Norwegian University of
Science and Technology

Parallelization of Artificial Spiking Neural Networks on the CPU and GPU

Tor Brede Vekterli

Master of Science in Informatics
Submission date: June 2009
Supervisor: Keith Downing, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Abstract

Conventional artificial neural networks have traditionally faced inherent problems with efficient parallelization of neuron processing. Recent research has shown how artificial spiking neural networks can, with the introduction of biologically plausible synaptic conduction delays, be fully parallelized regardless of their network topology. This, in conjunction with the influx of fast, massively parallel desktop-level computing hardware leaves the field of efficient, large-scale spiking neural network simulations potentially open to even those with no access to supercomputers or large computing clusters.

This thesis aims to show how such a parallelization is possible as well as present a network model that enables it. This model will then be used as a base for implementing a parallel artificial spiking neural network on both the CPU and the GPU and subsequently evaluating some of the challenges involved, the performance and scalability measured and the potential that is exhibited.

Contents

1 Introduction **5**
1.1 Purpose and motivation 5
1.1.1 Why focus on spiking neural networks? 6
1.1.2 Why multi-core CPUs 6
1.1.3 Why GPUs show a vast potential for neural networks processing 7
1.2 Existing research 7
1.3 Structure of this thesis 8

2 A parallelization primer **10**
2.1 Introduction 10
2.1.1 Challenges 12
2.2 Supporting tools 14
2.2.1 OpenMP 14
2.2.2 NVIDIA CUDA 17

3 Artificial Spiking Neural Networks **21**
3.1 Introduction 21
3.1.1 The biological spiking neuron 21
3.1.2 Simplifications in conventional neural networks 22
3.1.3 Spiking neural networks 23
3.2 Artificial neuron models 24
3.2.1 Leaky integrator model 24
3.2.2 Hodgkin-Huxley 25
3.2.3 Izhikevich model 25
3.3 Learning in SNNs 26
3.3.1 Hebbian learning 28
3.3.2 Spike-time dependent plasticity (STDP) 29
3.4 SNN parallelization 30
3.4.1 Limitations of conventional artificial neural networks 30

3.4.2 Synaptic conduction delays as an enabler of parallelization . . . 32
3.4.3 Learning-parallelization . . . 35
3.5 Spiking Neural Network model . . . 35
3.5.1 Structure . . . 35
3.6 Network construction . . . 38
3.7 Network simulation . . . 38

4 Implementation **50**
4.1 Introduction . . . 50
4.2 Parallel Izhikevich SNN on the CPU using OpenMP . . . 50
4.3 Parallel Izhikevich SNN on the GPU using NVIDIA CUDA . 50

5 Method **53**
5.1 Introduction . . . 53
5.2 CPU (OpenMP) . . . 54
5.2.1 Areas and procedures of testing . . . 54
5.3 GPU (CUDA) . . . 55
5.3.1 Areas and procedures of testing . . . 55

6 Results **56**
6.1 CPU . . . 56
6.2 GPU . . . 64

7 Discussion **70**
7.1 OpenMP implementation . . . 70
7.2 CUDA implementation . . . 73
7.3 Comparison . . . 75

8 Conclusions **76**
8.1 Summary . . . 76
8.2 Possible future work . . . 78

List of Figures

2.1 Overview of single-threaded and multi-threaded processes . 11
2.2 Diagram of a shared-memory system 12
2.3 Task dependencies 13
2.4 OpenMP task scheduling example 16
2.5 Threading hierarchy in CUDA 18
2.6 Coalescing of memory accesses 20

3.1 Figure of simplified biological neuron 22
3.2 Izhikevich Regular Spiking (RS) neuron 27
3.3 Izhikevich Fast Spiking (FS) neuron 28
3.4 LTP and LTD based on spike arrival and neuron firing times 29
3.5 STDP curve graph for interspike intervals 31
3.6 Parallelization issues with conventional, layered neural networks 33
3.7 Conceptual example of spike input over synapses with delays 39
3.8 ASNN construction phases 40
3.9 Network simulation task dependencies 44

4.1 Control-flow diagram of the kernels and their invocation . . 51

6.1 CPU network processing, 1–4 threads, $M = 100$ 57
6.2 CPU network processing, 1–4 threads, $M = 200$ 59
6.3 CPU network processing, 1–3 threads, $M = 300$ 60
6.4 Average neuron firing frequencies 61
6.5 Time spent during input buffer reset 62
6.6 Time spent during firing check+LTP stage 63
6.7 Time spent during delayed input+LTD stage 64
6.8 Time spent during membrane potential update stage 65
6.9 GPU SNN processing—10 seconds 66
6.10 GPU kernel timings for $M = 100$ 67
6.11 GPU kernel timings for $M = 200$ 68

6.12 GPU kernel timings for $M = 300$ 69

7.1 GPU and CPU SNN simulation comparison 75

Chapter 1

Introduction

For decades, Moore's "law" of steady leaps in the speed of microprocessors held true. In today's world, however, it finds itself increasingly challenged by the hard realities of physics and the speed of light itself. As a result, a shift has been made towards horizontal rather than vertical expansion of performance. This, of course, means *parallelization*—increasing the number of processing units working in tandem on problems that allow themselves to be solved by the power of divide and conquer, greatly improving the efficiency of producing solutions to these problems.

One of the immediately appealing aspects of biological neural networks is their ability to work completely in parallel, giving hopes that their artificial counterparts would be able to exhibit the same performance traits. However, artificial neural networks have traditionally faced many challenges in this area, and some are inherently incapable of being truly parallelized. This thesis aims to show, amongst other things, how artificial spiking neural networks do not suffer from these limitations given certain biologically plausible preconditions.

1.1 Purpose and motivation

The overall purpose of this thesis is to look at the properties of artificial spiking neural networks (generally used in the abbreviated form SNNs throughput this thesis) that make them possible to be parallelized, how this may be implemented on parallel hardware and to what extent the end result lives up to the expectations of scalable performance gains. All implementational work is done with a full focus on consumer (i.e. desktop) level hardware rather than the more classical high performance computers or computer clusters that you seldomly find outside large research

institutes or government agencies. The reasoning behind this is simple—enabling efficient large-scale simulation of SNNs on consumer level hardware opens up the possibility for more wide-spread research in this area, as the hardware costs and simulation-time spent are no longer excessive. There have also been significant recent advances in parallel consumer-level technology, giving a high factor of relevance to this approach.

To achieve this, a theoretical spiking neural network model and its components will be described, and this theoretical model will be used as a base for two implementations—one for multi-core PC Central Processing Units (CPUs) and one for multi-core Graphics Processing Units (GPUs). All hardware used is readily and widely available on the consumer market.

It is not the intention or goal to create any optimally parallelized network implementations. For reasons that will be discussed in later chapters, this is highly non-trivial and would go beyond the scope timewise for this thesis. The goal in this case is to give proof of concept examples of *how* one might go about creating a parallel artificial spiking network on commodity hardware, both on multicore personal computer systems as well as new generations of graphics hardware and *what* many of the challenges and issues are with this venture, the prototypical nature of the software notwithstanding.

1.1.1 Why focus on spiking neural networks?

The obvious answer as to what it is about artificial spiking neural networks that makes them so compelling and relevant to explore in the context of parallel computation is simply that they provide the closest (not counting very low-level simulations) approximation to their biological spiking counterparts, and that they can be used to replicate the emergent phenomena found in the brain, the most sophisticated parallel computer in existence. As hinted to in the introductory paragraphs, given certain biologically plausible preconditions, we can get one small step closer to such a level of massive parallelization on modern desktop hardware. This approach, as we shall see in Chapter 3, works even in the face of completely arbitrary network topologies.

1.1.2 Why multi-core CPUs

In today's hardware world, hardly any new computers are sold without a multi-core CPU onboard. Adding more cores allows for running more ap-

plication code simultaneously, which leads to increased performance and improved response time when the computer is under load. At the time of writing, most budget-level computers—be they desktops or laptops—come equipped with at least 2 cores, with 3 or 4 cores available for those who are willing to pay a bit more and 8-core setups just around the corner. The raw processing power available to e.g. a 4-core CPU with each core running at 2.5 GHz is formidable, and is something researchers only a decade ago would have problems achieving even on supercomputers.

1.1.3 Why GPUs show a vast potential for neural networks processing

Although the amount of cores present in CPUs are on a steady rise, the number of cores on GPUs have essentially exploded in comparison [NVIDIA, 2008, pg2]. Graphics hardware has always been focused on—and have excelled at—processing many small potentially computation-heavy items of data in parallel (e.g. vertices in a triangle, pixels on the screen), but this has for the longest time only been useful for purely graphical applications. Throughout the not too distant years, this limitation has been gradually removed, allowing people to begin thinking outside the box and using the available power for other purposes. Now, with the latest generations of graphics hardware, this limitation has effectively ceased to be, giving nearly free reigns to *general purpose computation* on the GPU (also known as GPGPU).

The cores found on GPUs are far less complex than those found on CPUs since they make the assumption that the code they will be running are especially tuned for minimizing their limitations and maximizing their potential. As such, the transistors that would have been used for eg. caches or program flow control on the CPU can rather be used for fitting in more cores [NVIDIA, 2008]. Combine this with stellar arithmetic computation capabilities and internal memory bandwidth that goes above and beyond that found on even very high-end PCs and you get a hardware platform that is highly attractive (albeit challenging) for AI tasks that benefit from high levels of parallelization.

1.2 Existing research

Parallelization of artificial neural networks is not by any stretch of the imagination a new phenomenon, but previous research has primarily focused on large, distributed supercomputing clusters [Ananthanarayanan

and Modha, 2007] or specialized hardware [Seiffert, 2004]. The topic of performing such tasks on single-machine commodity hardware has not yet been investigated nearly as in-depth. This is hardly a conspiracy or a product of oversight, but rather a natural consequence of desktop computing simply not being able to reach the required performance numbers in the past.

To the author's best knowledge, there are no existing published papers giving an explicit comparison of SNN parallelization on CPUs and GPUs, but a previous attempt at a GPU SNN implementation can be found in [Bernhard and Keriven, 2006], although it was based on older GPU technology and did not feature any synaptic delays.

Towards the end of the thesis period (late March 2009), a paper was released outlining a highly effective network architecture based on the same principles as the ones found herein, but with several important optimizations that makes it far more efficient, both memory and performance-wise [Nageswaran et al., 2009]. Although it offers very attractive improvements to the algorithms and datastructures used, due to time constraints and lack of published implementational details, there was only time to adapt a few of these changes.

1.3 Structure of this thesis

Chapter 1 gives an introduction into the thesis topic, its purpose and motivation as well as outlining some existing research.

Chapter 2 offers a brief excursion into the realm of parallel programming and some of the challenges and concepts it involves. The tools that will be used to create the CPU and GPU implementations are also covered in reasonable detail here.

Chapter 3 is the main thesis chapter, describing both biological and artificial spiking neurons and the networks they comprise, as well as creating an abstract model of a parallel spiking network and the theory behind its operation.

Chapter 4 gives a rundown of the most important aspects of creating the two implementations.

Chapter 5 outlines the methodology behind the experiments performed to evaluate the potential and shortcomings of the presented SNN model and its implementations.

Chapter 6 presents the results that the experiments given in Chapter 5 outlined.

Chapter 7 offers an in-depth discussion about the results from both implementations and considers some of the non-trivial details that can drastically skew the results negatively.

Chapter 8 concludes the thesis as well as outlining some potential future work aspects based on the results and author's experiences presented herein.

Chapter 2

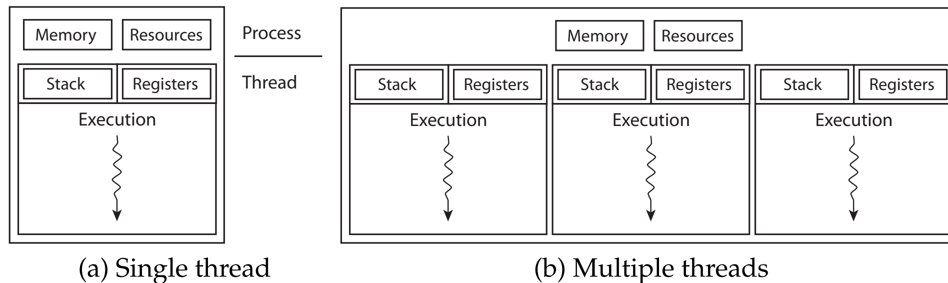
A parallelization primer

This chapter considers the computer science aspect of the thesis, and the main concepts and tools that enable the spiking neural network to be parallelized in the first place. It introduces concepts that will be used and referred to throughout the rest of the text. It is not meant to be in any way an exhaustive parallelization reference.

2.1 Introduction

As mentioned in Section 1.2, parallelization of (spiking) neural networks is not a new thing by far, but these efforts have predominantly been focused around high-performance computing, e.g. supercomputers such as IBM's BlueGene [Ananthanarayanan and Modha, 2007] or parallel clusters of computers [Izhikevich and Edelman, 2008, pg3595]. Given that this thesis focuses on consumer—that is to say, desktop—level hardware, these will not be covered in any detail.

Any attempt to parallelize a program or an algorithm (or parts of it) will be intimately tied to its problem domain. Some problems are what's referred to as "embarassingly parallel", meaning that they require very little effort to correctly parallelize. An example of this would be a genetic algorithm system wherein the fitness calculation of any given phenotype is completely independent from the calculations for all other phenotypes. The code for doing this might then have an arbitrary granularity in terms of how many processors it could be run on. Many problem domains are unfortunately not so easy to deal with. To return to the GA example, it becomes clear that if eg. the fitness calculation of a given phenotype requires it to interact in an environment that contains other phenotypes (that are also simultaneously trying to compute their own fitness), some sort of



(a) Single thread (b) Multiple threads

Figure 2.1: Overview of single-threaded and multi-threaded processes. Multiple threads in the same process share the same memory space and can therefore easily be set to work on different parts of the same datasets, improving performance.

overarching control—or *synchronization*—is required for these interactions in order to prevent total chaos.

When discussing parallel execution of any kind, the most central concept is usually that of the *thread*. In essence, a thread is a series of instructions executed to complete a task and the bookkeeping required to facilitate this (such as maintaining a per-thread stack, registers etc.). On modern operating systems, running a program involves creating a *process*, each of which has at least one or more threads. The threads in question may generally be scheduled on any available CPU. All threads in a process share the resources allocated to the process, including memory, file handles, network sockets et al. This shared access means that multiple threads can work on the same sets of data, but this comes with some snags, as outlined in the next section. Figure 2.1 shows a graphical example of how the threads in a given process operate with both private and shared resources.

Only so-called "shared memory" architectures will be considered here, meaning that all CPUs are able to access the same areas of memory in a uniform way. A very simplified diagram of this is shown in Figure 2.2. This is contrary to the *non*-uniform memory access architectures found in many supercomputers, in which different CPUs generally can access some parts of the memory fast and other parts slowly, depending on its locality relative to the CPU.

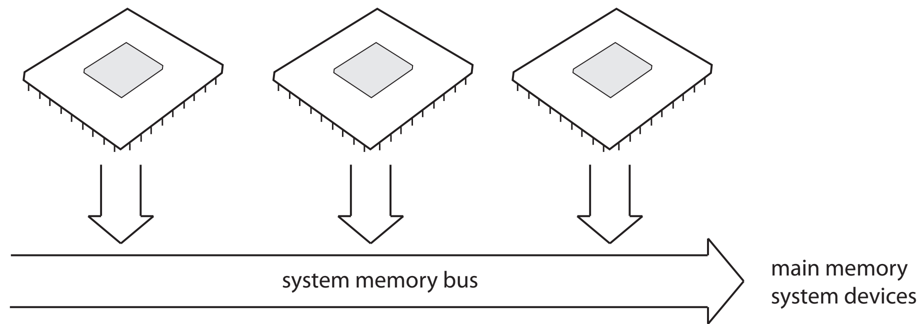


Figure 2.2: Diagram of a shared-memory system

2.1.1 Challenges

There are certain aspects of parallel programming that anyone attempting to parallelize a program will almost inevitable run into. Some of the more common ones (which are all relevant for this thesis) are:

For an overview of common approaches to parallelization, see Ambrus [2003].

Dependencies Whenever an operation requires to know the results of another operation before performing its own, there is a *dependency* between them. This may be seen as the primary limiting factor for whether or not (and in the case of the former—how much) a program may be parallelized. Whenever operations depend on each other, these will have to be performed in a proper order, that is to say *sequentially*. A graphical example of what this entails is given in Figure 2.3.

Non-linear performance gains When throwing n processors at a problem, the theoretically optimal result would be that the task would take n times shorter amount of time. In practice, this is almost never the case, as the bookkeeping operations needed for synchronization and scheduling, as well as inherently sequential operations will place an upper bound on the scalability achievable. This was postulated by Amdahl [1967]—later referred to as "Amdahl's law"—and formalized as the equation

$$S = \frac{1}{1 - P}$$

where S is the total possible speedup and P is the parallelizable fraction of the program. Example: if 90% of the program may be parallelized, and 10% must run sequentially, the maximum speedup $S = 1/(1 - 0.9) = 10$, or a factor of 10x over the original.

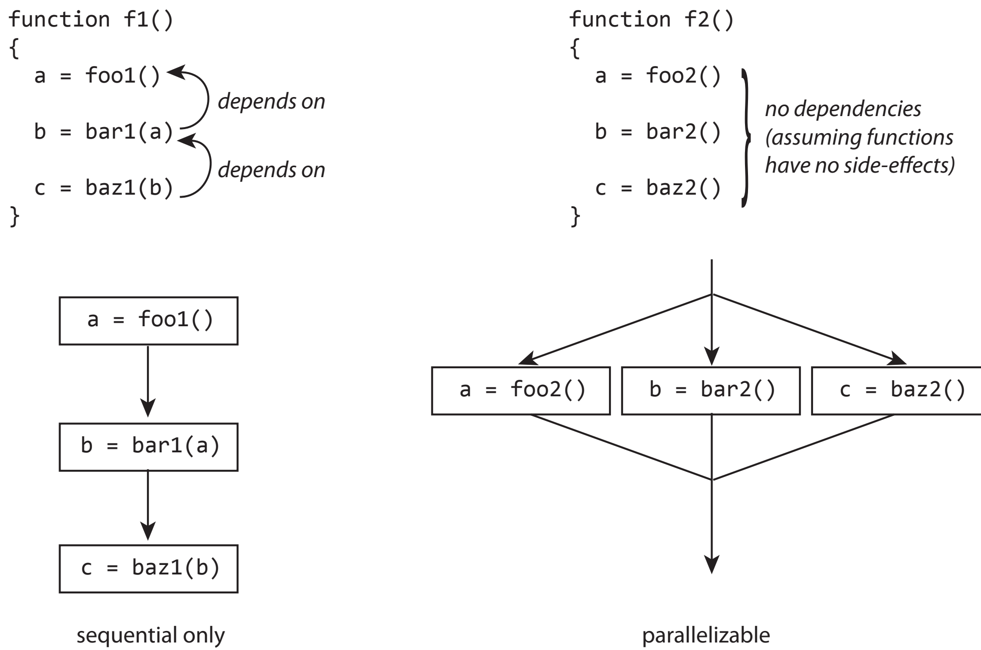


Figure 2.3: Dependencies may cause tasks to be impossible to perform in parallel by imposing a strong requirement of sequential ordering. On the contrary, a lack of dependencies would mean that we *may* perform the operations in parallel. The same principle applies if we imagine the 3 tasks in the figure to rather be 3 elements processed one at a time by a loop, with the sequential code not allowing the loop to be parallelized and vice versa.

A radically different computing model that is aimed at high performance parallelization from the bottom up is introduced in Section 2.2.2.

Synchronization When running code in parallel that operates on shared data, read-write operations on these will generally have to be synchronized somehow. This may range from something as simple as incrementing an integer to executing a "critical section" of code in which only one thread of execution may be at any given time. This is solved through the use of various synchronization primitives that ensure that this invariant is not broken. One important example of this is the use of so-called *atomic* memory operations. They are named as such due to their "indivisable" nature, that is, only one thread can perform the operation at a time. Using e.g. an atomic integer add is important because adding a number requires reading the value already stored at the location in memory, performing the addition and then writing the new value back. If a second thread that also wants to perform an addition comes in and reads the same memory location before the first thread has written to it, it will get a "stale" value, add to that value and then write it back, effectively losing the addition done by the first thread.

Failure to properly protect critical sections of code or shared data accesses will eventually lead to a *race condition*, which are notoriously hard to reproduce and debug. It is important to note that although critical for correctness, synchronization comes with some inherent overhead. This might be from having to "lock" the entire (or parts of) system memory bus when such an operation occurs, invoking the operating system's thread scheduler and so on.

2.2 Supporting tools

2.2.1 OpenMP

OpenMP [Gatlin and Isensee, 2005] is an established standard for easily parallelizing code in a cross-platform way on multi-CPU shared-memory systems, and is readily available for many of the biggest C/C++ and FORTRAN compilers, as well as a host of operating systems. Put succinctly, it allows a programmer to specify *regions* of code that would benefit from parallelization, which are then with the help of an OpenMP-supporting compiler transformed into implementation-defined thread handling and

`www.openmp.org`

task allocation code. The important part is that the programmer does not have to explicitly deal with any native thread management or the process of subdividing the tasks amongst the threads. Any OpenMP region will also run transparently on a single-processor system.

To give a brief example, consider a C/C++ for-loop processing a million elements:

```
for (int i = 0; i < 1000000; i++)
  do_work(i);
```

Given that all elements $i \in [0, 1000000)$ are independent of each other, we can use OpenMP to speed things up by utilizing a special compiler declaration that tells it that a parallel region should be created. In this case we can use a declaration for automatically handling for-loops.

```
#pragma omp parallel for
for (int i = 0; i < 1000000; i++)
  do_work(i);
```

Now the OpenMP runtime will, when we reach this part in the program, subdivide the loop iterations and schedule them on the available worker-threads (generally, the total number of threads available—including the master thread in which the program process runs—is equal to the number of processor cores on the system). A graphical example of a hypothetical system with 4 processor cores is given in Figure 2.4

The default scheduling policy is to simply statically schedule n/m operations on each thread, where n is the total number of operations and m is the number of threads. It may be noted that only "simple" for-loops where OpenMP can determine the number of operations in advance can be parallelized with this method [Gatlin and Isensee, 2005]. This means no `do-while` loop constructs can be work-shared.

Due to the relative ease of use from a multiprogramming point of view, OpenMP will be utilized as the method of parallelization for the CPU version of the spiking neural network model.

Although one can assume the that the OpenMP implementation used by the compiler is highly optimized, there is still inherent overhead present with the thread scheduling and task allocation. According to experiments performed in [Gatlin and Isensee, 2005], the minimum number of loop iterations that are required before a parallel region starts to show a measurable benefit is in the range of thousands when the amount of work done per iteration is small.

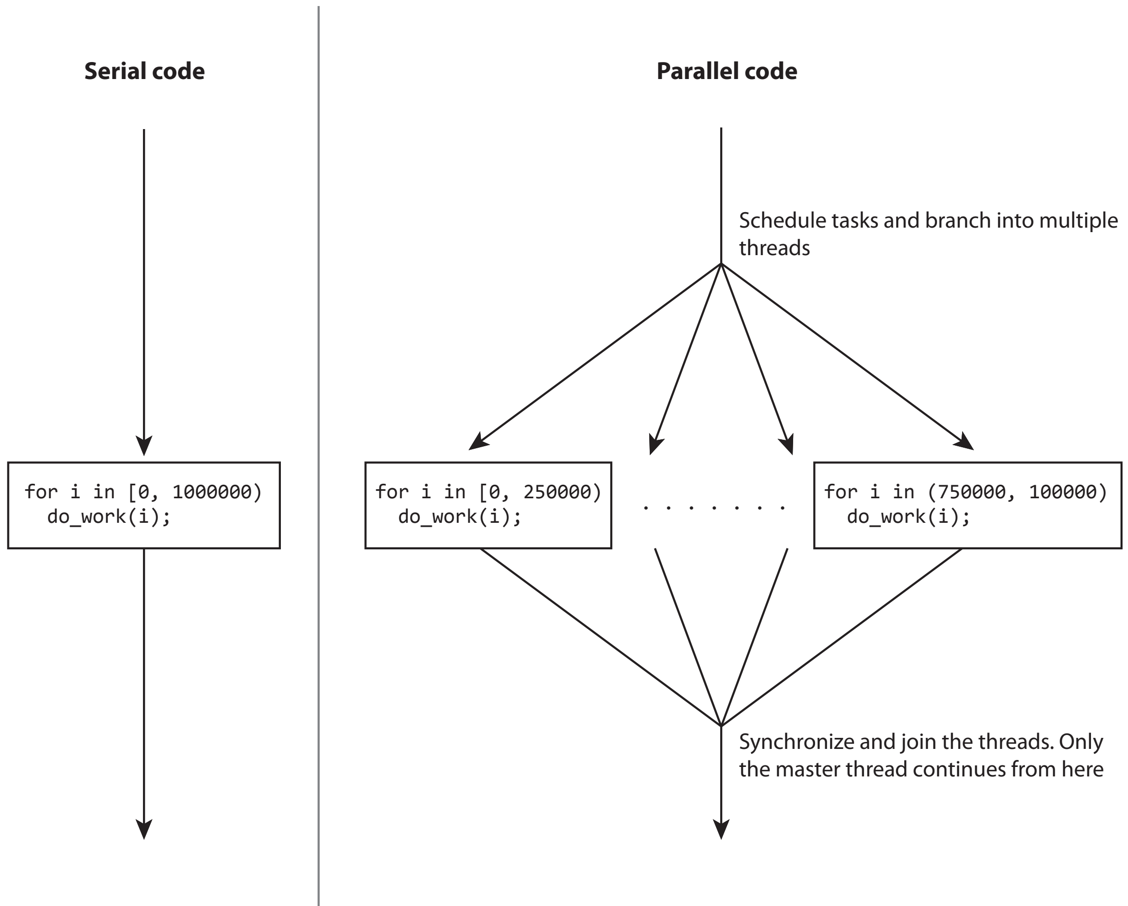


Figure 2.4: Control flows for serial and OpenMP parallel versions of the same code. In the serial version, a single thread (the program's "master thread") does all the work, while in the parallel version the work is divided into evenly sized chunks and processed by the available threads—in this case the program's master thread and 3 worker threads.

2.2.2 NVIDIA CUDA

CUDA (Compute Unified Device Architecture) is a platform for massively parallel high-performance computing on the recent generations of GPUs from NVIDIA. It has seen great reception for areas such as computational chemistry, physical modelling etc [Nickolls et al., 2008]. A core goal for CUDA is that parallelism should be *transparent*, meaning that software written for it should not have to know nor care how many physical processors are present, and they should be able to scale appropriately as a result.

www.nvidia.com/object/cuda_home.html

Typical modern CUDA-enabled GPUs have several dozen thread processors, each capable of executing multiple threads in parallal, as well as a memory bandwidth available to its internal memory that is many times greater than that available for a regular CPU to the standard system memory. Together this creates a formidable performance potential for code that can be adapted to its programming model.

The core concept in CUDA is the *kernel*, a small program that is compiled down to GPU-native code and can be executed in parallel over a set of threads. These threads may be organized into a hierarchy of *thread blocks*, offering a level of thread cooperativity across this block, as well as access to a block-private memory space. On the next level of the hierarchy we find the *grid* of blocks that may be executed in parallel. See Figure 2.5 for a graphical representation of this hierarchy. A CUDA application specifies the dimensions for the grid and blocks whenever it "launches" a kernel, enabling it to tailor these for the problem at hand (rather than the hardware at hand). An example might be running a 16×16 convolution kernel over an image of dimensions $W \times H$, where each kernel invocation outputs 1 pixel. In this case, the application simply requests that the kernel be launched with $W \times H$ blocks of 16×16 threads each. The number of blocks may be (read: usually is) far greater than the number of available processor cores on the hardware, as these are scheduled automatically to the available Streaming Multiprocessors (SMs) by the hardware [Nickolls et al., 2008].

The great level of parallelism comes from enforcing that different blocks must be able to be processed completely independently of each other, as there is no way to synchronize between kernels in different blocks (aside from atomic operations) and blocks may be run in any order, at any time. A CUDA application often consists of multiple kernels that can be launched in sequence (although *different* kernels aren't themselves run in parallel), with each kernel comprising the logic for a certain task.

There is a cost to all this potential, however, and it comes in the form of

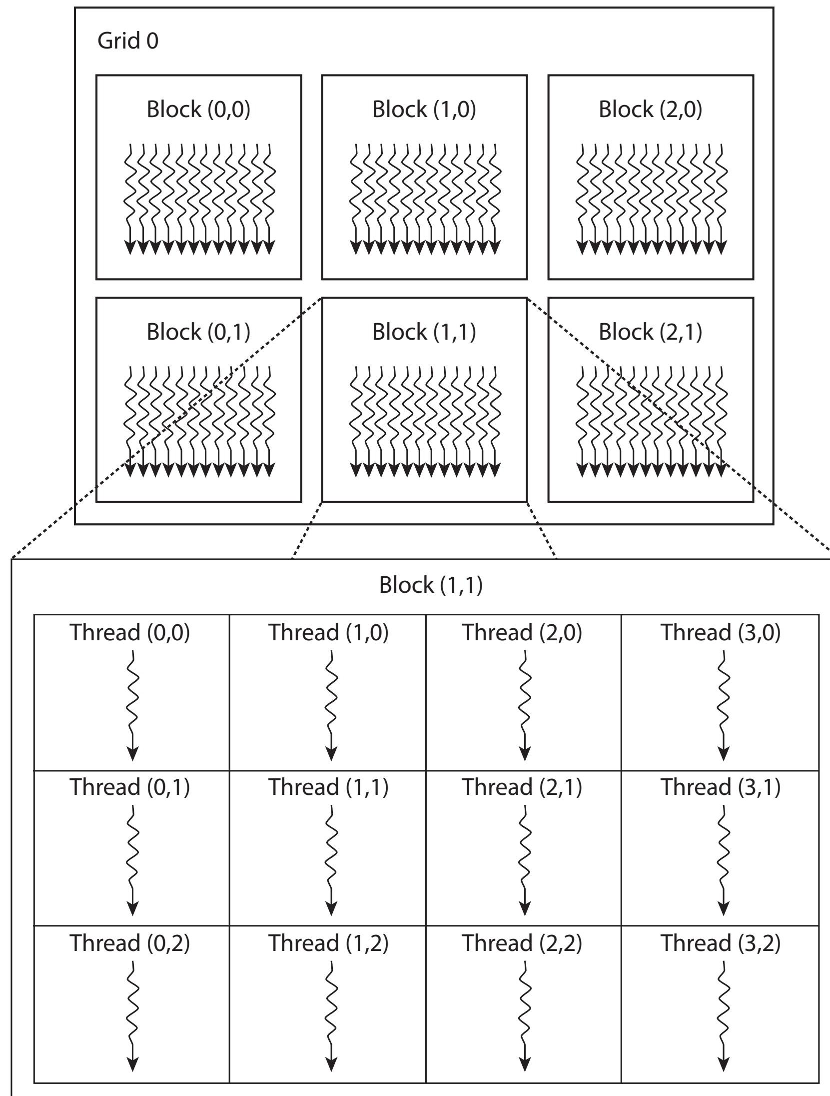


Figure 2.5: Threading hierarchy in CUDA (inspired by figure in [NVIDIA, 2008]). In this case, the kernel has been launched with a 2-dimensional grid of size 3×2 blocks and each block with a 2-dimensional set of threads of size 4×3.

certain requirements as to how the kernels must be programmed in order to actually truly benefit from it.

The first is a product of how the processors execute their threads. A single GPU multiprocessor is capable of running 32 threads *simultaneously* as long as these are all executing at the exact same point in the kernel (i.e. they are at the same instruction on the same path of execution) due to its ability to execute a single instruction in parallel for all its threads. If say just a single one of these 32 threads choose a different branch than the others (e.g. it executes the body of an if-statement while the others do not), all 32 threads must have their instructions executed *serially* until their paths once again converge, at which point the processor resumes parallel operation. This means that unless code is written to ensure such divergences are limited in their count and duration, execution throughput will be severly reduced. On the upside, thread scheduling and context switching is completely hardware-based on has *zero* overhead [NVIDIA, 2008]. This stands as a stark contrast to that of OpenMP and CPU threading in general.

The 32 thread grouping is referred to as a *warp*

The second is how memory must be accessed in order to give optimum throughput. As mentioned above, a single multiprocessor is capable of executing an instruction for many threads at once. An optimization technique is utilized so that when 16 threads simultaneously attempt to access memory locations that are continuous and fit certain alignment criteria, only a *single* memory transaction needs to be generated rather than 16, cutting the load on the GPU memory bus tremendously and leading to throughput in the 100 GB/sec range on newer GPUs. This process is referred to as memory *coalescing* [NVIDIA, 2008], and is vital for all CUDA applications to consider if they want to get anywhere near optimum performance. It may be mentioned that CUDA comes in several hardware versions, or *compute capability* versions, and version 1.2 removes many of the limitations for such coalescing [NVIDIA, 2008, pg57], but as the only hardware available for this thesis is a version 1.1 card, these limitations are all in place. Figure 2.6 shows a graphical representation of what is required for thread accesses to be coalesced into a single memory transaction.

The 16 threads come from the upper or lower part of a warp, and are referred to as a *half-warp*

There are additional challenges involved in programming for the CUDA platform, but these are sufficiently low-level that they are beyond the scope of this thesis. Interested readers are referred to [Nickolls et al., 2008] and [NVIDIA, 2008].

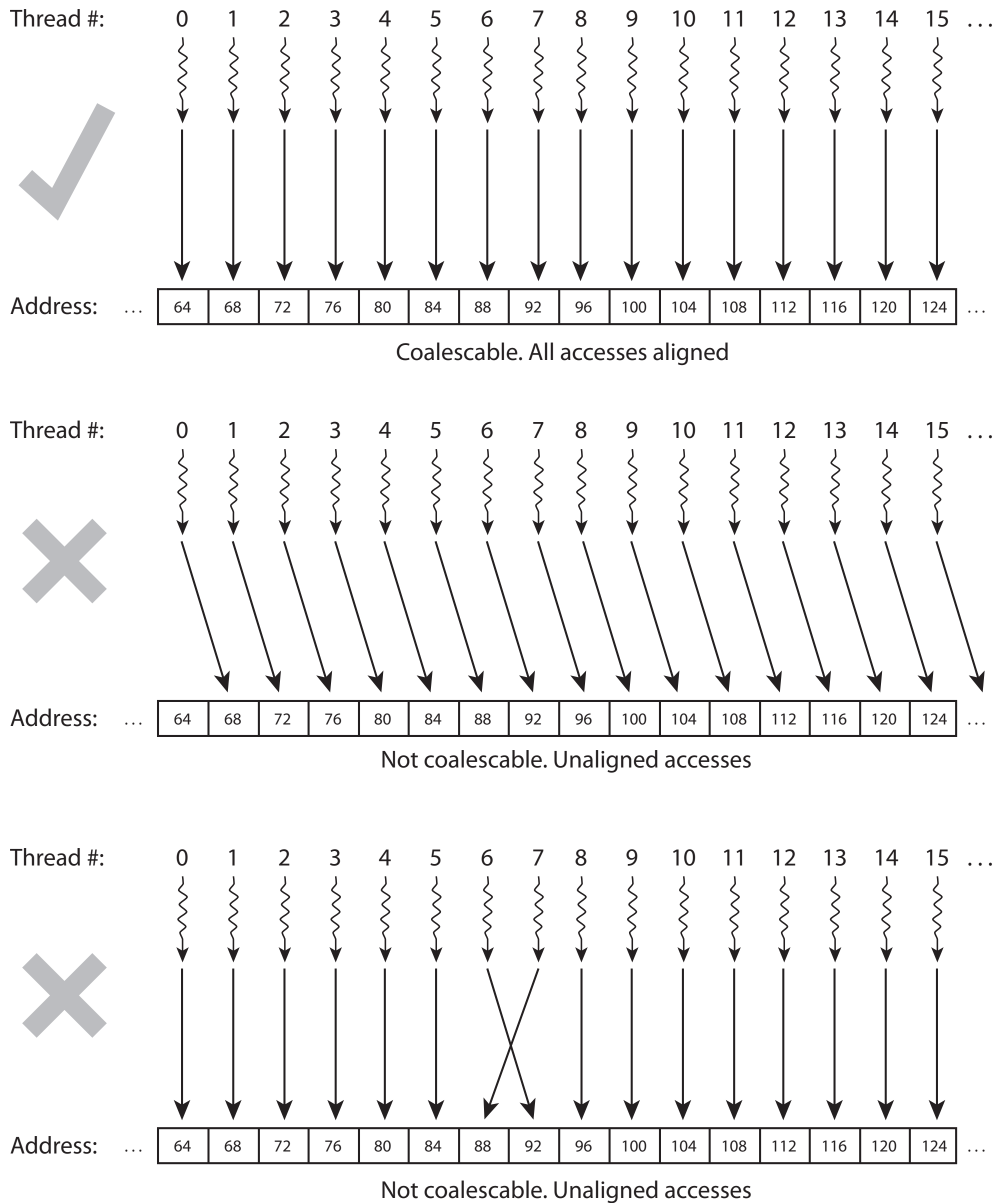


Figure 2.6: The first example has all threads accessing memory in an aligned way, thereby resulting in only 1 memory transaction. Both subsequent examples are *not* aligned, resulting in 16 memory transactions and with the resulting performance understandable impacted. Adapted from figure in [NVIDIA, 2008]

Chapter 3

Artificial Spiking Neural Networks

The first part of this chapter revolves around the abstract spiking network model and its biologically plausible components. Second part dives deeper into more concrete ways of representing such a network and then finally how to actually realize it in the form of pseudo-code, identifying and discussing areas of parallelization as it goes. The spiking network model presented herein will be used as a direct base for both the CPU and GPU implementations.

3.1 Introduction

3.1.1 The biological spiking neuron

To cover the terminology that will be used throughout the rest of this thesis, a simplified representation is given of a biological spiking neuron. Each neuron has an extended, tree-like structure of dendrites and axons, the *dendrites* bringing the input into the cell body (*soma*) and the *axon*, originating at this cell body, sends the output signal to other neurons (see Figure 3.1). These signals are based on the movement of charged atoms (ions). Whenever ions flow into and out of the neuron, the neuron's internal charge (voltage) changes in relation to that outside of its membrane. This charge is referred to as the neuron's *membrane potential*. If this potential reaches a certain threshold, the neuron is said to "spike" and an electrical pulse (the *action potential*) is sent down the axon, propagating this action potential down to other neurons it is connected to at their dendrites. This effectively resets the membrane potential and causes a refractory period in which the neuron can not fire. [O'Reilly and Munakata, 2000, pg27–

32]. There is a certain *conduction delay* involved when spikes travel down its transmission channels (axon, synapse, dendrite, ...), meaning that the transfer of action potential is not instantaneous [Izhikevich, 2006].

At the junction point between a sending neuron's axon and a receiving neuron's dendrite, we find the *synapse*. The synapse works as a modulator of the signal by affecting its strength (also known as synaptic efficiacy). The modification of this efficiacy is believed to be what causes learning in the brain.

Throughout this thesis, the term *presynaptic* will be used for the neuron sending action potentials (as it is before the synapse), and *postsynaptic* for the receiving neuron (as it is after the synapse).

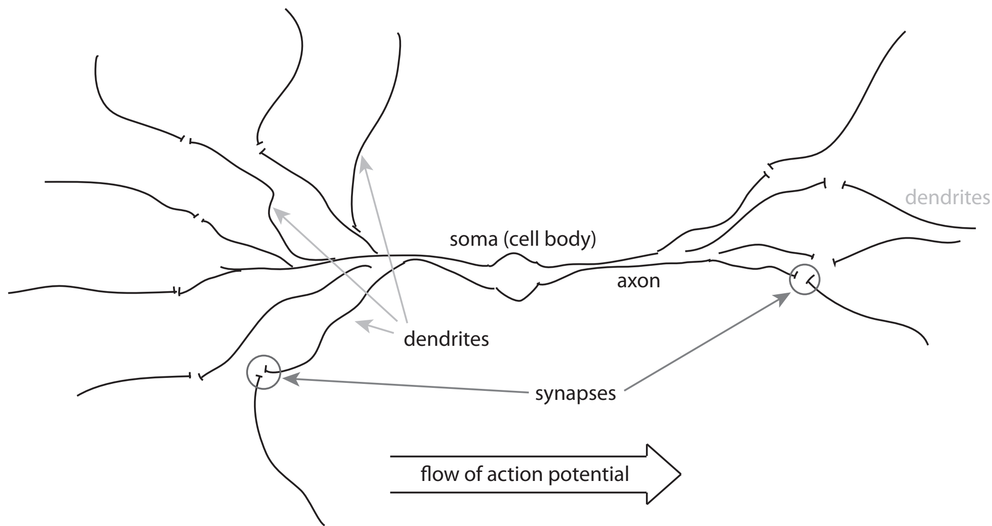


Figure 3.1: Simplified biological neuron and its connectivity.

Readers are referred to e.g. [O'Reilly and Munakata, 2000] for in-depth information on the electrochemical and biophysical properties of biological neurons and neural networks.

3.1.2 Simplifications in conventional neural networks

In conventional error-driven neural networks (primarily looking at the incredibly common backpropagation network [Callan, 1998]), the properties of biological neurons are simplified down by considering the axon→synapse→dendrite connection as one abstract synaptic weight factor for all of a neuron's synapses. The membrane potential and spiking is abstracted

away by having the output of a neuron be the real-valued integration of its input (weighted by the synaptic efficiacies per connection) be run through a function. A common choice here is the sigmoidal function, as it is continuous yet still provides a form of thresholding. This real-valued output is often considered to represent the "average firing rate" for a neuron. The notion of synaptic conduction delay is abstracted completely away, meaning transmission is instantaneous.

These neurons are then generally arranged in sequential interconnected layers that are processed in a "left to right" fashion. Learning takes place by modifying the weights of the synapses according to the difference (error) in actual output vs. desired output for each layer, and this is performed in a "right to left" fashion.

3.1.3 Spiking neural networks

A spiking neural network distinguishes itself from the majority of conventional neural networks in a very central way. Whereas the output of any neuron in eg. a feed-forward network is real-valued (usually in the range $[0,1)$), the output of a spiking neuron can be considered discrete and binary—either a spike is "fired" or it's not. Although one might be tempted to think this is the same as having a neuron with a real-valued output that is latched to either 0 or 1 depending on whether or not the sum of all inputs is above a certain value, biological spiking neurons (as well as the SNN model considered in this thesis) operate with some very different characteristics:

- Rather than depending only on the current input, the membrane potential gets modified by the input *over time* according to its neural dynamics [Izhikevich, 2003].
- Due to the transmission channel conduction delay, another aspect of time is introduced. This will prove critical for the parallelization of ASNNs.
- Inhibition is provided only from neurons with certain characteristics that allow them to fire rapidly and strongly, thus being able to prevent the network from having a "seizure" from uncontrolled firing. The non-inhibitory neurons are referred to as *excitatory* neurons, as they provide positive input to their receiving neurons. In eg. a backpropagation network, inhibition will be provided from weights with a negative sign, but all the neurons generating such inhibitory in-

puts will generally have the same characteristics as those generating excitatory ones.

Although there are many different types of neurons in a biological network [Izhikevich, 2003], only two will be used for the SNN model in this thesis.

3.2 Artificial neuron models

When working with a spiking network architecture, the choice of how to represent the neurons themselves is just as important as e.g. choosing a suitable transfer function for a real-value output architecture (such as the ubiquitous sigmoid function). Depending on the desired biological realism of the network, spiking neurons and their connections to other neurons may be represented at vastly different levels, ranging from near molecular level when the physical processes themselves are to be studied, to simply duplicating the end-result via mathematical equations. This end-result is usually centered around the neuron's membrane potential and the changes made to it by synaptic input and time.

It should come as no surprise that simulating many neurons at a very low level is incredibly computationally expensive, and not something that can reasonably be assumed to be possible to do efficiently on commodity hardware (at least not at the time of writing). As such, only those models that abstract away the underlying electro/biochemical processes will be considered for this thesis. But even these have significantly different performance and biological plausibility/accuracy. A very complete, comparative listing can be found in [Izhikevich, 2004].

Some models are briefly introduced here, going from the computationally simple leaky integrator, the "reference" model to which most artificial neural network architectures can trace their roots, the Hodgkin-Huxley model and finally to the recent Izhikevich model which combines good computational performance with high biological plausibility.

All models considered in this thesis revolve around point neurons, i.e. representations where the geometrical extent of the neuron has been shrinked down and simplified into a single point [O'Reilly and Munakata, 2000, pg24].

3.2.1 Leaky integrator model

Also known as the "leaky capacitator" model, a leaky integration neuron takes temporality into account by *decaying* a neuron's activation level with

time, meaning a neuron that does not receive sufficient input activation to fire will gradually see its membrane potential return to the resting level. A common analogy here is a water basin into which water may be poured in at the top, but also flow out (i.e. leak) at the bottom through a hole. The size of this hole determines the rate of leaking.

Although very simplistic in its representation, it's also correspondingly simple and fast to calculate. It lacks the ability to simulate many biological phenomena such as spike frequency adaptation [Izhikevich, 2004, pg1064], wherein the frequency of firing is high during the onset of stimulation but then adapts (an example of this can be seen in Figure 3.2[1]). Several enhancements to—and discussions of—the leaky integrate and fire model can be found in [Izhikevich, 2004, pg1067].

3.2.2 Hodgkin-Huxley

The Hodgkin-Huxley model is considered one of the most important and accurate neuronal models in computational neuroscience, utilizing a set of differential equations and tens of parameters that are based directly on the corresponding biological measurements [Izhikevich, 2004]. This has the side effect of leading to extremely high computational costs, and due to this and its overall complexity, it will not be covered here in any more detail. For its original paper, see [Hodgkin and Huxley, 1952].

3.2.3 Izhikevich model

Fairly recently introduced, the Izhikevich spiking neuron model [Izhikevich, 2003] offers highly biologically plausible neuron behavior at a relatively low computational cost. It is given as a pair of differential equations

$$\begin{aligned} v' &= 0.04^2 + 5v + 140 - u + I \\ u' &= a(bv - u) \end{aligned} \tag{3.1}$$

where v is the membrane potential and u is the recovery variable. The latter accounts for the activation of K^+ ionic currents and the inactivation of

[1] Purely anecdotally, a very early prototype of the spiking network implementation—arranged in a feed-forward manner—did in fact use leaky integrator neurons that spiked after a certain threshold, and in small networks the lack of such an adaptation mechanism seemed to make output neurons fire very seldomly due to the spikes reaching them being too sparsely distributed to provide sufficient activation. With a neuron model more sensitive to stimulus onset, this might have been mitigated.

Na^+ and provides negative feedback to v [Izhikevich, 2003] (cf. the integrate&fire activation leakage). a represents the time scale of the recovery variable and b represents the sensitivity of the recovery variable.

The neuron is determined to be firing if the membrane potential goes over a certain threshold, in this case 30 mV, in which case it is reset to a resting potential c and the recovery variable is updated with a parameter d.

$$\text{if } v \geq 30 \text{ mV, then } \begin{cases} v \leftarrow c \\ u \leftarrow u + d \end{cases} \tag{3.2}$$

A common value for c is -65 mV, which means that after a neuron has fired, its membrane potential goes low enough that it goes into a refractory period (see Section 3.1.1), preventing it from firing for a certain amount of time.

By choosing different values for parameters a and d, we effectively determine the behaviour of the neuron, in particular the duration of this refractory period. For "regular spiking" (RS) excitatory neurons (which is the most common type of neuron), we use the values $a = 0.02, d = 8$. For "fast spiking" (FS) inhibitory interneurons (which provide inhibition to the excitatory neurons) [Izhikevich, 2006, pg277], we use the values $a = 0.1, d = 2$. Figures 3.2 and 3.3 illustrate the behaviour of two randomly selected RS and FS neurons, respectively. It is easy to see how the FS neuron fires at a far higher rate than the RS neuron. FS is well suited to represent interneurons, as they must be able to provide sufficient inhibition even in the face of many firing excitatory neurons. As it strikes a fince balance between computational efficiency and biological plausibility (supposedly being at the level of Hodgkin-Huxley), the Izhikevich model is the one that will be used for all implementations in this thesis. There is however no inherent dependency on this model in the spiking network model itself—it should be fairly trivial to use any other set of spiking point neuron equations as a drop-in replacement, if this should be desired.

3.3 Learning in SNNs

Most artificial neural networks would be ultimately pointless unless they had some way of learning, be it supervised learning ("learning by example"), unsupervised learning (clustering, "learning by observing input") or otherwise[2]. As such, the SNN model considered herein will also re-

[2]Saying "most" here to avoid including networks whose synaptic weights have been modified through eg. evolution through genetic algorithms rather than through explicit or implicit learning.

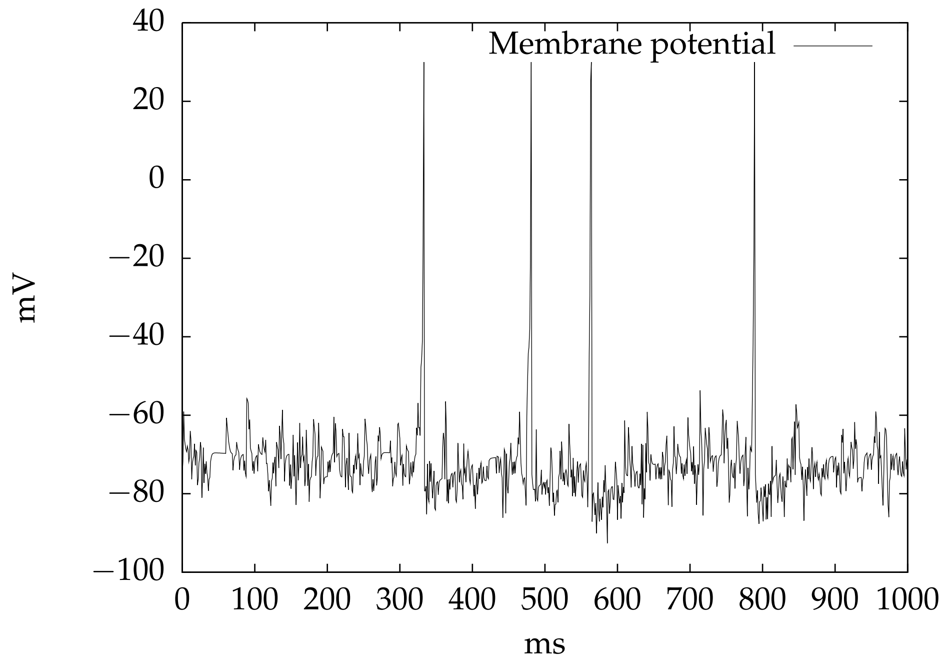


Figure 3.2: Izhikevich Regular Spiking (RS) neuron

quire that learning is capable of parallelization.

A common trait of most traditional artificial neural networks is that their learning is done through minimization of error, or *gradient descent*, wherein errors are propagated backwards in the network (and potentially time, such as with the Simple Recurrent Network [Callan, 1998] and its architectural siblings). However, SNNs are discontinuous in time due to their explicit spike timings, leaving such methods out of the question unless special considerations and simplifications are made [Kasiński and Ponulak, 2006]. This begs the question of *what* exactly to learn in a SNN, since the fairly intuitive notion of error generally speaking no longer applies. Panchev and Wermter [2004] rather eloquently summarizes it: *"The task of the plasticity algorithm is to adjust the weights of the neuron, so that for a particular set of spike trains, it is able to synchronise the peaks of the partial membrane potentials, and therefore maximise the response of the soma membrane potential"*. The actual task of training an SNN is far beyond the scope of this thesis, but interested readers may refer to [Kasiński and Ponulak, 2006], although it only considers supervised learning—for the purpose of this thesis, unsupervised learning is what will be used since we're not dealing with any explicit learning tasks.

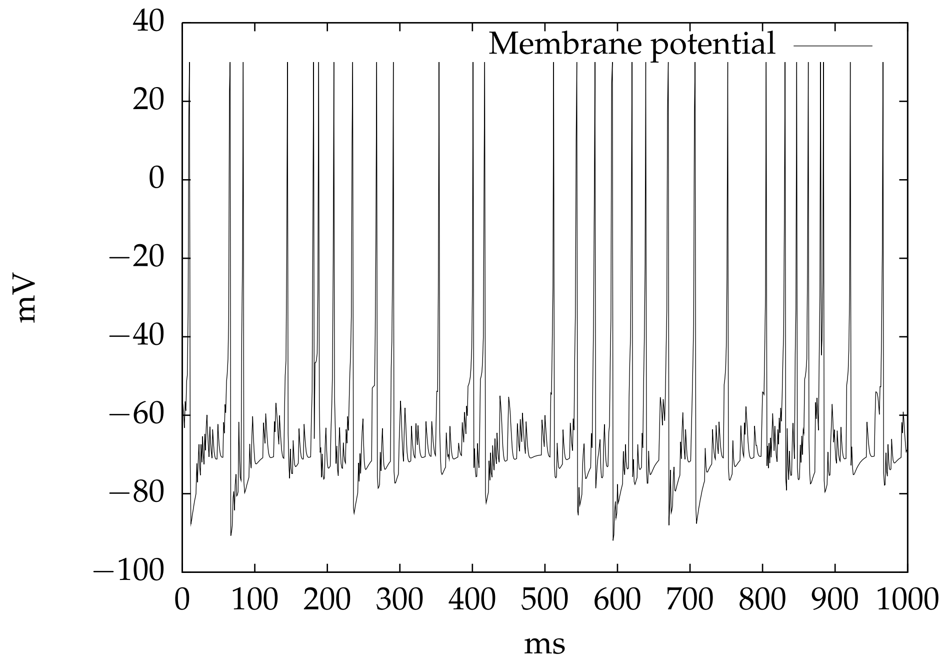


Figure 3.3: Izhikevich Fast Spiking (FS) neuron

3.3.1 Hebbian learning

Anyone interested in biological neural networks have most likely encountered citations of Donald Hebb's synapse postulate[3] often enough for it to take on a near catchphrase nature. It defines the strengthening (or weakening) of synaptic efficiacies based on the correlation of firings by the presynaptic and postsynaptic neurons. What we want to see is a strengthening of the synapse between two neurons if the first neuron contributes to the other neuron firing, meaning that a spike from the first neuron reached the second neuron *before* it fired. Conversely, if a spike from the first neuron reaches the second neuron *after* it has fired, we want the synapse to be weakened, since there's no correlation. This phenomenon is known as long-term *potentiation* (LTP) and long-term *depression* (LTP), of synaptic efficiacy, respectively. A graphical example of this process is shown in Figure 3.4

As the original Hebbian rule is inherently unstable (i.e. it causes weights

[3] *"When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased"* [Hebb, 1949]

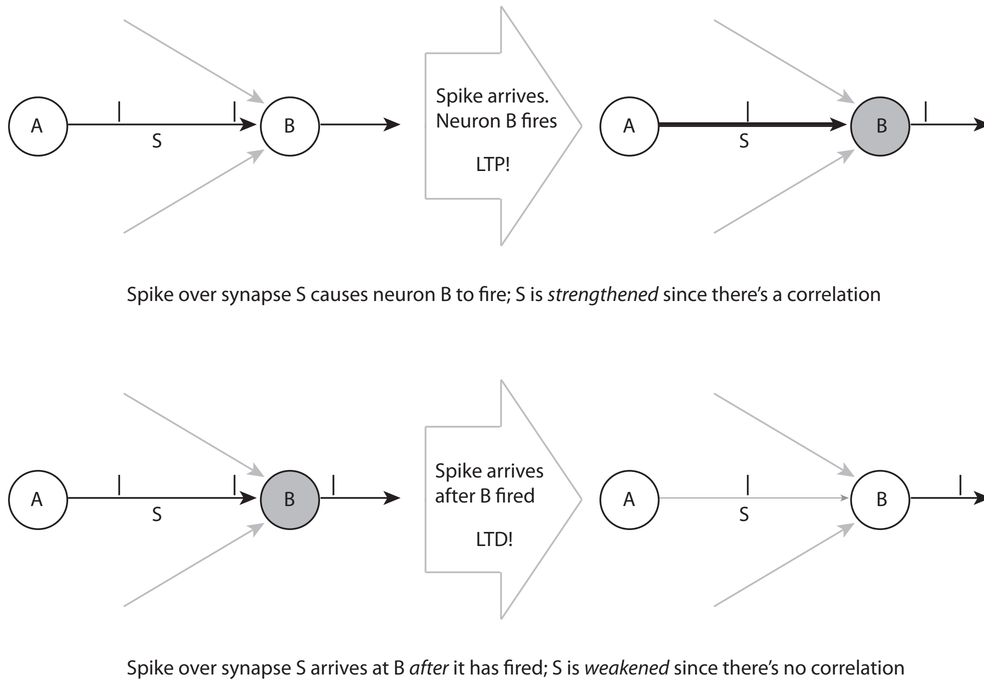


Figure 3.4: LTP and LTD based on spike arrival and neuron firing times

to grow exponentially), it faces severe problems when applied to network learning tasks [O'Reilly and Munakata, 2000, pg124]. As a result, modified versions such as the Generalized Hebbian algorithm are generally used in practice. These are still commonly rooted in linear neural outputs, so we need something that works with discrete spike timings rather than average firing frequencies and also has a very high level of biological plausibility.

3.3.2 Spike-time dependent plasticity (STDP)

Spike-time dependent plasticity (also known as Hebbian temporally asymmetric synaptic plasticity [Izhikevich, 2006, pg252]) offers a purely local mechanism of modifying synaptic efficiacy and encouraging synaptic competition that correllates with experimental observations and Hebbian principles. It provides for both LTP and LTD of synaptic efficiacy. Incidentally, this is exactly what we need.

From Song et al. [2000], we have the STDP equation

$$F(\Delta t) = \begin{cases} A_{+} \exp(\Delta t/\tau_{+}) & \text{if } \Delta t < 0 \\ A_{-} \exp(-\Delta t/\tau_{-}) & \text{if } \Delta t \geq 0 \end{cases} \tag{3.3}$$

Δt is the delta between the times when a spike last arrived at the postsynaptic neuron over a given synapse and when the postsynaptic neuron actually fired. A_{+} and A_{-} determine the maximum synaptic efficiacy modification for LTP and LTD, respectively. τ_{+} and τ_{-} specify the pre/post synaptic spike interval range itself over which STDP is performed [Song et al., 2000]. A common value for both values of τ is 20 ms. To examplify what this means for the shape of the function, see Figure 3.5.

As $F(\Delta t)$ is an exponential function whose rate of change increases as $\Delta t \to 0$, high synaptic modification changes will only occur when the time elapsed between the spike reaching the soma and the neuron firing is also close to zero. What this means for the learning-properties of the network is that those neurons that are strongly correlated will have their synaptic efficiacies proportionally strengthened, leading to a stronger binding for future spike firings, and vice versa for neurons that are strongly disassociated.

3.4 SNN parallelization

As we have looked at the properties of spiking neural networks and an applicable learning rule, it's natural that we explore how this allows us to process neurons in parallel. But before this, in order to better understand the problem, let's first have a quick look at why conventional neural networks are *not* optimal for such ventures.

3.4.1 Limitations of conventional artificial neural networks

Many artificial neural network architectures have inherent challenges when it comes to the issue of parallelization. This is due to the serial dependencies that are formed in any network where the output of a neuron requires the knowledge of the output of other neurons, and those outputs in turn require outputs from other neurons etc. and all these neurons take as their input the output of other neurons produced in the very same time step. Layered networks generally have each neuron and synapse in a given layer operate independently from all other neurons and synapses in the same layer, so *within the layer itself* there is room for parallelization for

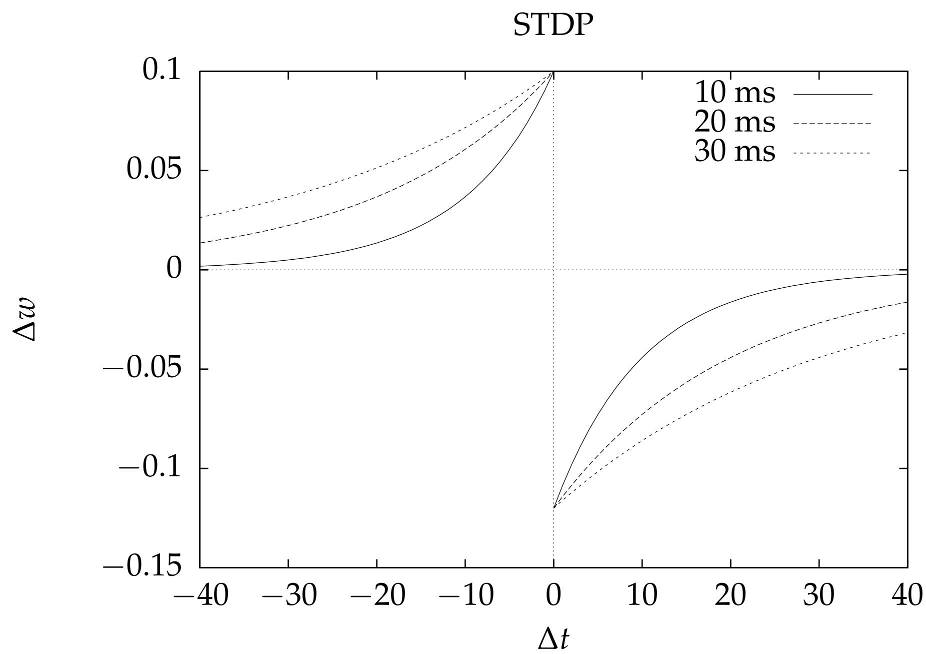


Figure 3.5: STDP for Δt showing the effect of different interspike intervals $\tau_+, \tau_- \in \{10, 20, 30\}$ ms. We can see how larger values of τ creates a greater window in which synaptic efficiacy may be changed, and vice versa for smaller values. For all these plots, $(A_+, A_-) = (0.1, 0.12)$.

both of these. There is however no possibility to parallelize the computation of the individual layers themselves, as (assuming a feed-forward network) layer L_n depends on L_{n-1}, which depends on L_{n-2} and so on. To amortize this problem, there have been significant efforts oriented towards specialized neuron processing hardware, wherein the number of neurons in a layer would optimally be equal to the number of processors, allowing the network to operate in synchrony [Seiffert, 2004], but the core issue of sequential layer processing still remains.

For learning, non-local learning rules such as backpropagation do not make the situation much better. Again, the parallelization potential is constrained to be layer-local as errors are propagated backwards in the net, the error computation of neurons in layer L_n being dependent on the error of neurons in layer L_{n+1}.

Although these dependency issues might not seem like much of a problem in practice with layered networks, as you still have a fair degree of parallelization, things quickly turn worse when network connectivity is arbitrary, as you would effectively have to build a full graph of dependencies in order to determine the proper order in which to process neurons and their synapses so that their outputs would reach the proper recipients. Needless to say, this is not a good starting-point for any parallelization attempts, as the number of neurons that are independent of each other will often be too small to counterweigh the overhead of thread synchronization et al.

3.4.2 Synaptic conduction delays as an enabler of parallelization

The nature of SNNs with synaptic conduction delays lets us overcome the integration and firing dependencies, and with a suitable local learning rule (such as STDP), lets us overcome the learning dependencies. It also allows for *completely arbitrary* network topologies without this impacting the parallellization potential, something which is critical for realistic simulations. In fact, the SNN model considered in this thesis is completely randomly connected.

The critical assumption for our parallel SNN model is that for any two neurons where there exists a connection between them, this connection will have a conduction delay > 0. That is to say, there are *no zero-delay synaptic connections* in the network. This is the most crucial assumption, and is also biologically plausible—propagating a signal from point A to point B will invariably incur some form of delay, no matter the conduction

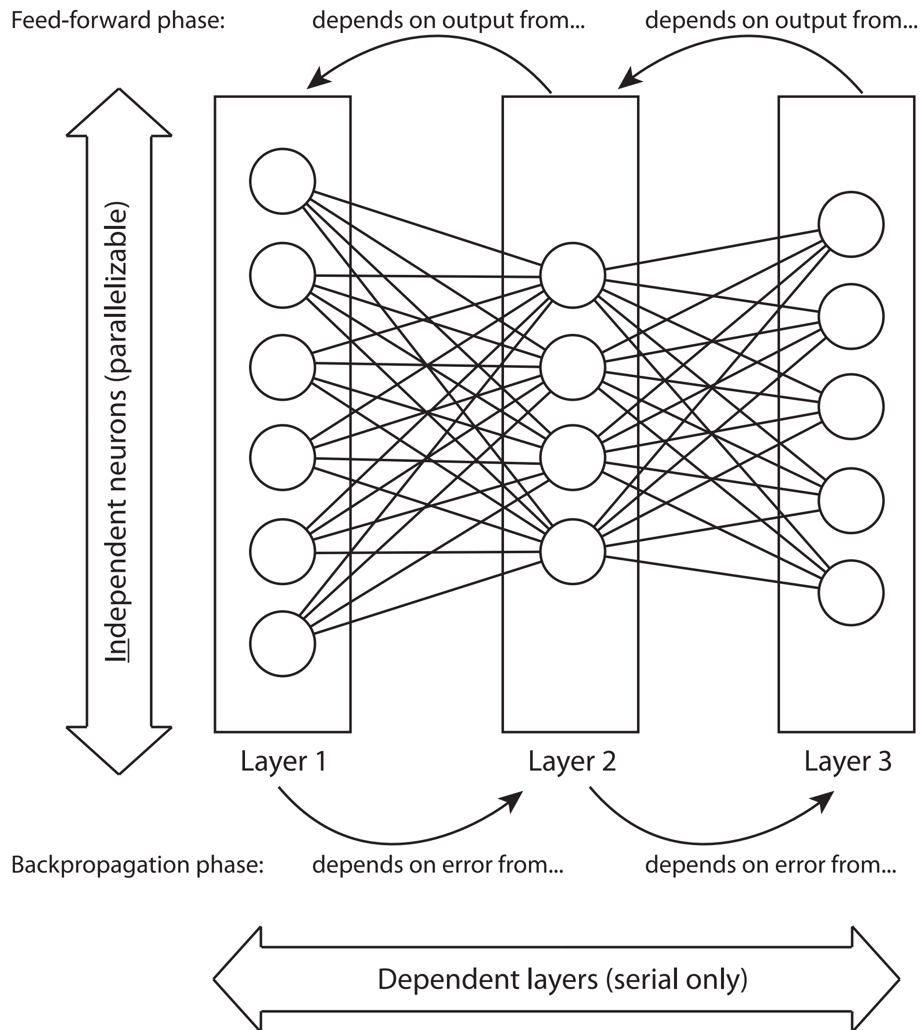


Figure 3.6: Parallelization issues with conventional, layered neural networks. The lack of synaptic connectivity between neurons *within* each layer ensures these can be processed in parallel. However, layer 1 must be processed in full before layer 2 may be processed at all and likewise with layer 2 before layer 3. With backpropagation learning, the dependency chain is reversed, but still in place.

speed of the underlying physical signal transmission channel. A discussion of this can be found in [Morrison et al., 2007, p52]. For the sake of simplicity, we'll assume that the SNN operates with a discrete simulation time step and that spike propagation takes at least 1 such time step.

The reason for synaptic delays being required is that they enable us to know that which is not possible for conventional (instantaneous) networks—all relevant network state for the *current* simulation time step! To see why, imagine a single spiking neuron firing its action potential at time step t. How many neurons require knowing this neuron's output at this time step, thus causing a dependency? The very short answer: *none*. Having an invariant that states that there is at least a 1 time step delay between when a neuron fires and when it reaches its destination means that it's impossible for the output of any neuron at time t to actually have any effect what so ever at that same time step t. Recall that a neuron's membrane potential is only affected when a spike reaches its soma, thus making it not care about those that are "in transit" to it. The soonest any neuron output at time t will be relevant is at time $t + 1$, at which point we *already know* everything we need, as the firings happened in the past. It then comes as an intuitive conclusion that in a SNN with synaptic delays, both input integration and output firings may happen in parallel, as the two are completely independent operations.

There are of course certain things we must take into account when implementing such a network model in practice. Although all neurons may be processed in parallel, having each neuron operate completely independently as its own thread of execution would be horribly inefficient, as constant synchronization would be needed for their communication (a computer system works very differently from a biological neural network after all). As such, we decompose the simulation of the network model into distinct *tasks* that run sequentially, but where each of these tasks can process all neurons completely in parallel. What this actual task decomposition involves will be covered in Section 3.7.

Although it's fairly intuitive, this approach to enable parallelization of spiking networks was seemingly first introduced in [Morrison et al., 2007]. The sheer simplicity of the approach allows for this parellization to be implemented easily on commodity—rather than specialized—hardware, as the task decomposition removes the need for expensive synchronization between the neurons themselves (although it does introduce the need for certain atomic operations—see Section 2.1.1).

3.4.3 Learning-parallelization

We've already seen how integration and firing can rely completely on conceptually "local" information only, and if this were the case with the learning mechanism as well, we'd be all set. Luckily, this is the case with spike-time dependent plasticity, which may be considered fully local. To see why, revisiting Section 3.3.2—more specifically Equation 3.3—shows both its prerequisite information and the information that gets modified as a result of its invocation. Δt requires knowledge of when the postsynaptic neuron fired and when a spike travelling down a synapse last arrived a the soma. This may be accomplished by eg. maintaining timestamps in the neuron (for firings) and in the synapses (for spike arrivals). By ensuring that the network model task that writes these timestamps runs before the one that reads them, we also remove any dependencies here, thus allowing parallel, local learning.

3.5 Spiking Neural Network model

The parallel SNN model described herein is an ehanced version of that found in [Izhikevich, 2006], more specifically, the C++ version of it[4]. The most prominent differences are that whereas Izhikevich's code uses compile-time, static-sized data buffers, this implementation is fully dynamic in terms of neuron population size, synaptic connectivity and the maximum synaptic conduction delay, as well as having several algorithmic redesigns/simplifications in order to enable proper parallelization. The construction algorithms have also been redesigned, yielding a potential speedup of a factor of N (N being the total number of neurons and compared to the original reference algorithm) for its most time-consuming aspects, in addition to being heavily parallelized. The reference algorithms will not be covered in any detail here—please see the original sources if they are of interest.

3.5.1 Structure

The behaviour of any neural network is completely dependent on its connectivity and the properties of the neurons in it. For the SNN, we'll be using the same approach as that found in [Izhikevich, 2006], as it allows us to observe the network emergently change its dynamics and go into waveform oscillations that resemble those of the brain (although this is

[4] `http://vesicle.nsi.edu/users/izhikevich/publications/spnet.cpp`

mostly so the STDP will have something to work upon, not something that will be covered. Interested readers are referred to the original text).

For this and subsequent sections and chapters, let N_E be the number of excitatory neurons, N_I the number of inhibitory interneurons, N the sum $N_E + N_I$ (i.e. the total number of neurons in the network, regardless of their type), M the number of synaptic connections from a given neuron to another (i.e. the number of axonal connections branching *out* from a neuron) and D the maximum synaptic conduction delay between two connected neurons.

We assume a linear, contiguous array of neurons, where each neuron is referred to by its index $i \in [0, N)$.

3.5.1.1 Neural dynamics

To control neuron membrane potential dynamics, four 1-dimensional arrays v, u, a and d, all of length N, are maintained. These correspond directly to the Izhikevich model parameters outlined in Section 3.2.3, and each neuron n_i, $i \in N$ will use and/or update elements of the tuple $\{v_i, u_i, a_i, d_i\}$ for its dynamics.

All excitatory neurons are initialized with $(a, d) = (0.02, 8.0)$, and all inhibitory with $(a, d) = (0.1, 2.0)$. These are the 2-dimensional differential variables for Regular Spiking and Fast Spiking neurons, respectively [Izhikevich, 2004]. For *all* neurons, $(v, u) = (-65.0, 0.2v)$, meaning that they start out with a membrane potential at resting levels and a default recovery variable. Note that unlike v and u, a and d are never changed after the initial network construction. This is because the intrinsic behaviours of the neurons themselves do not change during the course of the simulation (i.e. a Fast Spiking neuron will never cease to be a Fast Spiking neuron).

To update the membrane potential and recivery variables as the network is being simulated, we also maintain an input-array of length N where synaptic input to any given neuron is accumulated during a single time step. Once the variables have been updated, this array is reset to zero to prepare for the input from the next time step.

3.5.1.2 Synaptic connectivity

As with the implementation in [Izhikevich, 2006], the connectivity is completely random, with each neuron having exactly M axonal outputs to other neurons. For this model, excitatory neurons may connect to other excitatory and inhibitory neurons. Inhibitory neurons may only connect to excitatory neurons. No recurrent self-connections are allowed, and a

given presynaptic neuron may only connect at most once to another postsynaptic neuron.

Upon initialization, all synaptic weights are set to 6 for connections from excitatory neurons, and -5 for connections from inhibitory neurons. As inhibitory connections are considered non-plastic (i.e. have a fixed value), this initial weight value will not change during the network's lifetime. A maximum weight value of 10 is used for excitatory connections, and any STDP updates that cause the weight to go $>$ this value will be truncated. Similarly, 0 is the lowest value a weight may have, as negative weights are illegal for excitatory connections.

The synaptic connectivity is stored in two ways: a *post* 2-d matrix of size $N \times M$ containing for each neuron $i \in N$ the index of the postsynaptic neuron that neuron i's jth synapse points to. Inversely, we maintain for each neuron the indices of its incoming connections (i.e. dendrites) in a separate matrix *pre_syn_w* of size $N \times 3M$. $3M$ is because the random connectivity may lead to more than M neurons being connected to a given neuron, so we must be able to store the indices to all these.

3.5.1.3 Synaptic conduction delays

Synaptic delays for all excitatory neurons' outgoing connections are uniformly distributed, meaning that it will have M/D synapses for any value $\in [1, D]$. To allow the inhibitory connections to provide sufficient inhibition to their postsynaptic neurons, all their M synapses have a fixed conduction delay of 1 ms. D will be fixed at 20 ms throughout this thesis.

Referring back to Section 3.4, this guaranteed absence of zero-delay connections satisfies the requirements for a fully parallelizable network.

To know which postsynaptic neurons to distribute synaptic input to relative to when the presynaptic neuron has fired, we use two 2-matrices *delays_start* and *delays_length* of size $N \times D$ that for any delay delta $d \in D$ determine the range $[start, start + length)$ into a neurons *post* matrix row that should have their inputs modified. To take an example, if neuron i is an interneuron and d is 0, $delays_start[i][d]$ will be 0 and $delays_length[i][d]$ will be M, as interneurons have a fixed 1 ms delay. Any $d > 0$ here would leave both the length and start as 0, as no other delay delta is valid for these. This is an algorithmic improvement over the original implementation that came from [Nageswaran et al., 2009] and takes advantage of the fact that our synaptic delays are distributed sequentially, and interested readers are referred to it for further details. This approach may understandably not be the easiest to grasp and is also more of an implementational detail, so a purely conceptual figure outlining how delayed input

works in relation with previously fired neurons is given in Figure 3.7. This figure does not directly reflect the implementations.

3.5.1.4 Spike-time dependent plasticity

To enable efficient STDP, we use a $N \times M$ matrix of timestamps of when any given synapse $s_{i,j} \in N, M$ last had a spike arrive at its postsynaptic neuron, and a 1-d matrix of length N containing each neuron's last firing timestamp. This allows us to sparsely update parameters required for STDP only when it is absolutely necessary, i.e. when a neuron actually fires or a synapse actually carries a spike to the soma. Using timestamps lets us directly use the STDP equations from Song et al. [2000] as outlined in Section 3.3.2.

3.6 Network construction

Put briefly, this process is threefold:

1. Initialize the properties of all neurons and synapses.
2. Randomly connect each neuron to exactly M other neurons, ensuring that no interneuron connects to other interneurons, that any neuron attempts to recurrently connect to itself or that a neuron tries connecting more than once to another neuron. Synaptic weights are also set at this point, depending on which neuron they branch out from.
3. Assign conduction delays to each synapse. This is as mentioned the critical step for allowing the network to be parallelized.

A (simplified) graphical representation of this process is given in Figure 3.8. During the thesis-work, the construction stages were also parallelized and optimized, yielding a significant speedup over the serial reference-algorithm, but as the simulation is by far the most time-consuming (and relevant) aspect, this will not be covered.

3.7 Network simulation

Referring back to the Izhikevich equations in Section 3.2.3, they are all tuned to yield biologically plausible membrane potentials at an update rate of 1 millisecond. Going by this it comes as no surprise that we have

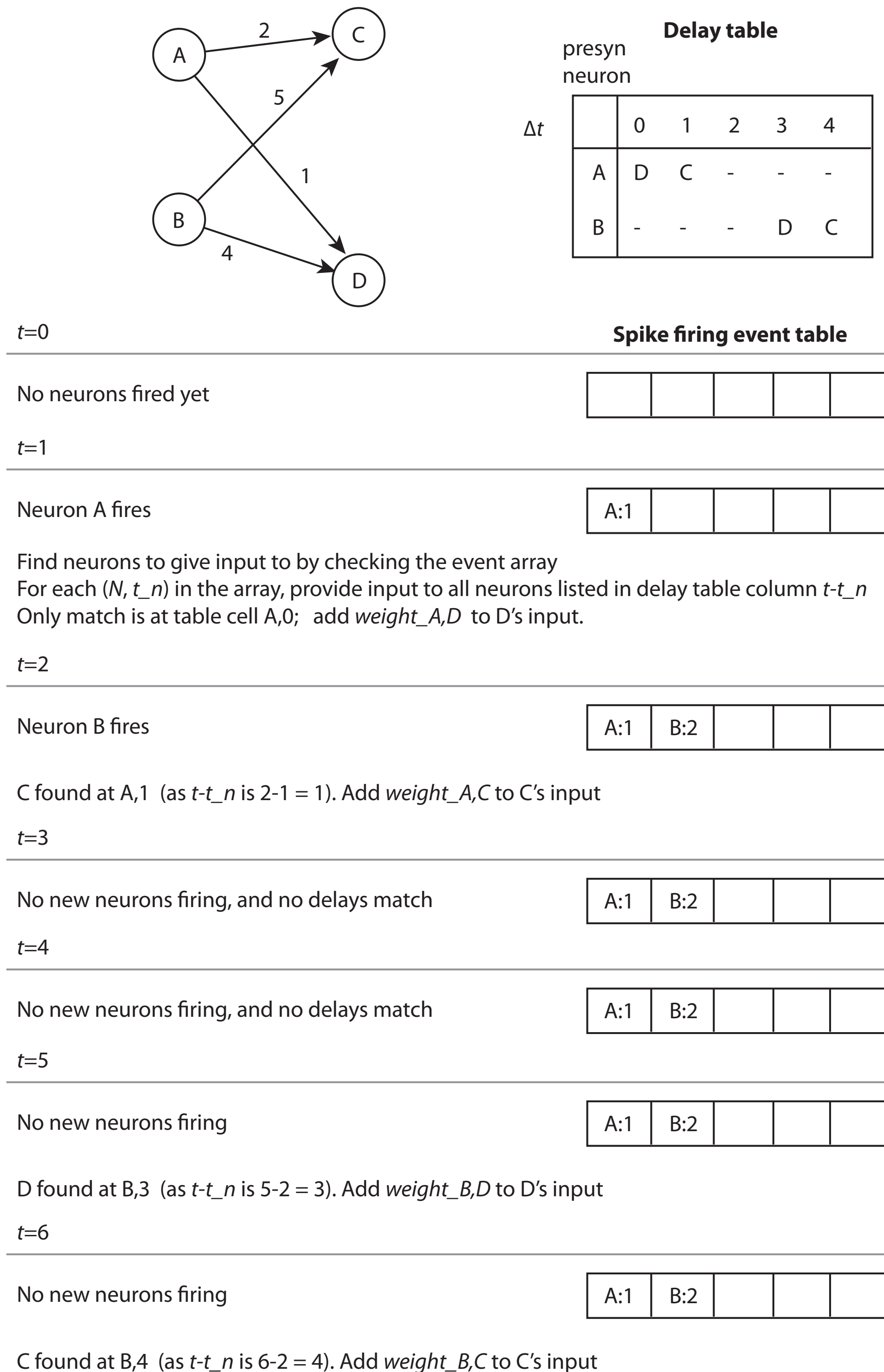


Figure 3.7: Conceptual example of spike input over synapses with delays. Each synapse has an integral delay associated with it.

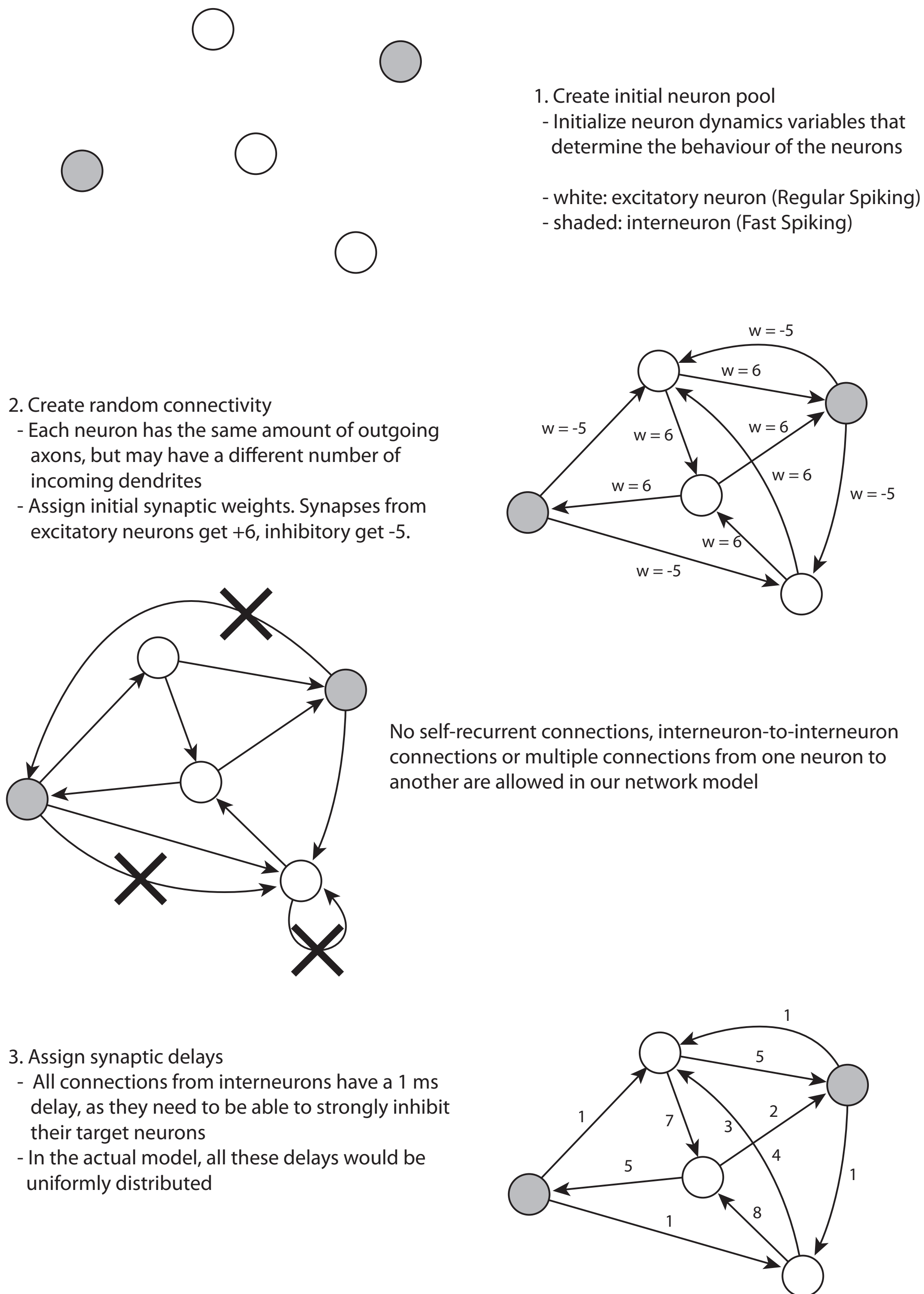


Figure 3.8: Randomly connected spiking network construction phases

to process 1000 individual time steps per simulated second. There are numerous stages/tasks that goes into a single time step:

1. (Re)set all elements of neuron input array to zero.

2. Simulate random thalamic input by setting neural input to +20 mV for a total of $N/1000$ randomly selected neurons. Even though this might seem like a very low amount of neurons, the relatively rich connectivity ensures that we get activity as that seen in [Izhikevich, 2006].

3. The membrane potential v is checked for *every* neuron to see if it's equal to, or beyond, the threshold required for it to fire. If so, reset its membrane potential, update its firing timestamp and add it to the list of fired neurons.

   In addition, all synapses that connect to the neuron have their weight derivatives updated with STDP LTP based on when they last carried a spike to the neuron relative to the current firing-time.

4. All neurons that have fired in the D previous time steps are processed, distributing input to their proper postsynaptic neurons based on the synaptic delays present. This updates the "spike last arrived from this synapse"-timestamps for all the synapses in question, as well as invoking STDP LTD on their weight derivatives relative to the time the postsynaptic neuron last fired.

5. Membrane potentials for all neurons are updated based on the input from the previous stage. It may be noted that it's fully possible to reset the input for the neurons in the same operation as this, avoiding having this as a separate task altogether, but the input task is kept as it (hopefully) makes the process more intuitive.

In addition, after each simulated second, other tasks have to be performed:

1. Every synapse has its actual weight modified by its weight derivative, which may be positive or negative, depending on the STDP performed on it in the course of the last 1000 ms. This derivative is then reset back to zero so that it may be used for another second of STDP. Weight bounds checking is also done here to ensure that no weight ends up outside the preset range.

2. The list of firing neurons is "shuffled" so that it only contains the neurons that have fired late enough in the last second that they'll

have an effect on the neurons that will fire at the start of the next second (consider a neuron firing at timestamp 995 which has synapses with up to 20ms of delay). The rest of the firing-list is then free to be used by firings in the next second.

3.7.0.5 Identifying parallelizable tasks

The approach used for identifying parallel regions is to look at the logical control flow loop regions that make up the overall simulation algorithm and ascertain the level of dependency between them (refer back to Section 2.1.1), and inside them. If there are dependencies between them it means they have to be run sequentially. Conversely, if they are free of dependencies they may be run in parallel. For the tasks' internal loops, if all iterations of a loop do not require the knowledge of any other iterations of the loop, the loop itself may be parallelized as well.

Due to the inherent nature of SNNs, we can already dismiss one sort of "embarassing parallelism" that was discussed in Section 2.1, meaning that no time step can be computed in parallel with another one. Millisecond 3 depends on millisecond 2 which in turn depends on millisecond 1 and so on. This is a natural consequence of the behaviour of the net being an emergent product of the individual spike timings, and simulating the time steps out of order would give completely different (and not to mention completely incorrect) outcomes.

Looking at the stages that make up a second and time step, it quickly becomes obvious that we're dealing with several chains of data-dependencies. Resetting the input array is required before any of its elements can be assigned during the random thalamic input (otherwise, the random input would be lost as the input buffer is cleared). The firing-check does *not* require any knowledge about the state of the input buffer, and does therefore not have a data-dependency here. This may seem very counterintuitive, but recall that the firing check only uses the membrane potentials of the neurons, which were already calculated at the end of the previous time step. Distributing the delayed inputs requires both knowledge of firings and the ability to add to the input of neurons, and does therefore have a dependency to both the input handling and firing-checks[5]. Updating the membrane potentials requires the final known state of the input buffer for all neurons, and therefore depends on all the tasks mentioned thus far.

[5] Although since all it does is add to existing values, the random thalamic input could conceivable be performed after the firing processing. There is still a dependency on having an initially reset input buffer, however.

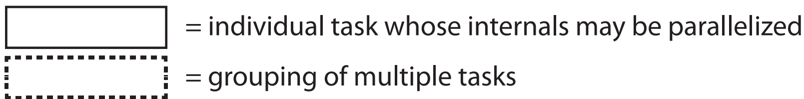
The tasks that are performed at the end of the simulated second—the weight updates on all the STDP operations and the firing buffer "shuffling" on all the firings—have an obvious dependency on all the millisecond time steps the second is comprised of. These two are on the other hand not dependent on each other, as they operate on completely different data. The final dependency graph is given in Figure 3.9.

All the tasks listed here are comprised of an inner loop, and *all* are conceptually *internally* parallelizable, as the individual loop iterations do not have dependencies on the results of other iterations. This does however require the proper use of synchronization operations, as outlined in Section 2.1.1 whenever a shared piece of data is updated. The shared pieces of data that are relevant for the network simulation is the value of the input buffer for any given neuron (as multiple neurons processed in parallel may be attempting to add their delayed input to the same postsynaptic neuron) and the current neuron firing count, as this value is used as an index into where information about the firing neuron should be placed. Failure to maintain such synchronization will mean that certain individual neuron firings eventually will appear to be randomly dropped or overwritten, resulting in a erroneous network operation (and a massive debugging headache).

3.7.0.6 SNN model simulation pseudocode

The realization of the network simulation and its identified parallel regions is given here in the form of pseudocode. The keyword **parallel-for** indicates that the body of its for-loop may be arbitrarily parallelized for the entire range that the loop in question covers. This also naturally implies that the correctness of the code would be unchanged if these loops were changed to be serial loops. Note that although the parallel task identification shows how certain phases may be run side-by-side, this code stays simple and only parallelizes *within* the tasks, not *between* them (i.e. only loop-level parallelism, not task-level parallelism). Also note that the actual pseudocode's operation deviates slightly from what has already been discussed, as it is meant to provide a decently close approximation to how the code *may* appear in anyone's (imperative) programming language of choice, rather than merely representing high-level concepts.

The pseudocode uses the same format as that from the ubiquitous Cormen et al. [2003], and should as a result hopefully be of a familiar format to most. All matrix/vector subscripts are shown using standard C-notation rather than mathematical notation. All variables used are presumed to be global and correspond directly to those discussed in the previous sec-



Arrows identify data dependencies and subsequent task ordering. Tasks that are on the same horizontal line may be executed in parallel.

for each simulated second:

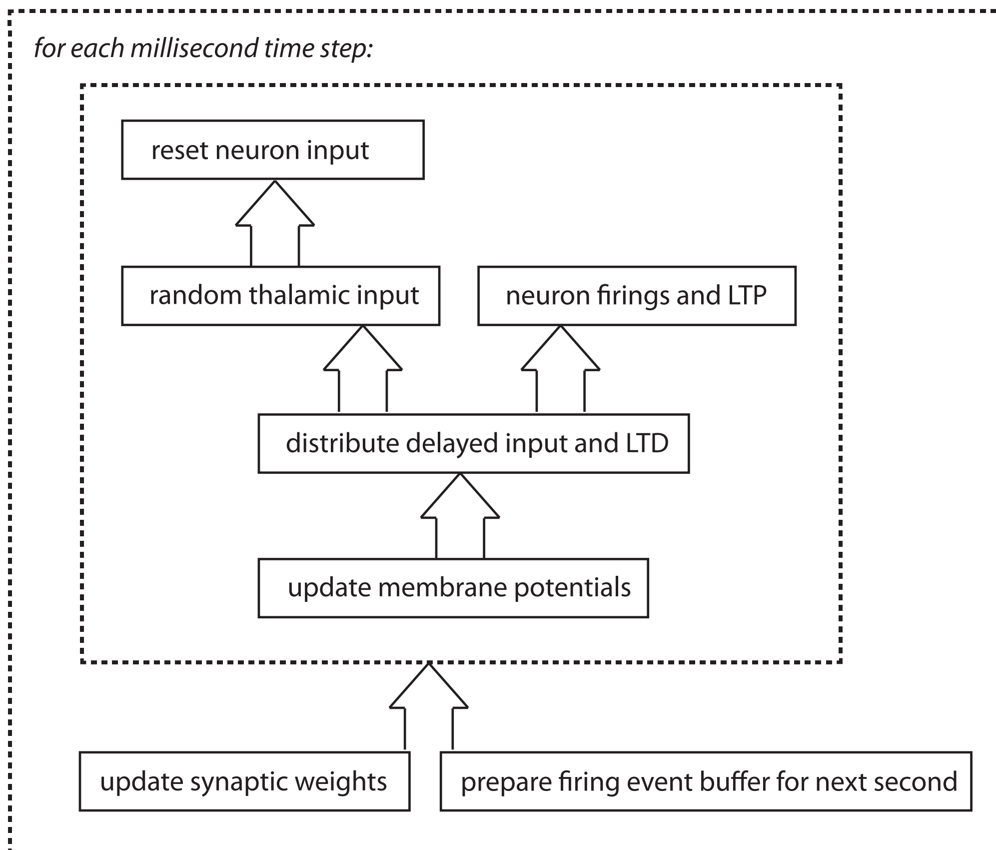


Figure 3.9: Network simulation task dependencies

tions. For those who aren't already mentioned, their function/contents is as follows:

timestamps and *syn_timestamps* correspond to the 1-dimensional neuron firing timestamp and 2-dimensional synapse spike arrival timestamp matrices outlined in Section 3.5.1.4, respectively. *sd* is the 2-d matrix holding synaptic weight derivatives for STDP, and *s* is the 2-d matrix holding the current active synaptic weights, both of dimensions $N \times M$. I is an 1-d matrix of size N that contains the current synaptic input to any given neuron (see Section 3.7).

firings is a 2-d matrix of size *max_firings* $\times 2$. The matrix's first column contains the time step at which the neuron fired and the second column contains the index of the fired neuron itself. *max_firings* is a "sufficiently high" value so that the number of firings in any given second does not overflow the firing buffer. A fairly safe value here could be e.g. $100N$.

```
PARALLEL-PROCESS-SPIKING-NETWORK(sec_total)
 1  ▷ Initialize STDP timestamp variables
 2  for i ← 0 to N
 3       do timestamps[i] ← −100000
 4          for j ← 0 to M
 5               do syn_timestamps[i][j] ← −100000
 6  for i ← 0 to D      ▷ Initialize firing history buffer
 7       do firing_start_history[i] ← 0
 8  n_firings ← 0       ▷ Number of neuron firings
 9  ts ← 0              ▷ Timestamp—not reset per second
10  ▷ Begin neural simulation
11  for sec ← 0 to sec_total
12       do ▷ 1 ms time step resolution
13          for t ← 0 to 1000
14               do ▷ Reset neural input
15                  parallel-for i ← 0 to N
16                       do I[i] ← 0
17                  ▷ Random thalamic input
18                  for i ← 0 to N/1000
19                       do I[RANDOM(N)] ← 20
20                  pre_firings ← n_firings
21                  parallel-for i ← 0 to N
22                       do if v[i] ≥ 30
23                             then HANDLE-NEURON-FIRING(i)
24                  ▷ Update firing history array
25                  for i ← 0 to D − 1
26                       do firing_start_history[i] ← firing_start_history[i + 1]
27                  firing_start_history[D − 1] ← pre_firings
28                  parallel-for k ← firing_start_history[0] to n_firings
29                       do time ← firings[k][0]
30                          neuron ← firings[k][1]
31                          length ← delays_length[neuron][t − time]
32                          post_start ← delays_start[neuron][t − time]
33                          for j ← post_start to post_start + length
34                               do i ← post[neuron][j]
35                                  ATOMIC-ADD(I[i], s[neuron][j])
36                                  syn_timestamp[neuron][j] ← ts
37                                  if neuron < N_E
38                                    then wd ← 0.12 exp(−(ts − timestamps[i])/τ_−)
39                                         sd[neuron][d] ← sd[neuron][d] − wd
40  (cont'd on next page)
```

```
41                 ▷ Update membrane potentials and recovery variables
42                 parallel-for i ← 0 to N
43                     do UPDATE-NEURON(i)
44                 ts ← ts + 1
45          ▷ Modify excitatory synapses
46          parallel-for i ← 0 to N_E
47              do for j ← 0 to M
48                     do sd[i][j] ← 0.9 sd[i][j]
49                        s[i][j] ← s[i][j] + 0.01 + sd[i][j]
50                        if s[i][j] > s_max
51                          then s[i][j] ← s_max
52                        if s[i][j] < 0
53                          then s[i][j] ← 0
54          ▷ Update firing count buffer for next second
55          k ← firing_start_history[1]
56          for i ← 0 to D
57              do firing_start_history[i] ← firing_start_history[i] − k
58          for i ← 0 to n_firings − k
59              do firings[i][0] ← firings[i + k][0] − 1000
60                 firings[i][1] ← firings[i + k][1]
61          n_firings ← n_firings − k
```

HANDLE-NEURON-FIRING(i)

```
 1  v[i] ← −65                ▷ Reset membrane potential
 2  u[i] ← u[i] + d[i]        ▷ Update recovery variable
 3  timestamps[i] ← ts        ▷ Set "last fired" timestamp
 4  for j ← 0 to pre_count[i]
 5       do ▷ Note: see the comments below for explanation on the indexing
 6          pre_syn_idx ← pre_syn_w[i][j]
 7          wd ← 0.1 exp((syn_timestamp[pre_syn_idx] − ts)/τ_+)
 8          sd[pre_syn_idx] ← sd[neuron][d] + wd
 9  fire_idx ← ATOMIC-ADD(n_firings, 1)
10  if fire_idx ≥ n_firings_max
11     then error "Too many neurons firing"
12  firings[fire_idx][0] ← t ▷ Time
13  firings[fire_idx][1] ← i ▷ Neuron index
```

UPDATE-NEURON(i)

1 $\triangleright$ Update membrane potential and recovery variable for neuron i
2 $v[i] \leftarrow v[i] + 0.5((0.04v[i] + 5)v[i] + 140 - u[i] + I[i])$
3 $v[i] \leftarrow v[i] + 0.5((0.04v[i] + 5)v[i] + 140 - u[i] + I[i])$
4 $u[i] \leftarrow u[i] + a[i](0.2v[i] - u[i])$

At the very top, lines 2–5 initialize the firing timestamps for all neurons and the spike arrival timestamps for all synapses. These are initially set to $-100,000$ so that when they're used in a STDP calculation without having first been overwritten by an actual timestamp, they will yield a number extremely close to zero, thereby not having an effect. Lines 6—7 set the contents of the firing history buffer. This is just a simple way to keep track of how many neurons were firing in the last D time steps, allowing us to know exactly how many neurons might have an effect on the input of neurons in the current time step.

The following ranges of lines correspond to the parallelizable tasks mentioned in Section 3.7:

Lines 15–16 reset neuron output.

Lines 18–19 random thalamic input. Note that this loop is *not* parallelized here, as the number of iterations is considered too small and it would require synchronization to ensure the pseudo-random number generator does not get any race conditions.

Lines 21–23 neuron firing and LTP.

Lines 28–39 distribute delayed input and LTD.

Lines 42–43 update neuron membrane potentials and recovery variables.

Lines 46–53 update synaptic weights.

Lines 58–60 prepare firing event buffer for next second. Note again that this loop is *not* parallelized here, as the number of iterations is not deemed worth it.

Lines 7 and 8 in HANDLE-NEURON-FIRING show a rather strange way of indexing into both the *syn_timestamp* and *sd* matrices. These are 2-dimensional matrices, but here we are treating them as an unwrapped *1-dimensional* matrix by considering each of its rows as laid out continuously after each other in memory. *pre_syn_idx* on line 6 is simply a precomputed 2-d index unwrapped into a single 1-d index which can be used

for both the synaptic timestamp and synaptic weight derivative matrices (they both have dimensions $N \times M$). The original implementation in [Izhikevich, 2006] achieves this by using pointers, but this is not portable for our later purposes, as pointers generated on the CPU are not valid on the GPU. Integral indices, on the other hand, are always valid.

As can be seen from UPDATE-NEURON(i), it updates v_i by splitting the computation into two 0.5 ms differential equations, thereby improving numerical accuracy [Izhikevich, 2006].

It should be mentioned that there exists a *potential* race condition on line 39. If an excitatory neuron fires twice or more within D time steps, it will get added to the list of firing neurons multiple times, and as such there may be multiple threads attempting to modify sd. For the purpose of this thesis, we make the simplifying assumption that no excitatory neuron does in fact fire twice within an interval of D time steps.

Chapter 4

Implementation

The previous chapter gave the theoretical and algorithmical foundations for how a parallel artificial spiking network may be realized. This chapter quickly outlines the creation of the CPU and GPU implementations based on this.

4.1 Introduction

During development, the correctness of the different implementations was verified by comparing their firing outputs over several seconds with that of the reference algorithm from [Izhikevich, 2006], ensuring that their pseudo-random number generators had identical starting points so that their sequence of events would also be identical.

4.2 Parallel Izhikevich SNN on the CPU using OpenMP

The CPU version of the code is in its essence a fairly direct mapping of the pseudocode in Section 3.7.0.6 into C++, with **parallel-for** replaced with the appropriate OpenMP directives, and the invocations of ATOMIC-ADD replaced with appropriate hardware platform support functions.

Since floating-point atomic operations are not natively supported, they had to be (losslessly) emulated with integer atomic operations.

4.3 Parallel Izhikevich SNN on the GPU using NVIDIA CUDA

To create the GPU CUDA implementation, we have to somehow map all the tasks outlined in Section 3.7 to a set of individual kernels that satisfy

the programming model constraints of the CUDA platform.

The GPU implementation is comprised of 7 kernels that each handle specific parts of the spiking network simulation process. 5 of these are invoked per time step, the remaining 2 at the end of each simulated second in order to update synaptic weights according to the STDP and to prepare for the next second. These kernels are all fairly functionally identical to the simulation stages we've already seen, so they will not be covered in great detail. There are however certain stages that are decomposed into several kernels for the sake of efficiency, and the reasoning behind this will be given. See Figure 4.1 for a diagram representation of how these kernels are launched.

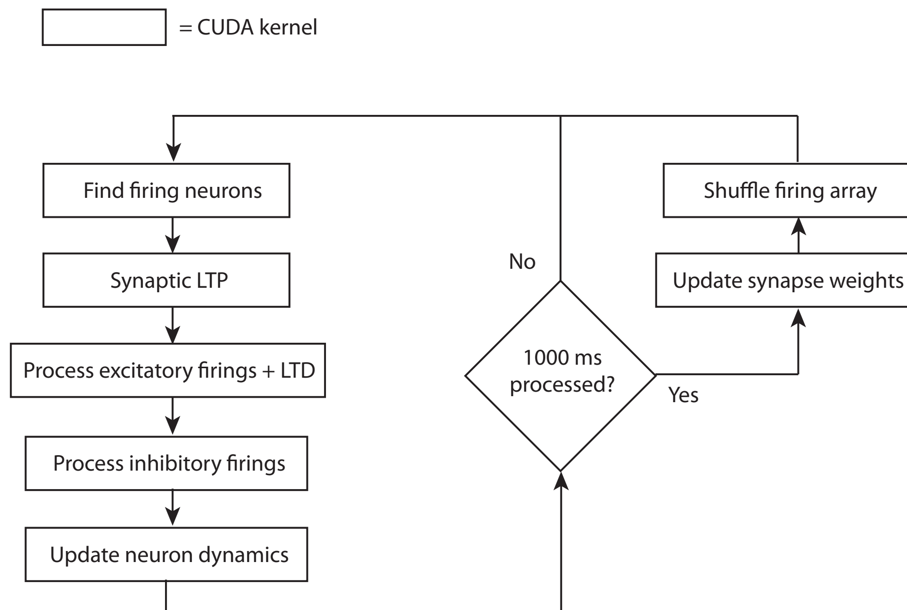


Figure 4.1: Control-flow diagram of the kernels and their invocation (inspired by diagram in [Nageswaran et al., 2009])

1. *Find firing neurons*. Runs through all N neurons and checks which of these have fired, adding the fired neurons to *separate* firing event lists for excitatory and inhibitory neurons. The reason for this distinction will hopefully soon be made clear.

2. *Synaptic LTP*. All incoming synapses for the neurons that fired in kernel 1 have their synaptic weight derivatives modified (potentiated).

3. *Process excitatory firings and LTD*. As with the CPU implementation, the recently firing neurons are processed to distribute synaptic input to their postsynaptic neurons based on the synaptic delays, and these synapses have their weight derivatives modified (depressed).

4. *Process inhibitory firings*. Since we know that all interneurons have a fixed, 1 ms delay in our model and that no STDP is performed on such synapses, a custom kernel is used that is especially tailored to be efficient for this case alone. This is the reason for the separate firing lists written to by the first kernel.

5. *Update neuron dynamics*. Updates membrane potentials and recovery variables for all N neurons according to the Izhikevich neuron model used. For this implementation, this kernel also resets the synaptic input for the neurons.

6. *Update synapse weight*. Updates all $N \times M$ synaptic weights based on the weight derivatives that were computed during the course of the previous second of simulation.

7. *Shuffle firing array*. Since all the neuron firing information is kept locally in the GPU's memory, we must run this code on it to prepare for the next second of simulations.

One challenge in creating these kernels lie primarily in the fact that there is no notion of *non*-parallel execution of code on the GPU, meaning that we're left handling e.g. non-parallel for loops on the CPU, such as that on pseudo code line 25, something that in the current implementation requires a small amount of memory to be copied from the GPU to the regular system memory (as the two cannot directly access each other) and back. Despite the amount of memory copied being on the scale of a few dozen bytes, it still leads to some overhead. The current implementation also requires this information from the GPU in order to determine the block-/thread ratio when launching certain kernels, such as those whose block count is dependent on the number of fired excitatory and inhibitory neurons.

The other challenges outlined in Section 2.2.2 were very relevant during the implementation of the GPU spiking network and were not completely solved. The inherent low-level nature of the approaches used and attempted in the implementation will not be covered, as they could nearly fill a separate thesis in their own right and would take focus away from the spiking networks.

Chapter 5

Method

This chapter outlines the experiments and methodologies utilized in order to determine how well the implementations from Chapter 4 realize their theoretically attainable goals of parallel scalability.

5.1 Introduction

As the purpose of this thesis to a large extent involves evaluating the benefits that desktop-level parallelization can bring to SNN model simulation, a very important concept is that of *time*. More spefically, how much time is spent during simulations in non-parallel vs. parallel test runs, and which parts of the algorithms that have already been outlined benefit the most and least from parallelization. Going from this, getting good timing results is important and for the results that will be presented in this thesis, a combination of high-resolution performance timers and amortization through averaging over multiple runs is used.

All network topology is derived from the output of pseudo-random number generators. Pseudo-random number generator seeds were unique for each invocation, which together with the result averaging should mitigate the effect of particularly "lucky" or "unlucky" seeds. Note that this means that invocations of the same network size with different thread counts were not with identical network topologies, but this should again be amortized away by the multiple runs. For all test runs, maximum delay $D = 20$.

5.2 CPU (OpenMP)

The hardware used for the testing was an Intel Core2Quad 9300 with each core running at 2, 5 GHz and with a total of 4 gigabytes of 800 Mhz memory. Being a quadcore system, this allowed for testing with 4 simultaneous threads, which should presumably provide a decent indication of scalability performance and parallelization viability.

To ensure that the results of the single-threaded tests would not be artificially skewed by OpenMP runtime overhead, these were run with OpenMP directives completely removed through conditional compilation. All C++ code was compiled in release mode on Visual C++ 2008 with optimizations enabled.

5.2.1 Areas and procedures of testing

For the CPU implementation, the following were tested, all over a simulated time period of 10 seconds:

1. Network processing times—serial, parallel and with different values of M
2. Total time spent for most of the individual parallelizable stages outlined in Section 3.7.0.5.

To determine the scalability of the parallel algorithms, total simulation times (not including construction or buffer allocations et al.) were logged for a total of 4 different runs of 10 simulated seconds sampled at 10k neuron intervals for all thread-counts. This was performed for $M \in \{100, 200, 300\}$ to see not only what increased synaptic connectivity does with the runtime and parallelism performance, but also to see what this does for the overall network activity. As the performance of the simulations are bound to the number of neurons being fired, this should help show to what extent network size and synapse count impacts neuron firing frequency. This was done by calculating the average firing frequency across several runs.

With M as usual being the outgoing synapse count from all neurons

Similarly, to identify the benefits (or lack thereof) of parallelizing the presumably most expensive tasks outlined in Section 3.7.0.5, the total time spent executing them in various neuron/synapse/thread configurations was recorded. This included the input buffer reset, the firing checks+LTP, the neuron delayed input+LTD and the per-second synapse weight adjustments. To avoid an excessive number of graphs, only the 1-thread and

4-thread test runs will be presented, as these provide the extrema of the results and should therefore serve as the biggest indication of performance. As with the other time-oriented results, these values were calculated as the average of 4 individual runs. Total simulation time spent to gather these measurements was around 24 hours.

5.3 GPU (CUDA)

The graphics hardware used for these tests was an NVIDIA 9800 GTX+ graphics card with 512 MiB RAM and 16 multiprocessors, totalling $16 \times 8 = 128$ concurrent thread execution units, otherwise running on the same hardware as the CPU tests.

5.3.1 Areas and procedures of testing

As with the CPU implementation, the following was tested over a simulated time period of 10 seconds:

1. Network processing times for $M \in \{100, 200, 300\}$.
2. Total time spent for the CUDA kernels outlined in Section 4.3.

Due to the explosive increase in memory requirements as M grows, it was only possible to test up to $N = 40,000$ for $M = 200$ and $N = 25,000$ for $M = 300$. There simply wasn't enough memory on the GPU to go any higher with the current datastructures and algorithms. To compensate for this, the interval between timing-samples was halved to $5,000$.

Chapter 6

Results

This chapter gives the results of the experiments outlined in Chapter 5. These results will be analyzed and discussed in Chapter 7.

6.1 CPU

For all test runs, T is used within the graphs to denote the thread count used for the given run.

Figure 6.1 shows the effect that an explicitly increasing neuron count has on a network with a connectivity factor of $M = 100$. Due to the connectivity, this comes with an implicit synapse count of NM, so the total count of synapses for the highest neuron count ($N = 100,000$) is $10,000,000$. Although the theoretical speedup is not reached, adding a 2nd and 3rd thread each yields a little less than 25% increase in performance over the single-threaded result and combined 4 threads manage to achieve almost 50% of their optimal speedups. For large network sizes this could be considered an improvement that is well worth it, but hardware with higher core counts would be required to ascertain how well the results scale beyond 4 threads. There is a fairly constant speed increase for each added thread (except from when the 4th thread is added). A constant speed increase is something that is highly desired, as it implies linear scalability is possible (i.e. "add n cores; get a factor of n speedup"). The reason for why the 4th thread did not reach the same benefits can really only be speculated at this point, due to a lack of proper low-level profiling tools, but a possible explanation is given in Section 7.1.

Profiling here refers to the ability to augment the generated executable code in a way that allows performance to be recorded on a function/instruction level

Figures 6.2 and 6.3 show a more troubling development. The relative speedup is severly reduced already at $M = 200$ and is nearly gone altogether at $M = 300$. This is a potentially very important set of results,

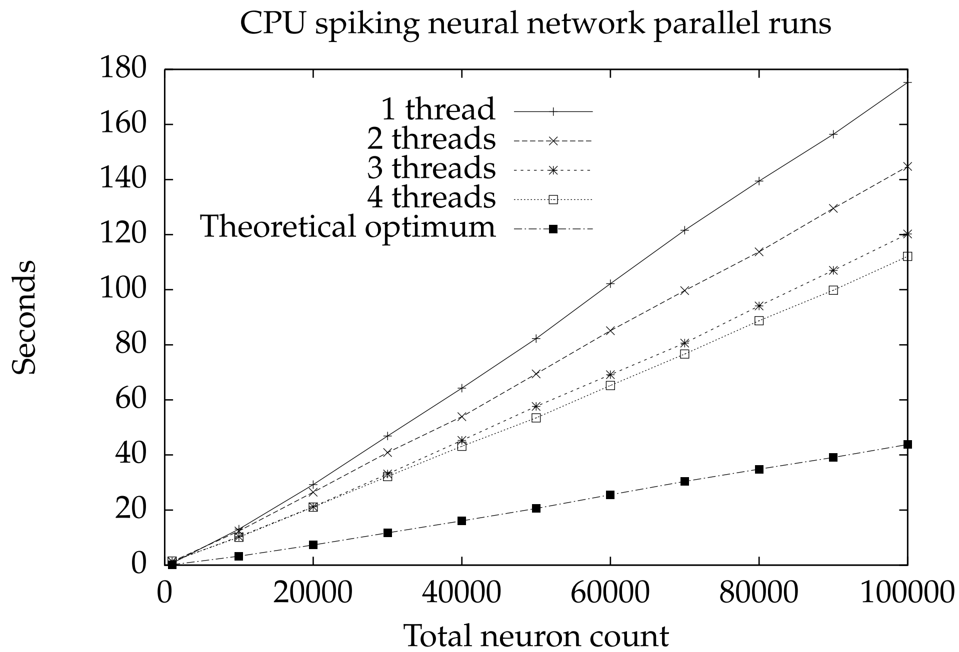


Figure 6.1: Time spent simulating 10 seconds of SNN time for increasing network sizes and thread counts with $M = 100$. Theoretical optimum is given as a function of the 1-thread result divided by 4.

as it could mean that the methods discussed in this thesis are incapable of dealing with SNNs that have a high degree of connectivity, which to a large degree is what one would find in a biological neural network and consequently in most realistic simulations.

Despite the vanishing performance numbers with high values of M, we can at least see that the increase in time per added 10k neurons also is essentially linear for all tested values of M. This is again a good thing, as a superlinear increase could quickly make computation of large networks infeasible. As the amount of synapse processing that goes on in a sparsely firing SNN is largely a product of how often neurons fire (delayed input integration, LTP/LTD), the increase in simulation time for each value of M being linear rather than superlinear can be readily explained by looking at Figure 6.4. It shows that despite an increasing neuron count, the network activity remains stable and with a near fixed frequency of firing (aside from a frequency-leap at the rise from $1,000$ to $10,000$ neurons in all cases). If the firing frequency had increased with the size of the network, the rate of change in time would presumably increase at a non-linear rate as although the number of neurons and synapses rises linearly, the amount of work that would have to be done upon them would be higher for each incremental step.[1]

The input reset stage in Figure 6.5, although contributing marginally to the total simulation time, is clearly not a very good choice for parallelization at the tested neuron counts. The graph is effectively divided in two, where the least amount of time is spent in the serial runs and the most amount of time is spent in the parallel ones (the serial runs being approximately *4 times faster* for low neuron counts).

The membrane potential firing check and synapse LTP stage in Figure 6.6 to a large extent reflects the results we've seen for $M \in \{100, 200, 300\}$, but actually with a higher level of displayed performance. It is interesting to note that the resulting benefit from parallelization goes down steadily for each increasing level of M, with the relative speedup for $M = 100$ being $\sim 50\%$, $\sim 30\%$ for $M = 200$ and $\sim 20\%$ for $M = 300$.

The real surprise arrives with the neuron delayed input and synapse LTD stage as given in Figure 6.7. Aside from when $M = 100$, the parallel runs take significantly *longer* time to execute than their serial counterparts! $M = 300$ incurs a $\sim 45\%$ slowdown when running with 4 threads, which is very high considering the amount of time that is spent in this stage in

[1] It may be noted that the recording of average frequencies did not differentiate between the firings of excitatory and inhibitory neurons. Since the excitatory neuron has to be processed over D time steps rather than just a single one, there might be discrepancies here that go unnoticed.

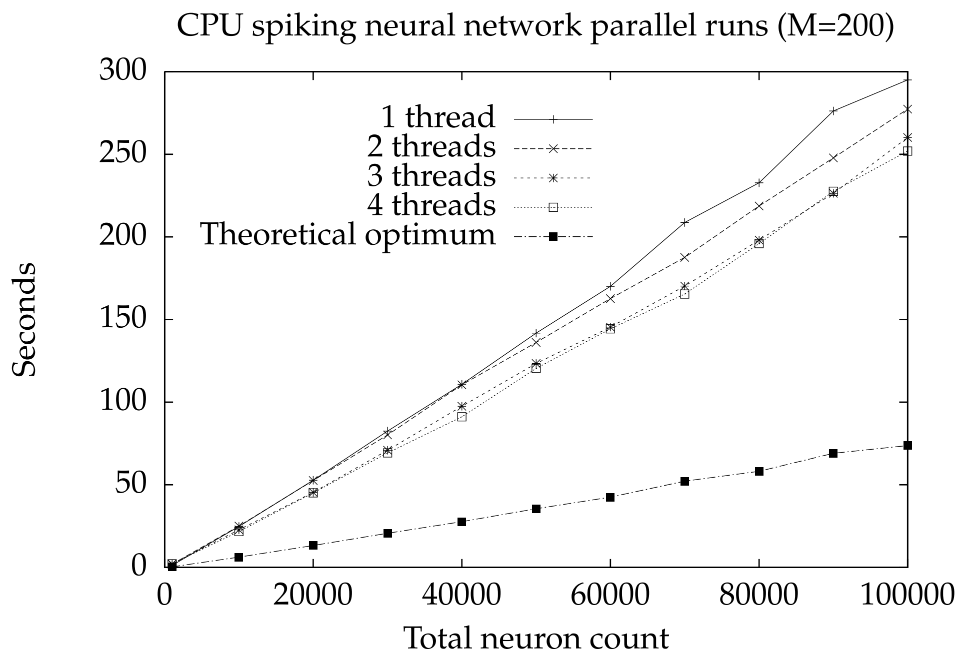


Figure 6.2: CPU: Time spent simulating 10 seconds of SNN time for increasing network sizes and thread counts with $M = 200$.

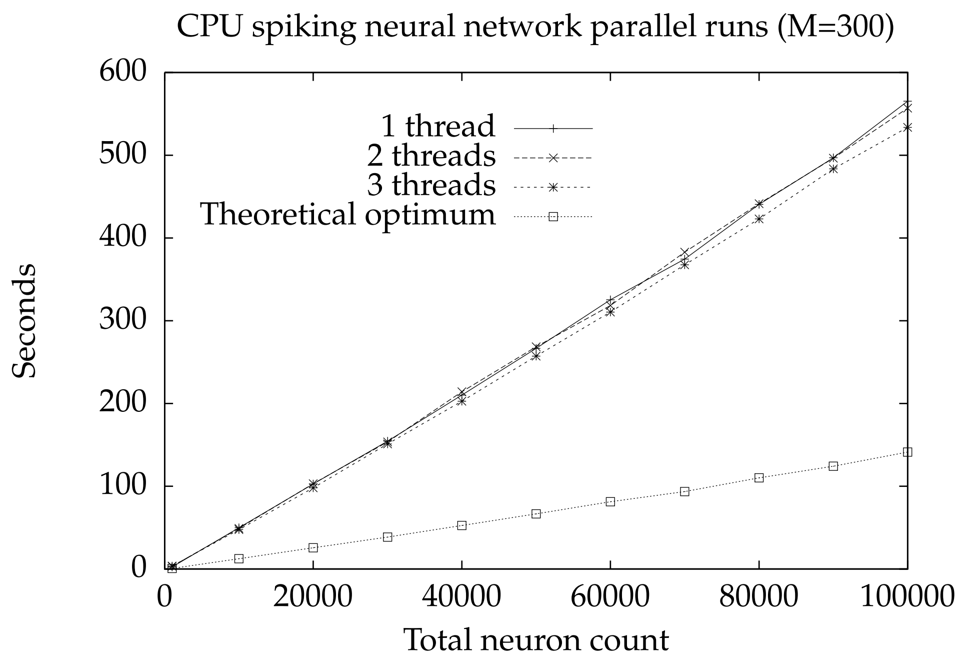


Figure 6.3: CPU: Time spent simulating 10 seconds of SNN time for increasing network sizes and thread counts with $M = 300$. Only 3 threads used due to the almost non-existing performance increase displayed.

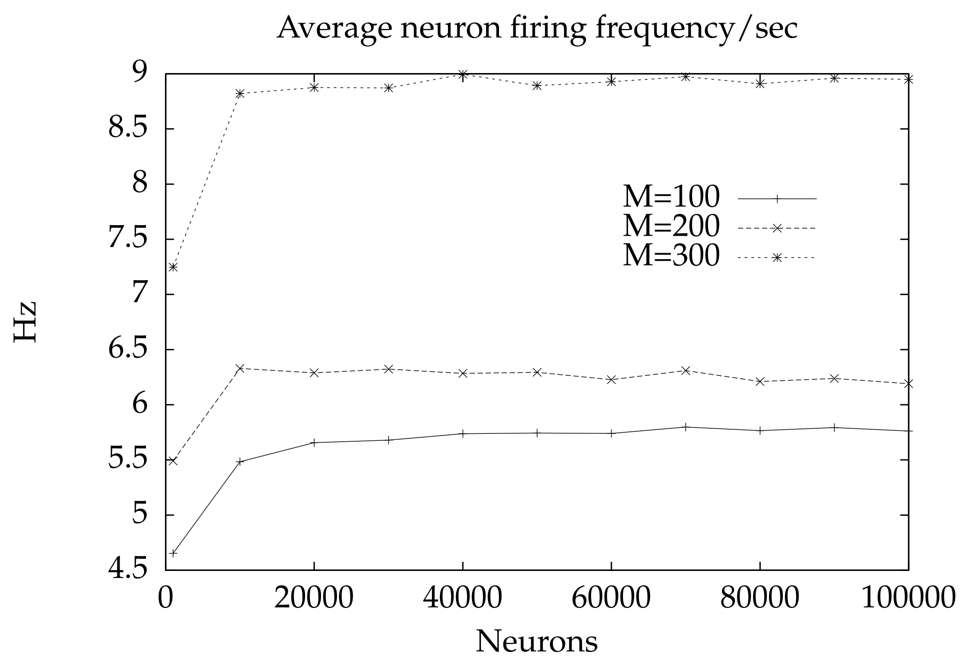


Figure 6.4: Average firing frequencies for SNN neurons in networks with synaptic connectivity $M \in \{100, 200, 300\}$.

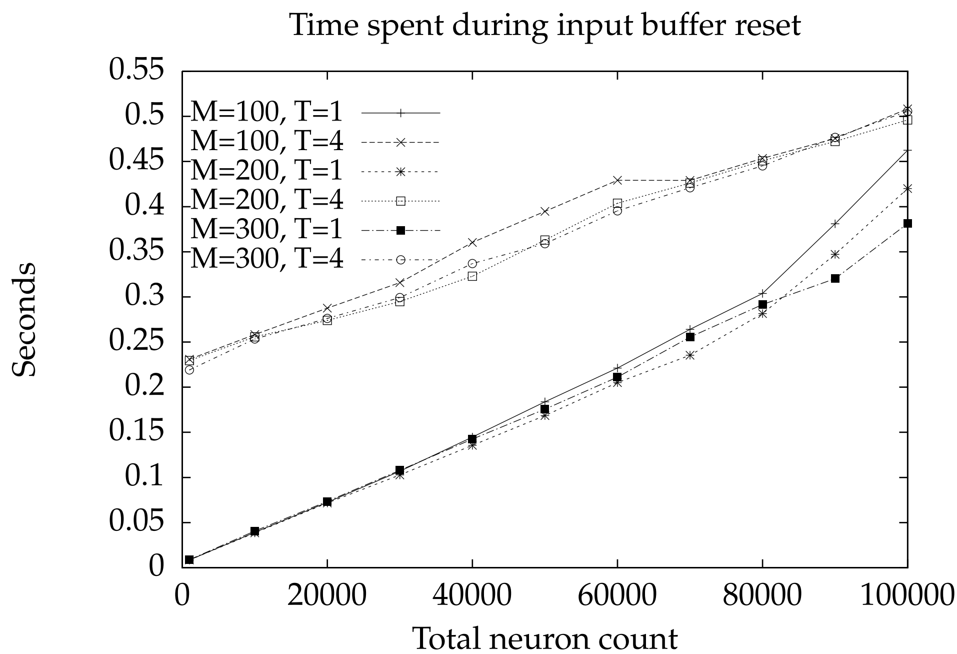


Figure 6.5: CPU: Time spent during input buffer reset for 1-thread and 4-thread test runs. Note that these test runs should be independent from the synapse count, so the inclusion of runs for additional values of M is primarily to ascertain this in practice.

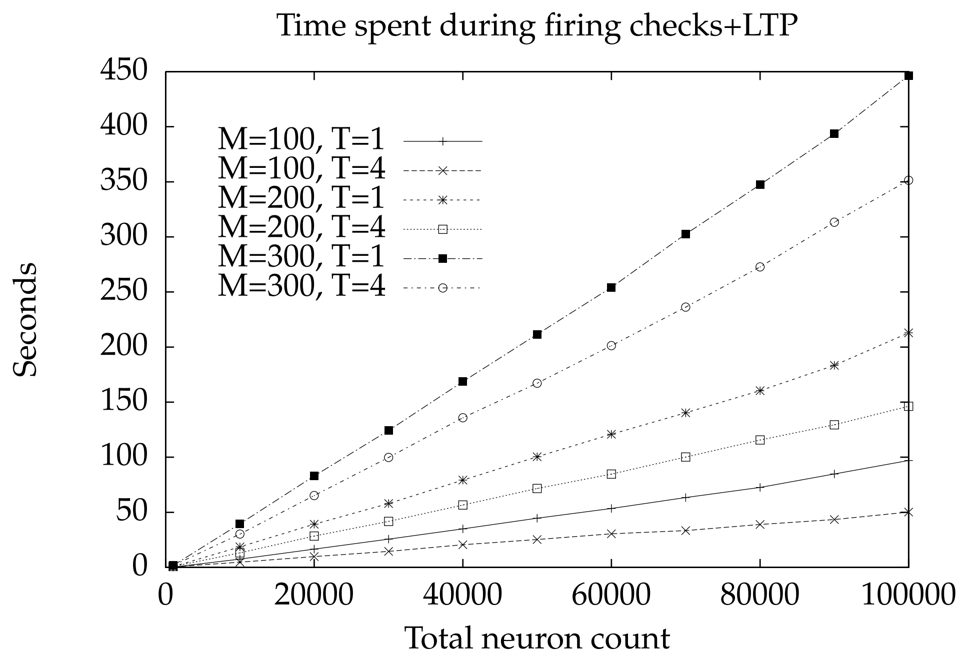


Figure 6.6: CPU: Time spent during the firing check+LTP stage for 1-thread and 4-thread test runs

general, and it would appear that for each stepwise increase in M, the gap widens.

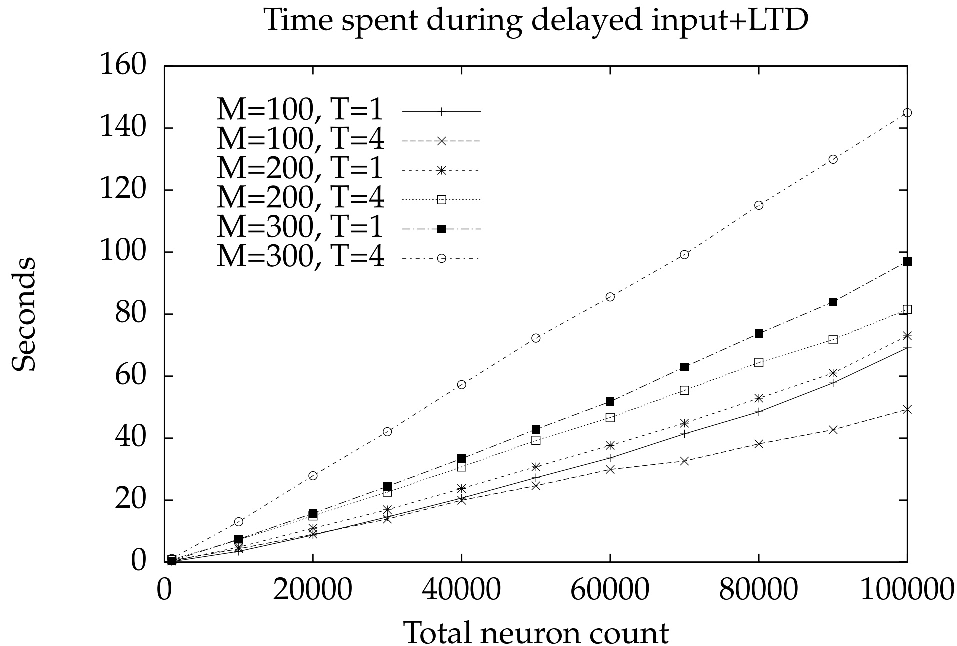


Figure 6.7: CPU: Time spent during the delayed input+LTD stage for 1-thread and 4-thread test runs

The membrane potential update stage in Figure 6.8 gives the overall best result in terms of theoretical vs. actual performance increase. Using 4 threads here actually incurs a speedup of nearly exactly 4x over the single-threaded version.

6.2 GPU

The big question is how the GPU compares up against the CPU, and whether or not it exhibits the same issues when exposed to dense synapse configurations. Due to the inherent parallel nature of the GPU, it makes sense to primarily compare it up the results of the parallel runs of the CPU, rather than the serial ones. Figure 6.9 shows the runtimes for the tested values of M. As mentioned in the previous chapter, an immediately visible problem here is the fact that the number of neurons testable for

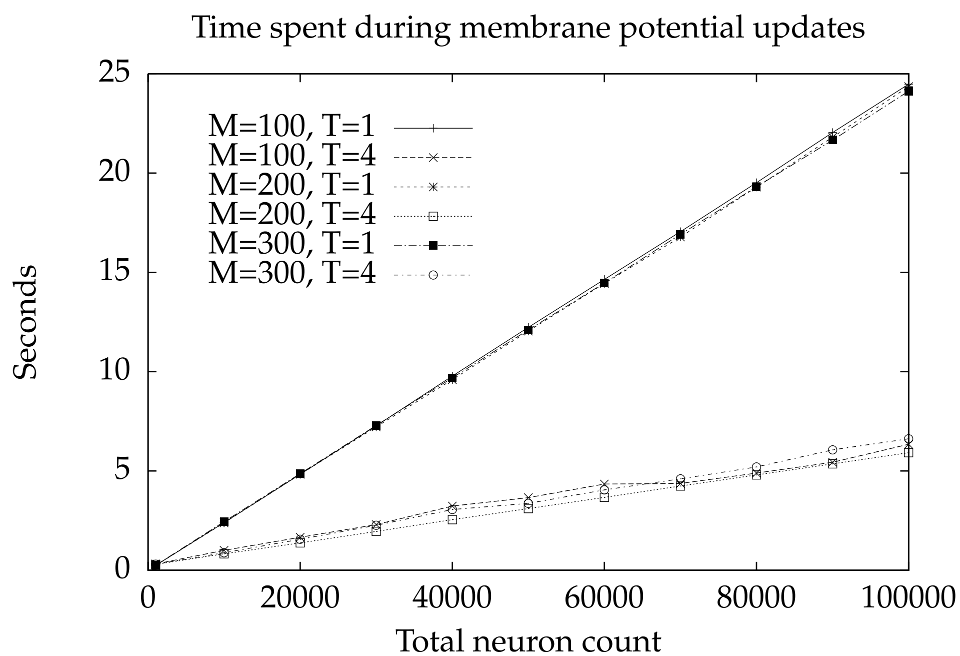


Figure 6.8: CPU: Time spent during the membrane potential update stage for 1-thread and 4-thread test runs

each increment of M is nearly halved due to the equally rapidly increasing memory requirements to represent the synaptic information (weights, derivatives, time stamps et al). Despite this, the results are fairly promising. The simulation times are consistenly lower than those of even the best performing $M = 100, T = 4$ run on the CPU, displaying a $\sim 34\%$ relative improvement for $N = 70,000$ (76 vs 50 seconds for the CPU and GPU, respectively). As M increases, the benefit seems to become even more apparent with a $\sim 61\%$ improvement for $N = 40,000$ and $M = 200$ (91 vs 35 seconds) and a $\sim 76\%$ improvement for $N = 25,000$ and $M = 300$ (~ 125 vs 30 seconds).

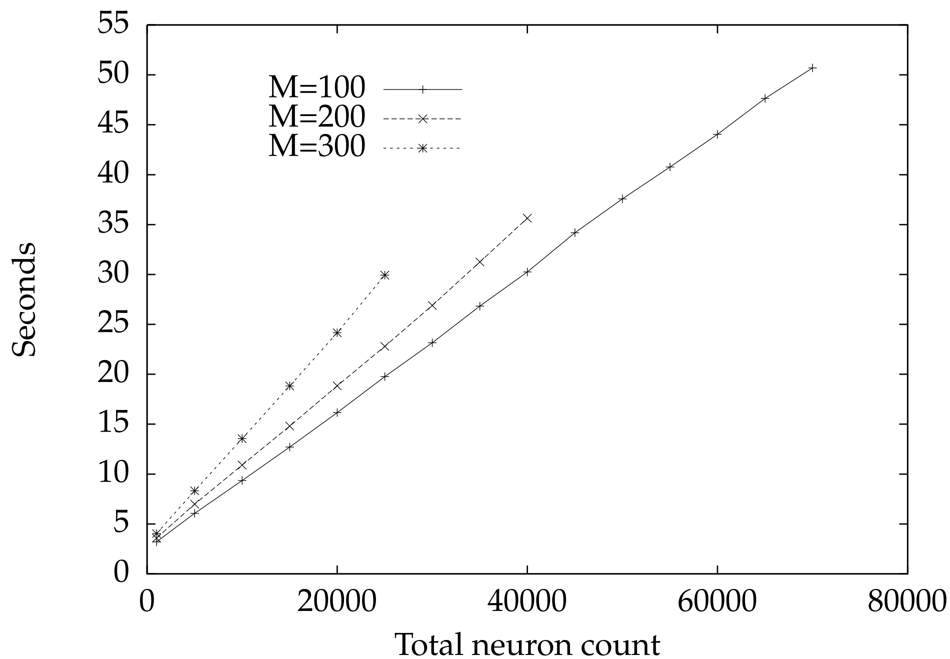


Figure 6.9: Total time spent processing 10 simulated seconds for the GPU SNN implementation $N \in [1000, 70000]$ for $M = 100$, $N \in [1000, 40000]$ for $M = 200$ and $N \in [1000, 25000]$ for $M = 300$ with 5k neuron increments

In order to get a better view on how different kernels scale with increasing neuron counts, Figures 6.10, 6.11 and 6.12 show the summation of timings for the kernels, including both the time it takes for the CPU to initiate the invocations and the actual processing time on the GPU itself. All tell essentially the same story—4 out of 7 kernels exhibit a very low growth in time as a function of N, while the remaining 3 have growths that far more resemble those of the CPU implementation. What this im-

plies will be discussed in the next chapter.

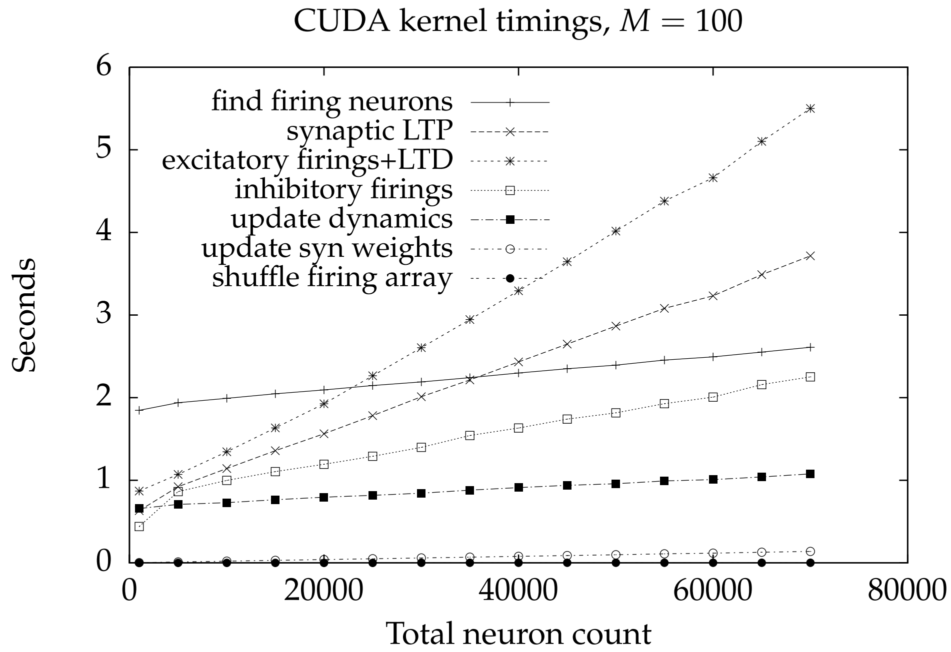


Figure 6.10: GPU kernel timings for $M = 100$. A very distinct difference in rate of change is noticeable between the kernels

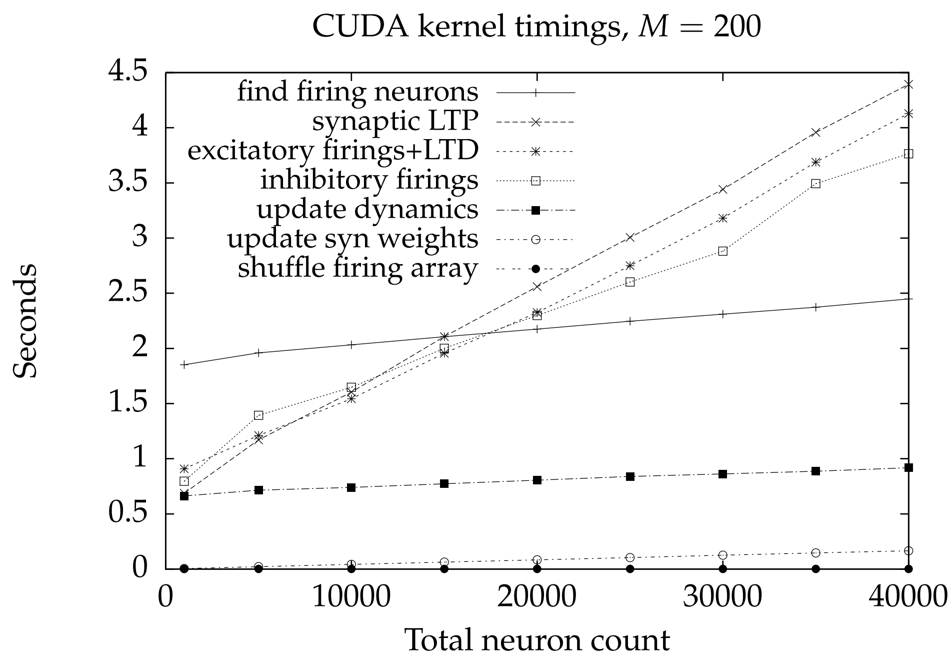


Figure 6.11: GPU kernel timings for $M = 200$.

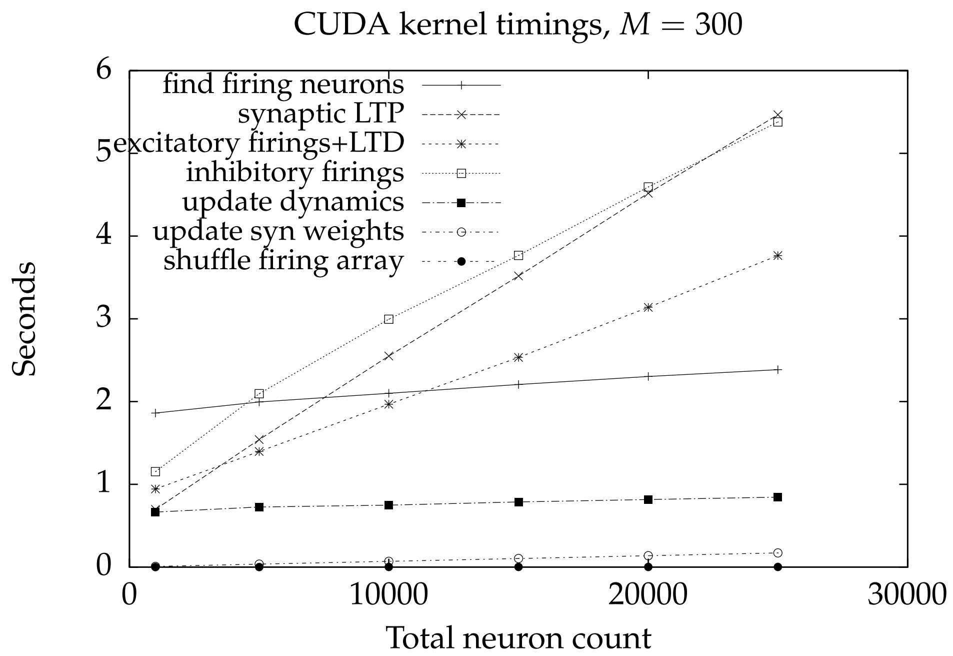


Figure 6.12: GPU kernel timings for $M = 300$.

Chapter 7

Discussion

In this chapter, the results from Chapter 6 will be analyzed and discussed, focusing on to which degree the implementations can be said to have achieved the goals they set out to, namely a notable speedup of the SNN simulation process on desktop hardware.

7.1 OpenMP implementation

The improvement results generated by the CPU implementation range from "good" (simulation time cut nearly in half) to "essentially not at all". The task here is to attempt to make sense of this and ascertain whether or not this is an issue that is inherent to the abstract problem domain—i.e. SNN parallelization on desktop hardware—or if it is primarily an issue with the concrete implementation.

The poor performance of the input reset task is a good sign that this operation should be incorporated into the membrane potential update task, as the value of a neuron's input can be safely discarded (i.e. reset) after its membrane potential v has been recalculated. Recall from Section 2.2.1 that subdividing and processing a task in parallel requires coordinating several threads both at the start of the task and after it has finished, something that is a fairly expensive operation when compared to simply running a tight inner loop on the CPU on a single thread. We can see from Figure 6.5 that the time spent for serial and parallel execution will converge if we extrapolate the graph slightly to around $110,000$ neurons, meaning that even keeping it the way it currently stands *will* be beneficial after a certain threshold is reached, but even then it stands to reason that it should be moved to the membrane potential update task.

Refer back to Section 3.2.3 for details

A very likely contributing factor for the membrane potential check and

LTP task not giving a full theoretical speedup is the fact that on average only a few of the neurons will actually have a membrane potential that is above the firing-threshold. Recalling the average firing frequencies in Figure 6.4, the probability for any given neuron firing when $M = 100$ is $\sim 5.5/N$. For those neurons that do not fire (i.e. the majority of them), the amount of code actually executed is minimal. As with the input reset task, if there is hardly any work for a thread to do, it's often simply not worth the overhead associated with distributing work to the threads, synchronizing their launches and rejoins et al. This is an issue that is very hard to avoid on a multiprocessor PC system. Despite this, there is still a tanglible benefit to keeping the parallelization, in turn justifying its presence.

The delayed neuron input and LTD task is somewhat bi-polar in its nature. It spans a 40–50% performance increase with the lowest synapse count to a 40–50% performance *loss* with the highest synapse count when run in parallel and compared to the serial version. Aside from the distribution of work between multiple threads, the only actual difference between the two is that the parallel version requires that all synaptic inputs to a neuron are added *atomically* to avoid multiple threads overwriting each other's results (see pseudo-code line 35 in Section 3.7.0.6). Floating point atomic operations are not natively supported by the hardware, meaning input addition must be emulated through other atomic operations. This works well and with no loss in precision, but requires the algorithm to continuously read back the memory location and retry an atomic write if other threads beat it to its attempt. In turn, this implies that each synapse does not mean 1 atomic operation, but rather *at least* 1 atomic operation. It also makes sense to assume that as the number of synapses (and/or CPUs) grow, so does the number of such "collisions" between threads attempting to write their inputs. Preliminary analysis indicates that in a network with $M = 300$, the atomic synaptic input alone accounts for around 30–40% of the *total* runtime.

Closely related to the previous paragraph, for any multiprocessor shared memory system, there's also a very important notion of how your implementation works with the CPUs' *caches*. In essence, there are mechanisms in place to ensure that the data kept in the CPUs' caches are coherent with each other so that no CPU attempts to read or write stale data. This mechanism will incur a not negligible scalability penalty when a lot of CPUs are attempting to read and write to regions of memory that share the same *cache line* (e.g. one can consider the main memory to be divided into regions of 64-byte cache lines that are aligned on 64-byte boundaries), as entire such cache lines will be copied back and forth between processors in a near ping-pong like fashion, constantly changing the ownership of the

The CPU cache is a low-latency area of on-chip memory that stores recently accessed parts of memory, reducing the amount of traffic to the relatively high-latency system bus.

cacheline in order to ensure no one gets out of date data[Smaalders, 2006]. Needless to say, this can essentially destroy scalability [Sutter, 2009]. The author was not aware of the significance of this mechanism before creating the implementation, so the memory buffers used were not in any way catered to avoid such problems. It is highly likely that this is phenomenon is a major overall contributing factor to the lack of synaptic scalability in the implementation, both for the cases where M was high and when $M = 100$ and the thread count was high, as well as being a presumed major reason for the performance problems faced by the CPU implementation in general. A potential way to solve this for delayed input without having to incur an algorithm redesign is to simply have each thread write to its own, non-shared input buffer of length N. This buffer could then have a *reduce* operation performed on it in parallel, adding up all the thread-owned buffers into the main, shared input buffer. Doing so should effectively both remove the need for any locking and stop the cache "ping pong" effect, at the cost of some extra memory used. This venue was not explored in the implementation itself due to lack of time, but intuitively it should scale far better than the current approach. For readers interested in parallel programming and how this phenomenon affects scalability, as well as strategies to avoid it, [Sutter, 2009] is a good starting point.

Also known as "false sharing" [Sutter, 2009]

Not directly related to parallelization, but still mentionable due to its significance is that both the two aforementioned stages use the STDP equations given in Section 3.3.2 and [Song et al., 2000] to adjust synaptic weight derivatives based on the delta between when a neuron fires and a synapse last carried a spike to the neuron. What these equations have in common is that they calculate the exponential of this delta, an operation that happens to be comparatively very expensive when run in such tight inner loops. Since all we're feeding to the exponential function is integers within a fixed range, it should be trivial to replace all instances of $exp(\Delta t / \tau)$ with precomputed lookup-tables such as *precomp_ltp*$[\Delta t]$. Testing showed that the speedup caused by removing the explicit (re)calculation of $exp(x)$ was on the order of 4–5x for the stages involved. Certainly, any "production-quality" SNN simulator should jump readily at such an easily implemented performance boost.

The remaining parallelized tasks offer promising, scalable results. Although the STDP weight update task at the end of each second was not included in graph form here, both it and the per time step membrane potential update task are trivial in their nature (i.e. no need for atomic operations) and therefore yield good scalability results, with the latter nearly reaching its theoretically optimal performance increase.

Summarizing the analysis for the various stages, we see that there are

some issues with the current algorithm and data structure design that together with certain aspects of modern multiprocessor mechanisms appear to cause the current OpenMP implementation to be unfit for the simulation of densely connected spiking neural networks. Despite of this, simulation of less dense networks see a relatively significant speedup, and it is important to note that all the problems exhibited here are rooted in the *implementation* (primarily choices on how to organize and access memory when multiple processors are involved), and not anything inherent to the PC multiprocessor platform. Given more time and capabilities to perform indepth profiling (such as being able to measure the aforementioned cache line inter-CPU copying et al) it is highly likely that a far greater factor of scalability could have been reached.

7.2 CUDA implementation

As mentioned in the CUDA results section, there is a rather massive divide in the neuron count vs. time spent per kernel, a trend that is particularly visible in Figure 6.11, which immediately indicates an implementational scalability issue.

The kernels that exhibit the worst scaling have a thing in common—they are the ones that only work on a small subset of the total neuron/synapse population based on the SNN's sparse firings. This might at first be very counter-intuitive—less usually means faster—but starts making sense when we consider how the GPU prefers to operate: it thrives on processing a great number of elements that are accessed sequentially in memory so that they can be read and written in batches, improving throughput by an order of a magnitude ("coalescing"—see Section 2.2.2). Since our SNN model is completely randomly connected, such sequential access is very tricky to achieve in the aforementioned kernels and the comparatively small amount of elements they process is not enough to keep all the GPU multiprocessors busy. When the inability to properly utilize the memory bandwidth and processing power the hardware offers is added to the fact that we have to read from—and write to—many different areas of memory just to deliver a spike over a single synapse (the index to its postsynaptic neuron, the weight derivative for STDP, the input buffer...), the resulting lack of performance becomes less of a mystery. To avoid such issues and start getting the most out of the hardware, it is likely that—as with the CPU implementation—parts of the algorithms and datastructures used would have to be redesigned entirely. This seems to be the approach taken in [Nageswaran et al., 2009], as they manage to reach speedups of

over *26x* compared to the serial version, but as few details were available for much of their model at the time of writing, no proper comparison can be made to it.

Luckily, there are also several kernels that exhibit the *exact* kind of behavior that we're looking for. The kernels for updating membrane potentials, checking for neuron firings and updating the synapse weights all work on the full population of neurons or synapses and are simple enough to allow for sequential access to memory without too many programming-related challenges. The effect of this is readily visible in the kernel time graphs—the rate of change for all these is incredibly low. To examplify, the membrane potential kernel—which runs once each time step and computes several differential equations for each and every neuron in the network—takes only ~ 0.35 seconds more to process $70,000$ neurons a total of $10,000$ times than doing this for only $1,000$ neurons! If the currently under-performing kernels could somehow be brought up to this level, this would mean some seriously impressive performance gains and scalability.

We can also see that the CUDA implementation scales much better than the OpenMP version when synaptic connectivity becomes more dense. As the techniques used for atomic updates of neuron input are essentially the same between the two, the speedup is likely both because the added synapses are amortized away by the GPU's high parallelization capabilities and memory bandwidth, as well as the fact that its multiprocessors *do not actually have any caches* that have to be kept synchronized when access occurs to the same area of memory. In this case, hardware simplicity seems to have paid off.

By further observing the evolution of kernel timings as a factor of M, we can see how the time spent processing inhibitory neurons goes steadily up, essentially becoming the kernel taking the most time to execute at $M = 300$. This can indicate that as the synaptic counts grow, the interneurons need to fire far more frequently in order to balance out all the activation going on in the network, something which also justifies having a separate kernel for handling exactly interneuron firings in a more efficient manner than doing everything in one, generic way as with the CPU implementation.

In summary, given the massive capabilities for parallelism offered by the GPU, the results offered by the SNN implementation did not quite meet the expectations, despite being consistently faster than even the fastest CPU runs. It does, however, show the *potential* power that lies in a GPU SNN when algorithms are designed in such a way that they take greater advantage of the hardware. Yet again, the problem lies in the implementation and not in the platform.

7.3 Comparison

Given what we have seen thus far (and ignoring for a moment the various performance snags that we've already discussed), the GPU comes out as the winner in essentially all tested cases, with the only real downside being its relatively limited memory capacity compared to the CPU, although this is due to thesis budget reasons [1] and not an inherent problem. Figure 7.1 shows this clearly, with the increase in time being nearly non-existent for increasing values of M when compared to the CPU.

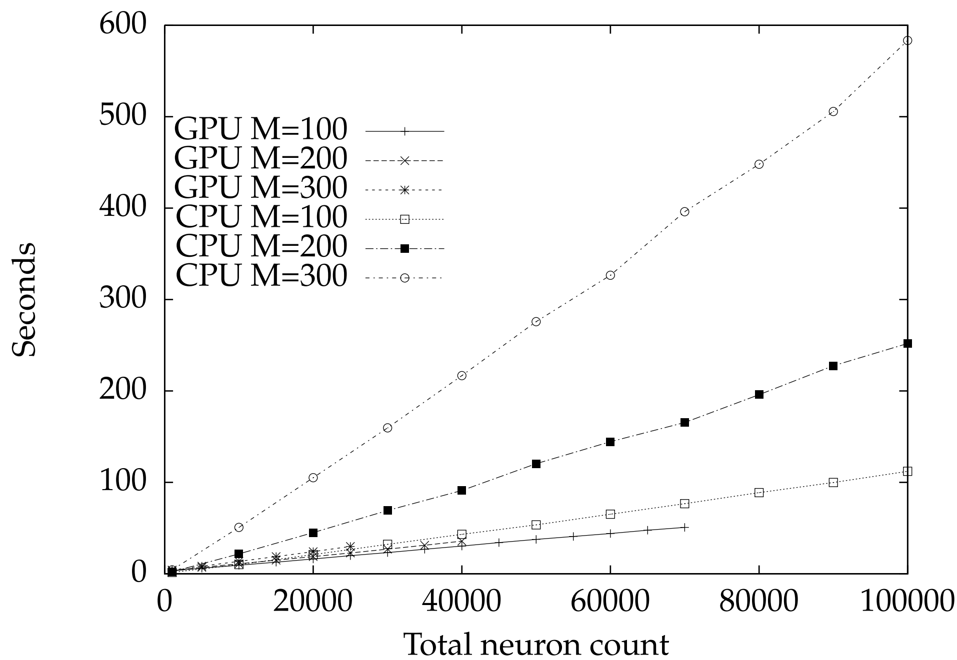


Figure 7.1: Comparison of the results from Chapter 6 for the CPU and GPU implementations. All CPU data points are from the 4-thread test runs.

[1] More specifically, none exists

Chapter 8

Conclusions

This chapter aims to bring together the discussion from Chapter 7 and the overarching motivations of the thesis and offer some thoughts and conclusions on their validity and the thesis process itself. Finally, some aspects that could warrant future work are considered.

8.1 Summary

Throughout this thesis, the capabilities of parallelization of artificial spiking neural networks have been investigated alongside strategies for implementing these on two significantly different multiprogramming platforms. By showing how SNNs exhibit very favorable traits for parallelization when conduction delays are taken into the equation, a parallel spiking network model and simulation algorithm was described that features a biologically plausible neuron model and a fully local learning mechanism based on Hebbian theory and neurobiology research.

This network model was tested with neuron counts ranging upwards to 100k neurons for the CPU OpenMP implementation and 70k neurons for the GPU CUDA implementation, all with synaptic connectivity per neuron ranging between 100–300. The OpenMP implementation reached speedups of nearly 2x over the single-threaded results on its best runs, which is half of its theoretically optimal result on a 4-core system. It did however exhibit serious scalability issues with high neuron counts, rendering the speed benefit minimal for these. The CUDA implementation consistently proved itself to be the fastest alternative, completely leaving the CPU behind at high synapse counts. In this case as well, scalability issues were identified. Reasons behind these were postulated for both the OpenMP and CUDA implementations.

If there is one definite conclusion that the author is left with after writing this thesis, it is that properly parallelizing even presumably fairly straight forward algorithms is a task filled with many different challenges—challenges certainly not lessened by programming up against two hardware platforms with diametrically opposed paradigms of how parallel programming is even performed and how performance is best achieved. The sheer amount of nuances and interplay of hardware and software that is relevant for how a given piece of parallelized code will behave in practice is often staggering and beyond the programmer's control (and perhaps even more frequently—knowledge). Whereas the majority of AI research is focused on making algorithms *work*, when one's area of focus is that of parallelization, one is suddenly faced with the problem of ensuring that it not only works, but overall works *faster*, and preferably in a way that scales as effortlessly as possible and with as much hardware investment return as possible. Otherwise, the point would be moot, as it would add complexity with no visible gains. A great deal of fairly low-level knowledge of the more esoteric aspects of the hardware is required to analyze the results and figure out exactly *why* a perfectly legitimate looking algorithm does not give the results that it theoretically should/could do, and even moreso when it comes to figuring out *how* it can be adapted into reaching such results. During development, even details such as upgrading the motherboard had more to say for GPU performance than completely overhauling several of the program kernels involved (although this probably says more about design-challenges related to the kernels than anything else).

The main limiting factor and challenge for both the OpenMP and CUDA SNN implementation can be summarized in one word: memory. Merely the patterns in which neurons and synapses are accessed can mean an order of a magnitude in difference for how the algorithm implementations behave in practice when multiple processors are involved. What this implies is that a parallel SNN system would preferably be implemented as a library by a group of experts in artificial intelligence, computer science and concurrent programming and made available to the research community. That this is in fact a highly relevant concept can be shown in [Nageswaran et al., 2009], as they appear to be contemplating the release of a highly optimized CUDA SNN library based on their research and results, closely matching the ideas that this thesis set out with (albeit with far better numbers to show for it in the end). Add this to the fact that the parallelization property of spiking networks with delays is also a fairly recent discovery ([Morrison et al., 2007]), the author finds it plausible to claim that there is still much work that can be done with developing efficient algorithms,

datastructures and synchronization methods to enable the simulation of ever larger networks, even bringing *real-time* large-scale simulation well into the realm of possibility.

Based on the results achieved in this thesis and the analysis performed upon them, the capabilities of the GPU vs. the CPU in conjunction with the results outlined in [Nageswaran et al., 2009], it seems like a logical step to conclude that if the performance problems were solved for both implementations, the GPU would pull ahead to an even greater extent than what we've seen here, making it an excellent and highly cost-efficient potential choice for SNN parallelization. This is to a large extent because the GPU directly caters to the kind of problems that highly parallelizable neural networks represent—many small elements, often a high degree of mathematical computation required and a high memory bandwidth.

With the current trend of increasing processor core counts—one that does not seem like it will taper off in the foreseeable future—and more and more of this technology finding its way into affordable consumer hardware, it is in the author's eyes not at all a far fetched concept that large-scale parallel spiking neural network simulations—as well as other parallel scientific simulations—can to an increasing degree be performed with the aid of these.

8.2 Possible future work

Chapter 7 revealed several more or less pressing issues related to realizing the potential of the hardware and tools used. A few approaches to deal with some of those are presented here, as well as other potential future tasks.

Algorithm redesign The algorithms used in this thesis are based closely on those given in [Izhikevich, 2006], which were presumably never initially designed with parallelization in mind. There are certain inherent issues with the current design, such as requiring a high number of atomic operations that is proportional the number of synapses from a neuron. This is again because neurons being processed by multiple threads may be attempting to add synaptic input to the same target (post-synaptic) neuron.

It seems plausible that this process could be "reversed" by taking inspiration from conventional integrate-and-fire networks. Rather than adding each firing neuron to a list and having them later "push" synaptic input into other neurons, it might be possible to add the

synapses themselves to a list, allowing the individual *post*synaptic neurons to "pull" their input from this, removing the need for synchronization (since a neuron's input will be touched by a single thread of execution only). Further inspiration here could be gotten from existing large-scale simulators such as [Ananthanarayanan and Modha, 2007], where synchronization overhead is already a major factor.

Improve memory access patterns As mentioned, getting good results from both the OpenMP and CUDA implementations require us to be very vigilant in how we write code that can access memory from the point of view of multiple processors. This is closely related to the previous point, but deserves its own mention.

Utilize STDP exponents lookup table It was already pointed out how removing the calculation of $\exp(x)$ essentially gave a speedup of 4–5x. Since we can do this without losing any numeric precision for the STDP calculations, this seems like a very logical step to implement in both the CPU and GPU versions of the algorithm.

(CUDA) Multi-GPU scalability Many motherboards sold today come equipped with so-called SLI (Scalable Link Interface) support, which allows multiple graphics cards to be used simultaneously. As CUDA has support for such arrangements, it would be interesting to determine the viability of running a randomly connected SNN on multiple GPUs, taking in mind such challenges as neurons having postsynaptic neurons that may exist solely in the other GPU's memory space.

Integrate into distributed system Although this thesis consciously did *not* consider large, distributed systems, it seems like a natural extension to eventually perform an integration into such a system, allowing far larger networks to be simulated by a scalable, connected array of desktop computers (should still be much more affordable than any supercomputing setup). This would naturally bring along a host of new challenges for keeping the state of the network coherent and in sync, but luckily there is already a well-established research area for such systems from which inspiration and techniques can be had.

Bibliography

Wagner Ambrus. A framework for automatic parallelization of sequential programs. In *Telecommunications, 2003. ConTEL 2003. Proceedings of the 7th International Conference on*, volume 2, pages 693–696 vol.2, June 2003. doi: 10.1109/CONTEL.2003.1215896.

Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, New York, NY, USA, 1967. ACM. doi: http://doi.acm.org/10.1145/1465482.1465560.

Rajagopal Ananthanarayanan and Dharmendra S. Modha. Anatomy of a cortical simulator. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–12, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-764-3.

Fabrice Bernhard and Renaud Keriven. Spiking neurons on GPUs. In *International Conference on Computational Science (4)*, pages 236–243, 2006.

Robert Callan. *Essence of Neural Networks*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1998. ISBN 013908732X.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. McGraw-Hill Science / Engineering / Math, 2nd edition, December 2003. ISBN 0072970545.

Kang Su Gatlin and Pete Isensee. OpenMP and C++: Reap the benefits of multithreading without all the work. *MSDN Magazine*, October 2005. URL `http://msdn.microsoft.com/en-us/magazine/cc163717.aspx`.

Donald O. Hebb. *The Organization of Behavior: A Neuropsychological Theory*. Wiley, New York, June 1949. ISBN 0805843000.

A. L. Hodgkin and A. F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of Physiology*, 117(4):500–544, 1952. URL http://jphysiol.highwire.org/content/117/4/500.short.

Eugene M. Izhikevich. Simple model of spiking neurons. *Neural Networks, IEEE Transactions on*, 14(6):1569–1572, 2003.

Eugene. M. Izhikevich. Which model to use for cortical spiking neurons? *Neural Networks, IEEE Transactions on*, 15(5):1063–1070, 2004. URL http://dx.doi.org/10.1109/TNN.2004.832719.

Eugene M. Izhikevich. Polychronization: Computation with spikes. *Neural Computation*, 18(2):245–282, February 2006. ISSN 0899-7667. URL http://vesicle.nsi.edu/users/izhikevich/publications/spnet.htm.

Eugene M. Izhikevich and Gerald M. Edelman. Large-scale model of mammalian thalamocortical systems. *Proceedings of the National Academy of Sciences*, 105(9):3593–3598, 2008. doi: 10.1073/pnas.0712231105. URL http://vesicle.nsi.edu/users/izhikevich/publications/large-scale_model_of_human_brain.htm.

A. Kasiński and F. Ponulak. Comparison of supervised learning methods for spike time coding in spiking neural networks. *International Journal of Applied Mathematics and Computer Science*, 16(1):101–113, 2006. URL http://baztech.icm.edu.pl/baztech/cgi-bin/btgetdoc.cgi?BPZ1-0028-0007.

Abigail Morrison, Sirko Straube, Hans Ekkehard Plesser, and Markus Diesmann. Exact subthreshold integration with continuous spike times in discrete-time neural network simulations. *Neural Comput.*, 19(1):47–79, 2007. ISSN 0899-7667. doi: http://dx.doi.org/10.1162/neco.2007.19.1.47.

Jayram Moorkanikara Nageswaran, Nikil Dutt, Jeffrey L Krichmar, Alex Nicolau, and Alex Veidenbaum. Efficient simulation of large-scale spiking neural networks using CUDA graphics processors. In *IJCNN '09*. University of California, Irvine, June 2009.

John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. In *SIGGRAPH '08: ACM SIGGRAPH 2008 classes*, pages 1–14, New York, NY, USA, 2008. ACM. doi: 10.1145/1401132.1401152. URL http://dx.doi.org/10.1145/1401132.1401152.

NVIDIA. NVIDIA CUDA^{TM} Programming Guide version 2.1, December 2008. URL http://developer.download.nvidia.com/compute/cuda/2_1/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.1.pdf.

Randall C. O'Reilly and Yuko Munakata. *Computational Explorations in Cognitive Neuroscience: Understanding the Mind by Simulating the Brain*. MIT Press, Cambridge, MA, USA, 2000. ISBN 0262650541.

Christo Panchev and Stefan Wermter. Spike-timing-dependent synaptic plasticity: from single spikes to spike trains. *Neurocomputing*, 58-60:365 – 371, 2004. ISSN 0925-2312. doi: DOI:10.1016/j.neucom.2004.01.068.

Udo Seiffert. Artificial neural networks on massively parallel computer hardware. *Neurocomputing*, 57:135–150, 2004. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.6.8003.

Bart Smaalders. Performance anti-patterns. *ACM Queue*, 4(1):44–50, 2006. ISSN 1542-7730. doi: http://doi.acm.org/10.1145/1117389.1117403.

S. Song, K. D. Miller, and L. F. Abbott. Competitive hebbian learning through spike-timing-dependent synaptic plasticity. *Nat Neurosci*, 3(9): 919–926, September 2000. ISSN 1097-6256. URL http://dx.doi.org/10.1038/78829.

Herb Sutter. Eliminate false sharing. *Dr. Dobb's Journal*, May 2009. URL http://www.ddj.com/go-parallel/article/showArticle.jhtml?articleID=217500206.