Simulating Spiking Neural Networks on Multiple GPUs

When a Microsecond is an Eternity

by

Dennis Bautembach

PhD Dissertation
Presented
in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

Heraklion, Feb 2022
UNIVERSITY OF CRETE
DEPARTMENT OF COMPUTER SCIENCE
Simulating Spiking Neural Networks on Multiple GPUs
PhD Dissertation Presented
by Dennis Bautembach
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

APPROVED BY:

Author: Dennis Bautembach

Supervisor: Prof. Antonis Argyros

Committee Member: Prof. George Papagiannakis

Committee Member: Prof. Panos Trahanias

Committee Member: Prof. Antonios Savvidis

Committee Member: Prof. Nikos Komodakis

Committee Member: Prof. Ioannis Tellis

Committee Member: Prof. Panagiota Poirazi

Department Chairman: Antonis Argyros, Professor, University of Crete

Heraklion, Feb 2022
I thank my supervisor Prof. Antonis Argyros and my co-authors Dr. Iasonas Oikonomidis and Dr. Nikos Kyriazis.
Contents

Contents vii
Figures ix
Tables x
Code Listings x
Foreword xi

INTRODUCTION 1

1 Abstract 3

2 Definitions 7

3 SNN Training 9
  3.1 Meta Algorithms ...................................................... 9
  3.2 Neuroplasticity ....................................................... 10

4 SNN Simulation 13
  4.1 Strategy ............................................................... 13
  4.2 Data Structures ..................................................... 15
  4.3 (Non-)instant. Interactions ......................................... 16
  4.4 Delays ................................................................. 16
  4.5 Plasticity ............................................................. 17
  4.6 Customizability ..................................................... 17
  4.7 API ................................................................. 19
  4.8 Biological Fidelity .................................................. 19
  4.9 Hardware .......................................................... 20

5 Related Work 21
  5.1 Clock-driven Simulators .............................................. 21
  5.2 Event-driven Simulators ............................................. 22
  5.3 Parallel Simulators .................................................. 22
  5.4 Neuromorphic Simulators .......................................... 24

6 GPGPU Primer 27
  6.1 Overall Architecture ............................................... 27
  6.2 SIMD vs. SIMT vs. SMT ............................................. 28
  6.3 Memory Coalescing ................................................. 29
  6.4 Cache ................................................................. 31
  6.5 Divergence .......................................................... 31
  6.6 Asynchronicity ...................................................... 32
Figures

2.1 LIF neuron charge-discharge pattern .............................................. 8
5.1 TrueNorth chip design ................................................................. 25
6.1 CPU versus GPU ....................................................................... 27
6.2 AoS versus SoA ........................................................................ 30
6.3 Effects of cache ........................................................................ 31
6.4 CUDA API asynchronicity ............................................................ 32
6.5 Dual-copy engines .................................................................... 33
6.6 CUDA API overhead .................................................................. 33
6.7 PCIe-bus ..................................................................................... 34
7.1 Spice’s data structures ................................................................. 38
7.2 Sorted, uniform random numbers on the GPU ............................... 38
7.3 Pipeline with fixed, global delay .................................................. 39
7.4 Pipeline with user-defined, global delay ....................................... 39
7.5 Pipeline with user-defined, per-synapse delays ............................. 40
7.6 Naive Spike Delivery ................................................................. 41
7.7 Naive vs. cache-aware spike delivery (sim. time) ........................... 41
7.8 Naive vs. cache-aware spike delivery (memory access) ............... 42
7.9 Shared Memory-based Spike Delivery ......................................... 42
7.10 Scheduling in shared memory-based spike delivery .................. 43
7.11 Shared memory-based vs. naive spike delivery (speedup) .......... 44
7.12 Plasticity at different optimization levels (pseudocode) .......... 45
7.13 Event-driven vs. lazy vs. naive plasticity (speedup) .................... 46
8.1 Dual-GPU pipeline ................................................................. 47
8.2 Distributing a SNN across multiple GPUs (mechanics) ............... 48
8.3 Spike synchronization ................................................................. 48
8.4 Latency hiding ........................................................................... 49
8.5 Profile-guided vs. static load balancing ..................................... 50
8.6 Distributing a SNN across multiple GPUs (load balancing) ....... 50
8.7 Multi-GPU scaling ................................................................... 51
8.8 Scaling under different scenarios ............................................... 51
8.9 Super-linear scaling .................................................................... 51
9.1 LOC distribution in Spice ............................................................ 53
9.2 Spice folder structure ................................................................. 54
9.3 Spice class diagram .................................................................... 56
9.4 Efficient task synchronization on the GPU ................................. 57
9.5 GeNN vs. Spice (API verbosity) .................................................. 58
9.6 Spice blueprint (single-GPU) ....................................................... 61
9.7 Spice blueprint (multi-GPU) ........................................................ 62
10.1 Topologies and firing patterns of Vogels and Brunel(+) .............. 66
10.2 Setup time ................................................................................ 67
10.3 Simulation time .......................................................................... 68
10.4 Multi-GPU scaleup ..................................................................... 69
10.5 Multi-GPU speedup .................................................................... 69
11.1 Strengths and weaknesses ........................................................ 72
Tables

1.1 ANNs vs. SNNs ........................................... 4
4.1 Clock vs. event-driven Simulators .......................... 15
5.1 Overview of clock- and event-driven Simulators .......... 23
5.2 Overview of neuromorphic Simulators .......................... 24
10.1 Test HW Specs ........................................... 66

Code Listings

4.1 Clock-driven Simulation Loop ............................... 13
4.2 Event-driven Simulation Loop ............................... 13
4.3 Clock-driven Poisson Neuron ............................... 14
4.4 Event-driven Poisson Neuron ............................... 14
4.5 NeuronGPU’s API ........................................... 18
4.6 GeNN’s API ................................................ 18
4.7 Spice’s API ................................................ 18
5.1 Brian’s API ................................................ 21

6.1 Example of the SIMT Dialect in CUDA C ................. 28
7.1 Naïve spike delivery ......................................... 40
7.2 Cache-aware Spike Delivery ................................. 41
7.3 Shared memory-based spike delivery ......................... 42

9.1 Unit-testing a SNN .......................................... 55
9.2 Spice Tutorial: Simplistic SNN .............................. 59
9.3 Spice Tutorial: SSSP ......................................... 60
Foreword

Everyone who has been remotely tuned in to recent progress in machine learning has heard of the current second generation artificial neural networks (ANNs) used for machine learning. These are generally fully connected, take in continuous values, and output continuous values. Although they have allowed us to make breakthrough progress in many fields, they are biologically inaccurate and do not actually mimic the actual mechanisms of our brain’s neurons.

The third generation of neural networks, spiking neural networks (SNNs), aims to bridge the gap between neuroscience and machine learning, using biologically-realistic models of neurons to carry out computation. A spiking neural network is fundamentally different from the neural networks that the machine learning community knows. SNNs operate using spikes, which are discrete events that take place at points in time, rather than continuous values. The occurrence of a spike is determined by differential equations that represent various biological processes; the most important of which is the membrane potential of the neuron. Essentially, once a neuron reaches a certain potential, it spikes, and the potential of that neuron is reset. The most common model for this is the leaky integrate-and-fire (LIF) model. Additionally, SNNs are often sparsely connected and take advantage of specialized network topologies.

At first glance, this may seem like a step backward. We have moved from continuous outputs to binary, and these spike trains are not very interpretable. However, spike trains offer us enhanced ability to process spatio-temporal data, or in other words, real-world sensory data. The spatial aspect refers to the fact that neurons are only connected to neurons local to them, so these inherently process chunks of the input separately (similar to how a convolutional neural network (CNN) would using a filter). The temporal aspect refers to the fact that spike trains occur over time; so what we lose in binary encoding, we gain in the temporal information of the spikes. This allows us to naturally process temporal data without the extra complexity that recurrent neural networks (RNNs) add. It has been proven, in fact, that spiking neurons are fundamentally more powerful computational units than traditional artificial neurons [1].

Given that these SNNs are more powerful, in theory, than second generation networks, it is natural to wonder why we do not see widespread use of them. The main issue that currently lies in practical use of SNNs is that of training. Although we have unsupervised biological learning methods such as Hebbian learning [2] and spike-timing-dependent plasticity (STDP) [3], there are no known effective supervised training methods for SNNs that offer higher performance than second generation networks. Since spike trains are not differentiable, we cannot train SNNs using gradient descent without losing the
precise temporal information in spike trains. Therefore, in order to properly use SNNs for real-world tasks, we would need to develop an effective supervised learning method. This is a very difficult task, as doing so would involve determining how the human brain actually learns, given the biological realism in these networks.

Another issue, that we are much closer to solving, is that simulating SNNs on normal hardware is very computationally-intensive since it requires simulating differential equations. However, neuromorphic hardware such as IBM’s TrueNorth aims to solve this by simulating neurons using specialized hardware that can take advantage of the discrete and sparse nature of neuronal spiking behavior.

The future of SNNs, therefore, remains unclear. On one hand, they are the natural successor of our current neural networks, but on the other, they are quite far from being practical tools for most tasks. There are some current real-world applications of SNNs in real-time image and audio processing, but the literature on practical applications remains sparse. Most papers on SNNs are either theoretical, or show performance under that of a simple fully-connected second generation network. However, there are many teams working on developing SNN supervised learning rules, and I remain optimistic for the future of SNNs.

Devin Soni [4] *

* with permission from the author
Introduction
Spiking neural networks (SNNs) are a class of artificial neural networks (ANNs) that attempt to, more accurately, model the processes inside biological neural networks such as the (human) brain. They are a generalization of “conventional” or “deep” ANNs. Conventional neural networks (be it convolutional, recurrent, or other networks) are typically layer-based and produce continuous outputs, which can be computed via simple matrix multiplications interleaved with non-linear (activation) functions. In contrast, SNNs can resemble arbitrary directed graphs. They consist of neurons, corresponding to the graph’s vertices, which are connected via synapses, corresponding to the graph’s edges. Both neurons and synapses have their own state, consisting of arbitrary attributes, which can be governed by arbitrary dynamics. In addition, neurons can fire or “spike” in which case a signal/message must be transmitted to their neighbors via their out-going synapses. A SNN’s output is its firing pattern.

We can immediately see how this behavior is quite similar to the electro-chemical processes happening in the brain. Several advantages can be derived from this similarity: Maass [1] proved in 1996 that SNNs are fundamentally more powerful computationally than conventional ANNs. In practice SNNs still lag behind ANNs but research around them remains vivid and promising. The gap is constantly shrinking so that SNNs may one day live up to their reputation. One aspect in which SNNs have already overtaken ANNs is power-efficiency [6], especially in combination with neuromorphic hardware (Section 4.9). In fact, they are so efficient that converting ANNs into SNNs has become its own field of research [7–9].

SNNs’ novelty also bears disadvantages. Many solved problems such as the efficient inference and training of ANNs, have to be re-thought for SNNs due to their drastically different nature. Inference requires full-blown simulation (Chapter 4). While training (Chapter 3) via meta-algorithms such as Backpropagation (BP) is possible (in fact, several attempts to adapt BP to SNNs have been made), SNNs lend themselves to a different kind of training: neuroplasticity. As the SNN is being simulated, it constantly self-adapts, producing ever-improving outputs. Training becomes an inherent part of the model and the simulator becomes responsible for driving it.

This is why I have dedicated all my research to simulation, which I see as an even more fundamental issue than training: A fast, resource-efficient, and user-friendly simulator not only speeds up existing simulations. It accelerates network design (prototyping/parameter tuning/etc.) and research into other algorithms, including training, advancing the field as a whole. To this effect I present Spice (/spack/), a state of the art SNN simulator. Spice is superior in terms of performance (speed, setup time, memory consumption) and ease of use. It is also the first simulator to scale linearly to eight GPUs. This is achieved by novel algorithms for spike
delivery (Section 7.3) and plasticity (Section 7.4), a novel parallelization scheme (Chapter 8), as well as a unique, modern API (Section 9.4). We will explore these algorithms and witness their evolution over several optimization levels, from naïve "first" implementations all the way to outperforming the state of the art.

Table 1.1 summarizes the key differences between ANNs and SNNs. Additionally, novices may want to read Kulkarni et al. [10] as a broad yet gentle introduction (touching on algorithms, hardware, and applications) before continuing onto the more thorough and technical discussion in this thesis.

<table>
<thead>
<tr>
<th>Table 1.1 Key differences between conventional ANNs and SNNs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ANN</strong></td>
</tr>
<tr>
<td>Topology</td>
</tr>
<tr>
<td>Output</td>
</tr>
<tr>
<td>Inference</td>
</tr>
<tr>
<td>Training</td>
</tr>
</tbody>
</table>
Προσομοίωση Νευρωνικών Δικτύων Δυναμικών Ενέργειας σε Πολλαπλές Μονάδες Επεξεργασίας Γραφικών

Τα Νευρωνικά Δίκτυα Δυναμικών Ενέργειας (ΝΔΔΕ) είναι μια κατηγορία Τεχνητών Νευρωνικών Δικτύων (ΤΝΔ) που επικεφαλής να μοντελοποιήσουν, με μεγαλύτερη ακρίβεια, τις διαδικασίες σε βιολογικά νευρωνικά δίκτυα όπως ο (ανθρώπινος) εγκέφαλος. Είναι μια γενικότερη των «συμβατικών» ή «βαθέων» ΤΝΔ. Τα συμβατικά νευρωνικά δίκτυα, είτε είναι συνελεκτικα, επαναλαμβανόμενα ή άλλης μορφής, τυπικά είναι διαστρωματωμένα και παράγουν συνεχείς εξόδους, οι οποίες μπορούν να υπολογιστούν μέσω απλών πολλαπλασιασμών μήτρας σε συνδυασμό με μη γραμμικές συναρτήσεις εγκέφαλος. Αντίθετα, τα ΝΔΔΕ μπορούν να μιαζόνται με αυθαίρετως κατευθυνόμενα γράφους. Αποτελούνται από νευρώνες, που αντιστοιχούν στις κορυφές του γράφου, οι οποίοι συνδέονται μέσω συνάψεων, που αντιστοιχούν στις ακμές του γράφου. Τόσοι οι νευρώνες όσοι και οι συνάψεις έχουν τη δική τους κατάσταση, που αποτελείται από ποικίλα χαρακτηριστικών, τα οποία μπορούν να διεπονταίνουν από αυθαίρετες δυναμικές. Επιπλέον, οι νευρώνες έχουν ένα σχήμα πυροδότησης με μορφή ακίδας, οπότε ένα σήμα/μήνυμα πρέπει να μεταδοθεί στους γείτονές τους μέσω των εξερχόμενων συνάψεων τους. Η έξοδος ενός ΝΔΔΕ είναι το μιζόνι πυροδότησης του. Μπορούμε αμέσως να δούμε την ομοιότητα αυτής της συμπεριφοράς με τις ηλεκτρονικές διεργασίες που συμβαίνουν στον εγκέφαλο. Πολλά πλεονεκτήματα μπορούν να προκύψουν από αυτή την ομοιότητα: Ο Maass απέδειξε το 1996 ότι τα ΝΔΔΕ είναι θεμελιωδώς πιο ισχυρά υπολογιστικά από τα συμβατικά ΤΝΔ. Στην πράξη, τα ΝΔΔΕ εξακολουθούν να υστερούν σε σχέση με τα ΤΝΔ, αλλά η έρευνα γύρω από αυτά παραμένει ζωτική και πολλά υποσχόμενη. Το τέλος συρρικνώνεται συνεχώς, έτσι ώστε τα ΝΔΔΕ να μπορούν μια μέρα να ανταποκρίνονται στη φήμη τους. Μα πτυχή στην οποία τα ΝΔΔΕ έχουν ήδη εξερευνήσει τα ΤΝΔ είναι η απόδοση ισχύος, ειδικά σε συνδυασμό με ουσιαστικά υλικά. Στην πραγματικότητα, είναι τόσο αποτελεσματικά που η μετατροπή των ΤΝΔ σε ΝΔΔΕ ορισθεί ένα συγκεκριμένο πεδίο έρευνας. Η καινοτομία των ΝΔΔΕ έχει επίσης μειονεκτήματα. Πολλά λυμένα προβλήματα, όπως η αποτελεσματική εξαγωγή συμπερασμάτων και η εκπαίδευση των ΤΝΔ, πρέπει να επανεξεταστούν για τα ΝΔΔΕ λόγω της δραστικής διαφορετικής φύσης τους. Η διαδικασία εξαγωγής συμπερασμάτων (inference) απαιτεί πλήρη προσομοίωση. Ενώ η εκπαίδευση μέσω μετα-αλγορίθμων όπως ο Αλγόριθμος Οπισθοδιάδοσης (ΑΟ) (Backpropagation) είναι δυνατή (έχουν γίνει αρκετές προσπάθειες προσαρμογής του ΑΟ σε ΝΔΔΕ), τα ΝΔΔΕ προσφέρονται για ένα διαφορετικό είδος εκπαίδευσης: τη νευροπλαστικότητα. Καθώς ένα ΝΔΔΕ προσομοιώνεται, αυτοπροσαρμόζεται συνεχώς, παράγοντας συνεχώς θεωρίαν συμπερασμάτων. Η εκπαίδευση γίνεται εγγενείς μέρος του μοντέλου και ο προσομοιωτικός γίνεται υπευθυνός για την οδήγησή του. Αυτός είναι ο λόγος για τον οποίο αφιερώσαμε τη συγκεκριμένη έρευνα στην προσομοίωση, την οποία θεωρούμε ακόμη πιο θεμελιώδες ζήτημα από την εκπαίδευση: Ένας γρήγορος, αποδοτικός από πλευράς πόρων και φιλικός προς το χρήστη προσομοιωτικός όχι
μόνο επιταχύνει τις υπάρχουσες προσομοιώσεις αλλά και επιταχύνει τη σχεδίαση
dικτύου (πρωτότυπο/συντονισμό παραμέτρων, κ.λ.π.) και την έρευνα σε άλλους
αλγόριθμους, συμπεριλαμβανομένης της εκπαίδευσης, προσάγοντας το πεδίο στο
σύνολό του. Για το σκοπό αυτό παρουσιάζουμε το Spice, έναν προσομιωτή ΝΔΔΕ
που αναπτύχθηκε. Το Spice υπερτερεί όσον αφορά την απόδοση (ταχύτητα, χρόνο
ρύθμισης, κατανάλωση μνήμης) και ευκολία χρήσης έναντι των υφιστάμενων
προσομιωτών. Είναι επίσης ο πρώτος προσομιωτής που κλιμακώνεται γραμμικά
σε οκτώ Μονάδες Επεξεργασίας Γραφικών (GPUs). Αυτό επιτυγχάνεται με νέους
αλγόριθμους για την δημιουργία των ακίδων και την πλαστικότητα, ένα νέο σχήμα
παραλληλοποίησης, καθώς και μία σύγχρονη Διεπαφή Προγραμματισμού
Εφαρμογών (Application Programming Interface - API). Η διατριβή αυτή
dιερευνά αυτά τα ζητήματα και παρουσιάζει την αντιμετώπισή τους σε διάφορα επίπεδα
βελτιστοποίησης, ξεκινώντας από κάποιες πρώτες βασικές υλοποιήσεις μέχρι την
tελευταία τους εκδοχή η οποία ξεπερνά σε επιδόσεις τις καλύτερες υφιστάμενες
τεχνικές.
Definitions

For our purposes a spiking neural network is a directed graph of **neurons** \( N \) and **synapses** \( S \) where the neurons represent the graph’s vertices and the synapses represent its edges. Neurons and synapses can have arbitrary attributes. \(^1\) Each synapse has a **delay** which determines how long spikes travel through it. Both neurons and synapses undergo state changes a.k.a **dynamics**.

A **model** is the totality of the directed graph \((N, S)\), the synapses’ delays, the neurons’ and synapses’ attributes, their initial values, and their dynamics. We differentiate between **static** and **plastic** models. In static models, all synapses’ attributes are static (once initialized they do not change their values anymore) while in plastic models synapses are subject to change as governed by their dynamics. Neurons are always plastic as otherwise the network would remain completely inactive.

In order to **simulate** a SNN we must advance its neurons’ and synapses’ dynamics from the network’s initial state to some desired point in time thereafter. As a result of their dynamics, neurons may fire or **spike**. When a neuron \( n \) fires, we must

- Immediately: Notify all synapses \((m, n)\) of a **post-synaptic** spike.
- After \( \text{delay}(n, m) \) time:
  - Notify all synapses \((n, m)\) of a **pre-synaptic** spike.
  - Notify \( n \)’s neighbors of a spike.

Simulation is discussed in depth in Chapter 4.

A SNN’s **output** is its **firing pattern** a.k.a spike train. \(^2\) This output can either be used directly or it can be further interpreted. For example, a pong playing SNN might have a dedicated subset of two output neurons whose spike trains directly correspond to the paddle’s up and down motions. Discrete spike trains can be converted back to continuous outputs by computing their (recent) firing rate. Instead of designating output neurons, we can also consider **all** neurons’ spike trains, as done by liquid state machines \cite{[11]}, for example, where a conventional ANN interprets a SNN’s (“reservoir”) spike trains and derives an output from it.

**Training** is the act of evolving a SNN, typically in an iterative fashion, from its initial state into one where it produces desired outputs. This can be done through a meta algorithm which adjusts the synaptic weights after which the network remains completely static and basically performs inference. Alternatively, training can be made an inherent part of the model where the network self-adapts and produces ever-improving outputs as the simulation progresses. Chapter 3 goes into more details.

Although SNNs can resemble arbitrary directed graphs, most SNNs are defined parametrically rather than explicitly. Neurons are organized into **populations**. These populations can be both inter- and intra-connected.

---

1 Different neurons/synapses can have different attributes.

2 A neuron’s firing times are called its spike train.
Popular connectivity types include (let $A$ and $B$ be potentially identical neuron populations):

- **complete**—every neuron in $A$ is connected with every neuron in $B$.
- **fixed out-degree**—every neuron in $A$ has a fixed number of out-going connections to neurons in $B$, picked uniformly at random.
- **fixed in-degree**—every neuron in $B$ has a fixed number of incoming connections from $A$, picked uniformly at random.
- **fixed probability**—every neuron in $A$ is connected to every neuron in $B$ with a fixed probability.

While there are dozens of popular neuron types and infinitely more that can be conceived, one of the most fundamental building blocks of SNNs is the **leaky-integrate-and-fire (LIF)** neuron. It is modelled after a capacitor—incoming spikes increase its potential asymptotically until it reaches a threshold, which is when it discharges abruptly, emitting a spike. After each spike it becomes inactive (it neither undergoes dynamics nor receives any spikes) for a **refractory period** (Figure 2.1). In the absence of any incoming spikes its potential slowly drops, hence it’s leaky.
For completeness I briefly cover training of SNNs. The interested reader will find a recent review by Tavanaei et al. \[^{[12]}\] in which the authors compare the accuracy, computational cost, and hardware friendliness of both supervised and unsupervised training methods for SNNs.

In addition to popular classifications from Deep Learning (supervised, unsupervised, etc.) SNNs introduce two new classifications orthogonal to the rest and unique to them: meta algorithms and neuroplasticity. Meta algorithms adjust synaptic weights after which the SNN remains static and basically performs inference. In neuroplasticity, training is an inherent part of the model and thus the simulation. As the network is being simulated it perpetually self-adapts, producing ever improving outputs. The line between training and inference dissolves. Since in conventional ANNs training and inference are completely separate steps, all training methods can be considered meta algorithms there.

### 3.1 Meta Algorithms

The wake-sleep algorithm \[^{[13]}\] is an unsupervised learning algorithm for multilayer networks with the goal of constructing optimal abstract representations of the input data using an objective metric of information density to determine weights. Networks trained using this algorithm can be used as a pre-process to other networks that perform the actual classification/regression task.

The purpose of each layer is to extract an underlying structure from the data it is being fed. To this end, the network is equipped with (forward) recognition connections and (backward) generative connections, whose purpose it is to reconstruct an approximation of the input data from the representation calculated by the recognition connections. Learning is then the process of determining weights that minimize the difference between this approximation and the input data. Learning happens in two phases. In the wake phase, the network receives input data, produces responses in the hidden layers using recognition connectivity weights and trains the generative weights to invert the recognition model. In the sleep phase, the network generates fantasy signals using generative connectivity weights and trains the recognition weights to invert the generative model.

Thiele et al. \[^{[14]}\] successfully use the wake-sleep algorithm to counter the effects of high learning rates on recurrent SNNs trained with neuroplasticity. These typically cause strong feedback loops inside the network which lead to the emergence of attractor states that strongly reduce the number of representable patterns and decrease the inference ability of the network. By employing Hebbian learning (Section 3.2) in addition to neuroplastic training, they are able to unlearn these attractor states,
which ultimately allows them to increase learning rates by a factor of ten while avoiding clustering.

Bohte et al. [15] approximate the discontinuities introduced by neuron potentials in SNNs by a linear function which allows the use of conventional backpropagation for training. However, this approximation is only valid for small learning rates which slows down convergence. The performance of so-trained SNNs was on par with that of ANNs for modelling the XOR function and small datasets of that time. SkipeProp’s performance should be re-evaluated under modern conditions regarding network and dataset sizes.

Lee et al. [16] model the activity of a spiking neuron with a single sum of exponentially decaying terms $X$. This equation is real-valued and continuous except for the time point where the spike occurs. The authors treat this spike as noise and apply a lowpass filter on $X$ which yields a derivable function that can be used to employ conventional backpropagation on SNNs. This technique reduces error rate by a factor of three compared to the best previous SNNs and also outperforms convolutional neural networks (CNNs) on the MNIST dataset. It would be worth evaluating the technique’s performance on more complex datasets.

O’Connor et al. [17], on the other hand, implicitly compute gradients from the dynamics of the neural network: The authors interpret a (spiking) neural network as a network of recurrently connected neurons with symmetric weights ($w_{ij} = w_{ji}$), also known as a “continuous Hopfield Network.” Hopfield [18] proposed an energy-function for such a network. However, this energy function is based on neuron’s continuous output values. The authors translate binary spiking neurons into continuous values via stochastic approximation. They then define the dynamics of the network as the derivative of this energy function with respect to its activations. This formulation allows them to employ “Equilibrium Propagation” [19] for training. SNNs trained with this algorithm were compared to state of the art ANNs on MNIST, where they performed comparably in both performance and convergence rate.

### 3.2 Neuroplasticity

**Hebbian learning** is a rule for optimizing weights in an existing, trained network with the goal of improving the network’s performance and simplifying its structure by increasing some weights and reducing others—and removing connections completely when their weights reach zero.

In *The Organization of Behavior* [20], first published in 1949, Hebb postulated that “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 on one or both cells such that A’s efficacy as one of the cells firing B, is increased.” This can be modelled in neural networks by increasing weights which connect neurons that fire synchronously, and decreasing weights that connect neurons which fire asynchronously.
**Spike-timing Dependent Plasticity (STDP)** [3] is another neuroplastic model which strengthens connections based on the relative timing of a particular synapse’s output and input action potentials (or spikes), in contrast to Hebbian learning which is concerned with neurons. Still, many researchers view STDP merely as a variation of Hebbian learning and accredit Hebb with the formulation.

Neuroplasticity was first applied to SNNs by Kempter et al. [2]. The authors consider a single neuron with one or more (input) synapses with efficacies (weights), and one output axon. Efficacies are changed proportionally to the time differences between all pairs of input and output spikes.

More recently, Kozdon and Bentley [21] combined a simple evolutionary approach with neuroplasticity by optimizing the entire population via the Hebbian rule between each generation. The authors showed that this combination leads to faster convergence on the optimal solutions, better stability of fit solutions and higher fitness of the whole population compared to using evolution alone.
Chapter 2 lists the minimum set of features any software must offer in order to qualify as a SNN simulator. In this chapter we discuss simulation of SNNs in depth, covering simulation strategies (Section 4.1), their implementation details (Sections 4.2–4.5), and other design considerations (Sections 4.6–4.9). For an even more thorough albeit slightly outdated overview the reader may consult the phenomenal review paper by Brette et al. [22].

It should be noted that no one architecture is superior—its choice is simply a question of usage requirements. For example, a customizable simulator (Section 4.6) is not better than a non-customizable one if the latter can simulate a given model and be \(10\times\) faster at that. If multiple simulators meet a set of requirements, the ultimate choice should come down to performance (build, setup, and simulation time) and user friendliness. Of course, with everything else being equal, we would prefer the more customizable/user-friendly/etc. simulator.

### 4.1 Simulation Strategy

There are two main simulation strategies for SNNs: **clock-driven** and **event-driven**. Clock-driven (a.k.a synchronous) algorithms revolve around a simulation loop and advance the entire network state in lock-step. Event-driven (a.k.a asynchronous) algorithms revolve around a priority queue and advance individual neurons and synapses on a per-need basis, that is, out of sync. Listings 4.1 and 4.2 contrast both approaches. For simplicity both listings omit delays and plasticity (more on those in Sections 4.4–4.5).

Listing 4.1: Clock-driven simulation loop. We advance all neurons’ dynamics by \(\Delta t\) (lines 4–5), noting which ones spike (lines 6–7). We then notify the neighbors of spiking neurons (lines 9–10).

```plaintext
// DT - time resolution
while true:
    spikes = []
    for n in neurons:
        bool spiked = advance(n, DT);
        if spiked:
            spikes.push(n)
    for n in spikes:
        deliverSpikeToNeighbors(n)
```

Listing 4.2: Event-driven simulation loop. We remove the most recent event from the priority queue (line 4) and set \(\text{NOW}\) to the event’s timestamp (line 5). We then process the event (line 6, see Listing 4.4 for details) which may generate zero or more future events, which we insert back into the priority queue (line 7).

```plaintext
// NOW - current point in time
// queue - chronologically sorted priority queue of events
while not queue.empty():
    e = queue.pop()
    NOW = e.time
    newEvents = process(e)
    queue.insert(newEvents)``
The difference between the two approaches will become more apparent if, instead of the overall simulation loop, we consider the dynamics of an individual neuron. A popular type of neuron that is used as a building block in many models is the Poisson firing neuron—a stateless neuron whose spike train is a rate-modulated Poisson process. Listings 4.3 and 4.4 show how one would implement said neuron under either doctrine.

**Listing 4.3:** Clock-driven Poisson neuron. We poll the neuron every \( DT \) to determine whether it fired.

```java
// DT - time resolution
// r - firing rate
poisson():
  return rand() < DT * r
```

**Listing 4.4:** Event-driven Poisson neuron. We directly calculate the timestamp of our next spike (line 4) and tell the simulator to, at that time, deliver a spike to our neighbors as well as invoke our `poisson()` method so we may calculate our next spike (lines 5–8).

```java
// NOW - current point in time
// r - firing rate
poisson():
  timeOfNextSpike = NOW + exprnd(1/r)
  return [
    {type: spike, time: timeOfNextSpike, neuron: this},
    {type: update, time: timeOfNextSpike, neuron: this}
  ]
```

We can immediately deduce some strengths and weaknesses of both approaches. **Clock-driven algorithms** are forced to quantize time. They can not differentiate between individual events within a single \( DT \). Research in neuroscience suggests that computations in the human brain are sensitive down to \( 10 \mu s \), and that they become chaotic if this resolution is significantly exceeded [23, Fig. 2(D)]. As of this writing, a time step of \( 10 \mu s \) is prohibitively computationally expensive, so most models settle on \( 1 \mu s \). In exchange, clock-driven algorithms are simple to program and offer a lot of parallelism that is easily exploited by GPUs.

**Event-driven algorithms**, on the other hand, practically offer infinite time resolution, making them the only choice for some scientific computations that require this level of precision. They are also theoretically more efficient than clock-driven algorithms: Since they only process events—which can have arbitrarily large pauses between them—event-driven algorithms can “skip ahead” in time while a clock-driven algorithm would be forced to loop over dozens of simulation steps just to determine that no activity has occurred. However, this seldomly applies in practice as most models have negligible neuron simulation costs and fire frequently (at least 1 spike every \( DT \)). In exchange, event-driven algorithms are more complex to program and require dynamics with closed-form solutions (so they can be computed at any point in time). Hodgkin-Huxley neurons [24] are a popular neuron model without closed-form dynamics expressions. In addition, event-driven algorithms offer little parallelism (only one event is processed at a time) and require maintaining a priority queue which does not map well to GPUs. Table 4.1 summarizes the key differences between clock- and event-driven simulators. See Naveros et al. [25] for an in-depth review of both strategies.

**Hybrid algorithms** attempt to combine the strengths of both approaches. Many otherwise clock-driven simulators employ event-driven plasticity for synapses only [26–28]. Furber et al. [29] developed an at its core event-driven algorithm that dynamically groups events together using a sliding (time) window, treating all events inside said window as simultaneous which allows them to be processed in parallel.
4.2 Data Structures

All simulators, regardless of simulation strategy, must store the SNN-induced graph somehow. They use fairly standard data structures such as:

**Adjacency list**: An adjacency list stores a compact list of neighbor IDs for every vertex. It is typically implemented using a compressed sparse row (CSR) representation: the contiguous neighbor lists of all vertices are concatenated; an additional offset table marks the beginning of each neighbor list, as opposed to using actual linked lists of individually allocated nodes.

**Adjacency matrix**: A (binary) adjacency matrix explicitly stores the existence of every potential edge in a $|V| \times |V|$ matrix/array, typically using a single bit to represent each connection. At a network density of ~3% an adjacency matrix starts becoming more memory efficient than an adjacency list (the former uses 1 bit per potential connection, the latter 32 bits per actual connection), promising not only larger networks but also less bandwidth usage. This efficiency is offset by iteration complexity: we now need to parse individual bits rather than simply read whole words. In order to compute synapse indices we need to calculate a costly prefix sum of the set bits.

**Procedural**: Since most SNNs are defined parametrically (Chapter 2) we do not actually need to store the graph at all but only the parameters that define it. We can then generate neighbor indices on the fly during the simulation. This idea was first implemented...

### Table 4.1 Key differences between clock- and event-driven simulators

<table>
<thead>
<tr>
<th></th>
<th>Clock-driven</th>
<th>Event-driven</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Core design</strong></td>
<td>Simulation loop&lt;br&gt;The entire network state is advanced in lockstep.</td>
<td>Priority queue&lt;br&gt;Neurons and synapses are individually advanced on a per-need basis.</td>
</tr>
<tr>
<td><strong>Dynamics</strong></td>
<td>Any</td>
<td>Closed-form only</td>
</tr>
<tr>
<td><strong>Time res.</strong></td>
<td>Quantized</td>
<td>Infinite</td>
</tr>
<tr>
<td><strong>Complexity</strong></td>
<td>$O(\max(1/DT, r \cdot #syn))$&lt;br&gt;Super-linear in activity and size: As DT shrinks, constants start to dominate simulation time.</td>
<td>$O(r \cdot #syn)$&lt;br&gt;Linear in activity and size. Negligible in practice.</td>
</tr>
<tr>
<td><strong>Parallelism</strong></td>
<td>High&lt;br&gt;All dynamics can be advanced and spikes delivered in parallel.</td>
<td>Low&lt;br&gt;Only one event can be processed at a time.</td>
</tr>
<tr>
<td><strong>GPU-suited</strong></td>
<td>High&lt;br&gt;High, SIMD-style parallelism</td>
<td>Low&lt;br&gt;Low parallelism, different event types require different processing, priority queue requires sorting/merging.</td>
</tr>
</tbody>
</table>

*DT* - time step<br>*r* - firing rate<br>*\#syn* - no. of synapses
by Knight et al. [30]. In their experiments, procedural connectivity performed on par with physical connectivity but allowed for much larger models. The authors reason that the extra computations and missing memory accesses cancel out.

Neurons and synapses are typically stored using a simple struct of arrays (SoA) representation to enable memory coalescing on GPUs (Section 6.3). Dynamic buffers (e.g., spike buffers) are typically conservatively allocated arrays (so as to avoid re-allocations) combined with atomic counters used to write to them.

### 4.3 (Non-)instantaneous Synaptic Interactions

(Non-)instantaneous synaptic interactions (not to be confused with delayed synaptic connections (Section 4.4)) define at what times a neuron may fire. In the case of instantaneous interactions, a neuron may only fire at times of incoming spikes. In the case of non-instantaneous interactions, a neuron can fire regardless of incoming spikes (e.g., due to a network-wide background current). Clock-driven simulators trivially support both interaction types: the `advance()` method, invoked on every neuron at each time step, determines whether said neuron spiked, which it can base on arbitrary conditions. On the other hand, non-instantaneous connections greatly complicate the design of event-driven simulators: the priority queue now has to support deletion and update in addition to extraction and insertion. The simulation loop becomes more complicated as well. This added complexity is why not all event-driven simulators (choose to) support non-instantaneous interactions.

### 4.4 Delays

For simplicity, I ignored transmission delays so far. However, they are not hard to implement. In the case of event-driven simulators it is trivial: simply add the delay \( d \) to `time0fNextSpike`, i.e. line 4 in Listing 4.4 would turn into:

\[
\text{time0fNextSpike} = \text{NOW} + \text{exprnd}(1/r) + d
\]

In the case of clock-driven simulators one has to differentiate between global and per-synapse delays. Global delays require that every spike be delayed by the same amount. Thus it is sufficient to store delay many spike arrays in a ring buffer. At time \( t \) spikes from slot \( t\%\text{delay} \) (which have been emitted at time \( t - \text{delay} \)) are delivered and then overridden with newly emitted spikes (which in turn will be delivered another delay steps from now). In the case of per-synapse delays we can take advantage of the quantized nature of our simulator which implies that there be only a finite number of distinct delays in the entire network. The graph \( G \) can therefore be split into \( \text{max}(\text{delays}) \) many sub graphs \( G_d \) that only consist of synapses with delay \( d \). Each of these sub graphs is an intact SNN with only a global delay, which we already know how to simulate. Delays in Spice are implemented almost verbatim (Section 7.2).
4.5 Plasticity

Plasticity refers to synapses undergoing dynamics—neurons are always plastic. ² An event-driven simulator will typically update synapses as part of processing events—before a spike is delivered, the dynamics of synapses over which it will be delivered are advanced. Plasticity can also trivially be implemented in clock-driven simulators: Simply update every synapse at each simulation step, together with neurons. However, this naïve implementation is so slow that nobody uses it. Due to the high cost of the synapse update (derived from their sheer number combined with the high computational cost of their dynamics) many clock-driven simulators optimize this step by employing event-driven techniques, such as only updating synapses upon transmitting spikes. Plasticity will be covered in depth in Section 7.4, when studying its implementation and optimization inside Spice.

4.6 Customizability

Most simulators allow control over minor details such as the machine word precision (through compilation flags or runtime parameters). This section refers to the ability and level of control over specifying custom models. Apart from proof of concept papers which may hard-code an entire model [31], most simulators allow the specification of custom models, albeit to varying degrees:

Some simulators [27, 32] opt to hard-code neuron and synapse types which become building blocks that the user can compose into models. They try to account for all potential use cases by parameterizing every conceivable aspect of their neurons/synapses. Having full control over the code can make it simpler and allow for special optimizations compared to a more generic simulator. On the other hand, if a particular neuron/synapse type is not supported or does not behave exactly as desired, the user is at a loss and has to wait on the developers to provide it.

Other simulators [26, 33] allow the user to specify custom dynamics expressions. However, they still make fundamental assumptions about SNNs, such as potentials, firing thresholds, or refractory periods. Within those assumptions the user gets control over, say, the update and reset logic.

To my knowledge, Spice [34] is the only completely generic simulator, making no such assumptions and allowing the user to specify arbitrary dynamics in the host language (C++) itself via callbacks. This makes it more akin to a general graph processing/message passing framework, allowing not only the implementation of SNN models but also of algorithms such as SSSP with relative ease. This generality does not come at the cost of performance or usability. It does put more responsibility on the user, who now has to avoid data races.
Listing 4.5: Excerpt of NeuronGPU’s [32] semi-procedural Python API. We create two neuron populations (lines 1–3), parameterize them (lines 5–7), and connect them using various connectivity types and synapse parameters (lines 9–12). Finally we run the simulation (line 14).

```python
neuron = ngpu.Create("iaf.psc.exp.g", N, 2)
E = neuron[0:NE] # excitatory neurons
I = neuron[NE:N] # inhibitory neurons
ngpu.SetStatus(
    neuron,
    {"V_m_rel": -60, "V_reset_rel": -60, "Theta_rel": -50, ...})
ngpu.Connect(
    E, E,
    {"rule": "fixed_total_number", "total_num": NE * NE // 50},
    {"weight": Wex, "delay": 0.8, "receptor": 0})
ngpu.Simulate()
```

Listing 4.6: Excerpt of GeNN’s [26] declarative API. We define a leaky integrate and fire style neuron LIF by inheriting from a base class (line 1). We then specify (part of) its behavior by passing strings written in a DSL to macros (lines 6–14). The rest of the code, dealing with instantiating neuron populations and connecting them, is very similiary in nature to Listing 4.5.

```cpp
class LIF : public NeuronModels::Base
{
public:
    DECLARE_MODEL(LIF, 6, 2);

    SET_SIM_CODE(
        "if ($(RefracTime) <= 0.0) {
        $(V) += DT / $(TauM) * ($(Vrest)-$(V)) + $(Isyn);
        } else {
        $(RefracTime) -= DT;
        }\n    ");

    SET_THRESHOLD_CONDITION_CODE(
        "$(RefracTime) <= 0.0 && $(V) >= $(Vthresh)";
    );
};
```

Listing 4.7: Excerpt of Spice’s [34] API. We define a neuron type LIF by inheriting from a templated base class and communicating the number and types of our neuron’s attributes to it (line 1). We specify its behavior in native C++ by overriding a series of callbacks (lines 3–5). We then create a SNN with a DT of 100 µs and a global delay of 15 (line 8). We then create two neuron populations A and B of 20 and 80 neurons respectively (lines 10–11), and connect A to B with a probability of 20% (line 13) while B is intra-connected with a probability of 10% (line 14). Finally, we run our simulation (line 16).

```cpp
class LIF : neuron<float, int>
{
    void init(...) override {...}
    bool advance(...) override {...}
    void receiveSpike(...) override {...}
};
snn net(0.0001, 15);
auto A = net.add<LIF>(20, ...);
auto B = net.add<LIF>(80, ...);
net.connect<STATIC>({A, B, fixed_probability(0.2), ...});
net.connect<STATIC>({B, B, fixed_probability(0.1), ...});
while (true) net.step();
```
4.7 Application Programming Interface (API)

C++ is the language of choice for most SNN simulators due to its performance and portability. Simulators typically expose their C++ interface directly to the user. Many also provide Python bindings and abstraction layers for simplicity and to reach a broader audience. The API ties in closely with customizability as it controls how the latter is exposed to the user. While certain designs seem to prefer certain APIs in practice, these ultimately remain orthogonal choices.

Procedural APIs are among the most popular ones [27, 32, 35–37]. They start with an empty SNN and, through a series of method calls, set parameters, create neuron and synapse populations and connect them—very similar to the state machine nature of OpenGL. These APIs can be mixed with varying degrees of object orientation. In all cases, at least the top level SNN is its own object so multiple independent instances can be created and RAII utilized for resource management. This style API is popular with non-generic simulators.

Generic simulators tend more towards declarative APIs [26, 34], especially for the task of specifying dynamics. Most generic simulators employ simple, custom domain-specific languages (DSLs) which can be transpiled to C++ via simple string replacement/regex operations. This obfuscates code generation, debilitates debuggers and profilers, and requires separate, proprietary compilation steps. The use of DSLs stems from the age of most simulators which were initially written at a time when such features could not easily be expressed in native C++. Once again to my knowledge, Spice is the only generic simulator written entirely in C++(17). Listings 4.5–4.7 contrast various API designs.

4.8 Biological Fidelity

The next design consideration is the biological fidelity of a simulator. Some simulators are merely based on the general principle of spiking neurons while others strive to model the electrochemical processes inside biological neurons (or the electrical processes inside capacitors—which some neurons are modelled after) with as much accuracy as possible. Only event-driven simulators can be truly biologically accurate as only they can simulate spikes with exact timing, at the aforementioned cost of not being able to simulate all neuron types. Since clock-driven simulators quantize time they are approximate by design, even when dynamics are computed exactly. However, even within clock-driven simulators, many different levels of biological accuracy can be achieved. They are most impacted by the following design choices:

1. The machine word precision (float/double/etc.) at which neuron/synapse attributes are stored and time is represented.
2. The integration method (Euler/Runge-Kutta/etc.) used to advance neuron/synapse dynamics.
3. The time keeping mechanism (floating point arithmetic with or without Kahan summation/integer arithmetic/etc.).

3 Many users of SNN simulation software are researchers themselves, albeit from different disciplines such as medicine, psychology, biology, or neuroscience. They might not possess the technical skills required to build and interface with a C++ library, but are competent with scripting in Python.

4 Models are specified in C++, compiled by and adhering to the same rules as the tool-chain used for the simulator itself. This has wide-reaching benefits regarding design patterns, code reusability, compilation, debugging, and profiling. More in Chapter 9.

5 In principle, it is possible to simulate a network exactly in a clock-driven fashion. Morrison et al. [38] present an algorithm revolving around a synchronous update loop that can simulate SNNs exactly, so long as the minimum delay in the network is larger than the time step. However, its design borrows heavily from event-driven simulators, using per-neuron event queues for example, making it more of a hybrid algorithm in my opinion.
4.9 Hardware

Many (mostly older) simulators target (not necessarily exclusively) CPUs for historical and compatibility reasons. They might have been authored before general-purpose GPU programming became ubiquitous or they want to take advantage of (predominantly) CPU-based supercomputers, of which there are still many economical ones around. However, modern simulators should no longer specifically and exclusively target CPUs as they have been overtaken by GPUs long ago, both in terms of absolute and per-dollar performance. To provide a point of reference: a single Tesla V100 rivals a 2048-core supercomputer instance from eight years ago. Even compared with modern CPUs, GPUs are $5\times\sim10\times$ faster per dollar. 

The speed and efficiency advantage combined with widespread availability and relative affordability makes GPUs the most popular target platform for SNN simulators. Even if some computational primitives do not map perfectly to GPU architectures, this is more than made up for by their massive parallelism. Algorithmic and implementation improvements in the order of $2\times$/year continue to be made and features such as Tensor Cores remain completely unutilized. In addition, SNNs are currently simulated at 1% of the GPUs theoretical maximum FLOPS. Furthermore, SNNs only very recently have been scaled to multiple GPUs [39] within a single node—they still need to be scaled to multi-node clusters. This indicates that GPU-based simulators will continue to scale and that it will take some time before we reach their full potential.

So-called neuromorphic hardware models SNNs directly in circuitry instead of emulating them with microcode running on a general-purpose processor. Early exemplars date back to 2006 [40] but only recently reached maturity and popularity with IBM’s TrueNorth [41]. It features a programmable field array which can represent a SNN of up to 1M neurons each with a maximum in/out-degree of 256. Spikes are transmitted by time-multiplexed wires enabling the modeling of delays. It has an average power density of $20 \text{ mW/cm}^2$. ROLLS [42] pursues a different route—it uses analog circuitry to capture the physics of biological neurons and synapses. For example, it simulates LIF neurons (which are modeled after capacitors) using actual capacitors. Neuromorphic hardware is already more power-efficient than GPUs but still lacks behind their absolute performance (density). However, given its scalable 2D design, this might be subject to change in the future. For a deeper dive into neuromorphic hardware I recommend the review by Nawrocki et al. [43].

---

6 I compared the performance of a Tesla V100 and an Intel Xeon E5-2699 v3. Pricing was taken from the Google Cloud Platform (preemptible prices). The difference would have been even more stark had we compared the Tesla T4, which is $2.5\times$ faster per dollar than the V100.

7 Analog circuitry has disadvantages, too. Differences in manufacturing or temperature fluctuations can impact the results of computations and thus simulations. There is a limit to how small analog circuitry can be shrunk. Digital circuits can be simulated exactly, making it possible to design and test an entire chip in software before even building the first prototype.
Related Work

Attempts to mathematically model the behavior of biological neurons date as far back as 1907 [44]. Research into SNNs as we know them today picked up in the 90s. Maass notably proved in 1996 [1] that SNNs are fundamentally more powerful computationally than conventional ANNs and subsequently formalized them for the first time in 1997 [45], dubbing them “the third generation of neural network models.” While GPGPUs popularized SNNs, they did so with some delay and to a lesser effect compared to ANNs. Most recently, neuromorphic hardware [43] opened a new era of SNN research and applications.

SNNs can be classified according to several key traits outlined in Chapter 4. We want to organize this chapter by the first trait simulation strategy since it is the most discriminating one and implies a fundamentally different design with major trade-offs, while the others can be viewed as features shared by all simulators (to various degrees).

5.1 Clock-driven Simulators

Clock-driven simulators quantize time and advance the whole simulation in lockstep, taking neurons and synapses from state \( t \) to state \( t + dt \). Likewise, spikes can only be generated at those discrete intervals and synaptic delays must be multiples of \( dt \). Clock-driven simulators can simulate all neuron and synapse types and map well to GPUs, resulting in very high-performance implementations. They pay for this with limited biological accuracy. See Section 4.1 for a detailed explanation of clock-driven simulators.

Nest [36] is one of the oldest and most established simulators. It is incredibly feature rich. Supporting dozens of neuron/synapse types and exact integration methods, it serves as a reference implementation for evaluating new simulators. Performance is not a priority for Nest.

GeNN [26] is a current, high-performance simulator accompanied by several publications. It is actively maintained and developed, constantly pushing the state of the art in terms of performance.

Brian [33] goes a different route. It aims to maximize ease of use before anything else, making it the prototyping tool of choice for many, especially users with little to no coding skills. Listing 5.1 shows its unique API. Being written in Python it is very slow. A cross-compiler [46] to GeNN exists, but not all dynamics are supported.

NeuronGPU [32] is a very recent simulator striving for maximum biological accuracy and feature-richness without sacrificing too much performance.

5.2 Event-driven Simulators

Event-driven simulators only advance neuron/synapse dynamics on a per-need basis when those are involved in the emission, transmission, or reception of spikes. They typically offer arbitrary time resolution and are biologically exact, making them the only choice for some scientific computations requiring this level of precision. In exchange they cannot simulate dynamics without closed-form solutions and are hard to map to GPUs efficiently. Thus, they are typically outperformed by clock-driven simulators, although they do excel in some models, especially low-activity ones where they can “skip ahead” in time while a clock-driven simulator is forced to loop. See Section 4.1 for a detailed explanation of event-driven simulators.

Some event-driven simulators [47, 48] still choose to quantize time because it simplifies code. Biological accuracy does not suffer since they can simply pick a very small time step (such as 1 ns) with close to no impact on performance. While CPU- and GPU-based event-driven simulators were very actively researched until ~2015, these efforts have been almost completely abandoned in favor of neuromorphic hardware which is predominantly event-driven (Section 5.4).

Two works that stand out in this space are the simulators by Mattia & Giudice [49] and FNS [50]. The former marks a milestone in SNN research as it was the first fully-fledged (event-driven) SNN simulation framework whereas prior work was mostly of a prototypical nature. The latter breaks with past year’s trend of moving to neuromorphic hardware by targeting GPUs, in 2021. The project is still in its infancy but has already produced promising results.

There are almost no pure clock- or event-driven simulators. Rather, they borrow design principles from each other where it makes sense, a trend already recognized in 2014 by D’Haene et al. [51] in their great meta paper in which they formulate general properties of a hypothetical hybrid simulator. Such simulators attempt to combine the strenghts of both clock- and event-driven simulators. For example, Furber et al. [29] developed an at its core event-driven simulator which however bins events together using a sliding time window. All events in the same bin are treated as simultaneous, allowing them to be processed in parallel. On the other hand, Naveros et al. [52] split the network according to activity: low-activity areas are simulated in an event-driven fashion on the CPU while high-activity areas are concurrently simulated in a clock-driven fashion on the GPU.

5.3 Parallel Simulators

Parallel simulators—ones that can distribute a single network across multiple (potentially heterogenous) processors in order to speed up their simulation or enable the simulation of larger instances—deserve their own section. Parallelization of SNNs is a non-trivial task if the history of multi-GPU-Backpropagation in ANNs is any indicator. From the Big Bang of Deep Learning in 2009, brought about by the advent of GPGPU hardware, it took ten years before PipeDream [53] scaled...
### Clock-driven sim.

<table>
<thead>
<tr>
<th>Simulato</th>
<th>Biologica</th>
<th>Custom.</th>
<th>Delays</th>
<th>CPU</th>
<th>GPU</th>
<th>Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nest [36, 37]</td>
<td>✓ ✓ ✓</td>
<td>✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>CARLSim [54, 55]</td>
<td>✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>GeNN [26, 30, 56]</td>
<td>✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>Brian [33, 46]</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>Spike [27]</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td><strong>Spice</strong> [28, 34, 39]</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>BSim [35]</td>
<td>✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>NeuronGPU [32]</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
</tr>
</tbody>
</table>

### Event-driven sim.

<table>
<thead>
<tr>
<th>Simulato</th>
<th>Non-inst. interact.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mattia et al. [49]</td>
<td>✓² ✓ ✓ ✓</td>
</tr>
<tr>
<td>SpikeNET³ [47]</td>
<td>✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>DAMNED³ [48, 57]</td>
<td>✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>NEVESIM [58]</td>
<td>✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>FNS [50]</td>
<td>✓ ✓ ✓ ✓ ✓</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Biological accuracy</td>
<td>✓ general principle of spiking neurons ✓ ✓ double precision/higher order integration/spikes as currents ✓ ✓ ✓ ✓ exact (as long as the minimum delay is greater than the time step)</td>
</tr>
<tr>
<td>Customizability</td>
<td>✓ parameterized neurons/synapses ✓ ✓ dynamics partially programmable ✓ ✓ ✓ ✓ dynamics fully programmable</td>
</tr>
<tr>
<td>Delays</td>
<td>✓ per network/synapse group ✓ ✓ per synapse</td>
</tr>
<tr>
<td>Parallel</td>
<td>✓ can distribute network over multiple processors</td>
</tr>
</tbody>
</table>

### Table 5.1 Famous clock- and event-driven simulators in chronological order of first appearance. All clock-driven simulators support non-instantaneous synaptic interactions. Conversely, virtually all event-driven simulators are biologically exact.

Backpropagation (BP) to eight GPUs (within a single node), achieving north of a 7.5x speedup. Scaling BP to clusters of nodes remains an active research field.

Attempts to harvest the computational power of multiple GPUs have been presented as early as 2010. In his thesis Krishnamani [31] notably utilized not only multiple GPUs but also the CPU and derived a spike synchronization algorithm that in variations is still used by multi-GPU simulators today. Apart from that, it was mostly a proof of concept work with hard-coded models.

Also notable is the work by Kunkel et al. [59] who scaled Nest to petascale supercomputers comprising some 100K cores with an efficiency of ~50%.² Its simple parallelization and static load balancing schemes greatly inspired Spice [39] which, running on a single V100, roughly matches the performance of a 2048-core instance or running on eight V100s, that of a 32K-core instance due to better scaling.³

CARLSim [55] is one of the few multi-GPU simulators. It requires massive network sizes to break even where it achieves ~50% efficiency on 4 GPUs [55, Fig. 7]. BSim [35] is another recent addition to the space of multi-GPU simulators. It benefits from multiple GPUs even for smaller network sizes, but still only achieves an efficiency of ~55% on 4 GPUs [35, Fig. 7]. Further, CARLSim cannot create larger models on more GPUs due to using a data-parallel work distribution scheme [60, Section 1.2.4.2]—it can only speed up existing models.

Spice (/spaik/), the simulator proposed and developed in this thesis, is currently the fastest clock-driven, GPU-based simulator both in terms of

---

² The baseline being a 512-core instance which already suffers from synchronization overhead
³ Based on extrapolated benchmarks
Table 5.2 Recent neuromorphic SNN simulators, grouped by commonalities and spread across two rows for space reasons. Would-be repeated values are left blank instead. Footnotes only concern the cell they occur in. The last column measures the average energy spent per sample per pixel in a classification task. It should be taken with a grain of salt. Not all simulators classified the same dataset. Even if they did, tasks efficiently processed by one architecture may be unsuitable for another. Only by comparing many benchmarks one can obtain a holistic picture of a particular simulator, including its trade-offs. Vineyard et al. [67] present pitfalls and challenges in benchmarking neuromorphic architectures. Overall, this table should be treated as an overview, not a comparison.

<table>
<thead>
<tr>
<th>Simulator</th>
<th>Hardware</th>
<th>Architecture</th>
<th>Neuron type</th>
<th>Synapse type</th>
<th>Learning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TrueNorth [41]</td>
<td>ASIC</td>
<td>event-driven</td>
<td>programmable</td>
<td>LTP, LTD, STP</td>
<td></td>
</tr>
<tr>
<td>ROLLS [42]</td>
<td></td>
<td>(L)IF</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>µBrain [62]</td>
<td></td>
<td>static</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zhang et al. [63]</td>
<td>FPGA</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fang et al. [64]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Li et al. [65]</td>
<td>hybrid</td>
<td>+[24, 66]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Simulator</th>
<th>Topology</th>
<th>Max neurons/synapses</th>
<th>Weight prec. (bit)</th>
<th>Area (mm²)</th>
<th>Energy (nj/px)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TrueNorth</td>
<td>flexible</td>
<td>1M / 256M</td>
<td>5,6,7 any</td>
<td>430.0</td>
<td>21.87</td>
</tr>
<tr>
<td>ROLLS</td>
<td></td>
<td>256 / 128K</td>
<td>n/a</td>
<td>51.4</td>
<td>-</td>
</tr>
<tr>
<td>µBrain</td>
<td>layers 4</td>
<td>256x64x16</td>
<td>6,7 4</td>
<td>2.8</td>
<td>1.33</td>
</tr>
<tr>
<td>Zhang et al.</td>
<td></td>
<td>784x512x384x10</td>
<td>7 8</td>
<td>1.6</td>
<td>-</td>
</tr>
<tr>
<td>Fang et al.</td>
<td></td>
<td>784x600x10</td>
<td>6 16</td>
<td>-</td>
<td>0.92</td>
</tr>
<tr>
<td>Li et al.</td>
<td></td>
<td>784x200x100x10</td>
<td>6 16</td>
<td>1.6</td>
<td>6430.00</td>
</tr>
</tbody>
</table>

1: no global clock  
2: computers responsible for learning  
3: with 30% lateral connections  
5: within memory limits  
6: floating point  
7: fixed point

4: Allowing not only the implementation of arbitrary models but also of graph algorithms such as SSSP

Simulation and setup time [28], according to our benchmarks (Chapter 10). It can take advantage of multiple GPUs with an efficiency of 100%, allowing the creation of 8x larger models and simulating them 8x faster on 8 GPUs [39]. Beyond that it is fully programmable and offers a modern, intuitive, and concise API written in C++.

Table 5.1 classifies relevant clock- and event-driven simulators along the categories discussed in Chapter 4: simulation strategy, biological accuracy, customizability, delays, and hardware.

5.4 Neuromorphic Simulators

Neuromorphic simulators are application-specific integrated circuits (ASICs) optimized for SNNs. Whereas a general purpose processor (CPU/GPGPU) is forced to simulate SNNs using arithmetic instructions and memory reads/writes, ASICs may replicate some or all SNN processes directly in circuitry. Transmitting a spike for example may be as simple as applying an electric current to a lane on a neuromorphic chip. This is a trade-off—generality is sacrificed for performance and efficiency.
Neuromorphic chips are still in their infancy. Most are research prototypes implemented on FPGAs with some custom, industry-grade, brand-name products slowly appearing as well. Even so, they have already surpassed CPUs and GPGPUs in terms of efficiency (energy per event) and power density (power consumption per area) by orders of magnitude. As a result, research into event-driven simulators has moved almost exclusively onto them. They still lack behind in terms of absolute performance (density) though. For details please see Section 4.9.

Two recent simulators that stand out are IBM’s TrueNorth [41] and ROLLS [42]. TrueNorth (Figure 5.1) stands out for its programmability. It consists of 4096 cores each of which represents a self-contained SNN with 256 axons freely connected to 256 neurons using up to 256² synapses. The cores are hierarchically interconnected. Each neuron on every core can target an axon on any other core. While its energy consumption is significantly higher than that of other simulators, it is still orders of magnitude below that of CPUs/GPGPUs. In exchange, it can simulate almost any model with arbitrary topology.

ROLLS stands out because it uses analog hardware. Take a LIF neuron for example, which is modelled after a capacitor. Whereas digital chips would simulate said neuron through several gate arrays and memory cells, ROLLS uses a single capacitor. This greatly reduces the number of components and simplifies design. ROLLS does not require any clock—the simulation is continuous and driven simply by applying a current. In exchange it is less generic, being able to simulate only LIF neurons. It is also susceptible to minute differences in the manufacturing process and operational conditions which could negatively affect simulation accuracy and repeatability.

Table 5.2 classifies recent neuromorphic simulators according to several key properties.

Figure 5.1 TrueNorth chip design. (a) The basic building block is a core, a self-contained neural network with 256 inputs (axons) and 256 outputs (neurons), connected via 256² directed, programmable synapses. (b) Cores are inter-connected by a mesh routing network. Each neuron on every core can target an axon on any other core. Cores operate in an event-driven fashion (a core is only powered when it receives a spike). Cassidy et al. [168, Fig. 11]

5 However, analog hardware tends to be larger than digital one
6 Which can be parameterized though (within limits)
This chapter aims to equip the reader with enough background knowledge on GPGPU programming in order for them to fully appreciate subsequent chapters. It gives a basic overview over GPGPU programming and then covers the most important performance considerations, to help understand why Spice’s optimizations are so effective on GPUs and where its major speedups are coming from. For a complete guide on the matter, I recommend Nvidia’s CUDA C++ Programming Guide. Another great resource is Nvidia’s developer blog with light-weight, self-enclosed stories covering all the topics discussed in this chapter.

We will use CUDA terminology throughout, but the concepts conveyed here are equally applicable to OpenCL or Vulkan for example. Likewise, GPU architecture is discussed on such a high level that it equally applies to both Nvidia and AMD GPUs, even though we only mention Nvidia.

### 6.1 Overall Architecture

![High-level overview of CPU](image)

A CPU (Figure 6.1(left)) roughly consists of multiple levels of cache and multiple cores. A core is equipped with

- its own, private L1 cache;
- a control unit responsible for program execution, including a program counter (PC), stack pointer (SP), instruction decoder, memory controller and prefetcher, branch unit and predictor, speculative execution unit, etc.;
- units responsible for the actual computations: general purpose registers to hold operands, arithmetic logic unit (ALU), floating point unit (FP), integer unit, vector registers and instructions (SIMD), etc.

Apart from the shared resources (L2 cache, RAM, and peripherals such as the PCIe bus, network adapter, etc.) each core forms an independent compute unit capable of executing its own program, in parallel—one core might be running a spreadsheet software while another is simultaneously running a music player. In addition to running separate programs (processes) we can run a single process cooperatively on multiple threads,
which can map to different hardware cores. Each thread could be executing its own independent task (for example, running the UI versus the business logic) or they could be cooperatively working on a single task. This is known as simultaneous multithreading (SMT).

In addition, CPU cores have vector registers and instructions allowing them to perform the same (arithmetic) operation on a fixed, small number of words in parallel. For example, we can add two 128-bit registers together not treating them as a single 128-bit integer, but as four independent 32-bit integers. This vector instruction completes in the same number of cycles as one operating on a single word, dramatically increasing throughput. This paradigm, known as single instruction, multiple data (SIMD), is often used to implement linear algebra routines.

The closest thing to a CPU core on the GPU (Figure 6.1(right)) is the streaming multiprocessor (SM). Like a CPU core it is equipped with its own L1 cache and control unit but unlike its CPU equivalent it lacks any computational units. For that purpose it commands “dumb” cores which are only equipped with compute units and nothing else. In other words, the SM handles program flow but delegates actual computations to its 32 cores. This makes GPUs a hybrid architecture, somewhere in-between SIMD and SMT. Nvidia calls this “single instruction, multiple threads” (SIMT).

6.2 SIMD versus SIMT versus SMT

By studying the differences between these architectures while already being familiar with one of them, we will also develop a better understanding of the other two. The best way to do so is via code examples. Take the task of adding two arrays of numbers together. This problem can be expressed in various dialects.

In a procedural dialect we would write:

```c
for (int i = 0; i < n; i++) a[i] = b[i] + c[i];
```

The SIMD dialect uses so-called intrinsics:

```c
for (int i = 0; i < n; i += 4) {
__m128 x = _mm_load_ps(&a[i]); // x = a[i..i+3]
__m128 y = _mm_load_ps(&b[i]); // y = b[i..i+3]
__m128 z = _mm_add_ps(x, y);  // z = x + y
__mm_store_ps(&c[i], z);     // c[i..i+3] = z
}
```

CUDA uses a SIMT dialect (Listing 6.1). All CUDA threads run the same instructions at the same time, but compute different values. Groups of 32 threads (so-called warps) will be scheduled to the 32 physical cores of a SM.

```c
__global__ void add(int *a, int *b, int *c) {
 int i = threadIdx.x + blockIdx.x * blockDim.x;
 a[i] = b[i] + c[i];
}
add<<<n/128, 128>>>(...);
```

---

1 Only very recently, CUDA cores got equipped with their own PC (previously they shared their SM’s PC). This allows them to keep track of where they are in a program’s execution and thus advance independently, enabling the implementation of some algorithms that previously lead to deadlock. From a performance standpoint, we still want to avoid such “divergence” (Section 6.5) as much as possible.

Listing 6.1: `__global__` marks `add()` as a thread entry point, similar to `main()` on CPUs. Notice the lack of a loop in the function body. Instead, line 6 launches `add()` with \( n / 128 \) blocks à 128 threads each (i.e., \( n \) threads total). Each thread computes its own, unique array index (line 3) and adds one array element (line 4).
Both SIMD and SIMT are forms of data-parallelism; they broadcast the same instruction to different compute units. This is efficient from a hardware design standpoint. The compute units are replicated but share the same control unit—this is how GPUs can fit thousands of “cores” while CPUs cap out at dozens. However, SIMD is more efficient than SIMT. In the example above, the SIMD version stores the array pointers a, b, c, and the loop counter i only once using one register each. In the SIMT version, every thread stores local copies of these (in registers). Similarly, in the SIMD version the loop counter is only incremented once while in the SIMT version every thread redundantly computes $i = \ldots$. So what is the point of SIMT then if it is less efficient? The fact that it is more powerful! Let us add a simple indirection to our add arrays-code, such as would be caused by a lookup table (LUT) for example:

```c
__global__ void add(int *a, int *b, int *c, int *LUT) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    a[i] = b[LUT[i]] + c[i];
}
```

The above semantics cannot be expressed in SIMD. Neither can the semantics below:

```c
__global__ void add(int *a, int *b, int *c) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if (b[i] % 2)
        a[i] = b[i] + c[i];
    else
        a[i] = b[i] - c[i];
}
```

Compared to SIMD, SIMT is more flexible, each thread:

- has its own set of independent registers;
- can operate on different memory addresses from other threads;
- can follow a different flow path (within the same stack frame) from other threads.

However, it is nothing compared to the flexibility of SMT where each CPU core can efficiently run a completely unrelated program from the other cores. Overall, SIMD is less flexible than SIMT, which is less flexible than SMT. Less flexible architectures are generally more efficient, unless their lack of flexibility prevents us from employing them for certain tasks altogether.

For further reading please see:

- The CUDA Programming Model [69]
- SIMD < SIMT < SMT [70]

### 6.3 Memory Coalescing

Main/system memory (RAM or VRAM, also known as global memory in CUDA) has a pretty high latency (the time delay from requesting a word to it arriving at the processor), some $400–600$ cycles on GPUs. 2

To make main memory accesses more efficient, both CPUs and GPUs access it in **transactions** (TAs), 64 B on the CPU and 128 B on the GPU. That is, the smallest requestable amount of memory on the GPU is a

---

2 Nvidia does not publish these numbers. However, Wong et al. did some reverse-engineering through microbenchmarking [71]. All numbers throughout this chapter are quoted/derived from their paper.
struct Point {
    float x;
    float y;
} points[4];

__global__ void mirror() {
    int i = ...; // as before
    points[i].x *= -1;
    points[i].y *= -1;
}

Figure 6.2 Array of structs (AoS) layout left versus struct of arrays (SoA) layout right and resulting memory access patterns.

contiguous block of 128 B. Even if only a single thread out of a warp requests a single (4-B) word, say from address 0x88, the hardware would still transmit all bytes from 0x80–0xff. We pay the same price no matter how many of the transmitted bytes we actually end up using. Thus, in order to maximize bandwidth, we want our code to result in as few TAs as possible. The GPU contains dedicated decision-making circuitry to split a memory request (by a warp) in as few TAs as possible and thus improve bandwidth. The act of minimizing the number of TAs is known as coalescing. Memory requests that are known to result in a single 128-B TA and completely utilize all its bytes, are known as (fully) coalesced memory accesses.

Some problems, like our add arrays-example can be trivially coalesced: 32 threads cooperatively read 32 consecutive 4-B words of memory. Other times, the programmer can re-structure their algorithm or data structure to turn non-coalesced into coalesced accesses. One popular way to do so on GPUs is to use the struct of arrays (SoA) layout. Take a simple data type such as a 2D-point for example. Reading a point requires two separate single-word reads of the x and y component respectively. If we were to store many of those points using an array of structs (AoS) layout and read consecutive points from different threads, the result would be a non-coalesced access, halving our effective bandwidth (Figure 6.2(left)). By using the SoA layout instead, we alleviate this problem (Figure 6.2(right)).

Many problems inherently generate random memory accesses which cannot be re-ordered and, thus, no or only poor coalescing can be achieved. SNN simulation is one of them: neurons are typically randomly inter-connected and fire in an unpredictable pattern. Additionally SNNs tend to be sparse. Suppose a density of 5%. A firing neuron’s spike would need to be delivered to its randomly distributed neighbors with an average distance of 20 neurons between them. Theoretically, our effective bandwidth would drop by 20× in this scenario. In practice, our performance drop depends on the cache.

For further reading please see:

▶ How to access global memory efficiently [72]
6.4 Cache

As a rule of thumb, the bigger memory gets, the slower it becomes (both in terms of latency and bandwidth). Just think of the performance characteristics of the following: mechanical HDDs, SSDs RAM, L3 cache, L2 cache, L1 cache, registers. Combined with the insight that we do not need to access all of our data all the time, memory hierarchies form a popular (hardware) optimization strategy. GPUs use them just like CPUs.

The idea of a cache is to buffer or cache many small, inefficient TAs and afterwards flush itself to main memory using few, large, and thus efficient TAs. The cache does operate with TAs and prefers to be accessed in a coalesced fashion, just like main memory. But even without coalesced memory accesses we still benefit from the order of magnitude higher speed of the cache compared to main memory (Figure 6.3).

This opens up new optimization avenues for programmers. Even if a given algorithm is not amenable to coalesced memory accesses, it could be possible to concentrate its memory accesses to a small region (at a time), maximizing the probability that they fit into and thus better utilize the cache (as done by Spice, Section 7.3).

In contrast to CPUs, GPUs allow the programmer to manually decide the placement of data between cache (a.k.a “shared memory”) and main memory. In CUDA, this is achieved by simply declaring a variable as __shared__. This makes cache hierarchies on GPUs so much more potent than on CPUs. On CPUs, the programmer can but write their program in as cache-friendly a manner as possible but then must hope that the CPU’s heuristics will lead to good cache utilizations. On the GPU however, as long as the requirements are met, the programmer can guarantee that certain data be placed in cache. Spice uses this mechanism to achieve its superior performance on static models (Section 7.3).

For further reading please see:

- Using shared memory in CUDA [73]
- An Efficient Matrix Transpose in CUDA [74]

6.5 Divergence

Divergence is of little concern to Spice but shall briefly be mentioned for completeness. Divergence is any condition that forces the SM to suspend some of a warp’s threads (and with them the cores they run on), resulting in reduced parallelism (cores are idling when they could be performing useful computations). Since the beginning of CUDA the most frequent source of divergence were branches such as the one in the example below:

```c
__global__ void add(int * a, int * b, int * c) {
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  if (b[i] % 2)
    a[i] = b[i] + c[i];
  else
    a[i] = b[i] - c[i];
}
```

Figure 6.3 Code snippet top and resulting memory access pattern bottom. Every four consecutive writes have the same color. Assume a TA-size of 4 words. Without a cache our effective bandwidth drops to $1 \div 4 = 25\%$: Writes are spread so far apart that each word requires its own TA. Now assume there is a cache large enough to buffer all our writes and is $10\times$ faster than main memory. Our effective bandwidth becomes $1 \div (4 \div 10 + 1) = 71\%$. (In practice more factors come into play.)

4 A GPU’s L1 cache, if accessed in a coalesced fashion, provides the same bandwidth as its registers!

5 Shared memory and (L1) cache used to be separate, fixed-size resources (represented by physically distinct memory chips on the graphics board). Nowadays both resources reside in L1 cache and the distinction between them is a logical one. Additionally, the programmer can control the amount of shared memory versus the amount of general-purpose memory.

For further reading please see:

▶ Using shared memory in CUDA [73]
▶ An Efficient Matrix Transpose in CUDA [74]
Because threads used to share their SM’s PC in the past, they were not able to execute different branches simultaneously. Rather, the SM was forced to execute the if-part while suspending all threads for which the condition evaluated to \texttt{false} and vice versa, effectively halving throughput. Recently, the Volta architecture relaxed this by equipping every SM core with its own PC, meaning in the above example all cores could in fact work in parallel, with some of them executing the if-case and the rest the else-case. However, cores still do not have a SP or their own stack, meaning they cannot execute divergent, non-inlined function calls:

\begin{verbatim}
__global__ void add(int * a, int * b, int * c) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if (b[i] % 2)
        a[i] = op1(b[i], c[i]); // non-inlined
    else
        a[i] = op2(b[i], c[i]); // non-inlined
}
\end{verbatim}

But even on Volta we have to be careful with branches because they can still lead to randomized memory accesses, hurting our performance just as much as reduced instruction throughput. Another form of divergence that affects all architectures without exception is tail-divergence, which occurs when threads have different runtimes:

\begin{verbatim}
__global__ void kernel(...) {
    ...
    while (condition) {...}
}
\end{verbatim}

If condition is evaluated differently per thread, each thread will have a different running time, causing the rest of the cores to idle while they are waiting for the longest-running thread’s completion. These underutilized resources again reduce our effective parallelism.

### 6.6 Asynchronicity

One of the most prominent features of CUDA is its non-blocking API. Take the code snippet in Figure 6.4(left) for example. Assuming that both \texttt{gpu	extunderscore calc()} and \texttt{cpu	extunderscore calc()} take significant time to complete (in the order of milliseconds), a possible timeline of the code’s execution may look like Figure 6.4(right).

We see that the call to \texttt{gpu	extunderscore calc()} returns immediately—we do not wait for its completion before executing \texttt{cpu	extunderscore calc()}. The asynchronous nature of CUDA’s API allows us to overlap operations. Not only GPU- and CPU-computations can be overlapped but, even more importantly, data transfers. In fact, most GPUs are equipped with dual-copy engines allowing them to simultaneously receive data, send data, and execute kernels (\textbackslash /computations).
copyToGPU(inputs[0]);
for (int i = 0; i < n; i++) {
    if (i < n-1) copyToGPU(inputs[i+1]);
    process<<<...>>>(inputs[i]);
    if (i > 0) copyFromGPU(outputs[i-1]);
}
copyFromGPU(outputs[n-1]);

This feature is also very simple to take advantage of (Figure 6.5). By breaking our computation up into batches and already uploading the inputs for step i+1 while, at the same time, processing step i and downloading step i-1’s results we almost completely negate the cost of all data transfers. The naïve approach of first uploading all our data, then performing our computations, and finally downloading the results would triple our runtime.

Spice makes extensive use of this feature in order to achieve linear scaling on multiple GPUs. By designing its data structures so that spike synchronization can be expressed as a series of cudaMemcpy() calls, it can synchronize spikes in the background, while the actual simulation is running un-interruptedly in the foreground (Chapter 8).

If we study Figure 6.4(right) closely, we observe a tiny bit of latency between the start of gpu_calc() and cpu_calc(). While, conceptually, CUDA API calls are instant, in practice they need to communicate with the GPU through the driver and all other OS abstraction layers in-between. This takes non-zero time. How much depends on the operation in question. For example, it is cheaper to launch a parameter-less kernel<<<...>>>(...) with parameters since the latter additionally needs to copy its parameters to the GPU. Spice observes API call overheads of 5 µs–10 µs. For many applications (including Spice which aims for 100 µs simulation steps) this tiny overhead can be prohibitive and requires careful implementation to negate (Chapter 9).

Typically (if our operations are long enough), we only pay for this overhead with latency, not throughput. That is because the CUDA runtime is able to queue several operations. That is, while the current operation is still running, the next one has already been transmitted to the device. Executing an already-queued operation takes only a fraction of a microsecond (Figure 6.6(top right)).

However, if our operations are significantly shorter than 5µs (such as cudaMemcpy()s of a few kilobytes), the API overhead starts dominating our total runtime and results in poor GPU utilization (Figure 6.6(bottom right)).

6 In fact, it is such a pressing problem for modern HPC applications that Nvidia introduced CUDA Graphs. These allow the programmer to upload large collections of tasks together with a dependency-graph to the GPU once and then execute them with a single API call.
A feature that goes hand in hand with asynchronicity is **peer-to-peer** (P2P) memory access: GPUs’ ability to directly communicate with one another over the PCIe bus, without involving the CPU. Combined with the duplex nature of the PCIe (Figure 6.7) this allows for efficient data exchange algorithms between GPUs as utilized by Spice for its spike synchronization (Section 8.2).

For further reading please see:

- How to Overlap Data Transfers in CUDA [76]
- Peer-to-Peer Multi-GPU Transpose in CUDA [77]
Method
Spice started out as a low-hanging fruit. SNNs were gaining popularity due to the rise in GPGPU and neuromorphic hardware and making a name for themselves in terms of power efficiency. GPU-based simulators started popping up and pushing the well-established (and CPU-based) simulators such as Nest off the performance throne. However, these early simulators were mostly written by biologists and neuro-scientists—not software engineers. During the work on Spice’s first incarnation, the competition improved by two orders of magnitude. Spice still came out 3x faster.

While Spice’s main objective was performance, we identified two other, major defects in then-state of the art simulators: initialization time and usability. Initialization/setup time had simply been ignored with the excuse that it only has to be performed once, taking up to half an hour in some instances, and completely neglecting the short-lived nature of most SNN simulations. Usability often suffered at the cost of other features or was not considered at all, so long as there was a way to implement models. A big offender in this was GeNN. It required learning a clunky (string & macro-based) DSL, running proprietary compilation steps, linking against the produced library, and repeating the whole process for any parameter change.

The goal with Spice was not only to excel in all these areas but, more importantly, also to demonstrate that they are not diametrically opposed, which we ultimately succeeded in doing (Section 11.1).

7.1 Data Structures

In Spice, a SNN consists of neuron populations which can be arbitrarily intra- and inter-connected via synapse populations. While each neuron can have multiple incoming and outgoing connections, a synapse only ever connects two neurons (and thus, a synapse population connects two and only two neuron populations). Different topologies are supported, such as fixed in/out-degree, fixed probability, or arbitrary connectivity via a user-specified adjacency list/matrix (Chapter 2).

Neuron and synapse pools are stored as structures of arrays (SoA) to enable memory coalescing (Section 6.3). Each synaptic connection is stored in its own adjacency list which greatly simplifies code since we only need to store a single synapse and connectivity type at once. This homogeneity allows us to use a padded or rectangular adjacency list which, compared to CSR, does not require an offset table (Section 4.2). Alongside each adjacency list, we store a synapse population with an implicit 1:1 mapping between the two: entry $i, j$ in the adjacency list corresponds to synapse $i, j$ in the associated synapse pool. Figure 7.1 illustrates how a network is mapped to memory.

---

1 I fondly remember speeding up a potential competitor by 3x with a single line code change.
Figure 7.1 (left) A simple SNN consisting of two populations \( A \) and \( B \) with 2 and 4 neurons respectively. Each neuron in \( A \) has a fixed out-degree of 3 (it is connected to 3 neurons from \( B \), picked uniformly at random). The neurons in \( B \) are intra-connected with a fixed probability of 50%.

(right) Each synaptic connection is stored inside its own adjacency list. Each row stores a neuron’s out-going connections as indices into the neuron pools. Non-connections are marked with sentinels \((\infty)\). The synapse pools have as many entries distributed random integers.

The allocation and initialization of neurons and synapses (on the GPU) is trivial. The generation of the adjacency list can also be efficiently done on the GPU, though it is a bit more complicated. Here, we will cover the fixed probability case—the other topologies can easily be derived from it.

Consider two neuron populations \( A \) and \( B \) (which may be identical) with connection probability \( p \). Every neuron in \( A \) has a probability \( p \) of being connected with every neuron in \( B \). The (out-)degree of each neuron in \( A \) is binomially distributed \( \sim (|B|, p) \). The degree many connections are uniformly drawn (without repetition) from \([0, |B|]\). We first conservatively estimate the total number of connections by approximating the binomial with a normal distribution \( \sim (\mu = |B| \cdot p, \sigma^2 = |B| \cdot p \cdot (1 - p)) \) and estimating the total maximum degree as \( d_{\text{max}} = \mu + 3\sigma \), and thus the total number of connections as \(|A| \cdot d_{\text{max}}\). This wastes a little bit of memory \(^2\) but in exchange allows us to pre-allocate the adjacency list and makes index calculations trivial, absolving us from an offset table (such as would be required by CSR, for example).

Each neuron’s connections can be created independently, so we only consider a single neuron. We assign a warp to each neuron. The warp generates a single binomially distributed random number \( d \) denoting the neuron’s degree. Then, we generate \( d \) exponentially distributed random numbers (by taking the negative logarithm of uniformly distributed random numbers), compute their prefix sum (which is uniformly distributed) and scale them to the interval \([0, |B| - d]\), adding \((0, \ldots, d - 1)\) to ensure uniqueness. Entries \((d, \ldots, d_{\text{max}} - 1)\) are filled with sentinels.

Figure 7.2 illustrates these steps for \( d = d_{\text{max}} = 5 \) and \(|B| = 100\). This algorithm is equivalent to drawing \( d \) numbers from \([0, \ldots, |B| - 1]\) uniformly at random without repetition. It has the added benefit of implicitly generating sorted sequences which improve cache coherency when delivering spikes. The algorithm requires two passes over the data since we need to know the maximum element (6.64) for the scaling step. Intermediate results are stored in shared memory with only the final values being written to global memory. The prefix sum can be efficiently calculated using warp-level primitives. Random numbers are generated on the fly using the “xoroshrio” RNG \([78]\) by Blackman and Vigna. Implemented in CUDA, this kernel generates synapses at close to the maximum bandwidth of a V100 (~800 GB/s).

\(^2\) In practice, we additionally round up \( d_{\text{max}} \) to the next multiple of 128 \( B \) which slightly increases bandwidth due to better memory alignment.

\[\text{exprn}(x)\]

\begin{array}{cccc}
0.56 & 2.68 & 1.12 & 0.95 & 1.33 \\
\text{prefix sum} & & & \\
0.56 & 3.24 & 4.36 & 5.31 & 6.64 \\
\text{round}(x \div (6.64 + \text{exprn}(x)) \cdot 95) & & & \\
8 & 45 & 60 & 73 & 92 \\
+ & & & \\
0 & 1 & 2 & 3 & 4 \\
= & & & \\
8 & 46 & 62 & 76 & 96 \\
\end{array}

Figure 7.2 An efficient, parallel algorithm for the generation of sorted, uniformly distributed random integers.
7.2 Pipeline

Next we want to discuss the simulation loop and flow of data through Spice. We will start with a basic implementation with a fixed, global delay of 1 dt. We will then extend said implementation to support user-defined global delays, and finally user-defined per-synapse delays.

The basic pipeline is straight-forward:

1. Neurons and synapses are updated.
2. As a consequence, neurons may spike.
3. The spikes are delivered to their recipients.

These steps constitute a single iteration of the simulation loop. They are visualized in Figure 7.3. Note that the topology being stored in multiple adjacency lists is a physical implementation detail—logically a SNN is a single, connected graph. Figure 7.3 provides a logical view of Spice’s pipeline. Also note that the spike condition is tested at the beginning of the Update step, which is why spikes carry timestamp \( t \) (this is done so spikes at \( t = 0 \) may be detected). The neuron (and synapse) update is embarrassingly parallel. Spike delivery can be parallelized as well but requires synchronization to avoid data races, as multiple spikes could arrive at the same neuron concurrently. In CUDA, we use atomic operations to resolve these potential data races. In practice, the synapse update is a bit more involved. Also, the naïve implementation shown here is infeasibly slow. Still, it results in a correct simulation and shall suffice for now. We will cover spike delivery and plasticity with all their facets and efficient implementations in Sections 7.3–7.4.

In order to go from a fixed to a user-defined global delay we need to introduce two new things: a delay parameter \( d \) and a circular buffer. Whereas before we used a singular spike list that we wrote during the update step and immediately consumed in the subsequent spike delivery step, we now require a circular buffer of \( d \) such spike lists. We write spikes into buffer \( i \) (mod \( d \)) while reading spikes from buffer \( i + 1 \) (mod \( d \)), deferring their delivery by \( d \) dt. Figure 7.4 shows the necessary changes to the pipeline.

To go from global to per-synapse delays we take advantage of the fact that time is quantized and thus there is a maximum delay \( d_{\text{max}} \). We can...
Figure 7.5 Same as Figure 7.4 but with support for per-synaps delay. The graph is partitioned into $d_{max}$ subgraphs, each consisting only of synapses with delay $d_i$. At every loop iteration, *all* spike lists are delivered: 1-$dt$ old spikes via the $d_1$-graph, 2-$dt$ old spikes via the $d_2$-graph, etc.

then split each synaptic connection into $d_{max}$ disjoint adjacency lists, each with a homogeneous delay which we already know how to handle. As before, we maintain a circular buffer of spike lists. But instead of only delivering spikes from $d$ steps ago using our singular adjacency list, we now deliver all spike lists via their respective adjacency lists (Figure 7.5). The delivery once again can be parallelized across all delays and adjacency lists so long as it is correctly synchronized. The total amount of work stays the same, the overhead of the additional kernel launches is hidden by the CUDA runtime.

Figure 9.6 brings everything we learnt so far together in one figure, showing not only a physical view of Spice’s data structures and pipeline, but also their inter-play.

### 7.3 Spike Delivery

In the following sections we want to zoom in on two parts of the pipeline: spike delivery and plasticity (synapse updates). While the implementation of both can be more or less straight-forward, it is not trivial to map these tasks efficiently to GPGPU hardware and maximize performance. Spice introduces two novel optimization techniques for this very purpose. The first is a type of blocking scheme allowing us to receive spikes in shared memory instead of global memory. The second is a combination of event-driven plasticity [26–28, 49, 79, 80] and lazy plasticity [34]. We will discuss all evolutionary stages of both pipeline steps, from the first naïve implementation, over several iterative improvements, to the final aforementioned optimizations.

Upon a neuron firing, all its neighbors must be informed of the spike. Conveniently, all its neighbors are already laid out in contiguous memory inside our data structure, making it very easy to iterate over them. Listing 7.1 illustrates this naïve approach:

Listing 7.1: Naïve algorithm for delivering spikes to target neurons for a single synaptic connection. All loop iterations are independent and thus trivially parallelized (for example, by assigning a CUDA block to each src).

```python
for src in spikes:
    for i in 0..maxDegree-1:
        dst = adj[src][i];
        if (dst < inf):
            deliver(src, neurons[dst], ...);
```

This algorithm is optimal in two ways:

- It does the minimum amount of work, only iterating over neighbors of spiking neurons (disregarding the < 3% overhead due to our padded adjacency list).
- Memory reads from the adjacency list get fully coalesced since they access contiguously stored elements.
It is sub-optimal in terms of writes to the target neurons, which are scattered across memory (Figure 7.6), the negative effects of which are mostly mitigated by the sorted nature of our adjacency list and the hardware cache. Sorting each row of the adjacency list minimizes the gaps between memory accesses. The average gap is antiproportional to the density of the network—a synaptic connection with 10% density will have an average gap of 10 neurons between neighbors. Theoretically this would cut our memory bandwidth by 90% which is mostly avoided by the sizable and well-performing cache (Section 6.4).

This brings us to the Achilles’ heel of this approach: the cache. Once we start dealing with massive networks with millions of neurons that exceed the cache, we begin suffering from the reduced effective bandwidth of our sub-optimal memory accesses. The solution to this problem is as simple as it is elegant and effective—we simply need to change the iteration order of our algorithm. Instead of delivering one spike to all neighbors of a neuron, we deliver all spikes to the first neighbor of each neuron, then the second, the third, etc. (Listing 7.2).

What this simple change achieves is keeping dsts close together, longer. Since our adjacency list is sorted, we expect the entries from column i to be close together. This results in the speedup shown in Figure 7.7 as it better utilizes the cache (Figure 7.8). Cache-aware spike delivery is strictly faster than the naïve one, regardless of network size, density, or activity. The algorithm is slightly sub-optimal because it needs to re-read meta data (src) for every target neuron but this extra cost is negligible in practice. Speaking of practice, we do not iterate over singular columns using singular threads but over three-column wide strips using warps so as to retain full coalescing of adjacency data reads.

GPGPUs are some of the few processors which give the programmer full control over the placement of data in either global memory (RAM) or shared memory (L1 cache) (Section 6.4). Shared memory is an order of magnitude faster than global memory and can be as fast as registers if we access it coalesced. As programmers we always want to keep data and perform computations inside the fastest available memory subsystem.

Spike delivery has an irregular memory access pattern and requires atomic operations, both of which can benefit immensely from shared memory—so what stops us from employing it?

---

**Figure 7.6** Memory access pattern of the naïve spike delivery algorithm. Reads from the adjacency list are coalesced. Writes to the neuron pool are scattered. However, as long as enough spikes are processed in parallel and the network is dense and small enough so that all targets fit into the cache, the bandwidth only drops marginally (ratio of white vs. gray cells).

**Listing 7.2**: Cache-aware algorithm for spike delivery. Same as Listing 7.1 but with the loops swapped.

---

3 CUDA devices access main memory in atomic transactions of 128 B. Even if only 10% of words in such a block are of interest to us, we still must transfer the entire block, dramatically reducing our effective bandwidth (Section 6.3).

4 Shared memory used to be its own, physically distinct resource. On modern GPUs, it is part of the L1 cache itself.
Comparison of memory access patterns between the naïve and cache-aware spike delivery algorithm. The latter are much more concentrated and thus result in better cache utilization.

- The need to stay generic—nothing stops the user from creating a graph with arbitrarily large gaps in its adjacency list which would exceed shared memory capacity.
- The fact that shared memory is fenced off from other CUDA blocks—if a neuron updated by block A spikes, that spike cannot be delivered to any neighbors held by other blocks, except through a complicated communication/synchronization scheme involving global memory—which would defeat the purpose.

In other words, if we manage to a) cap the range of spike recipients and b) establish an ownership between CUDA blocks and neurons, we will be able to utilize shared memory for spike delivery. We do so by logically dividing the neuron domain into chunks of equal size, say 1024 neurons each (the cap aspect). We then partition the adjacency list into slices so that each slice indexes one and only one chunk (the ownership aspect, see Figure 7.9). This is achieved by pre-computing \( \lceil n / 1024 \rceil \) pivots per row of the adjacency list via binary search. Our statistical expectation turns into mathematical certainty. Knowing that slice \( i \) indexes neurons and only neurons \( [i \cdot 1024, (i + 1) \cdot 1024) \) we can load them into shared memory and deliver spikes there (Listing 7.3). In a sense, shared memory-based spike delivery is the inverse of cache-aware spike delivery. Cache-aware delivery blindly accesses a fixed number of neighbors targeting a variable number of neurons (hoping the targets will be close-by). Shared memory-based delivery instead accesses a variable number of neighbors that in return target a fixed range of neurons.

Listing 7.3: Shared memory-based spike delivery. We load a 1024-neuron slice from global into shared memory (lines 4–5), where we deliver all spikes to our neurons (lines 9–12). When done we write the new neuron state back to global memory (lines 16–17). Once again, all loops are trivially parallelized. In practice we assign a (1024-thread) block to each slice, a warp to each spike, and a thread to each target neuron. This code assumes that \(|\text{neurons}|\) is a multiple of 1024, for brevity. Additional logic is needed to guard against out-of-bounds accesses.
The fundamental difference between the algorithm in Listing 7.3 and its previous two incarnations is that it establishes ownership between threads and neurons. This has the additional benefit of minimizing global memory traffic by only checking out the initial and committing the final neuron state. It also enables further optimizations: First, since one thread now owns one neuron, we can advance neuron dynamics directly inside the kernel. We would simply insert the following code:

```c
for i in 0..1023:
    advance(shared[i], DT, ...);
// insert new spikes into circ. buffer
__syncthreads();
```

in line 15 in Listing 7.3. This would save us one kernel invocation (reducing CUDA API overhead) and further reduce memory traffic. Having a combined kernel which both updates neurons and delivers spikes lends itself to another optimization: timestep grouping [27, Section 2.2]. Ahmad et al. observed that it is safe to blindly perform delay many updates in a row or batch, since any spikes generated within said batch will not arrive until after been completed. For us, this simply means wrapping lines 9–15 (including the new update snippet above) of Listing 7.3 into a loop with delay of many iterations, further minimizing API overhead and memory traffic. All these improvements combined yield a speedup of 2×–2.5×, 90% of which is due to the use of shared memory alone.

This optimization requires one small change to our hitherto pipeline. Recall that, for convenience, we store our graph across several adjacency lists—one per synaptic connection (Figure 9.6). Before, we were able to deliver all spikes over all synaptic connections concurrently (by launching kernels into different CUDA streams). Because we were using atomic operations and global memory, the hardware would resolve any conflicts for us. Now that threads own neurons though, we must avoid the same neuron being simultaneously checked out by two threads. That is, we must avoid simultaneously delivering spikes over synaptic connections which have common target neurons (i.e., ones that overlap horizontally in Figure 7.10). This problem is known as (job) scheduling under precedence constraints and is, in its most general form, NP-complete [81]. We use a simple greedy algorithm that processes as many non-conflicting synaptic connections in parallel as possible until none are left (a||c → b||d in Figure 7.10). It works well enough in practice since the number of connections is low and conflicts are rare.\footnote{Since the topology is static, an optimal schedule would only need to be created once. Thus, given a low enough number of synaptic connections, even an NP-complete algorithm may be feasible. It would be worth investigating what difference an optimal schedule makes.}

**Results**

Figure 7.11 shows the performance gain these optimizations culminate in, compared to the naïve approach. The respective speedups of 2× for the Vogels model and 2.5× for the Brunel model (Section 10.2) were achieved using the following parameterization:

- The neuron domain was divided into chunks of 1024 neurons.
- Each chunk was assigned to a CUDA block with 1024 threads.
- Each spike was assigned to a CUDA warp (with 32 threads).

Recently, Pronold et al. [82] implemented this very optimization on the CPU, which they refer to as “bwTS” in their paper. Even though CPUs offer much less control over the cache (the authors relied on the prefetch instructions which are generally treated as hints), they still observed a ~1.6× performance improvement.
The chunk size is a delicate, tunable parameter. Increasing it increases parallelism; the gap between pivots grows, more neurons fall into it, which results in more work for the warps. The effective bandwidth increases because meta data (pivots) only have to be read once per warp and divergence (Section 6.5) decreases due to more even loading of threads within the warp. However, increasing the chunk size also inhibits parallelism. Since more neurons have to be held in shared memory (a limited, finite resource), fewer CUDA blocks can be scheduled per SM (Section 6.1). The latency hiding ability of the GPU is reduced. 1024 neurons struck a good balance in our benchmarks and have the additional benefit of establishing a 1:1 mapping between neurons and threads, simplifying code.

### 7.4 Plasticity

Plasticity refers to a synapse’s ability to modify its state (typically its weight) as a function of its dynamics as well as local network activity (pre- and post-synaptic spikes). Plasticity is one of the main learning mechanisms of SNNs. The sheer number of synapses, the computational cost of their dynamics, and the non-triviality of computing pre- and post-synaptic spikes, makes plasticity a very costly operation worth optimizing.

Figure 7.12(a) shows an optimistically simplistic variation of plasticity, in which we update every synapse at each simulation step. This is infeasibly slow due to memory bandwidth requirements. For example, the Brunel model (Section 10.2) consumes 12 B per plastic synapse which make up 40% of all synapses. Thus, simulating a 1B-synapse instance of it in real-time would require ~87 TB/s, \(7 \times 100 \times \text{the bandwidth of a V100}\).

In addition to advancing each synapse’s dynamics, plasticity also needs to facilitate the efficient computation of pre- and post-synaptic spikes. A pre-synaptic spike occurs if a synapse’s source neuron fires; a post-synaptic spike occurs if its destination neuron fires. We must inform the synapse of either event. Pre-synaptic spikes are trivial to compute, especially if synapses are grouped by source neuron. The fact that a neuron fires implies that all its outgoing synapses experience a pre-synaptic spike (potentially after some delay). Luckily for us these same synapses are already laid out in a known, contiguous region of memory, making it very easy to efficiently iterate over and update them. Post-synaptic spikes are more tricky to compute since we need to consolidate a neuron’s incoming synapses. An infeasibly slow solution would be to iterate over the entire adjacency list and search for all occurrences of said neuron, and repeat this for all firing neurons (\(O(|\text{neurons}| \cdot |\text{synapses}|)\)).

Many clock-driven simulators [26, 27] borrow ideas from event-driven simulators which correctly observe that:

1. we only care about a synapse being up to date when it transmits a (pre-synaptic) spike, and
2. spikes are rare.
extern int delay; // delay (in time steps) of the synaptic connection
struct neuron { int64 hist; ... }; // firing history, each bit = one time step, LSB = most recent
struct synapse { neuron& dst; ...};

void updtd(synapses& syn, bool pre, bool post, int n) { // user-defined callback
range bits(int64 x); // returns indices of *set* bits of x

for n in neurons:
    for syn in neighbors(n):
        age = now-n.tLastSpike;
        for i in age..0:
            syn.dst.hist[i] = true;
            j = i;
        update(syn, j<age, false, age-j);
    n.tLastSpike = now;

Figure 7.12: Spicex’s evolution from naïve, over lazy, to event-driven plasticity. (a) Naïve plasticity. We loop over all synapses (lines 1–2) and invoke the user-provided callback updtd() (line 6) which is part of the synapse specification. We pass it a reference to each synapse (line 7) as well as information about the occurrence of pre- and post-synaptic spikes by reading the corresponding bits from the source and destination neuron’s firing histories (lines 8–9, x[i] := (x >> i) & 1). The for loops can be trivially parallelized because each loop iteration is independent. In the case of two nested for the outer loop iterations could be assigned to different CUDA blocks and the inner loop iterations could be assigned to different threads within a block. (b) Lazy plasticity. Instead of updating all synapses, we only update those originating from firing neurons (line 1), since they are about to transmit a spike and thus need be up to date. We compute the synapses’ age, i.e., the number of iterations for which we have intentionally neglected them (line 3) and replay all the missed updates (line 5). (c) Event-driven plasticity, by contrast, does not replay the entire firing history but only iterates over set bits, i.e., post-synaptic spikes, skipping ahead multiple update steps in-between (line 5). This is communicated to the user via updtd()’s fourth parameter which represents the number of time steps that have elapsed since the callback was last invoked (line 9). The updtd() call in line 11 handles the “tail.”

Thus, they opt for event-driven plasticity; that is, they only update synapses upon pre- or post-synaptic spikes. While this is a lot faster, it still has several major disadvantages:

- In order to update synapses based on post-synaptic spikes, they need to be congregated. For this purpose these simulators store a reverse adjacency list (of incoming connections), which doubles the memory consumption of the graph representation.
- Using a reverse adjacency list results in very poor memory bandwidth for the same reasons as discussed in Section 7.3. While the incoming connections (adjacency data) are stored contiguously and thus can be read in a coalesced fashion, the synapses they refer to are scattered all over memory, resulting in fragmented memory writes (over a much greater region compared to neurons).
- While event-driven plasticity is an order of magnitude faster, it is still not optimal in terms of bandwidth. It updates a synapse at every pre- and post-synaptic spike when we really only care about pre-synaptic spikes. In a model with twice as many post-synaptic as pre-synaptic spikes, for example, this would result in double the optimal work.

We will go a different route that does not require a reverse adjacency list, has a much better memory access pattern, and is optimal work-wise. Instead of employing a reverse adjacency list, we store the network’s recent firing history in the form of per-neuron bitsets (typically 64-bit integers). These bitsets are trivially updated during the neuron update step: n.history = (n.history << 1) | spiked;
Figure 7.13 Relative performance of native plasticity, lazy plasticity, and finally lazy plasticity combined with event-driven plasticity. Tested on the plastic variation of the Brunel model (Section 10.2).

Additionally we store a time of last spike per neuron which we initialize to 0. Now, whenever a neuron fires, we

1. Compute its \( \text{age} = \text{now} - t_{\text{LastSpike}} \);
2. Update each outgoing synapse by iterating over the \( \text{age} \) most recent bits of the target neurons’ firing histories (to determine post-synaptic spikes), followed by one final update for the pre-synaptic spike due to the source neuron firing;
3. Set \( t_{\text{LastSpike}} = \text{now} \).

This algorithm (detailed in Figure 7.12(b)) is known as Lazy Plasticity, first introduced in 2020 by Bautembach et al. [34]. Going from lazy to event-driven plasticity can be done trivially while retaining all of the benefits of lazy plasticity. Instead of iterating over the \( \text{age} \) most recent bits, we directly iterate only over the \( \text{set ones} \), which can be done in constant time using the \( \_\_\_\text{clz}() \) and \( \_\_\_\text{ffs}() \) intrinsics (Figure 7.12(c)). This is known as lazy event-driven plasticity [28].

In order to benefit from event-driven plasticity (whether lazy or not), synapse dynamics must have a closed-form solution so they can skip ahead multiple steps at a time. If this is not possible, users of Spice can simply loop internally in which case Spice gracefully degrades back to lazy plasticity.

Results

Figure 7.13 shows the relative performance of all three evolutionary stages of plasticity (naïve, lazy, and lazy+event-driven) as evaluated on the plastic variation of the Brunel model (Section 10.2). Note that lazy and naïve plasticity both perform the exact same number of computations. The 5\( \times \) speedup is only due to reduced memory traffic (the synapse state is read and written once per pre-synaptic spike, all updates are performed inside registers) and a better memory access pattern—only the reads of target neuron histories are scattered. However, they are 8 B small, are only read once per up to 64 updates, and are confined to a much smaller region of memory (KB vs. GB)—all synapse accesses are fully coalesced.

Lazy event-driven plasticity is 6\( \times \) faster than pure lazy plasticity (for a cumulative speedup of 30\( \times \)), this time due to actually performing fewer computations. Furthermore, it is still 2\( \times \) faster than our closest competitor relying on pure event-driven plasticity (Section 10.5).
If we want to advance the field of SNNs, not only should we strive to make SNN simulation as fast as possible, but also to efficiently parallelize it across several processors. CPUs’ single thread-performance stagnated years ago and GPUs, getting bigger and bigger and consuming more and more power, might fall victim to the same fate in the near future. Going forward, the biggest improvements will be achieved through parallelization. Kunkel et al. [59] have done an important first step in this regard on the CPU side—on the GPU side, until recently, results were underwhelming. The best simulator so far, BSim [35], achieved a ~2.2× speedup on four GPUs.

Spice on the other hand can utilize up to eight GPUs (in the same server) with an efficiency of 100%, allowing users to run the same model 8× faster or to run an 8x larger model in the same time. Spice’s parallelization strategy is its own chapter because it is completely orthogonal to all other features and optimizations presented so far. In fact, any simulator that meets the following criteria can employ Spice’s parallelization strategy, basically making it a meta algorithm:

- Spikes are stored contiguously.
- Spike buffers are large enough to hold up to \(|\text{neurons}\)| spikes.
- The spike buffers are exposed in the API (read & write access).
- The simulation runs in a non-default CUDA stream.¹

### 8.1 Parallelization Strategy

Spice’s entire parallelization strategy stems from a single principle:

> Every GPU must be able to deliver any spike to its neurons.

Subsequently, let us investigate how a network would be distributed across two GPUs which will simplify the discussion. Generalizing from two to \(n\) GPUs is straightforward. We already saw in Section 7.3 how a network can be partitioned into disjoint and thus non-conflicting slices (Figure 7.9). The key insight here is that each of these slices forms a SNN in its own right. While the original purpose of partitioning the network was to utilize shared memory, nothing stops us from distributing each slice to its own GPU, supplying both with the same spike lists, and letting them simulate their respective slice independently from one another.

In contrast to the single-GPU pipeline, the multi-GPU pipeline contains one additional step: spike synchronization (Figure 8.1). Since neurons stored on GPU A might have neighbors on GPU B (and vice versa), all GPUs must exchange their spikes after the update step. This synchronization step can be realized in the form of a meta algorithm—the existing pipeline need not change, i.e., it need not be made multi-GPU-aware.

---

1. For even better performance the simulator can choose to expose a CUDA event which marks the completion of each simulation step. This way the meta algorithm can use the more fine-grained and efficient `cudaStreamWaitEvent()` over `cudaStreamSynchronize()`.
Multi-GPU Parallelization

Figure 8.2 (left) A graph representing a SNN’s topology (see Figure 7.1 for details). (right) The same network distributed across two GPUs so that its size, topology, and semantics are maintained. Any spikes crossing the dashed line need to be synchronized.

Each slice forms a stand-alone SNN simulated on a dedicated GPU with no knowledge of any other slices.

Furthermore, we never need to physically instantiate the whole network just to distribute it again. Rather, we can directly construct the individual slices on their respective target GPUs from the model parameters. Given a graph which represents the topology of a SNN (Figure 7.1(left)), we can use the following algorithm to distribute it across two GPUs:

1. Split each neuron population (/node) \( A = \{a_1, \ldots, a_n\} \) into two populations \( A' = \{a_1, \ldots, a_{n/2}\} \) and \( A'' = \{a_{n/2+1}, \ldots, a_n\} \).
2. Split each synaptic connection (/edge) \((A, B, p)\) (read as: each neuron in \( A \) is connected to each neuron in \( B \) with probability \( p \)—other connection types are analogous) into four connections \((A', B', p), (A', B'', p), (A'', B', p), \) and \((A'', B'', p)\).
3. Form two new graphs \((\{X'\}, \{(\ast, X')\})\) and \((\{X''\}, \{(\ast, X'')\})\).

This is visualized in Figure 8.2. Note that each slice contains half the number of neurons and synapses as the original network, meaning the maximum network size grows linearly with the number of GPUs as well.

8.2 Spike Synchronization

Neurons on GPU A may have neighbors residing on GPU B (and vice versa). Hence, when these neurons fire, GPU A must communicate its spikes to GPU B. More generally, during the synchronization step all GPUs cooperatively compute the union of all spikes generated in the preceding update step. This way, each GPU obtains a complete list of all spikes in the system and can deliver them to its neurons. Since we merely exchange spike lists, i.e., lists of neuron IDs, we only need to transfer some data. Nevertheless, we want to make spike synchronization as fast as possible since it creates a synchronization point and thus serial step inside our pipeline which inhibits (parallel) scaling. We use a standard hierarchical algorithm [75, pp. 11–20] which takes advantage of GPUs’ dual-copy engines as well as the hierarchical and duplex nature of the PCI Express bus (Section 6.6). The algorithm is visualized in Figure 8.3.

---

2 All modern, higher-end GPUs are equipped with dual-copy engines, allowing them to concurrently send and receive data (while also executing kernels).
8.3 Latency Hiding

No matter how fast we make spike synchronization, Amdahl’s Law [83] teaches us that any serial step inside an algorithm caps the maximum obtainable speedup no matter the number of processors we throw at it. For example, assume that spike synchronization $s$ accounted for a mere 10% of the simulation step. In the case of eight GPUs that would already limit us to a speedup $S \approx 4.7$, wasting 40% of our hardware and energy:

$$S = \frac{1}{s + \frac{1-s}{N_{GPU}}} = \frac{1}{0.1 + \frac{0.9}{8}} \approx 4.7$$

Luckily, in most circumstances, the spike sync step is non-blocking: Remember that each synapse is associated with a transmission delay, meaning spikes do not arrive instantly at their destinations. If we manage to synchronize said spikes before their supposed arrival date, the simulation can continue uninterrupted. The sync cost effectively drops to zero, we have successfully hidden its latency and achieved linear scaling.

For now assume our network has a global delay $d$. We group the simulation into batches of $d \div 2$ steps. This creates two interleaved dependency chains where batch $i+2$ depends on batch $i$’s spikes but batches $i$ and $i+1$ are independent (Figure 8.4(alternating colors)). As soon as batch $i$ completes, we immediately begin synchronizing its spikes in the background, while concurrently running batch $i+1$. If the sync completes before batch $i+1$ does, batch $i+2$ will not be delayed (Figure 8.4).
In this case SNN simulation becomes a sequential problem (one in which task \( i \) depends on task \( i - 1 \)) for which we do not know any parallel solution at this time. (Whether *inherently* sequential problems, ones that *probably* cannot be sped up by parallelism, exist, is an open problem in computer science known as \( NC = P \).)

Fast spike synchronization and latency hiding are worthless if all GPUs are waiting for the longest-running one. This brings us to the third and final ingredient of Spice’s parallelization strategy which, together with the other two, achieves linear scaling—load balancing. We already learned how Spice distributes work across GPUs *mechanically*, on a data structure level. Load balancing is the act of finding a concrete neuron ↔ GPU mapping with the goal of loading each GPU equally. This is difficult because neurons’ simulation costs can vary greatly, for example due to their degree, computational cost of their dynamics, etc.

PipeDream [53], 4 which in large part solved multi-GPU backpropagation for conventional ANNs, opts for profiling. Before training, the cost of each layer is measured and layers are then optimally distributed across GPUs. This strategy has several disadvantages and does not apply to SNNs:

- The profiling time counts towards the total simulation time. A few seconds of profiling may be negligible in the face of hours of training, but they do add significant overhead in the case of short-lived SNN simulations.
- The thusly obtained partition would only be suitable for the profiling period itself. SNNs exhibit very complex dynamic behavior with greatly varying firing patterns over time. A partition which results in even load balancing at the beginning of a simulation, may become completely detrimental a few seconds into it (Figure 8.5(*top*)).

Instead, Spice adopts the simple static load balancing strategy proposed by Kunkel et al. [59]—over-subscription combined with round-robin scheduling. Instead of splitting the network into \#GPUs many slices, we split it into \( k \cdot \#GPUs \) slices (typically \( k > 10 \)) and assign them to GPUs in a round-robin fashion (Figure 8.6). Statistically, this averages out any variance or skew present in the slices’ simulation costs (Figure 8.5(*bottom*)) so long as their number outweighs the dynamic range of their costs—a necessary condition for any load balancing strategy. Adversarial cases where every \#GPU-th slice happens to be more expensive than the rest are rare and, if they occur, can be alleviated by simply changing \( k \).

While the basic concept of static load balancing is quite simple, Spice deserves extra credit for being able to efficiently split a network into (in practice) hundreds of slices, which enables us to employ static load balancing in the first place. Furthermore, we can reuse all the logic from Section 7.3.
8.5 Results

Spice scales with multiple GPUs in both space and time, allowing one to increase network size while maintaining simulation time (Scaleup, Figure 8.7(left)) or to cut down on simulation time while maintaining network size (Speedup, Figure 8.7(right)). Both scenarios suffer from a natural limit: spike synchronization time grows linearly with the number of GPUs—add enough and any simulation will eventually be bottlenecked by it, leading to sub-linear or even negative scaling (Figure 8.8(top)). This effect sets in much earlier in the speedup scenario where per-GPU simulation time goes down with the number of GPUs, shortening the spike synchronization window (Figure 8.8(bottom)). That is why I consider scaleup to be the predominant use case of our contribution.

As can be seen, Spice achieves close to linear scaleup except for Vogels (Section 10.2) running on eight GPUs. Vogels has a highly erratic firing pattern alternating between periods during which almost every neuron fires and periods during which few or no neurons fire. While we avoid synchronization altogether if no spikes occur, we still need to download spike counts to determine that no spikes have occurred. In the 8-GPU case this takes enough time to delay subsequent simulation steps and prevent linear scaling.

In some cases Spice even scales super-linearly. This is because making a network \( n \) times larger and then splitting it into \( n \) slices again does not yield the original network. If the original network needed to deliver \( S \) spikes to \( d \) recipients each, then a slice would need to deliver \( S \cdot \sqrt{n} \) spikes to \( d \div \sqrt{n} \) recipients each. While the total amount of work stays the same, the trade-offs change (Figure 8.9).

![Figure 8.7 Spice’s multi-GPU scaling. (left) Scaleup as a function of GPU count: How many times larger can we make a model on 2, 4, 8 GPUs while maintaining single-GPU simulation time? (right) Speedup as a function of GPU count: How much faster does the same model run on 2, 4, 8 GPUs compared to a single GPU?](image)

![Figure 8.8 Scaling by scenario. (top) Scaleup: As the number of GPUs increases so does the spike synchronization time, eventually surpassing simulation step time and delaying subsequent steps. (bottom) Speedup: Same as above. Additionally, the simulation time goes down with the number of GPUs. The described effect sets in much quicker.](image)

![Figure 8.9 Explanation of super-linear scaling. A SNN left, depicted as an adjacency matrix, is made 4x larger (referring to its synapse count) and split into 4 parts again right. While the total amount of work stays the same, the distribution of work changes: Each slice has to deliver twice the spikes to half the recipients.](image)
The discourse so far was mostly of conceptual nature. In addition to novel algorithms, a substantial portion of Spice’s success can be traced back to good software engineering, which we want to highlight in this chapter. Hopefully, by the end, the reader will have acquired enough knowledge to write their own, high-performance SNN simulator—without having to learn from their mistakes along the way.

Section 9.1 gives a high-level overview of the project structure. Section 9.2 illustrates to what lengths Spice went to ensure the correctness of its results including, but not limited to, its testing methodology. Section 9.3 tells the story of various implementation highlights, obstacles, and how they were overcome. These details are not at all evident from the conceptual discussion of preceeding chapters but contribute to Spice’s superior performance just as much as its novel algorithms and optimizations. Section 9.4 contains a complete, tutorial-style introduction to Spice’s API teaching the implementation of a simplistic SNN and full-blown SSSP. Finally, for the ambitious reader, Section 9.5 consists of a series of blueprints showing how Spice operates on a physical level.

### 9.1 Software Architecture

Spice is divided into four projects:

- `spice` (library)
- `spice_test` (executable)
- `spice_bench` (executable)
- `app` (executable)

`spice` is a shared library (a.k.a dynamic-link library on Windows) and contains all the business logic. It is designed as a library (rather than an application) so it can be effectively tested. `spice_test` is a test suite based on googletest responsible for ensuring the correctness of `spice`. `spice_bench` is a benchmark suite based on google benchmark responsible for ensuring spice’s performance. Finally, `app` is the actual user-facing application. It links against `spice` and provides a command-line interface (CLI) for its API. It ships with sample code for all models used in Spice’s evaluation. Users only need to build `spice` and `app`, for which they require a C++17 compiler and CUDA 11 (and a CUDA-compatible GPU). There are no other dependencies.

If we study Spice’s folder structure (Figure 9.2), we see that `spice` ships with two full implementations, one targeting the CPU (for testing purposes, Section 9.2), the other targeting the GPU. Both implementations consist of business logic and helper code (util subfolder). `spice_test` mirrors the folder structure of `spice` and provides tests for each class and method. Figure 9.1 shows the distribution of Spice’s 3.3K lines of code (LOC) across its various sub-projects.

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1 Software Architecture</td>
<td>53</td>
</tr>
<tr>
<td>9.2 Testing</td>
<td>54</td>
</tr>
<tr>
<td>9.3 Implementation Details</td>
<td>55</td>
</tr>
<tr>
<td>9.4 API</td>
<td>58</td>
</tr>
<tr>
<td>9.5 Blueprints</td>
<td>60</td>
</tr>
</tbody>
</table>

This chapter does not (nor does it aim to) cover Spice’s source code in its entirety. The interested reader can study it at github.com/denniskb/spice.
Design Philosophy

9.2 Testing

Spice’s correctness was of utmost importance to me. Typically, simulators compare their results against those from Nest [36] which is regarded as a de facto ground truth. Comparing Spice against Nest (or any other published simulator) with the goal of producing bit-for-bit identical results would have been difficult for two reasons: 1) All models rely on randomness to some degree. 2) Spice uses a fundamentally different way of generating random numbers. In Spice, each thread has its own RNG while other simulators read pre-generated random numbers from a common buffer. Even if I (temporarily) adopted the second scheme and found that Spice produces identical results, bugs might get introduced while porting back to per-thread RNGs.

Spice goes a different route instead. First, a single-threaded CPU implementation was written from scratch. This is orders of magnitude easier to reason about and debug compared to directly jumping into a GPU implementation. The CPU implementation was audited by multiple, un-biased developers to ensure its semantics reflect the mathematical definitions of our models. Additionally, its spike trains were compared quantitively against those of Nest by comparing the average firing rates of small clusters of neurons inside a small, sliding time window and qualitatively by visualizing both simulators’ firing patterns. Once enough confidence about the correctness of the CPU implementation had been gained, matching the GPU implementation against it became trivial—run a simulation on the CPU and make sure the GPU agrees with it bit-for-bit. Listing 9.1 shows an excerpt of the relevant test code.

---

**Figure 9.2** Folder structure of spice (shared library) left and spice_test (test suite) right.
Implementation Details

The CPU uses 80-bit floating point arithmetic and then truncates the results back to 32 bits (float) or 64 bits (double). The GPU performs arithmetic directly in either 32 bits or 64 bits.

Spice should be able to run the same model cross-platform simply by swapping `cpu::snn<model>` with `gpu::snn<model>` (write once, run everywhere). The CPU prefers an AoS layout, while the GPU prefers a SoA layout. At the same time, the user should not be burdened with any of these details. Spice abstracts away from the storage layout via iterators. But first the user must manually communicate their neuron/synapse layout to Spice since C++ does not (yet) support introspection. This is achieved via (variadic) templates:

```cpp
struct MyNeuron : public neuron<float, int> { ...);
```

By inheriting from a templated base class the user states, “my neuron type has two attributes with types `float` and `int`.” This description is then converted to either an AoS or a SoA representation for use on the CPU or GPU respectively:

```cpp
template <class... Fields>
struct neuron
{
  using aos_t = std::vector<std::tuple<Fields...>>;
  using soa_t = std::tuple<thrust::device_vector<Fields>...>;
  ...};
```

Then, when it comes to implementing the neuron’s dynamics, rather than accessing its attributes directly (and thus storage-aware), the user accesses them through an iterator (i.e., storage-agnostic):

```cpp
cpu::snn net1(...);
gpu::snn net2(net1);

for (int i = 0; i < ITER; i++) {
  ASSERT_EQ(net1.step(), net2.step());
  ASSERT_CLOSE(net1, net2);
}
```

The numerical states of both implementations never deviated by more than six decimal places for thousands of iterations. True bit-for-bit identity could have been achieved by using (single-float) SIMD instructions on the CPU to mirror the GPU’s floating point semantics, but given the already close results this was neither necessary nor worth it.

Spice tests more than just correctness. Performance regressions are avoided by running a benchmark as part of the test suite and comparing its score to past, stored high scores, rejecting any commit above a 3% regression.

Tests are not the only means of ensuring correctness either. Spice uses the contractual programming model, enforcing pre-conditions and invariants via assertions that, if violated, throw exceptions. Lastly, Spice’s entire API provides the strong exception guarantee [84, p. 72].
Design Philosophy

Figure 9.3 Spice class diagram.

Here, an `enum` is used to refer to our attributes by meaningful names rather than indices. `get<I>(it)` maps to `std::get<I>(*it)` on the CPU and `it.get<I>()` on the GPU. All that is left for the CPU and GPU implementations is to provide their own iterators. In the case of the CPU this is as simple as returning `std::vector::begin()`. On the GPU a little more coding is required:

```cpp
struct iter {
    int i;

    iter(int index) : i(index) {}  

    template <int I>
    __device__ auto get() {
        return soa[I][i];
    }

    // Rest of iterator interface
};
```

Here, `soa` is an array of pointers in constant memory pointing to arrays of `floats` and `ints`, that is the attributes of our neuron.

Figure 9.3 shows Spice’s class hierarchy. Both the CPU- and GPU-version of the simulator implement the same interface `snn` which allows the user to seamlessly switch between the two simply by replacing `cpu::snn<model>` with `gpu::snn<model>`—the model definition needs not be altered. This is useful for debugging models on the CPU before
// main thread
cudaSetDevice(0);
task0<<<...>>>();
cudaDeviceSynchronize();
cudaSetDevice(1);
task1<<<...>>>();

// thread A

cudaSetDevice(0);
cudaStreamCreate(s0);
cudaEventCreate(e0);
...
task0<<<..., s0>>>();
cudaEventRecord(e0, s0);
cudaStreamWaitEvent(s1, e0);
semaphore.signal();

// thread B

cudaSetDevice(1);
cudaStreamCreate(s1);
...
semaphore.wait();
task1<<<..., s1>>>();

simulating them on the GPU, for example. The multi-GPU version of
Spice is implemented using the composite design pattern [85, pp. 163–
173]. multi_snn is itself a snn but implements most of its functionality
by delegating to gpu::snn and adding spike synchronization on top. For
example, adding a new neuron population is implemented equivalent to
(edge cases omitted for brevity):

```
template <class T, class... Args>
void gpu::multi_snn::add<T>(int n, int offset, Args... args) {
  int slice = n / children.size();
  for (int i = 0; i < children.size(); i++)
    children[i].add<T>(slice, i * slice, args...);
}
```

The update step, which consists of delivering (the previous step’s) spikes
and updating neurons, is implemented equivalent to:

```
void gpu::multi_snn::step() {
  // net::step() performs:
  // - deliver previous spikes
  // - update neurons
  // - generate (new) spikes
  for (int i = 0; i < children.size(); i++) {
    cudaSetDevice(i);
    children[i].step();
  }
  synchronizeSpikes();
}
```

Every microsecond counts when aiming for 100 µs per simulation step.
Considering that CUDA API calls themselves have a latency of 1 µs–10 µs,
we must minimize the cost and count of such API calls before attempting
to optimize the code itself.

As we learned in Section 7.2, Spice uses a circular buffer to store spike
lists. As spike lists get recycled they need to be zero’ed. Of course we
do not override the entire list, we simply zero the counter. The naïve
approach would be to use cudaMemcpy(). I found a specialized kernel to be ~10× faster (1 μs vs. 10 μs):

```cpp
template <class T>
__global__ void _zero(T* x) {
    *x = T(0);
}
```

The effect of API calls-induced latency is exacerbated in the multi-GPU implementation. It needs to launch kernels to different GPUs. Typically, this is achieved via cudaMemcpy(). For example, the multi_snn might do:

```cpp
void update() {
    for (int i = 0; i < num_gpus; i++) {
        cudaMemcpy(i);
        _update<<<...>>>(...);
    }
}
```

Now consider that cudaMemcpy() takes ~10μs and our system contains eight GPUs. It would take us 70 μs (70% of our simulation budget) just to launch our kernels, nevermind for them to perform any meaningful computations. Spice alleviates this problem by launching a separate (CPU-)thread per GPU. Since each thread has its own CUDA context, we only need to call cudaMemcpy() once per thread. All subsequent kernel launches are issued to that device at no extra cost. All threads busy-wait until the simulation terminates. However, they do yield() to free up CPU resources when they are waiting for new tasks to arrive. Threads just serve to eliminate the cost of cudaMemcpy()—they are not involved in scheduling. This is done entirely on the GPU via cudaMemcpy() (Figure 9.4).

9.4 API

Spice draws inspiration from modern graphics pipelines such as DirectX and OpenGL. The GPU facilitates efficient transformation, tessellation, rasterization, etc. while allowing the user to customize the appearance of the final image through programmable shaders. Similarly, Spice facilitates efficient spike propagation, plasticity, delays, etc. while allowing the user to customize the behavior of their model through user-defined callbacks. Listing 9.2 shows the complete implementation of a simple SNN consisting of two inter-connected neuron populations that take turns exciting one another. As can be seen, the API does not make any assumptions about the layout and dynamics of neurons/synapses except for the update()→receive() loop. It is also written entirely in C++(17). This has several advantages:

- Any algorithm expressible in terms of the update()→receive() loop, can be simulated in Spice, including non-SNN algorithms such as SSSP (Listing 9.3).
- The user can take advantage of any C++/CUDA features (including any third party libraries) to implement their models.
Established reuse patterns within C++ can be used without restriction: inheritance, composition, etc. If, for example, a provided neuron model does not quite meet the user’s requirements, they can derive from it, specialize some of its behavior, and reuse most.

The code written is the same code that ends up being compiled (by the native toolchain). No proprietary compilation steps are required. Compilers are able to provide detailed warnings and error messages. The code can be statically analyzed, debugged, and profiled (e.g. using NVIDIA Nsight).

For comparison, implementing a model in GeNN [26] involves a few stages and languages. It amounts to providing the neuron and synapse models (using GeNN’s domain-specific language (DSL)), running a proprietary compilation step, which brings everything together in a new translation unit of CUDA C++, and linking against said unit from user code. Model, network size and connectivity parameters have to be provided at compile time, requiring re-compilation for any change.

Spice also aims for API calls to be minimal in count and verbosity. Figure 9.5 compares the coding effort required to implement the Brunel model [86] ex nihilo in both GeNN and Spice.

```cpp
struct myneuron : public neuron<bool>
{
    bool spiked;
    myneuron (bool init) : spiked(init) {}

    template <class Iter>
    void init(Iter self) { get<0>(self) = spiked; }

    template <class Iter>
    bool update(Iter self) {
        bool spike = get<0>(self);
        get<0>(self) = false;
        return spike;
    }

    template <class Iter, class Syn>
    void receive(Iter self, Syn) { get<0>(self) = true; }
};

snn ping_pong(1, 1);

auto A = ping_pong.add<myneuron>(100, true);
auto B = ping_pong.add<myneuron>(100, false);

ping_pong.connect<STATIC>(A, B, fixed_probability(0.01), 0);
ping_pong.connect<STATIC>(B, A, fixed_probability(0.01), 0);

while (true)
    ping_pong.step();
```

Listing 9.2: We define a new neuron type myneuron by inheriting from the neuron base class and communicate our layout through template parameters (line 1). Next we have to implement a series of callbacks to define our neuron’s behavior. We start by implementing init() which will be called once before the simulation and initializes each neuron to their initial state (line 7). We get access to individual neurons through an iterator. Next we implement update() which will be called at every simulation step. If we previously received a spike, we emit one and reset ourselves so we do not spike again until we receive another one (lines 11–13). Lastly, we implement receive() which will be called whenever we receive a spike from another neuron. If that happens, we simply set our flag to true, which will cause us to spike during the next update step (line 17). We instantiate a SNN with a DT and delay of 1 (line 20), create two neuron populations of 100 neurons each (lines 22–23), and connect them with a fixed probability of 0.01 (lines 25–26). Finally, we run our simulation (lines 28–29).
Listing 9.3: The SSSP algorithm inside Spice. The implementation is easiest to understand if we start with the receive() method. Whenever a neuron (vertex) receives a spike from one of its neighbors, it checks whether it discovered a shorter route from the source via said neighbor. If so, it updates its distance and marks itself so its spikes during the next update() step, giving, in turn, its neighbors a chance to also discover shorter routes. update() merely serves to broadcast new routes discovered during receive(). init() is responsible for setting the source’s distance to 0 and marking it, while setting all other vertices’ distances to infinity. Since the longest, acyclic path cannot be more than $|\text{neurons}| - 1$ steps, it is sufficient to run the simulation for that many steps in order to discover shortest paths to all vertices. Notice how simple and intuitive the code is by being able to think locally, rather than having to manage global queues as in Dijkstra for example. In a non-representative test this implementation performed just 3x slower than nvGRAPH. However, nvGRAPH is optimized for road networks which are incredibly sparse while I used a network with a density of 5%.

```cpp
struct sssp : public neuron<float, int, bool> {
    enum ATTR { d, prev, spike };

    int src;
    sssp (int src) : src(src) {} 

    template <class Iter>
    void init(Iter self) {
        if (self.id == src) {
            get<d>(self) = 0;
            get<prev>(self) = src;
            get<spike>(self) = true;
        } else {
            get<d>(self) = FLT_MAX;
            get<spike>(self) = false;
        }
    }

    template <class Iter>
    bool update(Iter self) {
        bool result = get<spike>(self);
        get<spike>(self) = false;
        return result;
    }

    template <class Iter, class Syn>
    void receive(Iter self, Syn edge) {
        if (get<d>(edge.src) + get<w>(edge) < get<d>(self)) {
            get<d>(self) = get<d>(edge.src) + get<w>(edge);
            get<prev> = edge.src.id;
            get<spike>(self) = true;
        }
    }
};
```

9.5 Blueprints

In this section we want to discuss Spice on a physical level. We want to see how its data structures are laid out in memory and how algorithms operate on them step by step. More importantly, we want to obtain a complete picture of Spice and learn how its components (that we discussed individually so far) inter-operate.

Figure 9.6 takes the reader from a parametric SNN description (in the form of actual code), over the induced graph, all the way to its representation in memory. We see how neuron update and spike delivery actually operate.

Figure 9.7 does the same but for the multi-GPU case. We see how a network is physically split in two and what communication is required between GPUs in order to run a distributed simulation.
With an implicit /one.prop:/one.prop mapping between the two (visualized by the stacked arrays behind the adjacency lists).

Starting the simulation step by first delivering last steps’ spikes simplifies code. It also has the added benefit of ending on the neuron update step. Some of our neurons (advancing their dynamics), some of which may fire as a result. Existing entries in the spike lists are shifted down, and new spikes are added to the lists. The colors correspond to the ones in the code listing. Solid edges represent synapses with delay (top right) Graph representation of a possible SNN created from our parameterization (after all, we’re dealing with random numbers). The colors correspond to the ones in the code listing. Solid edges represent synapses with delay and dashed ones with delay. (bottom right) Physical view of our SNN showing how all its entities are actually stored in memory. A neuron population consists of a) the neuron pool (Neurons) which stores the state of the actual neurons using a SoA layout (Section 6.3, visualized by the stacked arrays) and b) the population’s recent firing history in the form of a delayed spike list (Spikes). The lists are conservatively sized so they can accept up to [Neurons] many spikes. The atomic counters representing their sizes are stored contiguously rather than with their respective lists so they can be efficiently zero’ed with a single API call. The layout of populations B and C is identical, hence only their neuron pools are shown for brevity. Every synaptic connection consists of multiple adjacency lists, one for each delay present in the network (3 and 5 in our case). This seeming fragmentation has the benefit of ensuring that any adjacency list is only responsible for delivering spikes from a single list originating from a single neuron population, over a single synapse and connectivity type, to a single target neuron population. This homogeneity greatly reduces implementation complexity (loops and branches become obsolete) and increases storage efficiency. The synapse attributes are once again stored using a SoA layout. The synapse pools have the same size as the adjacency lists.

Neurons

Spikes

Step /1

Step /2

delay = 3

delay = 3

delay = 5

delay = 5

Figure 9.6 (top left) We create a SNN with three neuron populations with 4, 2, and 2 neurons respectively (lines 3–5). In addition to size, we also specify their types and initialization parameters (“...”). We connect neuron populations A and B via static synapses (lines 7–10) which have a fixed weight, and populations A and C via dynamic synapses (lines 12–15) whose weight changes according to spike-timing-dependent plasticity (STDP). Both synaptic connections have a connection probability of 0.5; that is, each of the 16 potential connections between A and B/C has a 50% probability of coming into existence. Furthermore, two distinct delays, 3 and 5, are uniformly distributed across the synapses. (top right) Graph representation of a possible SNN created from our parameterization (after all, we’re dealing with random numbers). The colors correspond to the ones in the code listing. Solid edges represent synapses with delay and dashed ones with delay. (bottom right) Physical view of our SNN showing how all its entities are actually stored in memory. A neuron population consists of a) the neuron pool (Neurons) which stores the state of the actual neurons using a SoA layout (Section 6.3, visualized by the stacked arrays) and b) the population’s recent firing history in the form of a delayed spike list (Spikes). The lists are conservatively sized so they can accept up to [Neurons] many spikes. The atomic counters representing their sizes are stored contiguously rather than with their respective lists so they can be efficiently zero’ed with a single API call. The layout of populations B and C is identical, hence only their neuron pools are shown for brevity. Every synaptic connection consists of multiple adjacency lists, one for each delay present in the network (3 and 5 in our case). This seeming fragmentation has the benefit of ensuring that any adjacency list is only responsible for delivering spikes from a single list originating from a single neuron population, over a single synapse and connectivity type, to a single target neuron population. This homogeneity greatly reduces implementation complexity (loops and branches become obsolete) and increases storage efficiency. The synapse attributes are once again stored using a SoA layout. The synapse pools have the same size as the adjacency lists.

Neurons

Spikes

Counters

Figure 9.6 (top left) We create a SNN with three neuron populations with 4, 2, and 2 neurons respectively (lines 3–5). In addition to size, we also specify their types and initialization parameters (“...”). We connect neuron populations A and B via static synapses (lines 7–10) which have a fixed weight, and populations A and C via dynamic synapses (lines 12–15) whose weight changes according to spike-timing-dependent plasticity (STDP). Both synaptic connections have a connection probability of 0.5; that is, each of the 16 potential connections between A and B/C has a 50% probability of coming into existence. Furthermore, two distinct delays, 3 and 5, are uniformly distributed across the synapses. (top right) Graph representation of a possible SNN created from our parameterization (after all, we’re dealing with random numbers). The colors correspond to the ones in the code listing. Solid edges represent synapses with delay and dashed ones with delay. (bottom right) Physical view of our SNN showing how all its entities are actually stored in memory. A neuron population consists of a) the neuron pool (Neurons) which stores the state of the actual neurons using a SoA layout (Section 6.3, visualized by the stacked arrays) and b) the population’s recent firing history in the form of a delayed spike list (Spikes). The lists are conservatively sized so they can accept up to [Neurons] many spikes. The atomic counters representing their sizes are stored contiguously rather than with their respective lists so they can be efficiently zero’ed with a single API call. The layout of populations B and C is identical, hence only their neuron pools are shown for brevity. Every synaptic connection consists of multiple adjacency lists, one for each delay present in the network (3 and 5 in our case). This seeming fragmentation has the benefit of ensuring that any adjacency list is only responsible for delivering spikes from a single list originating from a single neuron population, over a single synapse and connectivity type, to a single target neuron population. This homogeneity greatly reduces implementation complexity (loops and branches become obsolete) and increases storage efficiency. The synapse attributes are once again stored using a SoA layout. The synapse pools have the same size as the adjacency lists.

Neurons

Spikes

Counters

Figure 9.6 (top left) We create a SNN with three neuron populations with 4, 2, and 2 neurons respectively (lines 3–5). In addition to size, we also specify their types and initialization parameters (“...”). We connect neuron populations A and B via static synapses (lines 7–10) which have a fixed weight, and populations A and C via dynamic synapses (lines 12–15) whose weight changes according to spike-timing-dependent plasticity (STDP). Both synaptic connections have a connection probability of 0.5; that is, each of the 16 potential connections between A and B/C has a 50% probability of coming into existence. Furthermore, two distinct delays, 3 and 5, are uniformly distributed across the synapses. (top right) Graph representation of a possible SNN created from our parameterization (after all, we’re dealing with random numbers). The colors correspond to the ones in the code listing. Solid edges represent synapses with delay and dashed ones with delay. (bottom right) Physical view of our SNN showing how all its entities are actually stored in memory. A neuron population consists of a) the neuron pool (Neurons) which stores the state of the actual neurons using a SoA layout (Section 6.3, visualized by the stacked arrays) and b) the population’s recent firing history in the form of a delayed spike list (Spikes). The lists are conservatively sized so they can accept up to [Neurons] many spikes. The atomic counters representing their sizes are stored contiguously rather than with their respective lists so they can be efficiently zero’ed with a single API call. The layout of populations B and C is identical, hence only their neuron pools are shown for brevity. Every synaptic connection consists of multiple adjacency lists, one for each delay present in the network (3 and 5 in our case). This seeming fragmentation has the benefit of ensuring that any adjacency list is only responsible for delivering spikes from a single list originating from a single neuron population, over a single synapse and connectivity type, to a single target neuron population. This homogeneity greatly reduces implementation complexity (loops and branches become obsolete) and increases storage efficiency. The synapse attributes are once again stored using a SoA layout. The synapse pools have the same size as the adjacency lists.
We start with the same SNN as in Figure 9.6 (top right). It consists of three neuron populations (yellow, green, blue) with 4, 2, and 2 neurons respectively. They are connected by two synapse types (orange and purple). Synapses can have a delay of 1 (solid lines) or 3 (dashed lines). (top right) The network is distributed across two GPUs. Each neuron population is split in two with the first half assigned to GPU A and the second half assigned to GPU B. In addition to its respective halves, each GPU also stores every synapse pointing to any of its neurons. It only stores an entry in the adjacency list and the accompanying synapse state though, not necessarily the source neuron which may reside on a different GPU (depicted as small ghost neurons). (bottom left) Physical view of GPU A's portion of the network. Since each portion forms an intact SNN in its own right and we already know how to map a SNN to memory, it is of no surprise that this physical view is close to identical to that of Figure 9.6 (bottom right). There are only two differences: First, we introduce an offset. The single-GPU version has the added benefit that a neuron's index happens to be its ID by which we can look up its neighbors in the adjacency lists. However, we need not store an ID per neuron; a simple offset that is added to all neuron indices before they are written to the spike lists is enough. For example, in GPU A's slice of the yellow population, the indices coincide with the IDs, so its offset is 0. GPU B was assigned the second half of the yellow population. Since its neurons are preceded by two others (residing on GPU A), its offset is 2. Second, the spike lists are larger than the neuron populations. In fact, they are large enough to hold spikes from all slices combined. This is necessary for the spike synchronization step in the multi-GPU pipeline (Section 8.2). The simulation works the same as in the single-GPU case. However, after neurons have been updated and spikes generated, but not yet delivered, they are synchronized—the union of all $C_i$ lists per neuron population is computed and stored in each GPU. Then they are delivered as before. (bottom right) Same as (bottom left) but for GPU B. Some adjacency lists have all-∞ entries, i.e., they are empty. For example, GPU B’s orange list with delay 3. If we study the graph top right we notice that no dashed, orange synapses are pointing to any of GPU B’s neurons. In practice, empty adjacency lists would not be stored in the first place, as opposed to filling them up with sentinels. They are included here to maintain the symmetry of the diagram and make it easier to parse.
Results & Discussion
We have already studied the relative performance impacts of algorithmic choices and optimizations in Chapters 7–8, where they were introduced. In this chapter we want to analyze Spice’s absolute performance and compare it to that of other state of the art simulators, to see where we stand in the grand scheme of things.

### 10.1 Competitors

We compare Spice against three other, state of the art, clock-driven simulators. Event-driven simulators are not included in the comparison as the design goals and use cases are too different, as explained in Chapter 4. Subsequently, we briefly present and motivate each competitor—they have been covered in depth in Chapter 5.

**BSim** [35] is a recent, high-performance simulator. It is the only other simulator besides Spice and CARLSim [55] with multi-GPU support that also achieves halfway decent scaling. Since multi-GPU scaling is both crucial for advancing SNN research (as I have argued in Chapter 8) and a major selling point of Spice, a comparison is a must.

**GeNN** [26] is a well-established, high-performance, semi-generic simulator. It is actively maintained, has a wide user base, and is accompanied by several publications.

**NeuronGPU** [32] is a novel simulator that aims to maximize biological fidelity above everything else, while still offering a relatively fast GPU implementation. To this end, it supports double precision arithmetic and _exact integration_ [87] versus single precision arithmetic and Euler integration used by other simulators. NeuronGPU highlights how many design choices exist even within the clock-driven domain. Naturally, it will not perform on par with the rest of the simulators—it is included to demonstrate how much performance has to be sacrificed for higher biological accuracy.

### 10.2 Models

The performance of all simulators was evaluated on adaptions by Ahmad et al. [27] of three well-established SNN models:

- **Vogels-Abbott** [88], described in depth in [27, Appendix A]. Referred to as Vogels.
- **Brunel** [86], described in depth in [27, Appendix B]. Referred to as Brunel.
- **Brunel with plasticity turned on**, referred to as Brunel+.
Ahmad et al. tuned their adaptions for particular, fixed network sizes. Since we want to be able to vary network size, we need to scale synaptic weights by a factor $c$: \[ c_{\text{Vogels}} = \frac{16,000,000}{|\text{neurons}|^2} \quad c_{\text{Brunel}(+)} = \frac{20,000}{|\text{neurons}|} \]

As can be seen, substituting the original network sizes results in $c = 1$ and thus has no effect. For larger network sizes, scaling synaptic weights by $c$ ensures that the models retain their characteristic firing patterns and rates. Both models’ topologies and firing patterns are visualized in Figure 10.1.

### 10.3 Hardware & Benchmarking Methodology

All benchmarks were carried out on a dedicated, headless Google Cloud server with specifications according to Table 10.1. The CUDA runtime had exclusive access to the GPU driver. Therefore, even though we are not dealing with a real-time operating system, measurement outliers were rare and small in magnitude ($<1\%$). They were removed by running every experiment $3\times$ and reporting the median measurement.

### 10.4 Software

Spice, GeNN, and BSim employ single precision arithmetic and Euler integration while NeuronGPU employs double precision arithmetic and exact integration. The experiments use NeuronGPU’s Python interface and the other simulators’ C++ interfaces.
BSim suffered from race conditions due to implicitly assuming all threads in a CUDA warp executed in lockstep—an assumption that Volta invalidated with its independent thread scheduling feature. The bug was fixed without altering performance.

GeNN offers the choice between three different connectivity types: \texttt{SPARSE\_GLOBALG} is similar to an adjacency list, \texttt{BITMASK\_GLOBALG} is similar to a binary adjacency matrix, and \texttt{PROCEDURAL\_GLOBALG} does not store the graph at all but generates it on the fly. It turned out that \texttt{SPARSE\_GLOBALG} is faster for Vogels while \texttt{BITMASK\_GLOBALG} is faster for Brunel(+) so that is what the experiments used. The latter is also more memory efficient for dense networks, allowing GeNN to simulate much larger instances of Brunel(+).

Spice had all available algorithmic optimizations enabled to obtain its final performance numbers. For the impact of each individual optimization please see Chapters 7–8.

All the code used for the experiments can be found at:

- **BSim** github.com/denniskb/bsim, forked from master as of Feb 19, 2020.
- **GeNN** github.com/denniskb/genn, forked from tag “GeNN 4.4.0” as of Jan 5, 2021.
- **Spice** github.com/denniskb/spice/releases/tag/hpec2021.

### 10.5 Performance metrics and obtained results

#### Setup Time

We measure the time in seconds it takes to initialize Brunel for various network sizes (Figure 10.2(\textit{left})). Both axes are logarithmic. BSim and Spice, the only two multi-GPU capable simulators, are represented by a series of lines—one for each GPU configuration. GeNN is represented by two lines: the solid one represents the bare initialization time while the dashed one represents initialization + compilation. It is included because in GeNN, as opposed to the other simulators, \textit{any} parameter change requires re-compilation. Since parameter tuning is an important
Figure 10.3 Single-GPU simulation time as a function of network size for each model: We measure the time it takes to simulate 10 s of biological time for various synapse counts and report wall-clock time ÷ biological time. Note that in the Brunel(+) case both axes are logarithmic.

part of the training process, (re-)compilation should be counted towards GeNN’s setup time in my opinion.

We see that GeNN and Spice are lightning fast compared to BSim and NeuronGPU because they perform setup on the GPU. Spice is strictly faster than GeNN but not by a margin that would be considered a deal-breaker in practice. We also see that Spice’s initialization time is virtually constant with respect to network size. That is because it is entirely dominated by memory allocations and (CPU) thread launches. The actual setup kernel generates networks at ~200M synapses/ms.

BSim and NeuronGPU, as many other simulators, initialize on the CPU which is why they take in the order of minutes. The general sentiment seems to be that setup time is negligible because it only has to be carried out once. This is a fallacy, in my opinion. Fast setup is equally as important as fast simulation—it dictates how quickly we can iterate on our network designs or tune parameters and determines the efficiency of our experiments (fraction of total time spent in simulation). Let me put the numbers into perspective to illustrate.

1. If our experiments ran in the order of minutes as well, we would only achieve 50% efficiency—the simulator’s effective performance would be halved.
2. If we wanted to achieve >99% efficiency, we would have to run our experiments for dozens of hours—most SNN simulations are short-lived.
3. In the time it takes the competition to initialize one model, Spice can initialize and run ten 1-minute long experiments.

BSim and NeuronGPU also use very memory-inefficient intermediate data structures before compacting and uploading them to the GPU. This results in very high peak RAM usage (Figure 10.2(right)) which may be prohibitive for some users.

Simulation Time

We measure the time it takes to simulate 10 s of biological time for various network sizes (synapse counts) and models. We report wall-clock time ÷ biological time (Figure 10.3), i.e., the real-time-ness. Anything below 1 runs
faster than real-time, anything above 1 slower. Note that in the Brunel(+) plots both axes are logarithmic.

BSim and NeuronGPU are missing from the Brunel+ plot because, while both support STDP, their synapses’ behavior does now quite match that of Brunel+. According to the authors, modifying it “currently is a task for developers, not for users.”

All simulators scale close to linearly (with respect to synapse count), which is commendable. Notice that, not only is Spice ~2× faster than its closest competitor, that competitor changes from benchmark to benchmark, making Spice, not only the fastest, but also the most consistent simulator.

NeuronGPU uses double precision arithmetic and exact integration as opposed to single precision arithmetic and Euler integration used by the other simulators—it is not expected to perform on par but is included for completeness, to gauge how much performance one has to sacrifice for biological fidelity.

**Multi-GPU Scaling**

We want to study how BSim and Spice scale with multiple GPUs on various models. For this purpose we differentiate between two scenarios:

- **Scaleup** (Figure 10.4): By what factor can we increase the network size (synapse count) while maintaining single-GPU simulation time?
- **Speedup** (Figure 10.5): How much faster will the same network run on multiple GPUs?

Note that the bars are normalized with respect to the single-GPU column only, not with respect to each other (obviously Brunel and Brunel+ do not have the same simulation time as we saw in previous plots).
We see that Spice performs close to perfectly in the scaleup scenario and still offers decent improvements in the speedup scenario. The same plots have been presented and analyzed in depth in Section 8.5; they are repeated here for easier comparison with BSim only.

BSim, unfortunately, performs slower on multiple GPUs compared to a single GPU in our benchmarks, most likely due to poor load balancing which distributes the majority of the network to a singular GPU, coupled with the synchronization overhead introduced by multiple GPUs. According to the authors “(BSim) works well for huge networks with a large number of populations but does not perform well with small number of populations, especially when the numbers of neurons in each population are imbalanced.”\(^2\) Our benchmarks fall into this category. On models that do favor BSim, the authors report a speedup of 1.6×–2.3× using 4 GPUs [35, Fig. 7].

Please also note that, probably due to the same load balancing issues, BSim does not scale in space at all in our benchmarks. Its maximum network size is capped at 1.8B synapses regardless of the number of GPUs (Figure 10.2).
11.1 Achievements

Spice, a modern clock-driven SNN simulator was presented. Its key achievements are:

- **Performance.** Spice is not only the fastest clock-driven simulator by a substantial margin, but also the most consistent. While it is more than 2× faster than its closest competitor, that competitor changes from benchmark to benchmark. Furthermore, Spice offers lightning fast setup, an often neglected yet crucial feature: In the time it takes competitors to initialize a SNN with 2B synapses, Spice is able to run 40 different, 10-s long experiments in real time.

- **Multi-GPU Scaling.** Spice is the first simulator to scale to 8 GPUs at 100% efficiency in both space and time. Its closest competitor achieves a little over 50% efficiency on 4 GPUs.

- **Generality.** Spice is fully generic. Users can define arbitrary models with fully programmable dynamics. Furthermore, Spice makes no SNN-specific assumptions such as neurons having potentials or synapses having weights, making it incredibly flexible (allowing, for example, the implementation of graph algorithms).

- **Usability.** Spice offers a modern, simple, and concise API. Both its API and user-defined models are written entirely in C++/seven.prop, absolving us from the need for DSLs or proprietary compilation steps and enabling us to re-use the vast body of C++ and CUDA libraries. It has no third-party dependencies other than CUDA, making building it easy.

In my opinion, even more important than any of these individual accomplishments, Spice demonstrates that:

| Performance, generality, and usability are not diametrically opposed goals. |

This is neither expected nor intuitive, and certainly not supported by the collective software engineering experience. Both from intuition and experience, we regard at least two of all three pairs as opposing goals. Let me illustrate:

- **Generality ↔ Performance.** Typically, as a framework/library becomes more generic, it cannot assume domain-specific knowledge or employ domain-specific optimizations. Take for example a concrete SSSP implementation vs. a graph processing framework that must run any user-defined algorithm (including SSSP).
Figure 11.1  Strengths and weaknesses

**Performance** proportional to plots

**Generality**
1. parameterized neurons/synapses
2. dynamics partially programmable
3. dynamics fully programmable

**Biological accuracy**
1. general principle of spiking neurons
2. double precision/higher order integration/spikes as currents
3. exact (as long as the minimum delay is greater than the time step)

**Code Safety** inverse of generality

**Usability** API verbosity + author’s opinion/experience

► **Generality ↔ Usability.** Typically, as a framework/library becomes more generic, its API becomes more verbose. As it cannot assume what it will be used for, it has to rely on generic building blocks, lots of which need to be assembled into higher-level functionality. Take for example a 3D graphics API such as OpenGL vs. a general-purpose game engine vs. a genre-specific game engine: writing a roleplaying game, for example, becomes increasingly easier as the tools become tailored to the specific task at hand.

### 11.2 Shortcomings

Generality does come at the cost of code safety (the likelihood of shooting oneself in the foot) though. Having a constrained API has the benefit of having limited ability to abuse said API. In the case of BSim and NeuronGPU, for example, the worst that can happen is a wrong parameterization which should immediately become obvious during the simulation. In the case of Spice the user has to remember to *manually synchronize* target neurons during spike delivery. Failing to do so results in *non-deterministic* bugs that are incredibly hard to detect. Not to mention that the ability to use arbitrary third-party code creates an equal opportunity for arbitrary compilation/linking errors. In this sense, Spice offers its user exactly the same level of protection (or lack thereof) as its host language, C++. Figure 11.1 summarizes the most important strengths and weaknesses of the evaluated simulators.

### 11.3 Generalizability of Results

Conducting experiments is a major effort with quadratic time complexity (every model has to be implemented in each simulator). The four simulators and three models presented in Chapter 10 reached our maximum capacity in that regard. Within this capacity we wanted to maximize the representativeness of the results. In the case of simulators this was rather simple due to the manageable landscape. I feel confident that the
simulators we tested form a representative cross section of the state of the art in SNN simulation, published or not.

When it comes to models this is much harder due to their sheer number. We chose popular biological models that have been widely used in the past for benchmarking purposes so as to make our results comparable to simulators outside this study, by transitivity. Nevertheless, our models represent a tiny fraction of the dozens of popular models and infinitely more definable ones. Not only were they used for the final comparison, but they were also used during development to judge alternative implementations. And while Spice does not contain any model-specific optimizations or code paths, coding against one’s own benchmarks at least on a subconscious level cannot be avoided. As a result, Spice might very well perform worse than other simulators on radically different models (firing pattern/activity/density) from the tested ones, simply because it has not been exposed to them before. But I am also confident that any such potential performance gaps could be closed with minor tweaks.

11.4 Time to Market

Another important yet often neglected discussion is the maturity of the implementation. As is the nature of research, it often produces proof-of-concept prototypes which require extensive refactoring to be turned into production-level software. The most important part of this process is hardening where software is made robust against crashes (due to memory leaks for example), malformed user inputs, hardware failures, etc. Hardening is almost completely missing from research prototypes as their controlled environment does not require it—they often only need to run on a single machine with a fixed (software) configuration processing a single data set. Hardening is a non-trivial task that can add significant computational cost to an implementation [84], rendering results invalid. The most famous example is a vector (a resizable array) which, by definition, has to double its memory consumption if it wants to offer the “strong (exception) guarantee” [84, p. 72].

Spice’s authors have already gone to extreme lengths to ensure it is correct and robust:

- Spice’s output cannot directly be compared to ground truth data (outputs produced by well-established simulators that are generally accepted as correct) due to generating random numbers in a fundamentally different way. Instead, Spice’s developers wrote a naïve but trivial CPU implementation (whose core logic comprises some 20 LOC). Independent, unbiased developers audited it and its output was compared quantitatively and qualitatively to ground truth data with which it agreed within several decimal places. Finally, the developers asserted with tests that Spice’s GPU implementation is bit-for-bit identical with the CPU version.
- Spice has near full test coverage, comprised of unit, integration, and (performance) regression tests.
- Spice’s entire API offers the strong exception guarantee.

While Spice will require some refactoring before it can be turned into production-level software, this refactoring is of mostly cosmetic nature
(cleaning up code, re-organizing class hierarchy and structure, writing documentation, etc.).

**Portability.** Spice targets CUDA primarily because its developers were already familiar with it. While CUDA does have some exclusive features compared to OpenCL, Spice does not overly rely on them. In fact, warp-level primitives are the only CUDA-exclusive feature it uses for its setup code, which could easily be replaced by a slightly slower shared memory-based implementation. Thus, a port to OpenCL (and hence non-Nvidia GPUs) should be fairly easy.

### 11.5 Future Work

**Spice**

Spice’s adjacency list partitioning (Section 7.3) lends itself to a simple block compression scheme: Since at most 1024 neurons are being indexed at any given time, 10 bits are enough to uniquely identify each one. If we stored three consecutive indices in a single 4-B word (instead of one index per word right now), we could reduce the memory footprint of our graph representation by two thirds, regardless of network density.

Even though Spice already scales impressively with multiple GPUs, it could perform better in the speedup scenario (Section 8.5). The key to achieve this is to speed up the spike synchronization step which is bottlenecked by CUDA API overhead (Chapter 9). Early tests suggest that it might be faster to do this on the CPU by downloading the last delay \( \div 2 \) steps’ spikes, computing their union on the CPU, and uploading them again. While this involves going through slow RAM and congesting the PICe bus, it dramatically cuts down the number of API calls, by a factor of delay. Spikes would need to be stored compactly rather than in lists per simulation step.

**Tensor cores** are a recent feature of Nvidia GPUs. These are hardware units optimized for performing fused matrix multiplication and addition. They are orders of magnitude faster than software implementations. If we found a way to express neuron/synapse dynamics as matrix multiplications (at least in parts since most dynamics are non-linear), we could potentially dramatically speed up plastic models. Whether the benefit of faster calculations outweighs the cost of transforming the data into and from the matrix form, needs to be investigated.

**Field**

As mentioned in Section 11.2 there is a lack of agreement on models for benchmarking the performance of SNN simulators. I think the field as a whole would greatly benefit from having a universal standard to measure simulators, similarly to how 3DMark is used to gauge the gaming performance of desktop PCs. I may tackle the issue of developing such a benchmark suite myself and would like to encourage fellow researchers to do the same.
Another pressing issue is the lack of metrics for quantifying the biological accuracy of SNN simulators. In the pursuit of biological accuracy, developers reach for tools such as double precision arithmetic, higher order integration, or event-driven simulation. Intuitively this makes sense and part of the calculations becomes measurably more accurate as a result, but we cannot quantify the impact on real-world performance (whatever task the SNN is used for). While we can compare spike trains of individual neurons, we lack a metric that can say “this network is 90% biologically accurate” where the “90%” has a well-defined, sensible, and tangible meaning. As an intermediate band aid we could employ a more holistic way of comparing algorithms: Run the same real-world model on two different simulators and compare their performance (classification/prediction accuracy/etc.). Unfortunately, this approach has flaws as well—the results may not generalize to other SNN architectures or tasks.
References


Special Terms

A
ANN  Artificial neural network. Typically layer-based, fully connected, and produces continuous outputs.. 3
AoS  Array of structs. A collection of homogenous objects, grouped by the objects. Objects are stored contiguously in memory as \{obj1, obj2, ..., objN\}.. 30

C
CNN  Convolutional neural network. ANN with special “convolutional” layers mimicking the human retina.. xi
CSR  Compressed sparse row. A dense representation of adjacency lists consisting of a neighbor array and an offset table marking the beginning of each node’s neighbor list.. 15
CUDA  A programming framework and language for GPGPU computing on Nvidia GPUs.. 27
CUDA block  A logical ensemble of up to 1024 CUDA threads. The threads in a block share their SM’s L1 cache.. 42
CUDA core  A simple processing unit found on GPUs mostly capable of executing arithmetic instructions. In contrast to CPU cores, CUDA cores lack cache and most of the control unit.. 28
CUDA thread  A logical thread of execution that can be scheduled to a physical CUDA core.. 28

G
global memory  CUDA's terminology for a GPU’s RAM.. 29
GPGPU  General purpose GPU (computing). The practice of performing generic computations on graphics processors.. 21

P
PC  Program counter a.k.a instruction pointer (IP). A hardware register used to keep track where a processor is in its program’s execution.. 27

S
shared memory  CUDA’s terminology for a GPU’s cache. In contrast to the CPU, CUDA allows the manual placement of data inside the cache.. 31
SM  Streaming multiprocessor. The main building block of GPUs, consisting of L1 cache, control unit, and 32 simple cores. SMs share the GPU’s L2 cache and RAM.. 28
SNN  Spiking neural network. Generalization of ANNs. Typically randomly connected with discrete outputs.. 3
SoA  Structure of Arrays. A collection of homogenous objects grouped by the objects’ attributes. Objects are stored contiguously in memory as \{obj1.attr1, obj2.attr1, ..., objN.attr1, obj1.attr2, obj2.attr2, ..., objN.attr2, ..., objN.attrN\}.. 16
SP  Stack pointer. A hardware register that, together with the stack, enables the implementation of function calls.. 27
spike  Signal emitted by a neuron. Must be communicated to the neuron’s neighbors.. 3
spike train  A neuron’s firing pattern (the timings of its spikes).. 7
STDP  Spike-timing-dependent plasticity. A type of neuroplasticity, a learning mechanism of SNNs.. 11
synaptic delay  A spike’s travel duration through a synapse.. 21

W
warp  A group of 32 CUDA threads that is typically scheduled to the 32 physical cores of a SM. Threads of a warp can efficiently communicate with each other via “warp-level primitives”.. 28
Alphabetical Index

(neuro)plasticity, 9, 10, 17, 44
event-driven, 40, 45
lazy, 40, 45
STDP, 10, 69

adjacency list, 15, 37, 42
adjacency matrix, 15
Amdahl’s Law, 49
ANN, 3, 4
AoS, 30, 55
API, 19, 53
declarative, 19
overhead, 33, 58
procedural, 19
assertion, 55

batch, 49
benchmark, 53, 66, 74
circular buffer, 39, 57
CLI, 53
code safety, 72
composite pattern, 56
compression, 74
CSR, 15, 37, 38
CUDA, 27, 74
asynchronicity, 32
block, 42, 44
divergence, 31
latency, 33
P2P, 34
thread, 28, 43
warp, 28
cudaDeviceSynchronize(), 32
cudaEventCreate(), 58
cudaEventRecord(), 58
cudaMemcpy(), 33, 48, 58
cudaMemset(), 57
cudaSetDevice(), 58
cudaStreamCreate(), 58
cudaStreamSynchronize(), 58
cudaStreamWaitEvent(), 58
data race, 39, 43
DLL, 53

DSL, 19
dynamics, 7, 55
closed-form, 14, 46
exception safety, 55, 73
firing pattern, 7, 66

GPGPU, 27
hardware, 66
analog, 25
CPU, 20, 27, 53, 55
GPU, 20, 28, 53, 55
neuromorphic, 20

iterator, 56

latency, 57
hiding, 29, 49
load balancing, 50
static, 50

memory
bandwidth, 29
cache, 31, 41, 42
coalescing, 29, 30, 40
global, 29, 40
latency, 29
shared, 31, 40–42
model, 7, 65

neuron, 7, 40, 43
connectivity, 8
LIF, 8
population, 7
potential, 8
refractory period, 8

OpenCL, 27, 74
parallelization, 47, 71
strategy, 47
partition, 42, 47
pipeline, 39, 47
portability, 74

scaleup, 51, 69, 71

simulator, 13
biological accuracy, 19
clock-driven, 21
customizability, 17, 71
event-driven, 22
hybrid, 22
neuromorphic, 24
parallel, 22
SMT, 27, 28
SNN, 3, 4, 7
SoA, 30, 37, 55
speedup, 51, 69
Spice, 37
API, 58, 71
initialization, 38, 68
spike, 7, 40
post-synaptic, 7, 44
pre-synaptic, 7, 44, 46
synchronization, 74
synapse, 7, 44
delay, 7, 16, 39, 43, 49
non-instantaneous, 16
synchronization, 47, 48
tensor cores, 74
Timestep Grouping, 43
training, 3, 7, 9
Backpropagation, 10, 50
Hebbian Learning, 10
wake-sleep algorithm, 9

unit test, 53, 54, 73
Vulkan, 27