.Vm + rk4.z)     ) / n.tauRC;
    s.Vm += (rk4.x + 2*rk4.y + 2*rk4.z + rk4.w) / 6;

     // reset Vm and emit spike if threshold exceeded
    if(s.Vm > n.Vth) {
      s.Vm               = 0.0f;
      s.refEnd           = time + n.tauRef;
      dAxons[tid].buffer = 1;
      dMonitors[tid].spikeCount++;
    }
    dSomas[tid] = s;
  }
}
```

Listing 4.4: `updateSoma` kernel definition

calculation of the simulation time can have an effect on whether a spike occurs or not. To correct for this, the *machine epsilon*, the upper bound on error due to rounding, is incorporated into the simulation time calculation before passing it to the kernel, ensuring that the time will never be greater than expected and preventing spikes from generating prematurely.

Axons

Now that the action potentials have been generated, the next step is to communicate those spikes to the afferent neurons. However, they must first propagate down the axon according to a transmission delay. In the current simulation, all transmission is instantaneous (the transmission

```
__global__ void
updateAxons(Axon *dAxons)
{
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  dAxons[tid].spike  = dAxons[tid].buffer;
  dAxons[tid].buffer = 0;
}
```

Listing 4.5: `updateAxons` kernel definition

delay is zero), so the the incoming buffer is simply copied to the current spike status then reset (Listing 4.5).

Synapses

The last phase of the execution loop is to process the neuronal communication. As mentioned in Section 2.2.5, the simulation employs a linear filter to determine the value of the PSC. The filter is several milliseconds long and based on the particular PSC filter parameters for the neuron. In order to handle iterative convolution, a circular buffer is used to store the intermediate results. Originally the PSC buffer was updated by a single thread that incremented each value in the buffer iteratively at each time step. To increase performance, the buffer is only updated if a spike has arrived at the synapse. Further enhancements changed to a whole thread block to update the buffer in parallel with pre-fetching of filter data into shared memory to increase memory access performance. As it is still only a single synapse that is being updated, the kernel is handled similar to the MPI paradigm of having the master process (here thread 0) handle specific initialization and output functions. All threads participate in updating the PSC buffer, then the actual current value at the particular time step is placed in the associated dendrite by the master process (Listing 4.6).

The reason the synapse must update its corresponding dendrite and not the afferent neuron directly is to avoid *race conditions*, situations in which parallel processes acting on a single data source may interfere with the results of one another. For example, if synapse 1 acquired the input current value of the afferent neuron and, before writing the incremented value back to the neuron, synapse 2 read the same input current value, then synapse 2 would not know about synapse 1's pending change and ultimately overwrite it when writing back its own increment change. While CUDA does support *atomic* operations — operations that guarantee that the read and write events

needed to update a value occur as a single event — that can help avoid race conditions, these functions only apply to integer values precluding their use here.

```
__global__ void
updateSynapses(const Synapse *dSynapses,
               const float   *dPSCFilter,
               float         *dPSCBuffer,
               Dendrite      *dDendrites,
               Axon          *dAxons,
               const int     step)
{
  __shared__ int nOffset, sOffset, length, spike, idx;
  __shared__ float filter[256];

  int sid = blockIdx.x + blockIdx.y * gridDim.x;
  if(sid >= SYNAPSES) return;

  int tid = threadIdx.x;

  // load common references
  if(tid == 0) {
    Synapse s = dSynapses[sid];
    nOffset   = PSC_BUFFER_LENGTH * s.recvId;
    sOffset   = PSC_BUFFER_LENGTH * sid;
    length    = s.pscLength;
    idx       = step % s.pscLength;
    spike     = dAxons[s.sendId].spike;
  }
  __syncthreads();

  if(tid >= length) return;

  if(spike) {
    // prefetch filter
    filter[tid] = dPSCFilter[nOffset + tid];
    __syncthreads();

    int bid = tid + idx;
    if(bid >= length)
      bid -= length;

    // step-wise convolve w/ circular buffer
    dPSCBuffer[sOffset + bid] += filter[tid];
    __syncthreads();
  }

  if(tid == 0) {
    dDendrites[sid].Jin = dPSCBuffer[sOffset + idx];
    dPSCBuffer[sOffset + idx] = 0.0f;
  }

}
```

Listing 4.6: `updateSynapses` kernel definition

4.2 Performance Evaluation

Performance was evaluated as wall time needed to complete a full simulation run. The GPU version was compared against an MPI version running on one, two, four, and eight processors on a single node and 16 processors divided equally across four nodes. Times were acquired either from direct recording within the simulation using the `gettimeofday` system routine (GPU) or the UNIX `time` command (MPI). The average execution times of the individual GPU kernels were also recorded to identify performance bottlenecks. All runs evaluated a 10 second simulation using 0.25 millisecond time steps. Additionally, only one neuron (the input neuron) was provided external stimulus and all synaptic weights were set such that any received spike would immediately trigger the generation of another. This caused the network to be overdriven — the maximum number of spike events occurred given the connection patterns. All time trials recorded would then represent worst-case performance scenarios. All neuron specifications were identical and no stochastic elements were employed.

Four general simulation topologies were chosen to test the performance of the simulation under various conditions: ring, fully connected, randomly connected with fixed population size, and randomly connected with fixed number of synapses.

The ring topology (Figure 4.2) has as many synapses as neurons, with each neuron making and receiving only one connection. This network tests performance under constant communication for the duration of the simulation, as spikes propagate around the ring at each time step causing at least one neuron to be emitting a spike at any given time. This configuration was tested for population sizes ranging from 100 to 500,000.

The fully connected network (Figure 4.3), on the other hand, tests performance under periods of maximal communication load. Once the input neuron spikes, every other neuron in the population will spike at the next time step. This is followed by several time steps where no spiking occurs due to the refractory period of the neurons. Populations of up to 1000 neurons (1 million synapses) were simulated in the fully connected configuration.

The randomly connected networks aimed to test the simulation tool's ability to scale with respect to the number of synapses and neurons. To ensure consistency between trials, the connection specification file was generated once and used in the GPU and MPI trials. Random

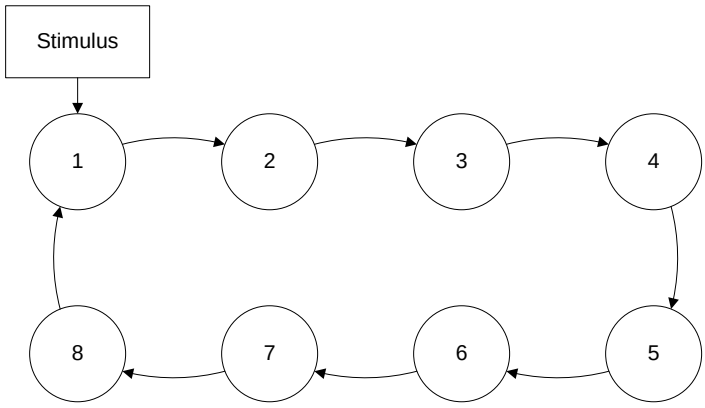

Figure 4.2: Simple schematic of an eight-neuron recurrent network in a ring topology. In this particular configuration, an additional external stimulus is applied to the first neuron in order to initiate network activity

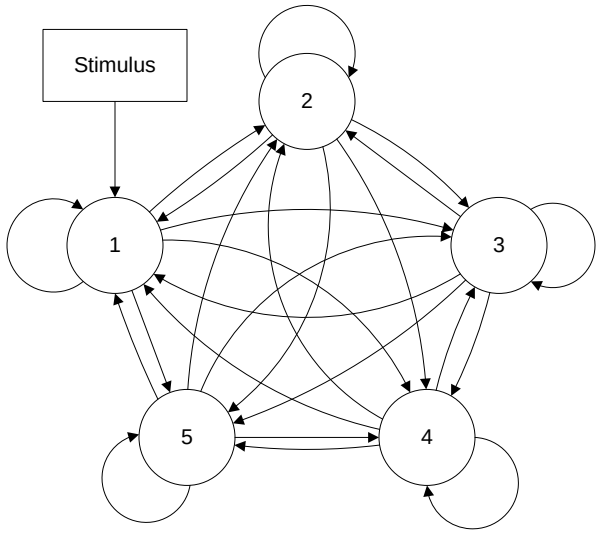

Figure 4.3: Simple schematic of a five-neuron recurrent network in a fully connected topology. Since the connections are directed, each neuron will have two connections with every other neuron in addition to a connection with itself for a total of 25 connections. In this particular configuration, an additional external stimulus is applied to the first neuron in order to initiate network activity.

connections were filtered to ensure no redundant connections existed. The random network tested scalability with respect to synapses by fixing the population size at 1000 neurons while the percent connectivity was varied from 0.1% (ring) to 100% (fully connected). This population size was chosen as the full range of connectivity configurations was possible given the limitations of the test GPU's memory. Likewise, to test the simulation tool's scalability with respect to neurons, the number of synapses was set at 500,000 while the population size was varied from 1000 neurons

(50% connectivity) to 500,000 neurons (0.0002% connectivity). Here, 500,000 was chosen for the number of synapses to allow a wide range of population sizes that were neither overly connected ($> 100\%$) nor sparse (less than one synapse per neuron).

For evaluating individual kernel performance, times were recorded for each kernel call at each iteration. The durations were summed and the average kernel time was determined by dividing the final totals by 40,000, the number of iterations for the simulation.

4.2.1 Test Environment

The GPU test runs were conducted on a machine with an AMD Phenom II X4 925 clocked at 2.8GHz with 2MB cache per core, 8GB system memory, running the 64-bit Fedora 12 Linux distribution. This machine had a GeForce GTX 260, a CUDA 1.3 device, with 216 cores, 16k registers, 896MB device memory operating at 1.35GHz, and a PCIe Gen2 connection. The MPI runs were tested on the Marquette Père compute cluster, with the single node trials conducted on the head node and the distributed node trials conducted on dedicated compute nodes. Each node had two Intel Nehalem X5550s for a total of 8 physical cores (16 with hyper-threading enabled) clocked at 2.67GHz with 8MB cache per core, 24GB system memory, running the 64-bit RedHat Enterprise Linux 5.3 distribution.

4.3 Results

For the ring topology (Figure 4.4), the GPU outperformed the MPI version in all core configurations once the population size reached 50,000 and was respectable at lower populations sizes as well. The 16-core trials appear to be dominated by communication costs as they consistently lasted at least a minute regardless of the problem size. At 500,000 neurons, only the 16-core MPI version remains competitive with the GPU.

When confronted with a fully connected network (Figure 4.5), the GPU struggled to keep up with the MPI version, only outperforming the communication-throttled 16-core configuration and only then at 10,000 neurons or less. There is no data for 16 cores at the 100 synapse, 10 neuron trial as the MPI tool requires at least as many neurons as cores to execute properly.

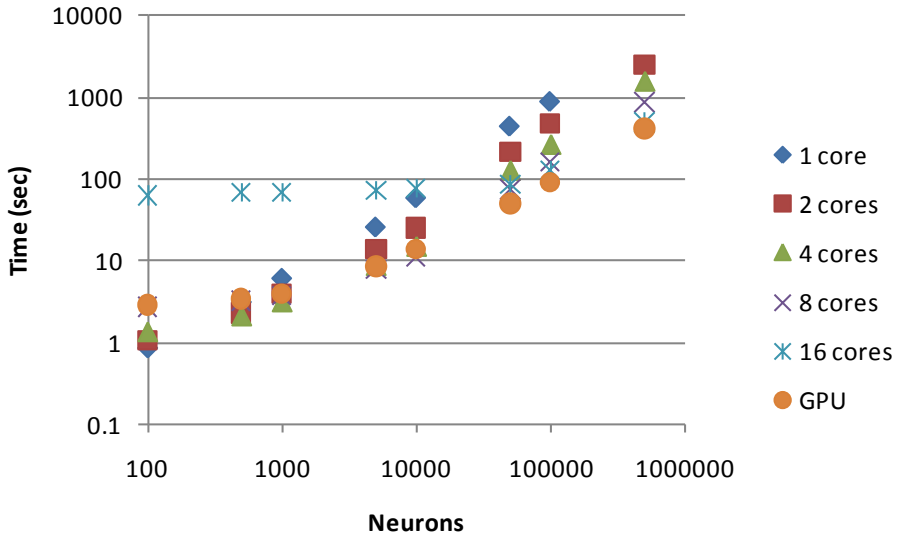

Figure 4.4: Time needed to complete a simulation for a neural population in a ring topology. The number of synapses equals the number of neurons.

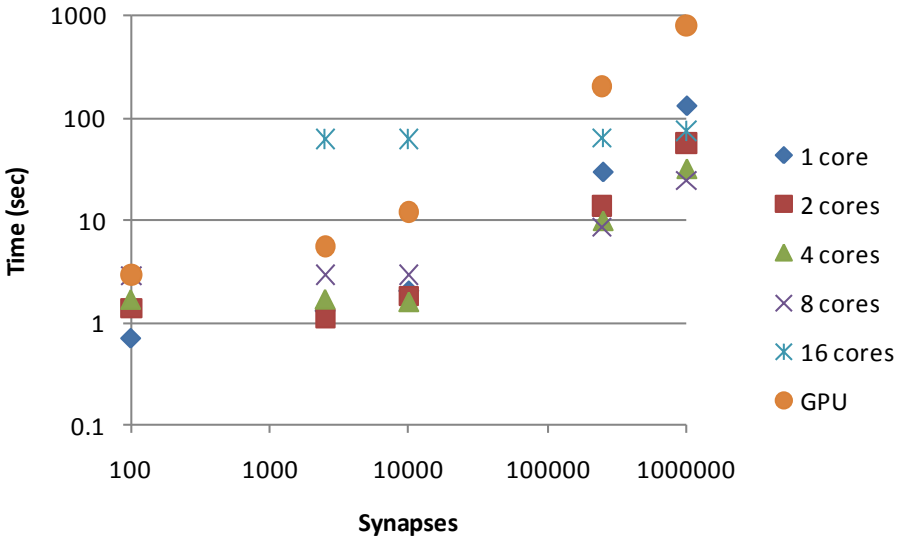

Figure 4.5: Time needed to complete a simulation for a fully connected neural population. The number of neurons equals the square root of the number of synapses.

For the fixed population size of 1000 neurons (Figure 4.6), the GPU again lags behind the MPI versions, slowing down significantly as the percent connectivity increases. When the number of synapses was held constant at 500,000 (Fig. 4.7), the GPU showed almost constant scaling, while the MPI versions appeared linear. The GPU starts out almost an order of magnitude slower

then the MPI trials, but by 50,000 neurons is close to the 16-core version and has become the best performer by 100,000 neurons.

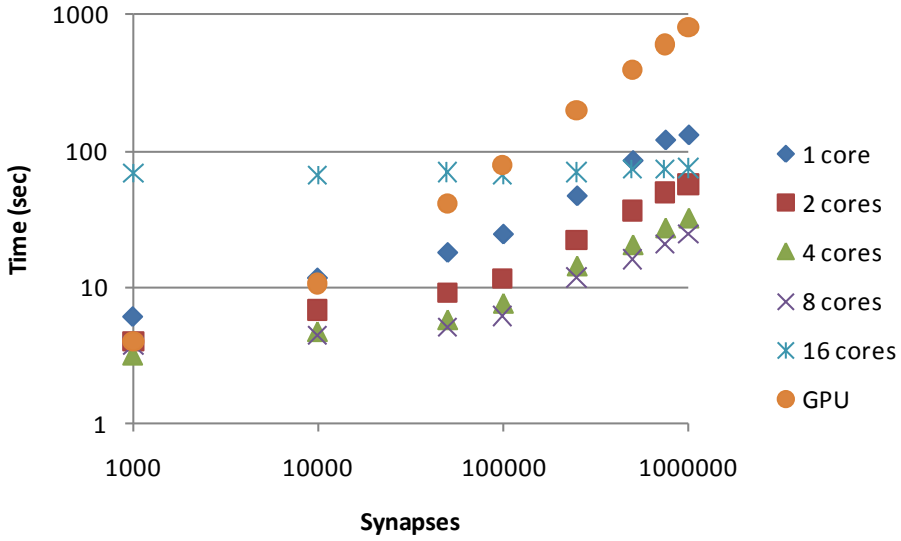

Figure 4.6: Time needed to complete a simulation for a 1000 neuron population. The percentage of connectivity ranges from 0.1% at 1000 synapses to 100% at 1 million synapses.

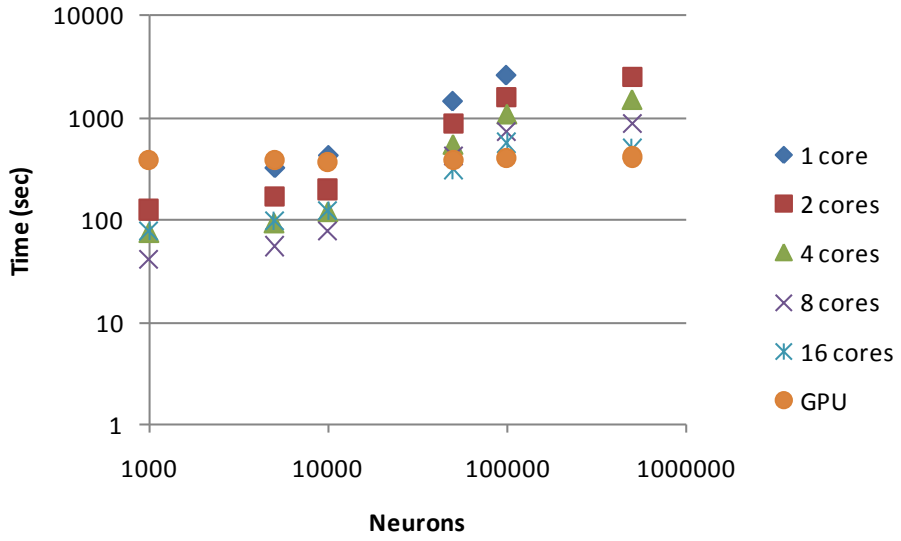

Figure 4.7: Time needed to complete a simulation for neural populations containing 500,000 synapses. The percentage of connectivity ranges from 50% at 1000 neurons to 0.0002% at 500,000 neurons.

Average times for each kernel were obtained for both the fixed neuron and fixed synapse trials (Figures 4.8, 4.9). In both cases the synapse kernel dominated all others, with obvious linear scaling demonstrated in the fixed neuron trial. There was noticeable scaling of the neuron-based kernels in the fixed synapse trial, but the time needed by them collectively was still significantly less than the synapse kernel.

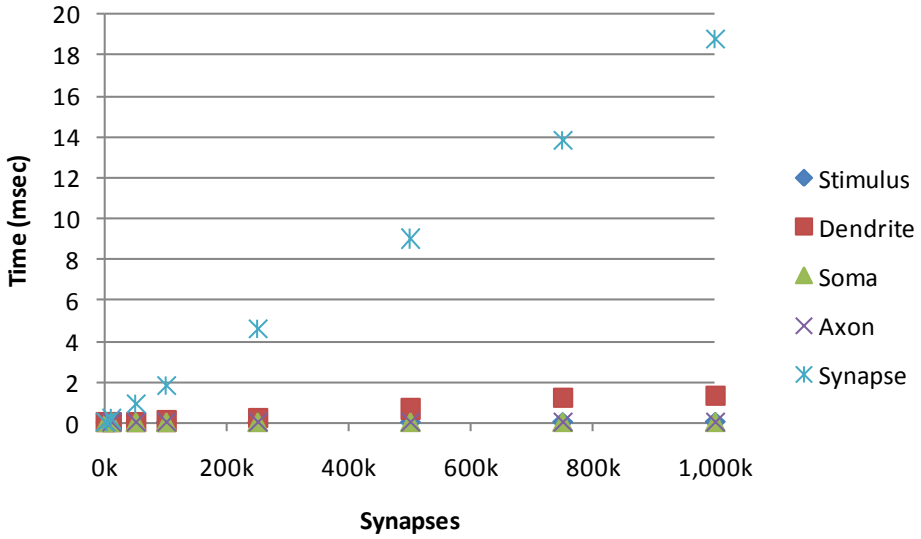

Figure 4.8: Average time (n=40,000) needed to complete the individual GPU kernels for a 1000 neuron population. The percentage of connectivity ranges from 0.1% at 1000 synapses to 100% at 1 million synapses.

4.4 Discussion

Due to the sheer dominance of the synaptic update kernel on the GPU, its performance can be characterized almost entirely by the number of synapses that exist within a population. The MPI version, on the other hand, appears to be sensitive to both neurons and synapses, as consistent scaling is observed when fixing either neurons or synapses. The almost constant time from the 16-core trials arises from communication and setup overhead needed for the multi-node MPI runs. As the compute cluster these trials were conducted on is a shared resource, it is not possible to get a completely controlled timing environment. Other research code was indeed running while acquiring this data. However, scaling consistent with the other core counts can be observed when the problem size is big enough to overcome the set cost of communication.

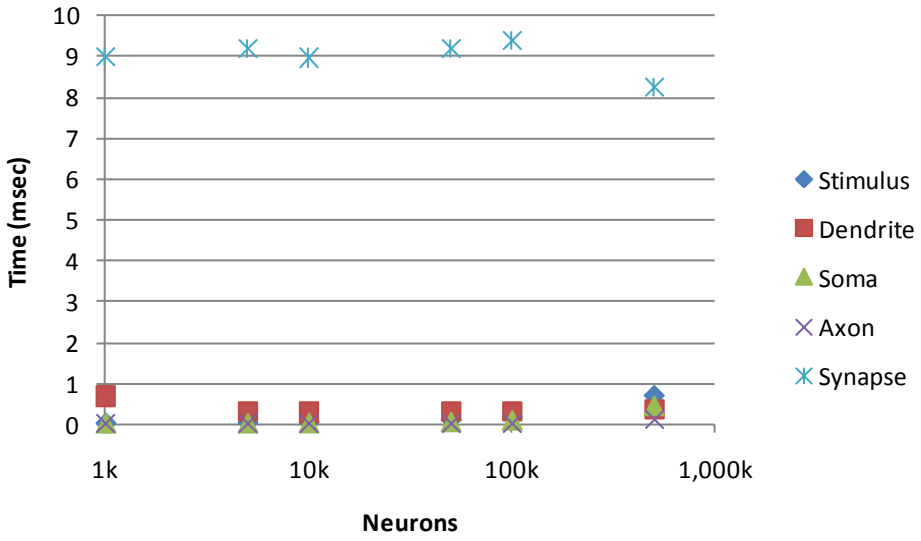

Figure 4.9: Average time (n=40,000) needed to complete the individual GPU kernels for neural populations containing 500,000 synapses. The percentage of connectivity ranges from 50% at 1000 neurons to 0.0002% at 500,000 neurons.

While the GPU is sensitive to the number of synapses present in the population, it is able to handle communication well and demonstrates a significant advantage over the MPI version for populations in which spiking events happen continuously. The overdriven nature of the trials demonstrated both the best and worst case scenarios for spike event occurrence. Most typically designed populations will probably behave somewhere in-between, as the neurons will not be identical, the percentage of connectivity will be low, and the synaptic weights will be realistic. In these situations, the memory of the GPU limits its usefulness as the GTX 260 used in these trials could not perform a simulation of a reasonable population size of 100,000 neurons with 1 million synapses. While more powerful GPUs certainly could accommodate this population, the purely GPU-based tool cannot scale up the same way the MPI version can without additional custom memory management.

4.4.1 Agent-Inspired Model

The agent-inspired model presented here has shown both strengths and weaknesses. The GPU simulation is tied, performance-wise, to the synaptic update kernel, which must evaluate each synapse at each time step. The overdriven fully connected model thus represents the worst case

scenario from a performance standpoint, as there are $N - 1$ spikes occurring at a time. As a typical simulation will only have a small percentage of neurons firing at any given time, checking every synapse introduces a significant amount of computational overhead. A reasonable solution, albeit one that departs slightly from the strict agent model, would be to separate the PSC filter update from the transmission of final input currents to the dendrites. The latter does happen at each time step, but the former is only needed when a spike has occurred. Predetermining those synapses requiring a PSC filter update could potentially reduce time spent in the collective communication kernels by moving the input current step to a single thread per synapse model. This would be much faster than the thread block per synapse currently employed, while it increases performance for the filter update step, it greatly reduces the number of active threads able to perform the current update step.

A major strength of the agent-inspired model is the relatively straightforward approach needed to change the behavior of any portion of the simulation. Replacing the current PSC filter method with an ion channel-based system for synaptic output or using Izhikevich dynamics instead of LIF for neural spiking is simply a matter of modifying the relevant kernel or creating a new one that can be swapped in through high-level configuration settings. This allows for a modular and customizable simulation setup.

In general, the GPU approach described above performed well in comparison to the parallel MPI approach. It is worth noting that the timing results are from a mid-grade consumer-class G200 architecture NVIDIA GPU that retails for around $250 at the time of writing, while the MPI version was conducted on high-end dedicated compute nodes within a $500,000 computational cluster. While true that the GPU was a component in a custom-built development machine and only a fraction of the total compute nodes within the cluster were employed, the actual price difference was still around an order of magnitude cheaper for the GPU-enabled machine. When considering performance relative to raw cost, the GPU clearly outperforms the compute cluster. Furthermore, the cluster has additional costs associated with personnel, maintenance, space, HVAC, and security. Given the additional cores and memory available on the dedicated compute GPUs, a larger investment could net significant performance gains without any modifications required. Finally, many currently available laptops contain CUDA-enabled GPUs allowing for a level of portability not feasible with a computer cluster or supercomputer.

4.4.2 Validation

It is important to note that the GPU and MPI versions of the simulation tool were developed by different programmers under different supervision. Efforts were made to ensure consistency between the two approaches where possible to ensure any comparisons were accurate. Unfortunately, there were small differences that resulted in slightly different behavior. The GPU version used a 0^{th}-order PSC filter while the MPI version used a 1^{st}-order filter. In the overdriven models tested, the only difference this made was that any input a neuron received in the GPU version would immediately emit a spike, whereas the MPI version's neuron would wait a time step before spiking. This resulted in a 4% increase in the number of spike events occurring in the GPU version. As the GPU's performance is dependent on spike activity, switching to a 1^{st}-order PSC filter should actually improve performance.

Another, more subtle difference between the two versions was the way the numeric integration of neuron membrane potential was handled. The GPU version incorporated the driving current at the current time step, while the MPI version used the current from the previous time step. The latter leads to slightly fewer spikes in the overdriven case as more time passes between when the spike arrives at the neuron and when it is able to be integrated into the membrane potential. Again, if the GPU version was to use this approach, it is probable that performance for longer simulation trials would experience minor improvement.

As these differences produce different spiking results, it is not possible to fully equate the spike counts of the GPU version's neurons with those of the MPI version. However, it is possible to analytically demonstrate the GPU version is behaving as expected using the overdriven ring topology. When a neuron spikes at a given time step, two deterministic events occur. One, the next neuron in the ring spikes at the next time step, and two, the neuron will not spike again for eight time steps due to the given the refactory period length of 2 msec and time step of 0.25 msec. No neuron spikes at $t = 0$, while the input neuron spikes at the first time step due to the applied external stimulus and then every nine time steps thereafter. These spikes propagate around the ring so that every neuron spikes every nine time steps until the end of the simulation once it receives its first spike. For a 10 second simulation run, the first three neurons will spike 4445 times ($\lfloor 40\,000/9 \rfloor = 4444, mod(40\,000, 9) = 4$), then every consecutive nine neurons will spike one less

than the previous group. This is, in fact, what the GPU version reports. Neurons 1-3 spike 4445 times, neurons 4-12 spike 4444 times, neurons 13-21 spike 4443 times, and so on.

4.4.3 Visualization

As seen in Chapter 3, an additional key benefit of CUDA is its interoperability with graphics libraries. By using a simple OpenGL program (Appendix C), a real-time raster plot of neural population spiking activity can be observed. While this slows the overall simulation down and may have limited utility from the performance side, it provides insight into how the neurons are behaving given a certain stimulus. This can both aid the modeler in verifying that the simulation is running as expected and also provide a useful demonstration for educational purposes (Figure 4.10).

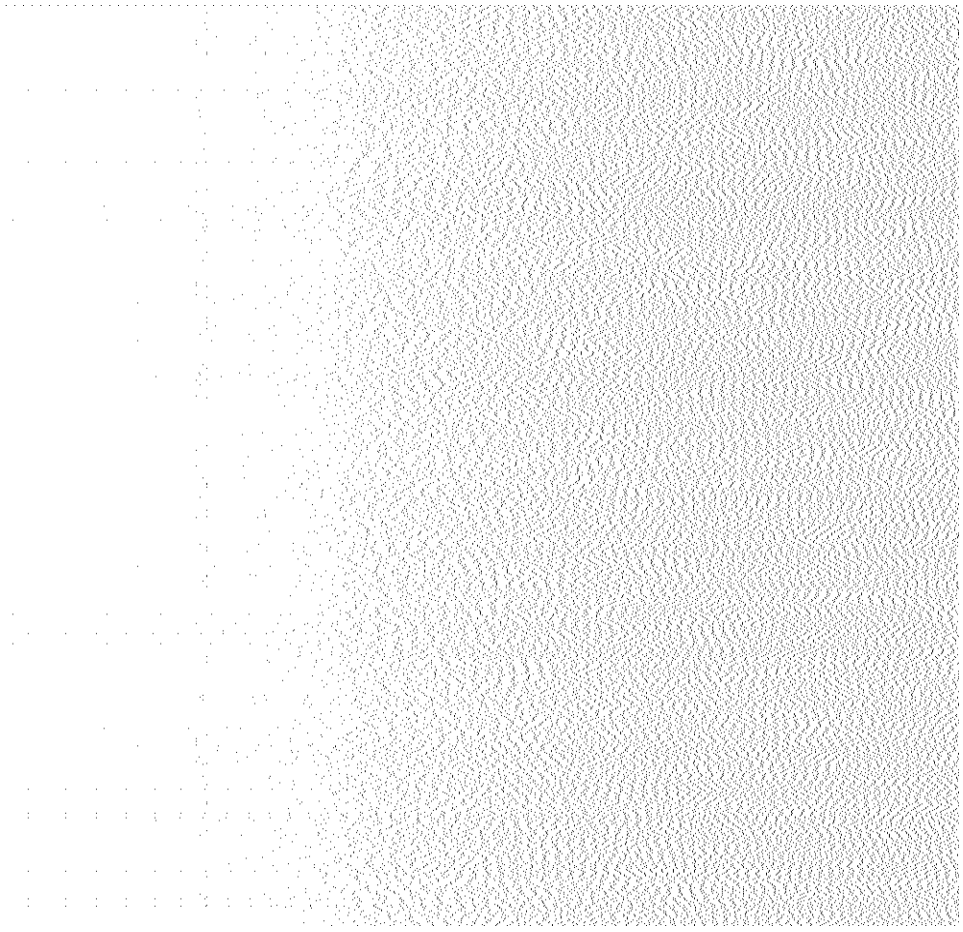

Figure 4.10: Snapshot of the real-time raster plot generator showing the rapid evolution of the 1% connected 1000 neuron network. Each pixel is a spike event, each row is a neuron, and each column is a time step. This window shows the last 1000 time steps corresponding to 0.25 msec.

4.5 Conclusion

In implementing this GPU-based simulation, several simplifying assumptions were made. First, there is no mechanism for handling action potential transmission delay beyond one time step. Adding this functionality will require additional development of the `Axon` data structure and associated methods, though the basic framework is in place to allow it without significant reformulation of the code. Second, synaptic weights are static, preventing the network from learning or evolving beyond its initial configuration. Third, the external stimulus signal was set as a hard-coded constant value. Providing an interface to acquire signals from arbitrary sources is needed. Finally, the only non-visual monitoring capability available is spike counting, which, while good for determining average firing rates, is not as useful as spike times or membrane potential traces for investigating neural dynamics. In order to enable more robust and practical options available to experimenters, these features need to be added. Doing so will, however, hamper the performance by increasing the computational and memory requirements.

In addition to addressing the above weaknesses, another area for future study would be adding support for automatic memory management for the GPU to enable larger population sizes and connectivities on limited hardware. Support for multi-GPU implementations, both local and distributed, could also be pursued. Finally, further enhancing the visualization tool as both a diagnostic and educational resource would be a valuable project in its own right.

CHAPTER 5

Neural Signal Decoding

An area of considerable interest in Biomedical Engineering research at present involves *Brain-Computer Interface (BCI) / Brain Machine Interface (BMI)*, which studies the use of computers and computational devices as prosthetic replacements for lost sensory and motor function in afflicted patients [19]. Probably the most well known example BCI is the cochlear implant [50], a device that restores some sense of sound to deaf patients by bypassing the defective part of the ear and stimulating the cochlea directly. The cochlea converts the auditory signal into neural signals which are then processed by the brain. The success of the cochlear implant can be attributed in part to the fact that researchers could target a single interface point (the cochlea) which then handled the much more complicated process of *encoding* the input signal into the appropriate *neural representation*. For other applications, such as neuromotor prosthetics, there is no single interface point. Instead, researchers must attempt to *decode* the intended motor action from neural signals in the appropriate region of the brain. Not only is the acquisition of these signals problematic, the processing and decoding of the signals into actionable data can present a significant computational challenge that inhibits the development of real-time BCI/BMI devices.

In Chapter 4, the GPU's massively parallel nature was harnessed to simulate how large populations of interconnected neurons responded to an externally applied stimulus signal. In this chapter, the focus shifts to decoding that neural response in order reconstruct the stimulus signal. Specifically, this chapter details efforts to adapt the population temporal linear filtering method for neural decoding put forth by Herzfeld and Beardsley [30] to the GPU. GPUs are attractive for certain BCI/BMI applications as they can provide the performance of a modest computational cluster while doing so at a fraction of the space, power, and maintenance required. Additionally,

CUDA-enabled GPUs exist in commercially available laptops, allowing for portability that would not be possible with dedicated processing computers. However, as seen with the spiking neural network simulator, GPUs are limited by available memory. Such a limitation presented itself in the scaling up of the decoding algorithm to handle large neural populations. This necessitated the development of the CUSUMMA algorithm for seamlessly handling large-scale matrix-matrix multiplications. The algorithm is described in-depth in Section 5.2.

5.1 Neural Decoding

The first step in neural decoding is to acquire and record the activity of the neurons under study. This can be done through a number of both invasive and noninvasive methods, though BMI applications typically are geared toward invasive, direct recording of neuron membrane potentials. One common way to do this is through microelectrode arrays, such as the Utah Intracortical Electrode Array [41]. This device contains 100 electrodes arranged in a 10 x 10 array that can be implanted onto the surface of the brain. This approach often will find a single electrode positioned by multiple neurons, resulting in complex signals comprising some combination of each neuron's activity. This *multi-unit (MU)* recording has to undergo additional processing to attempt to separate out the individual neural signals to approximate a *single-unit (SU)* recording paradigm. This provides greater signal fidelity which in turn allows for more accurate decoding, but at a greater computational cost. However, as shown by Herzfeld and Beardsley, working directly with the MU-recorded signal can still provide accurate decoding results.

Linear Filter Optimization

Using spike-based linear filter decoders, as opposed to more common firing-rate methods, they followed the approach described by Eliasmith and Anderson [21] to determine optimal linear decoders of the neurons. Given an input stimulus $x(t)$, the i^{th} neuron in a population has a certain response $a_i(t)$ that, along with the rest of the neurons, represents the encoded input signal. There exists some optimal decoder ϕ such that $x(t) = a(t)\phi$. Neurons are not perfect transducers, nor are the electrodes perfect recorders, so noise exists throughout the system. In order to find the optimal decoders in the presence of noise, a standard error minimization process is followed, ultimately

resulting in the relationship $\Gamma\phi = \Upsilon$, where

$$\Gamma = a_i(t)a_i(t)^T \tag{5.1}$$

and

$$\Upsilon = a_i(t)x(t). \tag{5.2}$$

Solving for ϕ gives

$$\phi = \Gamma^{-1}\Upsilon \tag{5.3}$$

which can then be used to estimate the stimulus signal $\hat{x}(t) = a_i(t)\phi$ and compared against the original source for accuracy.

Simulation

Herzfeld and Beardsley demonstrated through simulation that the spike-based, MU approach to finding optimal decoders was both more efficient and more accurate than previously established rate-based, SU methods. The simulation process (Figure 5.1) used for the spike-based, *population temporal (PT)*, simulation starts by generating a 100 second, 2D band-limited white noise training stimulus representing changes in X,Y velocity. The neural population to be stimulated is then initialized, using a LIF neural model with a set neural response tuning. All parameters are randomly selected from a uniform distribution across predefined ranges. The spiking response of the neural population is then determined by converting the input stimulus into a driving current based on the neural tuning profile. The driving current is used to determine individual spike events representing the SU recording model. To approximate the MU model, the individual spike trains are combined into compound signals. The neural output responses are then determined by convolving the spike events with PSC filters. These responses are then low-pass filtered and adjusted for delay. Decoding weights are optimized from the training response according to the equations listed above. In order to verify the success of the optimized decoding weights, a new ten second testing signal is generated, the spike events and neural response calculated, and the decoding weights used to reconstruct an estimate of the original test signal.

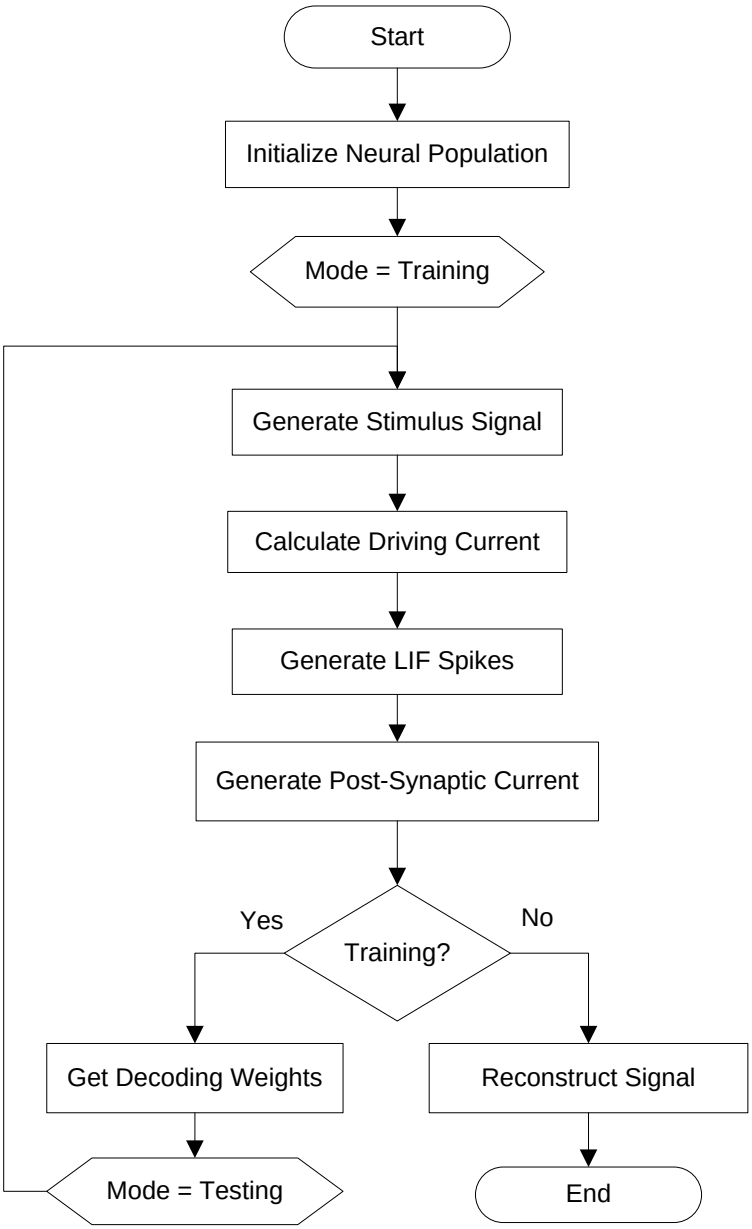

Figure 5.1: Simulation flow for the PT decoding trials. At the PSC generation, decoding weight determination, and signal reconstruction operations , both SU and MU results are calculated.

Their approach used a parallelization technique based on the MPI and ScaLAPACK libraries to successfully run experiments of up to one thousand independent neurons. At one thousand neurons, the simulation required an estimated hour to run on a 36-node cluster with 1Gb of memory per core. Most of the computational load came from the algorithm used to determine the optimal decoding weights based on neural responses to a training signal, though a number of large scale convolutions during the simulation of the neural population activity also contributed to the computational cost.

5.1.1 GPU Approach

As stated, the purpose of the simulation is to demonstrate that MU recordings of the direct spiking activity of a neural population can be used to achieve effective decoders of the neural response based on a stimulus. The key to the this approach is the process by which these decoders are determined from past neural response data, which consists of several matrix-based linear algebra routines. In order to obtain the neural response to begin with, recorded spike events are convolved with the PSC linear filters. Due to their importance and readily parallelizable nature, both the convolution and decoding weight optimization steps were targeted for implementation on the GPU.

Signal Convolution

Once the neural spike trains are acquired, a filter is applied to convert the spikes into post-synaptic currents. At the training phase, the resulting spike trains are 400k-element vectors, (100 seconds at 0.25 msec time step), so the DFT strategy for convolution is employed to achieve greater performance. In this approach, the two signals to be convolved are converted to the frequency domain, point multiplied, and the product is converted back to the time domain for the final answer. As this process has to be repeated for each neuron in the population, even a slight increase in performance for a single run will be magnified N times.

The CUFFT library [45] was used to handle the necessary DFTs. Additionally, a simple CUDA kernel function was created to perform the point multiplication directly on the card in order to a) achieve a high degree of parallelism for the operation and b) avoid unnecessary and costly

```
cufftPlan1d(&plan, fft_length, CUFFT_C2C, 1);
cufftExecC2C(plan, d_a, d_a, CUFFT_FORWARD);
cufftExecC2C(plan, d_b, d_b, CUFFT_FORWARD);

dim3 block(256);
dim3 grid(fft_length / block.x);

pointMultiply<<< grid, block >>>(d_c, d_a, d_b);
cufftExecC2C(plan, d_c, d_c, CUFFT_INVERSE);
```

Listing 5.1: Code demonstrating the convolution of two complex signals using the CUFFT library.

```
__global__ void
pointMultiply(float2 *d_c, float2 *d_a, float2 *d_b)
{
  unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;
  d_c[idx].x = d_a[idx].x * d_b[idx].x - d_a[idx].y * d_b[idx].y;
  d_c[idx].y = d_a[idx].x * d_b[idx].y + d_a[idx].y * d_b[idx].x;
}
```

Listing 5.2: The `pointMultiply` CUDA kernel used to pairwise multiply two signals in the frequency domain.

data transfers between the CPU and GPU. The CUFFT routines are highly optimized for when the length of the signals being transformed is a power of two, so all signals passed into the convolution routine were zero-padded up to the next greatest power of two.

Decoding Weight Optimization

Considered to be the bulk of the computational load of the simulation, the process for finding the optimal neural population decoding weights consists of two very large matrix-matrix multiplications (Γ, Υ), a matrix inversion (Γ^{-1}), and another matrix-matrix multiplication (ϕ). Given that each of these operations has a complexity of $O(n^3)$ for naive implementations, better performance is critical for achieving scalability to larger population sizes.

The main challenges to finding the weights were broken down into the two tasks of handling matrix multiplication and matrix inversion. The former was initially solved simply by the CUBLAS library's `cublasSgemm` routine [44], which is already heavily optimized to handle general matrix multiplication. Unfortunately, this could not account for the limitation in physical device memory. At 1000 neurons, the primary matrix being multiplied was 1,000 x 400,000 x 4 bytes or approximately 1.6GB. This is well beyond the capacities of all but GPUs dedicated solely as compute devices, so a new approach had to be devised to allow simulations to run on a wide

variety of GPUs. This led to the creation of the CUSUMMA algorithm, which is described in detail in Section 5.2.

The matrix inversion was handled in a more straightforward fashion. As the inversion is of a square matrix with order equal to the number of simulated neurons, the matrix being inverted is never truly large, so the strategy employed was to implement the LAPACK method for handling general matrix inversion, replacing the BLAS functions with CUBLAS calls. The required LAPACK functions were created using a naive implementation that focused on minimizing device-host data transfer as the primary optimization point (Listing D.2).

5.1.2 Results

First the performance of the convolution routines were evaluated. There were four points within the entire simulation run that convolution was needed – applying PSC filters to the training and testing spike response as well as performing a cross correlation operation on the resultant neural responses as part of a delay correction following low-pass filtering. CUFFT was compared to a CPU-only FFTW implementation. As can be seen in Figure 5.2, the CUFFT library performed significantly better than FFTW across all four convolution sizes.

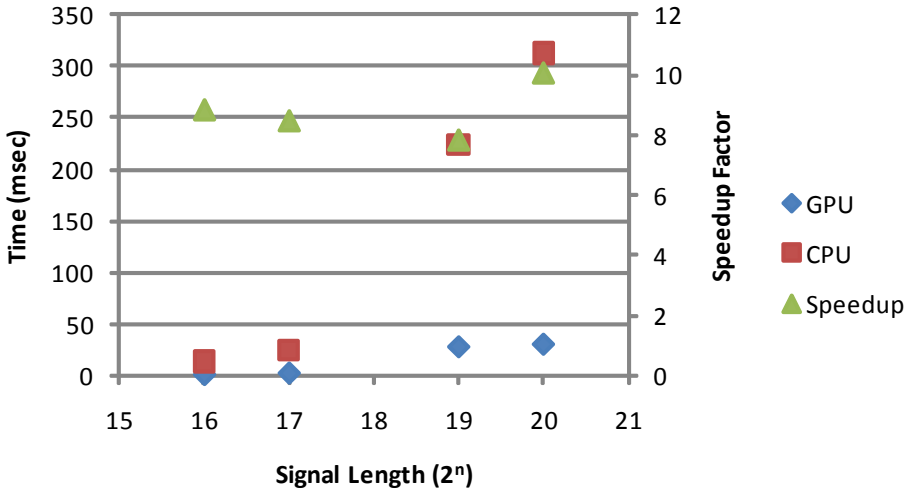

Figure 5.2: Performance of the GPU (CUFFT) vs. the CPU (FFTW) convolution routines for signal lengths of a given power of 2. Speedup is equal to CPU/GPU

For the neural weight decoding operation, the CUSUMMA algorithm was successfully employed to handle the Γ, Υ, and ϕ matrix multiplications. The largest multiplication was the Γ

calculation, as $a_i(t)$ was $N \times 400\,000$, where N was the number of neurons in the population and $400\,000$ was the number of time intervals in the signal ($\Delta t = 0.25$ msec over 100 sec). Each step was timed and compared across the CPU - SU, CPU - MU, GPU - SU, and GPU - MU modes for 10, 80, 150, 300, and 1000 neuron configurations (Figure 5.3). In the cases of Γ, Υ, and ϕ, the CUSUMMA routine was compared against ATLAS-tuned BLAS, while the CUDA-based matrix inversion routine was compared against ATLAS-tuned LAPACK.

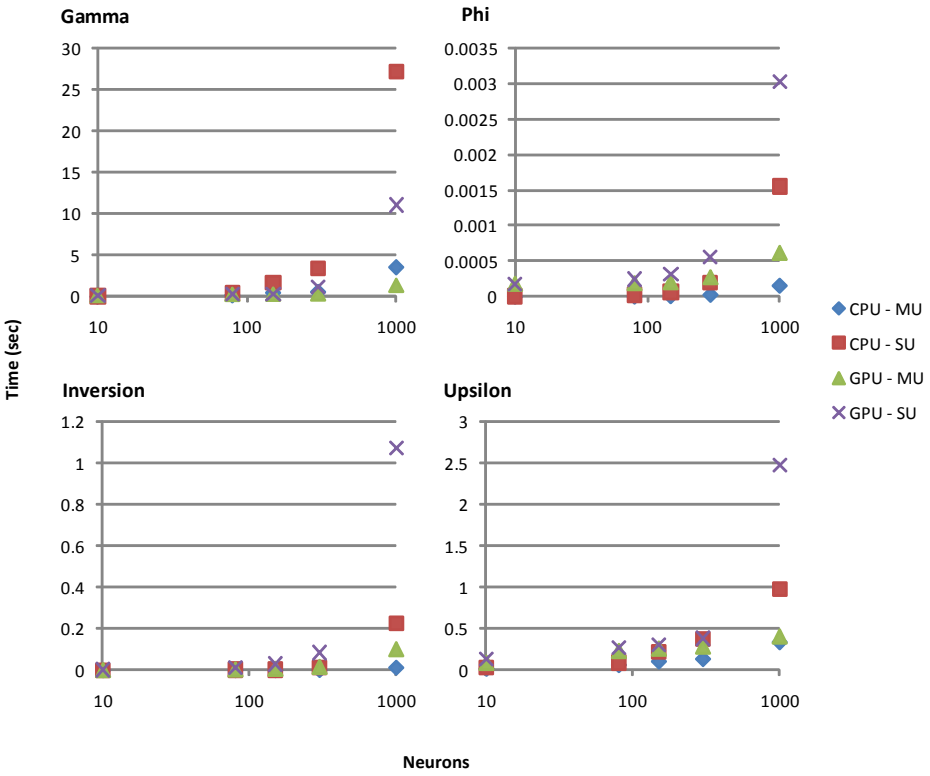

Figure 5.3: Performance comparison breakdown among the various components of the optimal decoding weight calculation process.

Interestingly, in all but the Γ calculation, the CPU version outperformed its GPU equivalent. However, this calculation dominates the total computational cost of the decoding weight process, so the GPU is able to find the optimal decoding weights faster than its equivalent CPU model provided there are at least 150 neurons in the population (Figure 5.4).

Finally, when evaluating the time to run through the entire simulation from initialization to result verification (Figure 5.5), the GPU does significantly better than the pure CPU version,

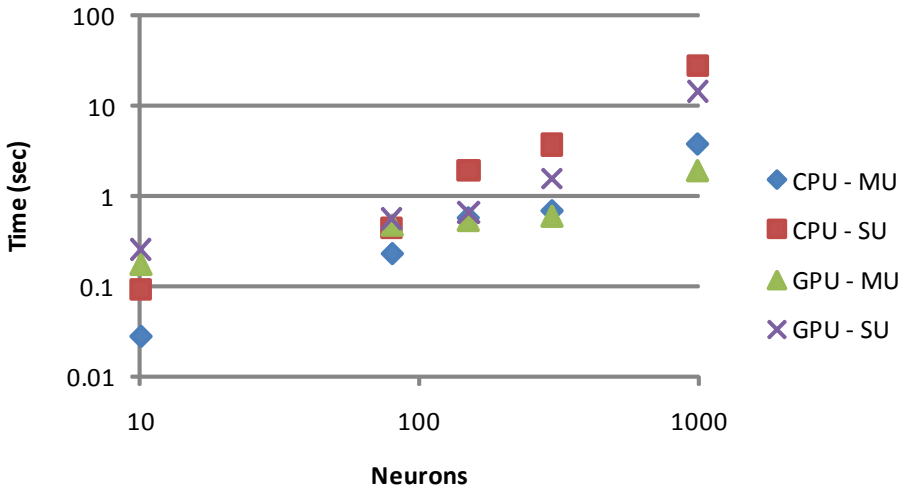

Figure 5.4: Sum of the four component times making up the process to find the optimal decoding weights.

maintaining about a 6x speedup for populations of 80 and greater. This can be attributed exclusively to the near 10x speedup gained from the convolution phases of the simulation.

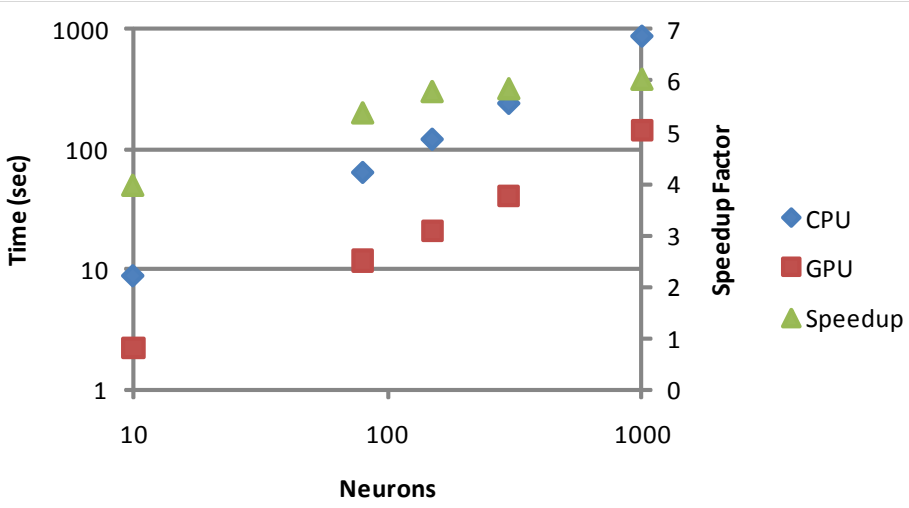

Figure 5.5: Performance of the GPU-enabled version of the simulation vs the CPU-only version. Speedup is defined as CPU/GPU.

Accuracy

The accuracy of the signal decoding process was also verified by comparing the original test signal the reconstructed signals acquired using both SU and MU methods. For a population of

150 neurons, the SU method achieved a normalized root mean square error (NRMSE) of 0.115 while the MU method achieved an NRMSE of 0.171, both comparable to the results reported by Herzfeld and Beardsley. The individual x and y components as well as the polar magnitudes and angles of the signals were plotted over time (Figure 5.6). As can be seen, there is very little difference between the original signal (black) and the SU (red) and MU (green) estimates. Only in the plot of magnitudes is there a consistently noticeable deviation from the input signal, mostly at minima and maxima, which translates into a slightly scaled version of the original 2D signal.

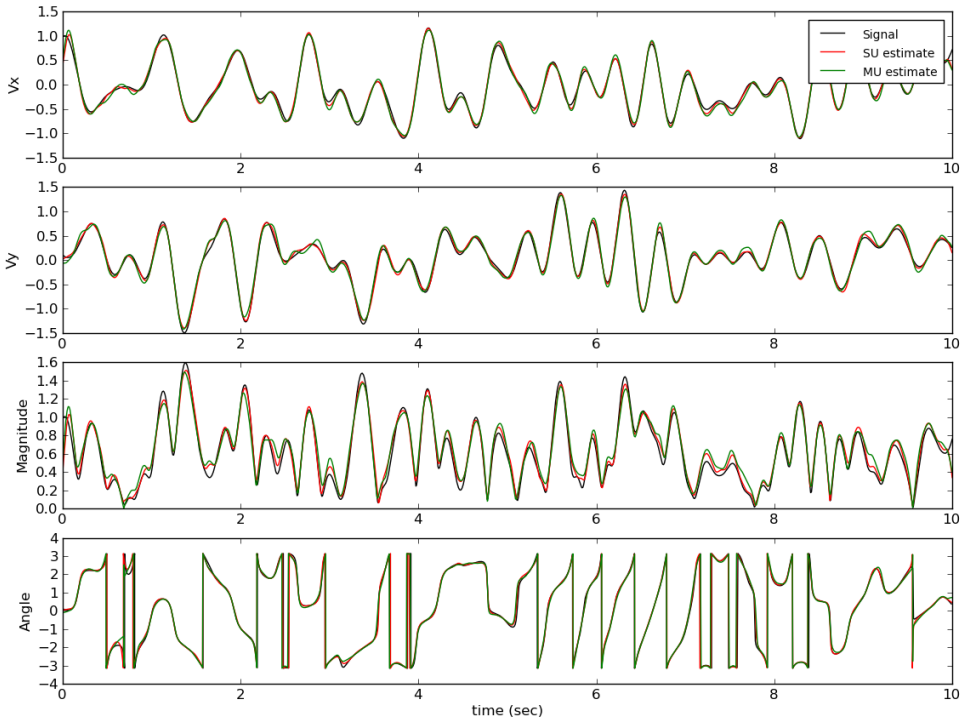

Figure 5.6: Signal reconstruction accuracy of the GPU-enabled version of the decoding algorithm. From the top, the charts depict the x and y components, the magnitude, and the angle of the 2D input signal (black) over time compared to the SU (red) and MU (green) reconstructed estimates.

5.1.3 Discussion

The simulation setup was used to validate the MU, PT decoding approach as superior to the others from a general accuracy vs computational cost standpoint. The process used to generate neural response was really just a fast and convenient way to generate the data to test the mode on.

In a real-world BMI application, adaptive or continuous applications would be required. Based on the current results, the GPU would be a positive addition if multiple large convolutions or Γ calculations were required. Otherwise, it appears that there just is not enough work for the GPU to do, or the overhead incurred outstrips any possible gains. That said, there is still utility in the current approach for exploratory testing of new or alternative methods, and the GPU definitely allows for faster runs. There are also several places where additional parallelization could be applied to the simulation, so rethinking the general approach may also increase performance.

It does, however, suggest a strong case for investigating hybrid GPU-CPU approaches. A simulation or tool that could profile itself to determine when the GPU would provide benefit over the CPU could achieve greater performance than either could separately.

5.2 CUSUMMA

As indicated previously, a main drawback to the GPU model is that, while GPU memory access is fast (141 GBps on the NVIDIA GTX280), getting data to that memory requires communication with the CPU over the PCIe bus (8Gbps peak on the PCIe x16 Gen2). Additionally, the amount of available memory to the GPU is typically a fraction of that available to the CPU, so while the dedicated compute GPUs provide 4GB of RAM, the majority of commodity GPUs available today are in the 256MB - 1GB range. As the space requirement for general matrix-matrix multiplication operations (GEMM) is $O(3(n^2))$, most GPUs will be unable to handle very large problems without some form of partitioning strategy.

One key to maximizing performance is to keep the number of data transfers on and off the device at a minimum. Due to the variety of GPU hardware configurations available, a common approach is to optimize the applications against the particular problem size and hardware constraints of the target GPU. While this can achieve good performance, the solution will more than likely not be ideal for different problem sizes or portable across a wide range of GPGPU capable devices, thus limiting the general effectiveness. I present CUSUMMA, an algorithm for achieving scalable GEMM performance on NVIDIA CUDA-enabled GPUs by automatically determining optimal submatrix partition sizes to minimize data transfers based on available GPU memory.

The name CUSUMMA comes from CUDA SUMMA, a matrix-multiplication algorithm inspired by the SUMMA algorithm [24] for CUDA-capable GPUs that minimizes the number of data transfers between the CPU and GPU. CUSUMMA achieves scalability and portability by automatically adjusting matrix partition sizes based on the problem size and amount of available GPU memory. CUSUMMA also only relies on the CUDA BLAS library, a standard part of the CUDA distribution, so any CUDA-capable GPU can run CUSUMMA without additional modifications required.

5.2.1 Matrix Multiplication

Given matrices A, B, and C of size $m \times k$, $k \times n$, and $m \times n$, respectively, the common way to perform the operation $A * B$ is to calculate each of the elements of the output C one at a time using the inner (dot) products ($\mathbf{u} \cdot \mathbf{v}$) of the rows of A and columns of B (Algorithm 2).

Algorithm 2 Matrix Multiplication - Inner Product

1: **for** $i \leftarrow 1, m$ **do**
2: **for** $j \leftarrow 1, n$ **do**
3: $C[i][j] \leftarrow A[i][1..k] * B[1..k][j]$
4: **end for**
5: **end for**

The same result can be achieved by using an outer (tensor) product ($\mathbf{u} \otimes \mathbf{v}$) approach. In this strategy columns of A are multiplied with rows of B to produce a matrix of partial values. The final answer is obtained by iterating over the shared dimension k between A and B and summing the results (Algorithm 3).

Algorithm 3 Matrix Multiplication - Outer Product

1: **for** $i \leftarrow 1, k$ **do**
2: $C \leftarrow A[1..m][i] * B[i][1..n] + C$
3: **end for**

While both techniques produce the same result and have the same number of total multiplication and addition operations, the inner product method is computationally more efficient as only one result value has to be determined at a time, so optimized multiply-accumulate operations can be employed. Furthermore, it allows for blocking approaches to add additional optimization by taking advantage of system memory hierarchies [40]. The main advantage of the

outer product method is found when attempting to calculate matrix products in a parallel, distributed environment.

Several ways to efficiently compute parallel matrix-matrix multiplication have been developed, such as the the block-cyclic approach described by the Parallel Universal Matrix Multiplication Algorithms (PUMMA) [16]. Block-cyclic techniques distribute submatrices, or blocks, of the input matrices across the different processing elements then locally computing a portion of the output at each node, cycling the blocks around the nodes until all subcalculations are performed. PUMMA, based on the inner product method, partitions and cycles blocks based on rows of A and columns of B, whereas the Scalable Universal Matrix Multiplication Algorithm (SUMMA) [24] takes the outer product method approach for partitioning the blocks. The key improvement observed with SUMMA was the lack of dependence on processing element mesh size to the overall performance, attributed to the increased flexibility of the algorithm to partition matrices. This flexibility comes from only having the single shared dimension k to iterate over as opposed to the row and column dimensions (m and n, respectively) of the output with the other approach.

5.2.2 The CUSUMMA Algorithm

CUSUMMA's scalability comes from being able to work with flexible partition sizes. Like SUMMA, CUSUMMA partitions the input and output matrices, but where SUMMA is based on distributing partitions over an arbitrary mesh of processing elements, CUSUMMA maximizes resource utilization on a single accelerator node – the GPU. By using the same summing over outer products concept, the partitions can be easily adjusted to cover the shared dimension k between the two input matrices. Adjusting the number of partitions over m or n is similar to adding additional processing elements to a mesh, as it increased the number of submatrices in the output that have to be recombined once the operation finishes.

Once the partition sizes have been determined, it is simply a matter of iterating over the partitions by the m (or n) and k dimensions, respectively. At each k partition iteration, the appropriate submatrices are isolated and transfered to the GPU where the CUBLAS SGEMM routine is performed, updating the result submatrix in the process. After each m (or n) partition iteration the finished result submatrix is transfered off the GPU and placed in the appropriate

location of the final result. For the approach described below (Algorithm 4) we assume $m \geq n$, though if the reverse is true, making the appropriate substitutions from m to n in the algorithm works just as well.

Algorithm 4 CUSUMMA

Require: $m \geq n$

procedure CUSUMMA(m, n, k, A, B, C)
  Compute m_{max}, k_{max}
  $m_{off} \leftarrow 0$
  $m_{part} \leftarrow m_{max}$
  while $m_{off} < m$ **do**
    $k_{off} \leftarrow 0$
    $k_{part} \leftarrow k_{max}$
    $\beta \leftarrow 0$
    while $k_{off} < k$ **do**
      $A_{gpu} \leftarrow A[m_{off}..m_{part}][k_{off}..k_{part}]$
      if $A = B$ **then**
        $B_{gpu} \leftarrow A_{gpu}$
      else
        $B_{gpu} \leftarrow B[k_{off}..k_{part}][0..n]$
      end if
      $C_{gpu} \leftarrow A_{gpu} * B_{gpu} + \beta C_{gpu}$
      $k_{off} \leftarrow k_{off} + k_{part}$
      if $k - k_{off} < k_{max}$ **then**
        $k_{part} \leftarrow k - k_{off}$
      end if
      $\beta \leftarrow 1$
    end while
    $C[m_{off}..m_{part}][0..n] \leftarrow C_{gpu}$
    $m_{oft} \leftarrow m_{off} + m_{part}$
    if $m - m_{off} < m_{max}$ **then**
      $m_{part} \leftarrow m - m_{off}$
    end if
  end while
end procedure

5.2.3 Determining Parameters

First, the optimal partition sizes for the input and output matrices need to be determined to minimize data transfer. Let $k_{max} \leq k$ and $m_{max} \leq m$ be the optimal sizes of the shared dimension between A and B and the leading dimensions of A, respectively. The number of partitions needed

to cover k and m individually are given by

$$t_k = \left\lceil \frac{k}{k_{max}} \right\rceil, \tag{5.4}$$

$$t_m = \left\lceil \frac{m}{m_{max}} \right\rceil. \tag{5.5}$$

The number of resultant submatrices needed is $t_m * t_k$ for A, t_k for B, and t_m for C, though each of the t_k submatrices from B will be transferred t_m times in order to properly perform the multiplication. The total transfer count can then be given as

$$t = 2t_m t_k + t_m \tag{5.6}$$

where the first term accounts for the number of input matrix transfers from the host onto the device while the second term is the number of output matrix transfers from the device back to the host. The minimum value of 3 is attained when $t_m, t_k = 1$, which corresponds to all the data being placed on the device at once (Figure 5.7a). If $t_m = 1, t_k > 1$, the entire output matrix can be kept on the card (Figure 5.7b), leaving the number of transfers proportional to t_k. Finally, when $t_m, t_k > 1$ the output matrix is not stored entirely on the device and must be computed in parts (Figure 5.7c).

As the device is constrained by the amount of total available memory, let

$$S = \frac{memory_{gpu} - memory_{work}}{memory_{elem}}$$

be the maximum number of data elements that can be stored on the device, where $memory_{gpu}$ is the total amount of device global memory in bytes available to CUSUMMA, $memory_{work}$ is an amount set aside for CUBLAS internal use, and $memory_{elem}$ is the number of bytes requires to store a single element of the data set. Minimizing t requires finding the largest integral values for k_{max} and m_{max} that satisfy

$$S \geq m_{max}k_{max} + k_{max}n + m_{max}n. \tag{5.7}$$

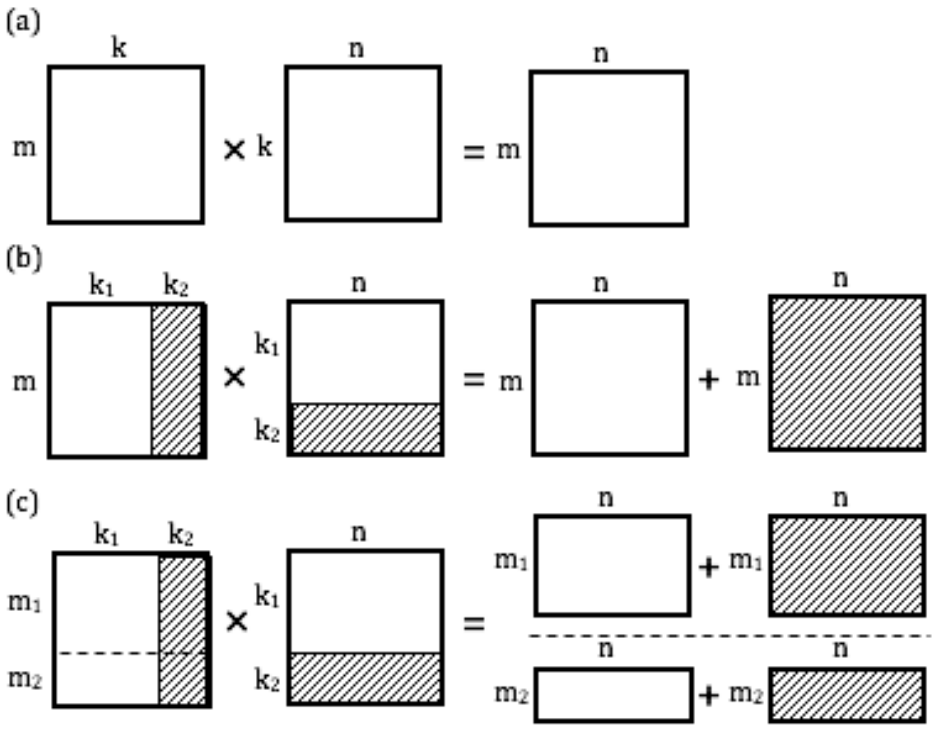

Figure 5.7: Partitioning strategies for $A * B = C$ - (a) no partitioning, (b) partitioning by shared dimension k, and (c) partitioning by leading dimension m of A and shared dimension k.

To solve this constraint minimization problem the basic assumption was made that $t_k = t_m$, which allowed for a straightforward analytical solution. This was then refined using an iterative solver based on empirical evidence.

Analytical Solution

Early block cyclic approaches such as those described by Cannon[14] and Fox[31] require a square grid of processing elements. Using a similar approach, the number of partitions needed was assumed to be square ($t_k = t_m$). k_{max} could then be rewritten as

$$k_{max} = \left\lfloor \frac{km_{max}}{m} \right\rfloor . \tag{5.8}$$

Substituting that into Equation 5.7 and solving for m_{max} produces

$$m_{max} = \left\lfloor \frac{\sqrt{4Skm + ((k+m)n)^2} - (k+m)n}{2k} \right\rfloor . \tag{5.9}$$

Thus, with just four parameters – the available space S and problem size dimensions m,n, and k – choices for k_{max} and m_{max} can be easily determined.

Iterative Solution

As t_m accounts for more of the total transfer count due to the additional term relating to retrieving the output matrix, a value as close to 1 as possible is desired. Rewriting Equation 5.7 in terms of k_{max} gives

$$k_{max} = \left\lfloor \frac{S - m_{max}n}{m_{max} + n} \right\rfloor, \tag{5.10}$$

while from Equation 5.5 we get

$$m_{max} = \left\lceil \frac{m}{t_m} \right\rceil. \tag{5.11}$$

Starting at $t_m = 1$ and working backwards, the corresponding value of t_k can be determined and t calculated. Each iteration t_m is incremented by one and the new value of t compared to the previous. When the value of t is valid (i.e. $t \geq 3$), t over t_m is roughly convex, so the iteration continues until t is no longer less than the previously recorded value (Algorithm 5).

5.2.4 Special Cases

The process detailed above is a general solution for finding the partition sizes that minimize data transfers for the $A * B$ operation. However, further optimization can be taken when looking at the $A * A^T$ operation. Here, only one copy of the partition from A needs to be transfered onto the device in order to perform the needed multiplication, provided the entire output matrix can be placed on the device (i.e. $m_{max} = m$). In this case, Equations 5.6 and 5.7 can be rewritten as

$$t = k_{max} + 1, \tag{5.12}$$

$$S \geq mk_{max} + m^2, \tag{5.13}$$

respectively. Solving Equation 5.13 in terms of k_{max} produces the optimal partition length

$$k_{max} = \frac{S}{m} - m. \tag{5.14}$$

Algorithm 5 Partition size determination

function GETPARTITIONSIZE(m, n, k, A, B)
  $S \leftarrow (memoru_{gpu} - memory_{work})/(memory_{elem})$
  if $A = B$ **then**
    $m_{max} \leftarrow m$
    $diff \leftarrow S - m * k - m * m$
    if $diff > 0$ **then**
      $k_{max} \leftarrow k$
    else
      $k_{max} \leftarrow \lfloor s/m \rfloor - m$
    end if
  else
    $t \leftarrow 0$
    $t_{last} \leftarrow 2 * m * k + m$ ▷ maximum possible transfers
    $t_m \leftarrow 0$
    while $t < t_{last}$ **do**
      if $t \geq 3$ **then** ▷ t is valid
        $t_{last} \leftarrow t$
      end if
      $t_m \leftarrow t_m + 1$
      $m_{max} \leftarrow \lceil m/t_m \rceil$
      $k_{max} \leftarrow \lfloor (S - m_{max} * n)/(m_{max} + n) \rfloor$
      $t_k \leftarrow \lceil k/k_{max} \rceil$
      $t \leftarrow t_m * t_k + t_m$
    end while
    $m_{max} \leftarrow \lceil m/(t_m - 1) \rceil$
    $k_{max} \leftarrow \lfloor (S - m_{max} * n)/(m_{max} + n) \rfloor$
  end if
end function

5.2.5 Performance Results

CUSUMMA was implemented (Listing D.1) using the CUBLAS library v2.1 and run on two test machine configurations in order to demonstrate portability. The first was an Intel Core 2 Duo E4500 clocked at 2.2GHz with 2MB cache per core, 2GB system memory, and running the 64-bit Fedora 9 Linux distribution. This machine had a Quadro FX 3700, a CUDA 1.1 device, with 112 cores, 8k registers, 512MB device memory operating at 1.24GHz, and a PCIe Gen1 connection. The second machine was an AMD Phenom 9500 Quad-Core clocked at 2.2GHz with 512MB cache per core, 8GB system memory, and running the 64-bit Fedora 10 Linux distribution. This machine had a GeForce GTX 260, a CUDA 1.3 device, with 216 cores, 16k registers, 896MB device memory operating at 1.35GHz, and a PCIe Gen2 connection. As the amount of working

memory needed by CUBLAS is not documented, $memory_{work}$ was determined experimentally to be 15MB by steadily increasing its value until performance of the algorithm was consistent across test data sets. GEMM operations in a number of matrix configurations were performed using CUSUMMA and compared to purely serial operations using ATLAS-tuned BLAS SGEMM.

In Figure 5.8, the number of transfers required based on set problem sizes using the iterative method were calculated for both architectures in order to demonstrate how the partitioning algorithm works to adapt based on both the supplied data set and available device memory. The GeForce GTX 260 had available memory approximately twice that of the Quadro FX 3700, which allowed for significantly larger partitions and thus significantly less required transfers. As can be seen, the number of iterations needed to find the minima is low, with the best answer frequently being when $t_m = 1$.

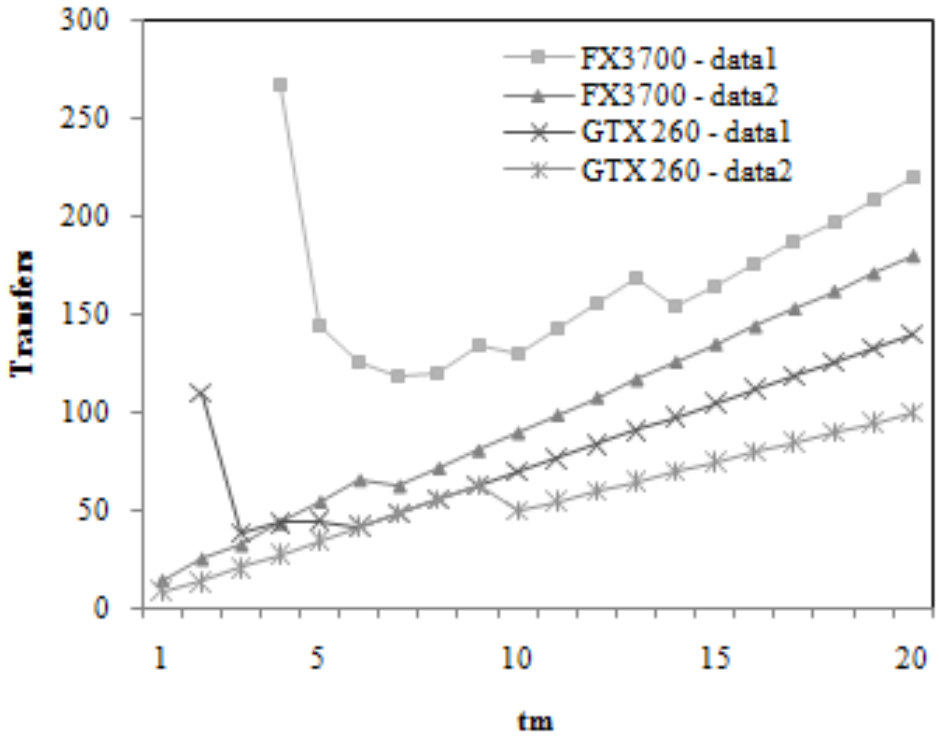

Figure 5.8: Iterative Solver - data1 is $m = n = k =$ 20k while data2 is $m = n =$ 1k, $k =$ 400k. Both sets have the same number of elements in the input but drastically different output space requirements. Missing values indicate the parameter configuration is invalid.

In Figure 5.9, the speedup of CUSUMMA compared to ATLAS-tuned BLAS for $A * B$ square matrices ($m = n = k$) is shown as performed on the AMD/GTX 260 test machine.

CUSUMMA achieves a consistent speedup of around x12 through the multiplication of two 20k $\times$ 20k matrices. Performance spikes were observed at 4k and 8k, similar to that reported by Barrachina et al.[5]. This is due to all the dimensions of the matrices being multiples of 32, the size of a warp, which follows from the fact that the hardware achieves maximum performance when all threads in a warp are active. Incorporating additional constraints that partitions should be multiples of 32 may offer additional performance improvements, though initial efforts in this direction proved inconclusive for very large matrices. It should be noted that for the GeForce GTX 260, shared dimension partitioning only comes into effect at the 9k mark, while leading dimension partitioning is not added until 15k, yet the performance is maintained.

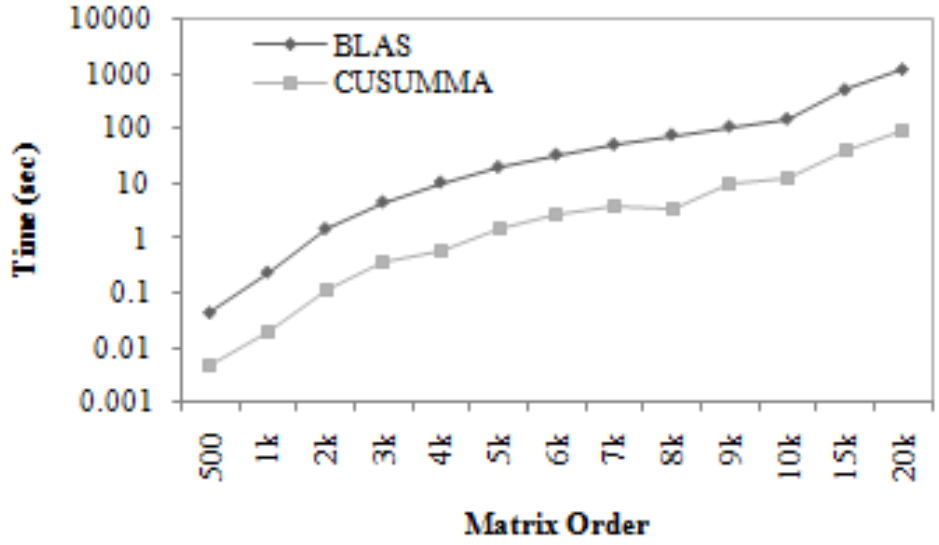

Figure 5.9: Performance of CUSUMMA vs. BLAS for square matrices.

Figure 5.10 shows the performance of CUSUMMA compared with ATLAS-tuned BLAS for rectangular matrices, where m and n are held constant at 10k and k is varied. Here again we see consistent order of magnitude performance increase that is consistent with that seen for the 10k $\times$ 10k result from before. In this trial, only shared dimension partitioning was required, though increases in total partition number did not change the behavior of the performance curve.

As CUSUMMA does not transfer all submatrices out at the beginning to a mesh of processing elements like SUMMA, but rather extracts them from the original sources when particular tiles are requested, it is dependent on the host system to copy submatrix data from the source to a temporary destination before transferring it. When dealing with very large matrices that

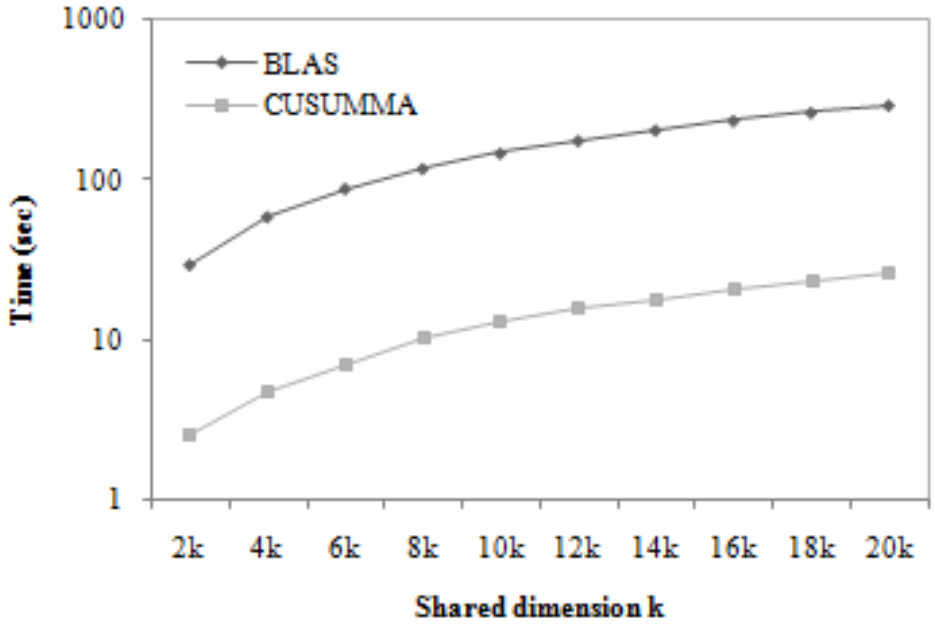

Figure 5.10: Performance of CUSUMMA vs. BLAS for rectangular matrices ($m = n =$ 10k).

can potentially exceed even system memory, paging must occur which can have a significant impact on CUSUMMA's performance. Even in these cases, CUSUMMA can still provide solid performance gains over the serial approach from the parallelization benefit alone.

5.2.6 Discussion

The performance results presented here compare the CUSUMMA algorithm, a parallel GPU-based approach, with an optimized serial BLAS routine. Not unexpectedly, the parallel implementation outperformed the serial one. While further comparisons of CUSUMMA with typical non-GPU parallel routines is needed to further vet the actual performance gains, the main outcome here is that the GPU is able to scale dynamically beyond its memory constraints without significant loss of performance. Whether CUSUMMA's outer product partitioning approach really yields the best GPU solution also needs to be explored, though it can be shown that the number of data transfers needed to execute a typical inner product partitioning scheme greatly exceeds that of CUSUMMA for large problem sizes. Additional work is needed to flesh out the specific performance profiles of these various approaches in order to make a strong statement regarding CUSUMMA as an optimal solution.

CHAPTER 6

Conclusions And Future Work

The implementation of three neural network models on the GPU was presented in this work. The Self-Organizing Map demonstrated proof of concept for accelerating neural networks using the GPU. In addition to providing a first pass at the benefits and challenges of parallelizing algorithms on the GPU, the SOM also served as a test case for visualizing the internal activity of a neural network in real-time. The spiking neural network model described in Chapter 4 took the one thread per neuron strategy employed by the SOM and extended it to a full agent-inspired model of biological neural activity. The neuron model was increased in complexity to mimic biology by adding spike-based communication between connected neurons. This model was compared against a similar version that used MPI instead of the GPU for parallelization, with positive and encouraging results. Using the visualization techniques developed with the SOM, the spiking activity of neural populations was also able to be observed in real-time. Finally, in Chapter 5, the focus shifted toward efforts to decode information represented by the spiking response of a population of neurons. Unlike the previous two chapters, the role of the GPU in the decoding operation was to increase the performance of the process used to obtain neural linear decoders. Once obtained, this transform could be applied to the neural response to reconstruct a stimulus signal. As with the spiking neural network results, the decoding process on the GPU showed considerable gains over an equivalent optimized serial process.

A key result of both the spiking neural network and decoding simulations was that the GPU can provide significant performance and functional benefits, but only under certain conditions. Due to its inherent overhead in launching and executing kernels, the data sizes presented to the GPU need to be large enough for the gain from parallelization to become apparent.

On the other hand, if the data sizes are too large, the GPU cannot handle them without additional help from the programmer. The GPU is not a panacea, nor is achieving optimal performance something that comes after the first or second round of optimization efforts. It does, however, contain tremendous potential for certain application areas such as portable high-performance computing or as a low cost alternative to compute nodes for researchers on a budget.

The recurrent spiking neural network simulator has a lot of promise. The strict adherence to the agent-inspired concept drove development and resulted in respectable performance. However, moving away from the restriction of one-to-one mappings between threads and biological elements would allow for new code optimizations and constructs that could increase performance without sacrificing the current conceptual benefits. The massively parallel nature of the GPU coupled with the shared memory system makes it an ideal platform for more neuromorphic or neuro-inspired computing applications.

Bibliography

[1] Java bindings for cuda, 5 2010.

[2] ABBOTT, L. F. Lapicque's introduction of the integrate-and-fire model neuron (1907). *Brain Research Bulletin 50*, 5-6 (1999), 303–304.

[3] ANANTHANARAYANAN, R., AND MODHA, D. S. Anatomy of a cortical simulator. SC07.

[4] ARYA, S., MOUNT, D. M., NETANYAHU, N. S., SILVERMAN, R., AND WU, A. Y. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. In *ACM-SIAM SYMPOSIUM ON DISCRETE ALGORITHMS* (1994), pp. 573–582.

[5] BARRACHINA, S., CASTILLO, M., IGUAL, F. D., MAYO, R., AND QUINTANA-ORTI, E. S. Evaluation and tuning of the level 3 cublas for graphics processors. In *IPDPS* (2008), IEEE, pp. 1–8.

[6] BENETIS, R., JENSEN, C. S., AND ET AL. Nearest neighbor and reverse nearest neighbor queries for moving objects, 2002.

[7] BENTLEY, J. L. Multidimensional binary search trees used for associative searching. *Commun. ACM 18*, 9 (September 1975), 509–517.

[8] BERNHARD, F., AND KERIVEN, R. Spiking neurons on gpus. 2006, pp. 236–243.

[9] BLACKFORD, L. S., DEMMEL, J., DONGARRA, J., DUFF, I., HAMMARLING, S., HENRY, G., HEROUX, M., KAUFMAN, L., LUMSDAINE, A., PETITET, A., POZO, R., REMINGTON, K., AND WHALEY, R. C. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software 28* (2001), 135–151.

[10] BONABEAU, E. Agent-based modeling: methods and techniques for simulating human systems. *Proceedings of the National Academy of Sciences of the United States of America 99 Suppl 3* (May 2002), 7280–7287.

[11] BOWER, J. M., AND BEEMAN, D. *The book of GENESIS (2nd ed.): exploring realistic neural models with the GEneral NEural SImulation System*. Springer-Verlag New York, Inc., New York, NY, USA, 1998.

[12] BRETTE, R., AND GERSTNER, W. Adaptive exponential integrate-and-fire model as an effective description of neuronal activity. *Journal of Neurophysiology 94*, 5 (November 2005), 3637–3642.

[13] BRETTE, R., RUDOLPH, M., CARNEVALE, T., HINES, M., BEEMAN, D., BOWER, J., DIESMANN, M., MORRISON, A., GOODMAN, P., HARRIS, F., ZIRPE, M., NATSCHLÄGER, T., PECEVSKI, D., ERMENTROUT, B., DJURFELDT, M., LANSNER, A., ROCHEL, O., VIEVILLE, T., MULLER, E., DAVISON, A., EL BOUSTANI, S., AND DESTEXHE, A. Simulation of networks of spiking neurons: A review of tools and strategies. *Journal of Computational Neuroscience 23*, 3 (December 2007), 349–398.

[14] CANNON, L. E. *A cellular computer to implement the kalman filter algorithm*. PhD thesis, Bozeman, MT, USA, 1969.

[15] CHAPMAN, B., JOST, G., AND VAN DER PAS, R. *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, October 2007.

[16] CHOI, J., DONGARRA, J. J., AND WALKER, D. W. Pumma: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers. Tech. rep., 1993.

[17] DAVISON, A., BRUDERLE, D., EPPLER, J., KREMKOW, J., MULLER, E., PECEVSKI, D., PERRINET, L., AND YGER, P. Pynn: a common interface for neuronal network simulators. *Frontiers in Neuroinformatics 2* (2008).

[18] DIESMANN, M., PLANCK INST FR STRMUNGSFORSCHUNG, M., AND OLIVER GEWALTIG, M. Nest: An environment for neural systems simulations. In *In T. Plesser and V. Macho (Eds.), Forschung und wisschenschaftliches Rechnen, Beitrage zum Heinz-Billing-Preis 2001, Volume 58 of GWDG-Bericht* (2002), pp. 43–70.

[19] DORNHEGE, G. *Toward Brain-Computer Interfacing (Neural Information Processing)*. The MIT Press, September 2007.

[20] DOYA, K., ISHII, S., POUGET, A., AND RAO, R. P. N., Eds. *Bayesian Brain: Probabilistic Approaches to Neural Coding*, 1 ed. The MIT Press, January 2007.

[21] ELIASMITH, C., AND ANDERSON, C. H. *Neural Engineering: Computation, Representation, and Dynamics in Neurobiological Systems (Computational Neuroscience)*, new edition ed. The MIT Press, September 2004.

[22] ERMENTROUT, G. B., AND KOPELL, N. Parabolic bursting in an excitable system coupled with a slow oscillation. *SIAM Journal on Applied Mathematics 46*, 2 (1986), 233–253.

[23] FRIGO, M., STEVEN, AND JOHNSON, G. The design and implementation of fftw3. In *Proceedings of the IEEE* (2005), pp. 216–231.

[24] GEIJN, R. A. V. D., AND WATTS, J. Summa: Scalable universal matrix multiplication algorithm. Tech. rep., Concurrency: Practice and Experience, 1995.

[25] GEORGOPOULOS, A. P., KALASKA, J. F., CAMINITI, R., AND MASSEY, J. T. On the relations between the direction of two-dimensional arm movements and cell discharge in primate motor cortex. *The Journal of neuroscience : the official journal of the Society for Neuroscience 2*, 11 (November 1982), 1527–1537.

[26] GODDARD, N. H., HUCKA, M., HOWELL, F., CORNELIS, H., SHANKAR, K., AND BEEMAN, D. Towards neuroml: model description methods for collaborative modelling in neuroscience. *Philos Trans R Soc Lond B Biol Sci 356*, 1412 (August 2001), 1209–1228.

[27] GOODMAN, D. Brian: a simulator for spiking neural networks in python. *Frontiers in Neuroinformatics 2* (2008).

[28] GROPP, W., HUSS-LEDERMAN, S., LUMSDAINE, A., LUSK, E., NITZBERG, B., SAPHIR, W., AND SNIR, M. *MPI: The Complete Reference*, 2nd ed. The MIT Press, September 1998.

[29] GUTTMAN, A. R-trees: a dynamic index structure for spatial searching. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD international conference on Management of data* (New York, NY, USA, June 1984), vol. 14, ACM, pp. 47–57.

[30] HERZFELD, D. J., AND BEARDSLEY, S. A. Improved multi-unit decoding at the brain-machine interface using population temporal linear filtering. *Journal of Neural Engineering 7* (2010).

[31] HEY, A. J. G., FOX, G. C., AND OTTO, S. W. Matrix algorithms on a hypercube 1: Matrix multiplication. Parallel Computing 1, 17.

[32] HINES, M. L., AND CARNEVALE, N. T. The neuron simulation environment. *Neural computation 9*, 6 (August 1997), 1179–1209.

[33] HODGKIN, A. L., AND HUXLEY, A. F. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of physiology 117*, 4 (August 1952), 500–544.

[34] IZHIKEVICH, E. M. Simple model of spiking neurons. *IEEE transactions on neural networks / a publication of the IEEE Neural Networks Council 14*, 6 (November 2003), 1569–1572.

[35] JAROSZ, Q. Neuron hand-tuned [licensed under cc-by-sa-3.0].

[36] KANDEL, E. R., SCHWARTZ, J. H., AND JESSELL, T. M. *Principles of Neural Science*. McGraw-Hill Medical, January 2000.

[37] KIRK, D. B., AND HWU, W.-M. W. *Programming Massively Parallel Processors: A Hands-on Approach*, 1 ed. Morgan Kaufmann, February 2010.

[38] KLOCKNER, A., PINTO, N., LEE, Y., CATANZARO, B., IVANOV, P., AND FASIH, A. Pycuda: Gpu run-time code generation for high-performance computing. *Parallel Computing* (In Process).

[39] KOHONEN, T. Self-organized formation of topologically correct feature maps. *Biological Cybernetics 43*, 1 (January 1982), 59–69.

[40] LAM, M. S., ROTHBERG, E. E., AND WOLF, M. E. The cache performance and optimizations of blocked algorithms. In *In Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (1991), pp. 63–74.

[41] MAYNARD, E. M., NORDHAUSEN, C. T., AND NORMANN, R. A. The utah intracortical electrode array: a recording structure for potential brain-computer interfaces. *Electroencephalography and clinical neurophysiology 102*, 3 (March 1997), 228–239.

[42] MUNSHI, A., Ed. *The OpenCL Specification*. Khronos OpenCL Working Group, 2009.

[43] NAGESWARAN, J. M., DUTT, N., KRICHMAR, J. L., NICOLAU, A., AND VEIDENBAUM, A. V. A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors. *Neural Networks 22*, 5-6 (2009), 791 – 800. Advances in Neural Networks Research: IJCNN2009, 2009 International Joint Conference on Neural Networks.

[44] NVIDIA. *CUDA CUBLAS Library 3.0*. NVIDIA Corporation, 2010.

[45] NVIDIA. *CUDA CUFFT Library 3.0*. NVIDIA Corporation, 2010.

[46] NVIDIA. *CUDA Programming Guide 3.0*. NVIDIA Corporation, 2010. Reference for permission to use: http://forums.nvidia.com/index.php?showtopic=162107.

[47] SHERIDAN, R. Wallpaper: Year zero #3 (standard) [licensed under cc-by-nc-2.0], 11 2008.

[48] SLEATOR, D. D., AND TARJAN, R. E. Self-adjusting binary search trees. *J. Assoc. Comput. Mach. 32*, 3 (1985), 652–686.

[49] VESANTO, J., HIMBERG, J., ALHONIEMI, E., AND PARHANKANGAS, J. Self-organizing map in matlab: the som toolbox. In *In Proceedings of the Matlab DSP Conference* (2000), pp. 35–40.

[50] WILSON, B. S., AND DORMAN, M. F. Cochlear implants: a remarkable past and a brilliant future. *Hearing research 242*, 1-2 (August 2008), 3–21.

[51] WRIGHT, R. S., LIPCHAK, B., AND HAEMEL, N. *OpenGL(R) SuperBible: Comprehensive Tutorial and Reference (4th Edition) (OpenGL)*. Addison-Wesley Professional, June 2007.

[52] ZEZULA, P., AMATO, G., DOHNAL, V., AND BATKO, M. *Similarity Search: The Metric Space Approach (Advances in Database Systems)*. Springer, November 2005.

[53] ZHOU, K., HOU, Q., WANG, R., AND GUO, B. Real-time kd-tree construction on graphics hardware. *ACM Trans. Graph. 27*, 5 (2008), 1–11.

APPENDIX A

CUDA Hello World Code Listing

```
#include <cuda.h>
#include <stdio.h>

// Prototypes
__global__ void helloWorld(char*);

// Host function
int
main(int argc, char** argv)
{
  int i;

  // desired output
  char str[] = "Hello World!";

  // mangle contents of output
  // the null character is left intact for simplicity
  for(i = 0; i < 12; i++)
    str[i] -= i;

  // allocate memory on the device
  char *d_str;
  size_t size = sizeof(str);
  cudaMalloc((void**)&d_str, size);

  // copy the string to the device
  cudaMemcpy(d_str, str, size, cudaMemcpyHostToDevice);

  // set the grid and block sizes
  dim3 dimBlock(6); // threads per block
  dim3 dimGrid(2);  // blocks per grid

  // invoke the kernel
  helloWorld<<< dimGrid, dimBlock >>>(d_str);

  // retrieve the results from the device
  cudaMemcpy(str, d_str, size, cudaMemcpyDeviceToHost);

  // free up the allocated memory on the device
  cudaFree(d_str);

  // everyone's favorite part
  printf("%s\n", str);
```

```
  return 0;
}

// Device kernel
__global__ void
helloWorld(char* str)
{
  // determine where in the thread grid we are
  int idx = blockIdx.x * blockDim.x + threadIdx.x;

  // unmangle output
  str[idx] += idx;
}
```

Listing A.1: Hello World example using CUDA

APPENDIX B

SOM Visualization Code Listings

The visualization code was inspired from and uses parts of the `imageDenoising` example in the CUDA SDK.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cublas.h>
#include "selfOrganizingMap.h"

__device__ float vecLen(float4 a, float4 b) {
    return (
        (b.x - a.x) * (b.x - a.x) +
        (b.y - a.y) * (b.y - a.y) +
        (b.z - a.z) * (b.z - a.z)
    );
}

__device__ TColor make_color(float r, float g, float b, float a) {
    return
        ((int)(a * 255.0f) << 24) |
        ((int)(b * 255.0f) << 16) |
        ((int)(g * 255.0f) <<  8) |
        ((int)(r * 255.0f) <<  0);
}

//Texture reference and channel descriptor for image texture
texture<uchar4, 2, cudaReadModeNormalizedFloat> texImage;
cudaChannelFormatDesc uchar4tex = cudaCreateChannelDesc<uchar4>();

//CUDA array descriptor
cudaArray *a_Src;

extern "C"
cudaError_t CUDA_Bind2TextureArray()
{
    return cudaBindTextureToArray(texImage, a_Src);
}

extern "C"
cudaError_t CUDA_UnbindTexture()
{
    return cudaUnbindTexture(texImage);
}
```

```
extern "C"
cudaError_t CUDA_MallocArray(uchar4 **som, int somW, int somH)
{
    cudaError_t error;

    error = cudaMallocArray(&a_Src, &uchar4tex, somW, somH);
    error = cudaMemcpyToArray(a_Src, 0, 0,
                              *som, somW * somH * sizeof(uchar4),
                              cudaMemcpyHostToDevice
                              );

    return error;
}

extern "C"
cudaError_t CUDA_FreeArray()
{
    return cudaFreeArray(a_Src);
}

__global__ void
findDistance(float *dist, float4 p, int somW)
{
  const int ix = blockDim.x * blockIdx.x + threadIdx.x;
  const int iy = blockDim.y * blockIdx.y + threadIdx.y;
  //Add half of a texel to always address exact texel centers
  const float x = (float)ix + 0.5f;
  const float y = (float)iy + 0.5f;

  const int idx = iy * somW + ix;

  float4 w = tex2D(texImage, x, y);

  dist[idx] = vecLen(w,p);
}

__global__ void
updateWeights(TColor *som, uchar4 *buffer, uint2 bmu, float4 p, float radius,
    float alpha, int somW)
{
  const int ix = blockDim.x * blockIdx.x + threadIdx.x;
  const int iy = blockDim.y * blockIdx.y + threadIdx.y;
  //Add half of a texel to always address exact texel centers
  const float x = (float)ix + 0.5f;
  const float y = (float)iy + 0.5f;

  int idx = iy * somW + ix;

  int u = ix - bmu.x;
  int v = iy - bmu.y;
  int d = u*u + v*v;

  if(d < radius) {
    float4 w = tex2D(texImage, x, y);
    float wx = w.x + alpha*(p.x - w.x);
    float wy = w.y + alpha*(p.y - w.y);
    float wz = w.z + alpha*(p.z - w.z);
```

```
    buffer[idx] = make_uchar4((int)(wx*255.0f),(int)(wy*255.0f),(int)(wz*255.0f)
        ,0);
    som[idx] = make_color(wx, wy, wz, 0.0f);
  }
}

extern "C" void
cuda_SOM(TColor *som, uchar4 *buffer, float *dist, float radius, float4 p, int
    somW, int somH)
{
  dim3 block(16,16);
  dim3 grid(somW / block.x, somH / block.y);

  findDistance<<< grid, block >>>(dist, p, somW);
  cudaThreadSynchronize();

  int minDist = cublasIsamin(somW * somH, dist, 1);
  uint2 bmu = make_uint2(minDist % somW, minDist / somW);
  updateWeights<<< grid, block >>>(som, buffer, bmu, p, radius, 0.05f, somW);
  cudaThreadSynchronize();

  cudaMemcpyToArray(a_Src, 0, 0, buffer, somW * somH * sizeof(uchar4),
      cudaMemcpyDeviceToDevice);
}
```

Listing B.1: `selfOrganizingMap.cu` - SOM CUDA kernels

```
#ifndef SELF_ORGANIZING_MAP_H
#define SELF_ORGANIZING_MAP_H

typedef unsigned int TColor;

// functions to load images
extern "C" void LoadBMPFile(uchar4 **dst, int *width, int *height, const char *
    name);

// CUDA wrapper functions for allocation/freeing texture arrays
extern "C" cudaError_t CUDA_Bind2TextureArray();
extern "C" cudaError_t CUDA_UnbindTexture();
extern "C" cudaError_t CUDA_MallocArray(uchar4 **som, int somW, int somH);
extern "C" cudaError_t CUDA_FreeArray();

// CUDA kernel functions
extern "C" void cuda_Copy( TColor *d_dst, int somW, int somH);
extern "C" void cuda_SOM( TColor *d_dst, uchar4 *buffer, float *dist, float
    radius, float4 p, int somW, int somH);

#endif
```

Listing B.2: `selfOrganizingMap.h` - SOM CUDA headers

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <GL/glew.h>
#include <cuda_runtime.h>
#include <cublas.h>
#include <cutil.h>
#include <cutil_gl_error.h>
#include <cuda_gl_interop.h>
#include <GL/glut.h>

#include "selfOrganizingMap.h"

////////////////////////////////////////////////////////////////////////////////
// Global data handlers and parameters
////////////////////////////////////////////////////////////////////////////////
//OpenGL PBO and texture "names"
GLuint gl_PBO, gl_Tex;
//Source image on the host side
uchar4 *som, *buffer;
int somW, somH, imageSize, pixel;
float *dist, radius;
unsigned char *image;
FILE *fp;

////////////////////////////////////////////////////////////////////////////////
// Main program
////////////////////////////////////////////////////////////////////////////////
int  g_Kernel = 0;
bool    g_FPS = false;
bool   g_Diag = false;

const int frameN = 24;
int frameCounter = 0;


#define BUFFER_DATA(i) ((char *)0 + i)

// Auto-Verification Code
int fpsCount = 0;        // FPS count for averaging
int fpsLimit = 1;        // FPS limit for sampling
unsigned int frameCount = 0;
unsigned int g_TotalErrors = 0;
bool g_Verify = false, g_AutoQuit = false;

void displayFunc(void){
    TColor *d_dst = NULL;

    CUDA_SAFE_CALL( cudaGLMapBufferObject((void**)&d_dst, gl_PBO) );

    CUDA_SAFE_CALL( CUDA_Bind2TextureArray()                      );

    float4 p;
    p.z = image[pixel]     / 255.0f;
    p.y = image[pixel + 1] / 255.0f;
    p.x = image[pixel + 2] / 255.0f;
    p.w = 1.0f;
```

```
    pixel += 3;
    if(pixel >= imageSize) {
      radius = radius <= 1 ? 1.0f : powf(sqrtf(radius)-1.0,2);
      pixel = 0;
    }

    cuda_SOM(d_dst, buffer, dist, radius, p, somW, somH);

    CUDA_SAFE_CALL( CUDA_UnbindTexture()     );
    CUDA_SAFE_CALL( cudaGLUnmapBufferObject(gl_PBO) );

    glTexSubImage2D( GL_TEXTURE_2D, 0, 0, 0, somW, somH, GL_RGBA,
        GL_UNSIGNED_BYTE, BUFFER_DATA(0) );
    glBegin(GL_TRIANGLES);
        glTexCoord2f(0, 0); glVertex2f(-1, -1);
        glTexCoord2f(2, 0); glVertex2f(+3, -1);
        glTexCoord2f(0, 2); glVertex2f(-1, +3);
    glEnd();
    glFinish();

glutSwapBuffers();

glutPostRedisplay();
}


void shutDown(unsigned char k, int /*x*/, int /*y*/)
{
    switch (k){
        case '\033':
        case 'q':
        case 'Q':
            printf("Shutting down...\n");
                CUDA_SAFE_CALL( cudaGLUnregisterBufferObject(gl_PBO) );
                glBindBuffer(GL_PIXEL_UNPACK_BUFFER_ARB, 0);
                glDeleteBuffers(1, &gl_PBO);
                glDeleteTextures(1, &gl_Tex);

                CUDA_SAFE_CALL( CUDA_FreeArray() );
                free(som);
                //fclose(fp);
            printf("Shutdown done.\n");
            exit(0);
        break;
    }
}



int main(int argc, char **argv){

    CUT_DEVICE_INIT(argc, argv);

    int seed;

    seed = atoi(argv[1]);

    srand(seed);
```

```
somH = 512;
somW = 512;

som = (uchar4*) malloc(somH * somW * sizeof(uchar4));
for(int i = 0; i < somH * somW; i++) {
    som[i].x = rand() % 256;
    som[i].y = rand() % 256;
    som[i].z = rand() % 256;
    som[i].w = 255;
}

printf("Loading␣in␣image␣file...\n");
fp = fopen(argv[2],"rb");
if(fp == 0) {
  printf("file␣not␣opened\n");
  return 1;
}
fseek(fp, 0, SEEK_END);
imageSize = ftell(fp) - 53;
rewind(fp);
fseek(fp, 54, SEEK_SET);
image = (unsigned char*) malloc(imageSize * sizeof(unsigned char));
fread(image, 1, imageSize, fp);
fclose(fp);
pixel = 0;
radius = 256.0f;

printf("Allocating␣host␣and␣CUDA␣memory...\n");

CUDA_SAFE_CALL( CUDA_MallocArray(&som, somW, somH) );

CUDA_SAFE_CALL(cudaMalloc((void**)&dist, somW * somH * sizeof(float)));
CUDA_SAFE_CALL(cudaMalloc((void**)&buffer, somW*somH*sizeof(uchar4)));
CUDA_SAFE_CALL(cudaMemcpy(buffer, som, somW*somH*sizeof(uchar4),
    cudaMemcpyHostToDevice));
printf("Data␣init␣done.\n");

printf("Initializing␣GLUT...\n");
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGBA | GLUT_SINGLE);
    glutInitWindowSize(somW, somH);
    glutInitWindowPosition(512 - somW / 2, 384 - somH / 2);
    glutCreateWindow(argv[0]);
    printf("Loading␣extensions:␣%s\n", glewGetErrorString(glewInit()));
    if(!glewIsSupported(
        "GL_VERSION_2_0␣"
        "GL_ARB_pixel_buffer_object␣"
        "GL_EXT_framebuffer_object␣"
    )){
        fprintf(stderr, "ERROR:␣Support␣for␣necessary␣OpenGL␣extensions␣
            missing.");
        fflush(stderr);
        return CUTFalse;
    }
printf("OpenGL␣window␣created.\n");

printf("Creating␣GL␣texture...\n");
    glEnable(GL_TEXTURE_2D);
    glGenTextures(1, &gl_Tex);
```

```
        glBindTexture(GL_TEXTURE_2D, gl_Tex);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, somW, somH, 0, GL_RGBA,
            GL_UNSIGNED_BYTE, som);
    printf("Texture␣created.\n");

    printf("Creating␣PBO...\n");
        glGenBuffers(1, &gl_PBO);
        glBindBuffer(GL_PIXEL_UNPACK_BUFFER_ARB, gl_PBO);
        glBufferData(GL_PIXEL_UNPACK_BUFFER_ARB, somW * somH * 4, som,
            GL_STREAM_COPY);
        //While a PBO is registered to CUDA, it can't be used
        //as the destination for OpenGL drawing calls.
        //But in our particular case OpenGL is only used
        //to display the content of the PBO, specified by CUDA kernels,
        //so we need to register/unregister it only once.
        CUDA_SAFE_CALL( cudaGLRegisterBufferObject(gl_PBO) );
        CUT_CHECK_ERROR_GL();
    printf("PBO␣created.\n");

    printf("Starting␣GLUT␣main␣loop...\n");
    printf("Press␣[q]␣to␣exit\n");

    glutSetWindowTitle("Self␣Organizing␣Map");
    glutIdleFunc(displayFunc);
    glutDisplayFunc(displayFunc);
    glutKeyboardFunc(shutDown);
    glutMainLoop();

    CUT_EXIT(argc, argv);
}
```

Listing B.3: `selfOrganizingMap.cu` - SOM CUDA main OpenGL methods

APPENDIX C

Spiking Neural Network Visualization Code Listings

The OpenGL visualization code was inspired from and uses parts of the `imageDenoising` example in the CUDA SDK.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <float.h>
#include <math.h>

#include "spikingNeuralNetwork.h"


#include "gpu_snn_kernels.cu"

////////////////////////////////////////////////////////////////////////////////
// Helper functions
////////////////////////////////////////////////////////////////////////////////
__device__ unsigned int
make_color(float r, float g, float b, float a){
    return
        ((int)(a * 255.0f) << 24) |
        ((int)(b * 255.0f) << 16) |
        ((int)(g * 255.0f) <<  8) |
        ((int)(r * 255.0f) <<  0);
}


////////////////////////////////////////////////////////////////////////////////
// Global data handlers and parameters
////////////////////////////////////////////////////////////////////////////////
//Texture reference and channel descriptor for image texture
texture<uchar4, 2, cudaReadModeNormalizedFloat> texImage;
cudaChannelFormatDesc uchar4tex = cudaCreateChannelDesc<uchar4>();

//CUDA array descriptor
cudaArray *a_Src;

////////////////////////////////////////////////////////////////////////////////
// Filtering kernels
////////////////////////////////////////////////////////////////////////////////
extern "C" cudaError_t
CUDA_Bind2TextureArray()
{
  return cudaBindTextureToArray(texImage, a_Src);
```

```
}

extern "C" cudaError_t
CUDA_UnbindTexture()
{
  return cudaUnbindTexture(texImage);
}

extern "C" cudaError_t
CUDA_MemcpyToSymbols(int nNeurons, int nSynapses, int maxPSC, float dt)
{
  cudaError_t error;
  error = cudaMemcpyToSymbol(NEURONS, &nNeurons, sizeof(int));
  error = cudaMemcpyToSymbol(SYNAPSES, &nSynapses, sizeof(int));
  error = cudaMemcpyToSymbol(PSC_BUFFER_LENGTH, &maxPSC, sizeof(int));
  error = cudaMemcpyToSymbol(DT, &dt, sizeof(float));

  return error;
}

extern "C" cudaError_t
CUDA_MallocArray(uchar4 **rasterPlot, int windowWidth, int windowHeight)
{
  cudaError_t error;

  error = cudaMallocArray(&a_Src, &uchar4tex, windowWidth, windowHeight);
  error = cudaMemcpyToArray(a_Src, 0, 0,
                            *rasterPlot, windowWidth * windowHeight * sizeof(
                                uchar4),
                            cudaMemcpyHostToDevice
                            );

  return error;
}

extern "C" cudaError_t
CUDA_FreeArray()
{
  return cudaFreeArray(a_Src);
}

__global__ void
updateRasterPlot(unsigned int *rasterPlot, uchar4 *plotBuffer, Axon *dAxons, int
     windowWidth, int windowHeight)
{
  const int ix = blockDim.x * blockIdx.x + threadIdx.x;
  const int iy = blockDim.y * blockIdx.y + threadIdx.y;

  if(ix >= windowWidth || iy >= windowHeight) return;

  //Add half of a texel to always address exact texel centers
  const float x = (float)ix + 1.5f;
  const float y = (float)iy + 0.5f;

  int idx = iy * windowWidth + ix;
  float p;

  if(ix == windowWidth - 1) {
    p = dAxons[windowHeight - 1 - iy].spike ? 0.0f : 1.0f;
```

```
  }
  else {
    float4 w = tex2D(texImage, x, y);
    p = w.z;
  }

  int q = (int)(255.0f*p);
  plotBuffer[idx] = make_uchar4(q,q,q,0);
  rasterPlot[idx] = make_color(p,p,p,0.0f);
}

extern "C" void
CUDA_UpdateRasterPlot(unsigned int *rasterPlot, uchar4 *plotBuffer, Axon *dAxons
    , int windowWidth, int windowHeight)
{
  dim3 block(16,16);
  dim3 grid(ceil(1.0*windowWidth / block.x), ceil(1.0*windowHeight / block.y));

  updateRasterPlot<<< grid, block >>>(rasterPlot, plotBuffer, dAxons,
      windowWidth, windowHeight);
  cudaThreadSynchronize();

  cudaMemcpyToArray(a_Src, 0, 0, plotBuffer, windowWidth * windowHeight * sizeof
      (uchar4),
                            cudaMemcpyDeviceToDevice
                            );
}

extern "C" void
CUDA_AdvanceSimulation(float *dStimulus,
                       float *hStimulus,
                       size_t stimSize,
                       Soma *dSoma,
                       TuningProfile *dProfiles,
                       Dendrite *dDendrites,
                       Neuron *dNeurons,
                       Axon *dAxons,
                       LIFNeuron *dLIFNeurons,
                       Monitor *dMonitors,
                       int step,
                       Simulation *sim,
                       Synapse *dSynapses,
                       float *dPSCFilter,
                       float *dPSCBuffer)
{
  int dimX = 1;
  int dimY = 1;
  if(sim->nSynapses <= 65535) {
    dimX = sim->nSynapses;
  }
  else {
    dimX = ceil(sqrt(sim->nSynapses));
    dimY = dimX;
  }
  dim3 synapseBlock(sim->pscFilterSize,1,1);
  dim3 synapseGrid(dimX,dimY,1);
  dim3 neuronBlock(64,1,1);
  dim3 neuronGrid(ceil(sim->nNeurons/64.0),1,1);
```

```
  // apply external stimulus to neurons
  cudaMemcpy(dStimulus, hStimulus, stimSize, cudaMemcpyHostToDevice);
  applyStimulus<<<neuronGrid, neuronBlock>>>(dSoma, dProfiles, dStimulus);
  cudaThreadSynchronize();

  // sum spike-related input currents
  updateDendrites<<<neuronGrid, neuronBlock>>>(dDendrites, dSoma, dNeurons);
  cudaThreadSynchronize();

  // update neuron membrane voltages and emit spikes
  updateSomas<<<neuronGrid, neuronBlock>>>(dSoma, dAxons, dLIFNeurons, dMonitors
      , step*(sim->dt-FLT_EPSILON));
  cudaThreadSynchronize();

  // update axons
  updateAxons<<<neuronGrid, neuronBlock>>>(dAxons);
  cudaThreadSynchronize();

  // process spikes, apply psc filter
  updateSynapses<<<synapseGrid, synapseBlock>>>(dSynapses, dPSCFilter,
      dPSCBuffer, dDendrites, dAxons, step);
  cudaThreadSynchronize();
}
```

Listing C.1: `spikingNeuralNetwork.cu` - Spiking Neural Network CUDA/OpenGL interface methods

```
#ifndef SELF_ORGANIZING_MAP_H
#define SELF_ORGANIZING_MAP_H

#include "gpu_snn.h"

////////////////////////////////////////////////////////////////////////////////
// Filter configuration
////////////////////////////////////////////////////////////////////////////////
// functions to load images
extern "C" void LoadBMPFile(uchar4 **dst, int *width, int *height, const char *
    name);

// CUDA wrapper functions for allocation/freeing texture arrays
extern "C" cudaError_t CUDA_Bind2TextureArray();
extern "C" cudaError_t CUDA_UnbindTexture();
extern "C" cudaError_t CUDA_MallocArray(uchar4 **rasterPlot, int windowWidth,
    int windoeHeight);
extern "C" cudaError_t CUDA_FreeArray();
extern "C" cudaError_t CUDA_MemcpyToSymbols(int nNeurons, int nSynapses, int
    maxPSC, float dt);
// CUDA kernel functions
extern "C" void CUDA_UpdateRasterPlot(unsigned int *rasterPlot, uchar4 *
    plotBuffer, Axon *dAxons, int windowWidth, int windowHeight);

extern "C" void
CUDA_AdvanceSimulation(float *dStimulus,
                       float *hStimulus,
                       size_t stimSize,
                       Soma *dSoma,
                       TuningProfile *dProfiles,
                       Dendrite *dDendrites,
                       Neuron *dNeurons,
                       Axon *dAxons,
                       LIFNeuron *dLIFNeurons,
                       Monitor *dMonitors,
                       int step,
                       Simulation *sim,
                       Synapse *dSynapses,
                       float *dPSCFilter,
                       float *dPSCBuffer);
#endif
```

Listing C.2: `spikingNeuralNetwork.h` - Spiking Neural Network header file

```
#include <math.h>
#include <float.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <GL/glew.h>
#include <cuda_runtime.h>
#include <cutil.h>
#include <cutil_gl_error.h>
#include <cuda_gl_interop.h>
#include <GL/glut.h>

#include "spikingNeuralNetwork.h"
#include "gpu_snn.h"

////////////////////////////////////////////////////////////////////////////////
// Global data handlers and parameters
////////////////////////////////////////////////////////////////////////////////

//OpenGL PBO and texture "names"
GLuint gl_PBO, gl_Tex;

//Source image on the host side
uchar4 *rasterPlot, *plotBuffer;
int windowWidth, windowHeight;

////////////////////////////////////////////////////////////////////////////////
// Main program
////////////////////////////////////////////////////////////////////////////////
const int frameN = 24;
int frameCounter = 0;
int step = 0;
#define BUFFER_DATA(i) ((char *)0 + i)

// Auto-Verification Code
int fpsCount = 0;        // FPS count for averaging
int fpsLimit = 1;        // FPS limit for sampling
unsigned int frameCount = 0;
unsigned int g_TotalErrors = 0;
bool g_Verify = false, g_AutoQuit = false;

char spikes[80];
  Neuron    *dNeurons;   // constant neuron parameters
  LIFNeuron *dLIFNeurons;
  Synapse   *dSynapses;  // array of the model's synapses
  Soma     *hSoma,      *dSoma;      // represents the soma integration step
  Dendrite  *dDendrites; // input to each dendrite at a given time step
  Axon     *hAxons,     *dAxons;     // spike events
  float    *hPSCFilter, *dPSCFilter; // constant array storing PSC filter values
  float    *hPSCBuffer, *dPSCBuffer; // spike history per synapse

  float    *hStimulus,  *dStimulus;
  Monitor  *hMonitors,  *dMonitors;

  TuningProfile *hProfiles, *dProfiles; // tuning profile parameters

  size_t neuronSize;              //
  size_t synapseSize;             //
```

```
  size_t pscFilterSize;            // memory size of synapse arrays
  size_t pscBufferSize;            // memory size of synapse arrays
  size_t dendriteSize;
  size_t somaSize;
  size_t profileSize;
  size_t axonSize;
  size_t stimSize;
  size_t monitorSize;

  Simulation *sim;


void
displayFunc(void)
{
  unsigned int *dBuffer = NULL;

  CUDA_AdvanceSimulation(dStimulus,
                         hStimulus,
                         stimSize,
                         dSoma,
                         dProfiles,
                         dDendrites,
                         dNeurons,
                         dAxons,
                         dLIFNeurons,
                         dMonitors,
                         step,
                         sim,
                         dSynapses,
                         dPSCFilter,
                         dPSCBuffer);
  step++;

  CUDA_SAFE_CALL(cudaGLMapBufferObject((void**)&dBuffer, gl_PBO));
  CUDA_SAFE_CALL(CUDA_Bind2TextureArray());

  CUDA_UpdateRasterPlot(dBuffer, plotBuffer, dAxons, windowWidth, windowHeight);

  CUDA_SAFE_CALL(CUDA_UnbindTexture());
  CUDA_SAFE_CALL(cudaGLUnmapBufferObject(gl_PBO));

  glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, windowWidth, windowHeight, GL_RGBA,
      GL_UNSIGNED_BYTE, BUFFER_DATA(0));
  glBegin(GL_TRIANGLES);
    glTexCoord2f(0, 0); glVertex2f(-1, -1);
    glTexCoord2f(2, 0); glVertex2f(+3, -1);
    glTexCoord2f(0, 2); glVertex2f(-1, +3);
  glEnd();
  glFinish();

  glutSwapBuffers();
  glutPostRedisplay();

}

void
shutDown(unsigned char k, int /*x*/, int /*y*/)
{
```

```
    switch (k){
        case 'w':
            hStimulus[0] = 0.0f;
            hStimulus[499] = 1000.0f;
            break;
        case 's':
            hStimulus[0] = 1000.0f;
            hStimulus[499] = 0.0f;
            break;

        case '\033':
        case 'q':
        case 'Q':
            printf("Shutting down...\n");
                CUDA_SAFE_CALL( cudaGLUnregisterBufferObject(gl_PBO) );
                glBindBuffer(GL_PIXEL_UNPACK_BUFFER_ARB, 0);
                glDeleteBuffers(1, &gl_PBO);
                glDeleteTextures(1, &gl_Tex);

                CUDA_SAFE_CALL( CUDA_FreeArray() );
                free(rasterPlot);

CUDA_SAFE_CALL(cudaMemcpy(hMonitors, dMonitors, monitorSize,
    cudaMemcpyDeviceToHost));
  FILE *fd;
  fd = fopen(spikes,"w");
  for(unsigned int i =0; i<sim->nNeurons; ++i) {
    fprintf(fd, "%d ", hMonitors[i].spikeCount);
    if((i+1) % 20 == 0)
      fprintf(fd, "\n");
  }
  fclose(fd);

  // free allocations
  cudaFree(dMonitors);
  cudaFree(dStimulus);
  cudaFree(dProfiles);
  cudaFree(dAxons);
  cudaFree(dSoma);
  cudaFree(dDendrites);
  cudaFree(dPSCBuffer);
  cudaFree(dPSCFilter);
  cudaFree(dSynapses);
  cudaFree(dLIFNeurons);
  cudaFree(dNeurons);

  free(hMonitors);
  free(hStimulus);
  free(hProfiles);
  free(hAxons);
  free(hSoma);
  free(sim->dendrites);
  free(hPSCBuffer);
  free(hPSCFilter);
  free(sim->synapses);
  free(sim->tauPSCs);
  free(sim->lifNeurons);
  free(sim->neurons);
  free(sim);
```

```
printf("steps: %d\n",step);

            printf("Shutdown done.\n");
            exit(0);
        break;
    }
}

int
main(int argc, char **argv)
{
  CUT_DEVICE_INIT(argc, argv);

  if(argc == 1)
    return printf("usage: sim path/to/spec\n");

  char spec[80], topo[80];
  strcpy(spec,argv[1]);
  strcat(spec,"/timing.xml");
  strcpy(topo,argv[1]);
  strcat(topo,"/connections.bin");
  strcpy(spikes,argv[1]);
  strcat(spikes,"/spikes.txt");

  sim = initSimulation(spec);
  initConnections(sim, topo);

  neuronSize = sim->nNeurons * sizeof(Neuron);

  // init PSC filter array
  int totalPSCLength = (sim->neurons[sim->nNeurons-1].pscOffset + sim->neurons[
      sim->nNeurons-1].pscLength);
  totalPSCLength += totalPSCLength % 16;

  pscFilterSize = totalPSCLength * sizeof(float);
  hPSCFilter    = (float*)calloc(totalPSCLength, sizeof(float));
  for(int i = 0; i < (int)sim->nNeurons; ++i) {
    Neuron n  = sim->neurons[i];
    float sum = 0;

    for(int j = 0; j < (int)n.pscLength; ++j)
      sum += hPSCFilter[n.pscOffset + j] = (float)(pow(j*sim->dt,sim->
          pscFilterOrder)*exp(-j*sim->dt/sim->tauPSCs[i]));

    for(int j = 0; j < (int)n.pscLength; ++j)
      hPSCFilter[n.pscOffset + j] /= sum;
  }

  // init synapses
  synapseSize   =  sim->nSynapses * sizeof(Synapse);

  // init tuning profile
  profileSize = sim->nNeurons * sizeof(TuningProfile);
  hProfiles = (TuningProfile*)malloc(profileSize);
  for(int i = 0; i < (int)sim->nNeurons; ++i) {
    LIFNeuron n = sim->lifNeurons[i];
    float maxResp = 100.0; //getRandom(sim->maxRespRange[MIN], sim->maxRespRange
        [MAX]);
    float minResp = maxResp * sim->noiseError;
```

```
  hProfiles[i].Jbias = minResp == 0 ? 0 : n.Vth * (1.0 / (1.0 - exp((n.tauRef
      * minResp - 1.0) / (n.tauRC * minResp))));
  hProfiles[i].alpha = n.Vth * (1.0 / (1.0 - exp((n.tauRef * maxResp - 1.0) /
      (n.tauRC * maxResp)))) - hProfiles[i].Jbias;
}

// init PSC buffer array
// tk: backfill buffer with background spiking behaviour to avoid "cold start"
    bias
pscBufferSize = sim->nSynapses * sim->pscFilterSize * sizeof(float);
hPSCBuffer    = (float*)calloc(sim->nSynapses * sim->pscFilterSize, sizeof(
    float));

// init dendrites / soma
dendriteSize = sim->nSynapses * sizeof(Dendrite);
somaSize     = sim->nNeurons * sizeof(Soma);
hSoma        = (Soma*)malloc(somaSize);
for(int i=0; i<(int)sim->nNeurons; ++i) {
  hSoma[i].Jspike = 0;
  hSoma[i].Jstim = 0;
  hSoma[i].Vm = 0;
  hSoma[i].refEnd = 0;
}
axonSize     = sim->nNeurons * sizeof(Axon);
hAxons       = (Axon*)malloc(axonSize);
for(int i=0; i<(int)sim->nNeurons; ++i) {
  hAxons[i].buffer = 0;
  hAxons[i].spike = 0;
}

monitorSize  = sim->nNeurons * sizeof(Monitor);
hMonitors    = (Monitor*)calloc(sim->nNeurons,sizeof(Monitor));

// init stimulus
// tk: needs to be pushed to an interface of some kind
stimSize     = sim->nNeurons * sizeof(float);
hStimulus    = (float*)calloc(sim->nNeurons,sizeof(float));
//hStimulus[0] = 10000.0f;

size_t lifNeuronSize = sim->nNeurons * sizeof(LIFNeuron);

// allocate arrays on device
CUDA_SAFE_CALL(cudaMalloc((void**)&dNeurons,   neuronSize));
CUDA_SAFE_CALL(cudaMalloc((void**)&dLIFNeurons,   lifNeuronSize));
CUDA_SAFE_CALL(cudaMalloc((void**)&dSynapses,  synapseSize));
CUDA_SAFE_CALL(cudaMalloc((void**)&dPSCFilter, pscFilterSize));
CUDA_SAFE_CALL(cudaMalloc((void**)&dPSCBuffer, pscBufferSize));
CUDA_SAFE_CALL(cudaMalloc((void**)&dDendrites, dendriteSize));
CUDA_SAFE_CALL(cudaMalloc((void**)&dSoma,      somaSize));
CUDA_SAFE_CALL(cudaMalloc((void**)&dAxons,     axonSize));
CUDA_SAFE_CALL(cudaMalloc((void**)&dProfiles,  profileSize));
CUDA_SAFE_CALL(cudaMalloc((void**)&dMonitors,  monitorSize));
CUDA_SAFE_CALL(cudaMalloc((void**)&dStimulus,  stimSize));

// transfer arrays to device
CUDA_SAFE_CALL(cudaMemcpy(dNeurons,    sim->neurons,   neuronSize,
    cudaMemcpyHostToDevice));
CUDA_SAFE_CALL(cudaMemcpy(dLIFNeurons, sim->lifNeurons,   lifNeuronSize,
    cudaMemcpyHostToDevice));
```

```
  CUDA_SAFE_CALL(cudaMemcpy(dSoma,      hSoma,      somaSize,
      cudaMemcpyHostToDevice));
  CUDA_SAFE_CALL(cudaMemcpy(dSynapses,  sim->synapses,  synapseSize,
      cudaMemcpyHostToDevice));
  CUDA_SAFE_CALL(cudaMemcpy(dPSCFilter, hPSCFilter, pscFilterSize,
      cudaMemcpyHostToDevice));
  CUDA_SAFE_CALL(cudaMemcpy(dPSCBuffer, hPSCBuffer, pscBufferSize,
      cudaMemcpyHostToDevice));
  CUDA_SAFE_CALL(cudaMemcpy(dDendrites, sim->dendrites, dendriteSize,
      cudaMemcpyHostToDevice));
  CUDA_SAFE_CALL(cudaMemcpy(dAxons,     hAxons,     axonSize,
      cudaMemcpyHostToDevice));
  CUDA_SAFE_CALL(cudaMemcpy(dProfiles,  hProfiles,  profileSize,
      cudaMemcpyHostToDevice));
  CUDA_SAFE_CALL(cudaMemcpy(dMonitors,  hMonitors,  monitorSize,
      cudaMemcpyHostToDevice));

  CUDA_MemcpyToSymbols(sim->nNeurons,sim->nSynapses,sim->pscFilterSize,sim->dt);

  windowWidth  = 1000; // ~0.25 seconds
  windowHeight = 1000;
  if(sim->nNeurons < 1000)
    windowHeight = sim->nNeurons;

  rasterPlot = (uchar4*)malloc(windowHeight * windowWidth * sizeof(uchar4));
  for(int i = 0; i < windowHeight*windowWidth; ++i)
    rasterPlot[i] = make_uchar4(255,255,255,0);

  CUDA_SAFE_CALL(CUDA_MallocArray(&rasterPlot, windowWidth, windowHeight));
  CUDA_SAFE_CALL(cudaMalloc((void**)&plotBuffer, windowWidth * windowHeight *
      sizeof(uchar4)));
  CUDA_SAFE_CALL(cudaMemcpy(plotBuffer, rasterPlot, windowWidth * windowHeight *
       sizeof(uchar4), cudaMemcpyHostToDevice));

  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_RGBA | GLUT_SINGLE);
  glutInitWindowSize(windowWidth, windowHeight);
  glutInitWindowPosition(512 - windowWidth / 2, 384 - windowHeight / 2);
  glutCreateWindow(argv[0]);
printf("Loading␣extensions:␣%s\n", glewGetErrorString(glewInit()));
        if(!glewIsSupported(
            "GL_VERSION_2_0␣"
            "GL_ARB_pixel_buffer_object␣"
            "GL_EXT_framebuffer_object␣"
        )){
            fprintf(stderr, "ERROR:␣Support␣for␣necessary␣OpenGL␣extensions␣
                missing.");
            fflush(stderr);
            return CUTFalse;
        }

  glEnable(GL_TEXTURE_2D);
  glGenTextures(1, &gl_Tex);
  glBindTexture(GL_TEXTURE_2D, gl_Tex);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
```

```
  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, windowWidth, windowHeight, 0, GL_RGBA
      , GL_UNSIGNED_BYTE, rasterPlot);

  glGenBuffers(1, &gl_PBO);
  glBindBuffer(GL_PIXEL_UNPACK_BUFFER_ARB, gl_PBO);
  glBufferData(GL_PIXEL_UNPACK_BUFFER_ARB, windowWidth * windowHeight * 4,
      rasterPlot, GL_STREAM_COPY);
  CUDA_SAFE_CALL(cudaGLRegisterBufferObject(gl_PBO));
  CUT_CHECK_ERROR_GL();

  printf("Starting␣GLUT␣main␣loop...\n");
  printf("Press␣[q]␣to␣exit\n");

  glutSetWindowTitle("Spiking␣Neural␣Network");
  glutIdleFunc(displayFunc);
  glutDisplayFunc(displayFunc);
  glutKeyboardFunc(shutDown);


  glutMainLoop();

  CUT_EXIT(argc, argv);
}
```

Listing C.3: `spikingNeuralNetworkGL.cpp` - Spiking Neural Network main routine and OpenGL methods

```
#include <libxml/parser.h>
#include <libxml/tree.h>
#include <math.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

#include "gpu_snn.h"

void parseNeurons(Simulation*, xmlNode*);

float
getFloatProp(xmlNode *root, const char *name)
{
  xmlChar *prop = xmlGetProp(root, (const xmlChar*)name);
  float result  = atof((const char*)prop);

  xmlFree(prop);

  return result;
}

unsigned int
getUIntProp(xmlNode *root, const char *name)
{
  xmlChar *prop = xmlGetProp(root, (const xmlChar*)name);
  unsigned int result  = (unsigned int)atoi((const char*)prop);

  xmlFree(prop);

  return result;
}


Simulation*
initSimulation(const char *filename)
//int main(int argc, char **argv)
{
  Simulation *sim;
  xmlDoc *doc;
  xmlNode *root, *node;

  sim = (Simulation*)calloc(1,sizeof(Simulation));

  doc  = xmlReadFile(filename, NULL, XML_PARSE_DTDVALID);
  root = xmlDocGetRootElement(doc);


  // simulation duration (T)
  sim->T = getFloatProp(root,"T");

  // simulation time step (dt)
  sim->dt = getFloatProp(root,"dt");

  // general psc data default values
  sim->pscFilterOrder = 0;
  sim->pscFilterScale = 5;
  sim->pscFilterSize  = 0;
```

```
  node = root->xmlChildrenNode;
  while(node != NULL) {
    if(!strcmp((const char*)node->name, "pop")) {
      parseNeurons(sim, node);
    }
    node = node->next;
  }

  xmlFree(node);
  xmlFree(root);
  xmlFree(doc);

  return sim;
}

void
parseNeurons(Simulation *sim, xmlNode *pop)
{
  xmlNode *node;
  unsigned int id, pscLength, pscOffset;

  // TK: assumes single population
  sim->nNeurons   = getUIntProp(pop, "num_neurons");
  sim->neurons    = (Neuron*)   calloc(sim->nNeurons, sizeof(Neuron));
  sim->lifNeurons = (LIFNeuron*)calloc(sim->nNeurons, sizeof(LIFNeuron));
  sim->tauPSCs    = (float*)    calloc(sim->nNeurons, sizeof(float));

  pscOffset = 0;

  node = pop->xmlChildrenNode;
  while(node != NULL) {
    id     = getUIntProp (node, "id");

    sim->tauPSCs[id] = getFloatProp(node, "tau_psc");

    pscLength = ceil(sim->pscFilterScale * sim->tauPSCs[id] / sim->dt);
    sim->neurons[id].pscLength = pscLength;
    sim->neurons[id].pscOffset = pscOffset;
    pscLength += pscLength % 16; // cuda memory access optimization
    if(pscLength > sim->pscFilterSize)
      sim->pscFilterSize = pscLength; // fixed width filter buffers, even if
          actual filter size is smaller
    pscOffset += pscLength;

    sim->lifNeurons[id].tauRC  = getFloatProp(node, "tau_rc");
    sim->lifNeurons[id].tauRef = getFloatProp(node, "tau_ref");
    sim->lifNeurons[id].Vth    = getFloatProp(node, "v_th");
    sim->lifNeurons[id].Rleak  = getFloatProp(node, "r_leak");

    node = node->next;
  }

  xmlFree(node);
}

void
initConnections(Simulation *sim, const char *filename)
{
```

```
  FILE *fd;

  fd = fopen(filename, "rb");
  unsigned int src, dst;
  float weight, delay;

  unsigned int *synapseCounts, *synapseOffsets, nSynapses;
  synapseCounts  = (unsigned int*)calloc(sim->nNeurons, sizeof(unsigned int));
  synapseOffsets = (unsigned int*)calloc(sim->nNeurons, sizeof(unsigned int));
  nSynapses = 0;
  // tk: use a data structure to store and sort values to avoid two passes
  while(fread(&src,    sizeof(unsigned int), 1, fd) != 0 && fread(&dst,
      sizeof(unsigned int), 1, fd) != 0 &&
        fread(&weight, sizeof(float),        1, fd) != 0 && fread(&delay,
            sizeof(float),        1, fd) != 0) {
if(dst >= sim->nNeurons) printf("%u <- %u, %u\n",dst, src, sim->nNeurons);
    synapseCounts[dst]++;
    nSynapses++;
  }

  int offset = 0;
  for(unsigned int i=0; i<sim->nNeurons; ++i) {
    synapseOffsets[i] = sim->neurons[i].dendriteOffset = offset;
    offset           += sim->neurons[i].dendriteLength = synapseCounts[i];
    synapseCounts[i]  = 0;
  }

  sim->nSynapses = nSynapses;
  sim->synapses  = (Synapse*)malloc(nSynapses*sizeof(Synapse));
  sim->dendrites = (Dendrite*)malloc(nSynapses*sizeof(Dendrite));

  rewind(fd);
  while(fread(&src,    sizeof(unsigned int), 1, fd) != 0 && fread(&dst,
      sizeof(unsigned int), 1, fd) != 0 &&
        fread(&weight, sizeof(float),        1, fd) != 0 && fread(&delay,
            sizeof(float),        1, fd) != 0) {
    offset = synapseOffsets[dst] + synapseCounts[dst];
    synapseCounts[dst]++;

    sim->synapses[offset].sendId = src;
    sim->synapses[offset].recvId = dst;
    sim->synapses[offset].pscOffset = offset * sim->pscFilterSize;
    sim->synapses[offset].pscLength = sim->neurons[dst].pscLength;

    sim->dendrites[offset].weight = 5;//weight;
    sim->dendrites[offset].Jin    = 0;
  }

  fclose(fd);

  free(synapseOffsets);
  free(synapseCounts);
}
```

Listing C.4: `initialize.cpp` - Spiking Neural Network specification file parser

```
/*
** Synapses do the following at each time step:
**   1) Check for a efferent neuron spike
**   2) Update PSC result buffer accordingly
**   3) Send input to afferent neruon dendrite
**   TODO:
**   Support STDP via LTP/LTD using GABAa,GABAb,AMPA,NMDA channels
*/

__constant__ float DT;
__constant__ int NEURONS;
__constant__ int SYNAPSES;
__constant__ int PSC_BUFFER_LENGTH;

/*
__global__ void
updateSynapses(const Synapse *dSynapses,
               const Neuron  *dNeurons,
               const float   *dPSCFilter,
               float         *dPSCBuffer,
               Dendrite      *dDendrites,
               Axon          *dAxons,
               const int     step)
{
  // load synapse
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  if(tid >= SYNAPSES) return;

  // tk: mod is expensive.. maybe calculate on cpu and load for lookup instead
  Synapse  s = dSynapses[tid];
  Dendrite d = dDendrites[tid];
  Neuron  n   = dNeurons[s.recvId];
  int     idx = step % s.pscLength;

  // tk: only update buffers that have new spikes?
  // update PSC buffer
  if(dAxons[s.sendId].spike) {
    for(int i = 0, j = idx; i < n.pscLength; ++i) {
      dPSCBuffer[s.pscOffset + j] += dPSCFilter[n.pscOffset + i];
      j = ++j == n.pscLength ? 0 : j;
    }
    dAxons[s.sendId].spike = 0;
  }

  // tk: this assumes synapses are contiguous by neuron
  // Send input to afferent neurons and reset buffer
  dDendrites[tid].Jin = dPSCBuffer[s.pscOffset + idx];
  dPSCBuffer[s.pscOffset + idx] = 0.0f;
}
*/


__global__ void
updateSynapses(const Synapse *dSynapses,
//               const Neuron  *dNeurons,
               const float   *dPSCFilter,
               float         *dPSCBuffer,
               Dendrite      *dDendrites,
```

```
                Axon          *dAxons,
                const int     step)
{
  __shared__ int nOffset, sOffset, length, spike, idx;
  __shared__ float filter[256];

  int sid = blockIdx.x + blockIdx.y * gridDim.x;
  if(sid >= SYNAPSES) return;

  int tid = threadIdx.x;

  // load common references
  if(tid == 0) {
    Synapse s = dSynapses[sid];
    nOffset   = PSC_BUFFER_LENGTH * s.recvId;
    sOffset   = PSC_BUFFER_LENGTH * sid;
    //dOffset   = n.dendriteOffset;
    length    = s.pscLength;
    idx       = step % s.pscLength;
    spike     = dAxons[s.sendId].spike;
  }
  __syncthreads();

  if(!spike || tid >= length) return;

  // prefetch filter
  filter[tid] = dPSCFilter[nOffset + tid];
  __syncthreads();

  int bid = tid + idx;
  if(bid >= length)
    bid -= length;

  // step-wise convolve w/ circular buffer
  dPSCBuffer[sOffset + bid] += filter[tid];
  __syncthreads();

  if(tid == 0) {
    dDendrites[sid].Jin = dPSCBuffer[sOffset + idx];
    dPSCBuffer[sOffset + idx] = 0.0f;
  }

}

__device__ float
applyTuning(TuningProfile profile, float Jd)
{
  return Jd; //profile.alpha * Jd + profile.Jbias;
}

__global__ void
updateSomas(Soma *dSomas, Axon *dAxons, const LIFNeuron *dLIFNeurons, Monitor *
    dMonitors, const float time)
{
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  if(tid >= NEURONS) return;

  Soma   s  = dSomas[tid];
  LIFNeuron n  = dLIFNeurons[tid];
```

```
  float  Jd;
  float4 rk4;

  // apply tuning transform
  Jd = s.Jspike + s.Jstim;

  // only integrate if allowed
  if(time > s.refEnd) {
    // RK4
    rk4.x = DT * (Jd * n.Rleak -  s.Vm             ) / n.tauRC;
    rk4.y = DT * (Jd * n.Rleak - (s.Vm + 0.5f*rk4.x)) / n.tauRC;
    rk4.z = DT * (Jd * n.Rleak - (s.Vm + 0.5f*rk4.y)) / n.tauRC;
    rk4.w = DT * (Jd * n.Rleak - (s.Vm + rk4.z)     ) / n.tauRC;
    s.Vm += (rk4.x + 2*rk4.y + 2*rk4.z + rk4.w) / 6;

    // EULER
    // s.Vm += DT * (Jd * R_LEAK - s.Vm) / n.tauRC;
    //dTrace[tid*Nt+step] = s.Vm;

    // reset Vm and emit spike if threshold exceeded
    if(s.Vm > n.Vth) {
      s.Vm              = 0.0f;
      s.refEnd          = time + n.tauRef;
      dAxons[tid].buffer = 1;
      dMonitors[tid].spikeCount += 1;
    }
    dSomas[tid] = s;
  }
}
//updateDendrites(Synapse *dSynapses, Soma *dSoma, Neuron *dNeurons, float *
    dPSCBuffer, const int step, const int nNeurons)
__global__ void
updateDendrites(Dendrite *dDendrites, Soma *dSoma, Neuron *dNeurons)
{
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  if(tid >= NEURONS) return;

  Neuron n = dNeurons[tid];
  Soma   s = dSoma[tid];

  // reset input current
  s.Jspike = 0.0f;

  // tk: should be parallel reduction
  for(int i = 0; i < n.dendriteLength; ++i) {
    Dendrite d = dDendrites[n.dendriteOffset + i];
    s.Jspike += d.Jin * d.weight;
  }
  dSoma[tid] = s;
}

__global__ void
applyStimulus(Soma *dSoma, const TuningProfile *dProfiles, float *dStimulus)
{
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  if(tid >= NEURONS) return;

  dSoma[tid].Jstim = applyTuning(dProfiles[tid], dStimulus[tid]);
}
```

```
__global__ void
updateAxons(Axon *dAxons)
{
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  dAxons[tid].spike  = dAxons[tid].buffer;
  dAxons[tid].buffer = 0;
}
```

Listing C.5: `gpu_snn_kernels.cu` - Spiking Neural Network agent-based kernels

```
#ifndef GPU_SNN_H
#define GPU_SNN_H

// Macros
#define MIN 0
#define MAX 1

// Structures

// Constant neuron parameters assigned at initialization
typedef struct
#ifdef __CUDACC__
__align__(16)
#endif
{
  unsigned int dendriteOffset; // dendrite offset
  unsigned int dendriteLength; // number of dendrites
  unsigned int pscOffset;      // PSC filter offset
  unsigned int pscLength;      // length of PSC filter
  //float tauPSC;                // PSC time constant
} Neuron;

typedef struct
#ifdef __CUDACC__
__align__(16)
#endif
{
  float tauRef;
  float tauRC;
  float Rleak;
  float Vth;
} LIFNeuron;

// Dynamic neuron properties
typedef struct
#ifdef __CUDACC__
__align__(16)
#endif
{
  float Jspike;                // summed input from dendrites
  float Jstim;                 // stimulation input
  float Vm;                    // current membrane voltage
  float refEnd;                // time when refractory period ends
} Soma;

// Spiking reference
typedef struct
//#ifdef __CUDACC__
//__align__(8)
//#endif
{
  char buffer;                 // single time step delay buffer
  char spike;                  // boolean spike, char used for CUDA-
      compatability
} Axon;

// Constant synaptic properties assigned at initialization
typedef struct
```

```
#ifdef __CUDACC__
__align__(16)
#endif
{
  int sendId;                   // efferent neuron
  int recvId;                   // afferent neuron
  int pscOffset;                // psc buffer offset
  int pscLength;                // psc buffer length
} Synapse;

// Input current reference
typedef struct
#ifdef __CUDACC__
__align__(8)
#endif
{
  float Jin;                    // input current
  float weight;                 // input weight
} Dendrite;

// Linear tuning profile parameters
typedef struct
#ifdef __CUDACC__
__align__(8)
#endif
{
  float alpha;
  float Jbias;
} TuningProfile;

// Monitor to record spike counts
// tk: monitors probably don't need need special structs
typedef struct {
  int spikeCount;
} Monitor;

typedef struct {
  unsigned int nNeurons;
  unsigned int nSynapses;
  float T;
  float dt;
  float tauRefRange[2];
  float tauRCRange[2];
  float tauPSCRange[2];
  float maxRespRange[2];
  float noiseError;

  // PSC filter related values
  float *tauPSCs;
  float pscFilterOrder;
  float pscFilterScale;
  unsigned int pscFilterSize;

  Neuron    *neurons;
  LIFNeuron *lifNeurons;
  Synapse   *synapses;
  Dendrite  *dendrites;
} Simulation;
//
```

```
// Prototypes
Neuron* initNeurons(Simulation*);
float getRandom(float, float);

#ifdef __cplusplus
extern "C" {
#endif
Simulation* initSimulation(const char*);
void initConnections(Simulation*, const char*);
#ifdef __cplusplus
}
#endif

#endif // GPU_SNN_H
```

Listing C.6: `gpu_snn.h` - Spiking Neural Network data structure definitions

APPENDIX D

Neural Decoding Code Listings

```
/*
** cusumma.cu
**
** Resource-aware dense matrix multiplication using GPUs
**
** Provided as is with no warranty of any kind.
** Byron Galbraith (c) 2009
*/

#include <cuda.h>
#include <cublas.h>
#include <math.h>
#include <stdio.h>
#include <time.h>
#include <sys/time.h>

extern "C" int
cusumma(unsigned int transA,
        unsigned int transB,
        unsigned int m,
        unsigned int n,
        unsigned int k,
        float *A,
        float *B,
        float *C)

{
  float *hA, *dA, *hB, *dB, *dC;
  int i, j, diff, tm, tk, tp, tp_last, tmp1, tmp2, _kmax, _mmax, _m, _moff, _k,
      _koff;
  char opA, opB;
  float factor;
  unsigned int gpu_mem, lda, ldb, ldc;

  cublasInit();

  // get total free memory available to CUSUMMA
  cuMemGetInfo(&gpu_mem, NULL);

  // take 15MB off the top for CUBLAS working memory
  // this is a guess that seems to work, replace with actual numbers when known
  gpu_mem -= 15*1048576;

  // convert gpu_mem from bytes into matrix elements (floats) for simplicity
```

```
gpu_mem /= sizeof(float);

// determine optimal partition dimensions
tp = 2*m*k+m;
tm = 0;
do {
  if(tp > 0)
    tp_last = tp;
  _mmax = ceil(1.0*m/++tm);
  tmp1 = gpu_mem - _mmax * (A == B ? _mmax : n);
  tmp2 = _mmax + (A == B ? 0 : n);
  _kmax = tmp1 / tmp2; //(gpu_mem - n * _mmax)/(n + _mmax);
  tk    = ceil(1.0*k / _kmax);
  tp    = (A == B ? 1 : 2)*tm*tk + tm;
} while(tp < 0 || tp < tp_last);
tp = tp_last;

_mmax = ceil(1.0*m/--tm);
if(A == B) {
  _kmax = gpu_mem / _mmax - _mmax;
} else {
  _kmax = (gpu_mem - _mmax*n )/(_mmax + n);
}

// assumes input matrices are in row-major order
opA = transB ? 't' : 'n';
opB = transA ? 't' : 'n';

_m    = _mmax;
_moff = 0;
ldc = n;
while(_moff < m) {
  cublasAlloc(_m * n, sizeof(float), (void**)&dC);
  if(A == B) {  // op(A) * op(A)
    diff = gpu_mem - m*k - m*m;
    if((_m == m) && (diff > 0)) {
      cublasAlloc(m * k, sizeof(float), (void**)&dA);
      cublasSetVector(m * k, sizeof(float), A, 1, dA, 1);

      lda = transA ? m : k;
      ldb = transB ? k : n;
      cublasSgemm(opA, opB, n, m, k, 1.0f, dA, ldb, dA, lda, 0.0f, dC, ldc);
      cublasFree(dA);

    } else {
      _koff  = 0;
      _k     = _kmax;
      factor = 0.0f;
      while(_koff < k) {
        cublasAlloc(_m * _k, sizeof(float), (void**)&dA);

        hA = (float*)malloc(_m * _k * sizeof(float));
        for(i = 0; i < _m; ++i)
          for(j = 0; j < _k; ++j)
            hA[i*_k + j] = A[(i+_moff)*k + j + _koff];
        cublasSetVector(_m * _k, sizeof(float), hA, 1, dA, 1);
        free(hA);

        lda = transA ? _m : _k;
```

```
        ldb = transB ? _k : n;
        cublasSgemm(opA, opB, n, _m, _k, 1.0f, dA, ldb, dA, lda, factor, dC,
            ldc);
        cublasFree(dA);

        _koff += _k;
        _k     = k - _koff > _kmax ? _kmax : k - _koff;
        factor = 1.0f;
      }
    }

  } else { // op(A) * op(B)
    diff = gpu_mem - (m*k + k*n + m*n);
    if((_m == m) && (diff > 0)) {
      cublasAlloc(m * k, sizeof(float), (void**)&dA);
      cublasSetVector(m * k, sizeof(float), A, 1, dA, 1);

      cublasAlloc(k * n, sizeof(float), (void**)&dB);
      cublasSetVector(k * n, sizeof(float), B, 1, dB, 1);

      lda = transA ? m : k;
      ldb = transB ? k : n;
      cublasSgemm(opA, opB, n, m, k, 1.0f, dB, ldb, dA, lda, 0.0f, dC, ldc);

      cublasFree(dA);
      cublasFree(dB);

    } else {
      opA = opB = 'n'; // transpose manually ahead of time
      _koff  = 0;
      _k     = _kmax;
      factor = 0.0f;
      while(_koff < k) {
        cublasAlloc(_m * _k, sizeof(float), (void**)&dA);
        cublasAlloc(_k * n, sizeof(float), (void**)&dB);

        hA = (float*) malloc(_m * _k * sizeof(float));
        for(i = 0; i < _m; ++i)
          for(j = 0; j < _k; ++j)
            hA[i*_k + j] = transA ? A[(j+_koff)*k + i + _moff] : A[(i+_moff)*k
                + j + _koff];
        cublasSetVector(_m * _k, sizeof(float), hA, 1, dA, 1);
        free(hA);

        hB = (float*) malloc(_k * n * sizeof(float));
        for(i = 0; i < _k; ++i)
          for(j = 0; j < n; ++j)
            hB[i*n + j] = transB ? B[j*k + i + _koff] : B[(i+_koff)*n + j];
        cublasSetVector(_k * n, sizeof(float), hB, 1, dB, 1);
        free(hB);

        lda = _k;
        ldb = n;
        cublasSgemm(opA, opB, n, _m, _k, 1.0f, dB, ldb, dA, lda, factor, dC,
            ldc);

        cublasFree(dA);
        cublasFree(dB);
```

```
          _koff += _k;
          _k = k - _koff > _kmax ? _kmax : k - _koff;
          factor = 1.0f;
        }
      }
    }

    cublasGetVector(_m*n, sizeof(float), dC, 1, C+(_moff*n), 1);
    cublasFree(dC);

    _moff += _m;
    _m = m - _moff > _mmax ? _mmax : m - _moff;
  }

  cublasShutdown();

  return tp;
}
```

Listing D.1: `cusumma.cu` - Implementation of the CUSUMMA algorithm

```
/*
** Simple matrix inversion routine using CUDA
**
** This reimplements the LAPACK sgetri and associated required routines
** by replacing the BLAS calls with CUBLAS calls.
**
** Byron Galbraith
** Department of Mathematics, Statistics, and Computer Science
** Marquette University
** 2009-04-30
*/
#include <cublas.h>

// Prototypes
int* cudaSgetrf(unsigned int n, float *dA);
void cudaSgetri(unsigned int n, float *dA, int *pivots);
void cudaStrtri(unsigned int n, float *dA);

/*
** cudaInvertMatrix
** Inverts a square matrix in place
**   n - matrix dimension
**   A - pointer to array of floats representing the matrix in column-major
    order
*/
extern "C" void
cudaInvertMatrix(unsigned int n, float *A)
{
  int *pivots;
  float *dA;

  cublasInit();

  cublasAlloc(n * n, sizeof(float), (void**)&dA);
  cublasSetMatrix(n, n, sizeof(float), A, n, dA, n);

  // Perform LU factorization
  pivots = cudaSgetrf(n, dA);

  // Perform inversion on factorized matrix
  cudaSgetri(n, dA, pivots);

  cublasGetMatrix(n, n, sizeof(float), dA, n, A, n);

  cublasFree(dA);
  cublasShutdown();
}

/*
** cudaSgetrf
** Performs an in-place LU factorization on a square matrix
** Uses the unblocked BLAS2 approach
*/
int *
cudaSgetrf(unsigned int n, float *dA)
{
  int i, pivot, *pivots;
  float *offset, factor;
```

```
  pivots = (int *) calloc(n, sizeof(int));
  for(i = 0; i < n; ++i)
    pivots[i] = i;

  for(i = 0; i < n - 1; i++) {
    offset = dA + i*n + i;
    pivot = i - 1 + cublasIsamax(n - i, offset, 1);

    if(pivot != i) {
      pivots[i] = pivot;
      cublasSswap(n, dA + pivot, n, dA + i, n);
    }

    cublasGetVector(1, sizeof(float), offset, 1, &factor, 1);
    cublasSscal(n - i - 1, 1 / factor, offset + 1, 1);

    cublasSger(n - i - 1, n - i - 1, -1.0f, offset + 1, 1, offset + n, n, offset
        + n + 1, n);
  }

  return pivots;
}

/*
** cudaSgetri
** Computes the inverse of an LU-factorized square matrix
*/
void
cudaSgetri(unsigned int n, float *dA, int *pivots)
{
  int i;
  float *dWork, *offset;

  // Perform inv(U)
  cudaStrtri(n, dA);

  // Solve inv(A)*L = inv(U)
  cublasAlloc(n - 1, sizeof(float), (void**)&dWork);

  for(i = n - 1; i > 0; --i) {
    offset = dA + (i - 1)*n + i;
    cudaMemcpy(dWork, offset, (n - 1) * sizeof(float), cudaMemcpyDeviceToDevice)
        ;
    cublasSscal(n - i, 0, offset, 1);
    cublasSgemv('n', n, n - i, -1.0f, dA + i*n, n, dWork, 1, 1.0f, dA + (i-1)*n,
        1);
  }

  cublasFree(dWork);

  // Pivot back to original order
  for(i = n - 1; i >= 0; --i)
    if(i != pivots[i])
      cublasSswap(n, dA + i*n, 1, dA + pivots[i]*n, 1);
}

/*
** cudaStrtri
```

```
** Computes the inverse of an upper triangular matrix in place
** Uses the unblocked BLAS2 approach
*/
void
cudaStrtri(unsigned int n, float *dA)
{
  int i;
  float factor, *offset;

  for(i = 0; i < n; ++i) {
    offset = dA + i*n;
    cublasGetVector(1, sizeof(float), offset + i, 1, &factor, 1);
    factor = 1 / factor;
    cublasSetVector(1, sizeof(float), &factor, 1, offset + i, 1);

    cublasStrmv('u', 'n', 'n', i, dA, n, offset, 1);
    cublasSscal(i, -1 * factor, offset, 1);
  }
}
```

Listing D.2: `cudaMatrixInversion.cu` - Matrix inversion using CUBLAS routines