

Fast Algorithms for Spiking Neural Network Simulation with FPGAs

Björn A. Lindqvist and Artur Podobas
KTH Royal Institute of Technology
Stockholm, Sweden

May 6, 2024

Abstract

Using OpenCL-based high-level synthesis, we create a number of spiking neural network (SNN) simulators for the Potjans-Diesmann cortical microcircuit for a high-end Field-Programmable Gate Array (FPGA). Our best simulators simulate the circuit 25% faster than real-time, require less than 21 nJ per synaptic event, and are bottle-necked by the device’s on-chip memory. Speed-wise they compare favorably to the state-of-the-art GPU-based simulators and their energy usage is lower than any other published result. This result is the first for simulating the circuit on a single hardware accelerator. We also extensively analyze the techniques and algorithms we implement our simulators with, many of which can be realized on other types of hardware. Thus, this article is of interest to any researcher or practitioner interested in efficient SNN simulation, whether they target FPGAs or not.

Keywords: cortical microcircuit, fpga, hls, hpc, opencl, simulation, spiking neural networks

1 Introduction

Makin (2019) outline what they see as the four biggest challenges for human brain simulation; scale, complexity, speed, and integration. Scale refers to the enormous size of the brain – billions of neurons and trillions synapses – which is difficult to simulate at acceptable speeds even for the fastest super-

computers. There are two reasons for this difficulty. First, the conceptual mismatch between how brains and computers operate; the former can be viewed as collections of billions of processing nodes, communicating with sparse events called spikes (Ghosh-Dastidar and Adeli, 2009), while the latter is composed of imperative programs, vulnerable to the von Neumann bottleneck (Efimushcheva et al., 2017), and executed on general-purpose devices (GPUs or GPUs). Power *in-efficiency* is the second reason. General-purpose devices are jacks of all trades and can execute a wide variety of workloads, ranging from word processing to scientific simulations. Their generality comes at a high price and executing a single instruction consumes up to three orders of magnitude more energy than the computation itself (Jouppi et al., 2018). This causes brain simulations to consume far more energy than the roughly 20 Watts a human brain draws (Versace and Chandler, 2010). Finally, the 2004 end of Dennard’s scaling (Bohr, 2007), and the impending termination of Moore’s law (Theis and Wong, 2017), forces researchers to reconsider how to compute efficiently in a post-Moore future. Among the more promising options in a post-Moore world is the use of reconfigurable systems such as Field-Programmable Gate Arrays (FPGAs) (Kuon et al., 2008).

FPGAs, along with their siblings, Coarse-Grained Reconfigurable Systems, CGRAs (Podobas et al., 2020), belong to the reconfigurable family of computing devices. Unlike traditional general-purpose pro-

processors (CPUs and GPUs), their underlying compute fabric is composed of a large number of reconfigurable blocks of different types. The most common blocks are look-up tables (LUTs, often several hundred thousands), digital signal processing (DSPs – capable of tens of TFLOP/s), or on-chip random-access memory (BRAM – tens-to-hundreds of MB) (Langhammer et al., 2021; Murphy and Fu, 2017). These resources allow designers to create custom hardware that sacrifice generality for better performance and lower energy consumption than general-purpose devices, as well as transcending the latter’s limitations (i.e., the mentioned von Neumann-bottleneck). Importantly, these devices can, and often already are (Sano et al., 2023; Meyer et al., 2023; Boku et al., 2022), live side-by-side in HPC nodes and can be reconfigured before runtime (Vipin and Fahmy, 2018): when the user is running a brain simulation, the FPGA will be configured as an efficient brain simulator, while for a different application a different accelerator will be used. In short, FPGA facilitates the use of special hardware accelerators during application runtime but is general enough so that different accelerators can be configured between applications. This makes them an attractive choice for neuroscience simulation since they can be reconfigured for different neuron, synapse, and axon models, which dedicated neuromorphic ASIC hardware, whose silicon is immutable, cannot be.

Historically, FPGAs have been designed using low-level hardware description languages (HDLs) such as VHDL and Verilog (Perry, 2002). These languages have a steep learning curve and require specialist knowledge to be comfortable with. However, with the increased maturity of High-Level Synthesis (HLS) tools in the last two decades (Nane et al., 2015), there has been a resurgence of interest in using FPGAs for HPC. HLS allow designers to describe hardware in relatively high-level languages such as C and C++ whose learning curves are shallower (Podobas et al., 2017; Zohouri et al., 2016). For example, FPGAs have been used to accelerate Computational Fluid Dynamics (Karp et al., 2021; Faj et al., 2023), Quantum circuit simulations (Podobas, 2023; Aminian et al., 2008), Molecular Dynamics (Sanaullah and Herbordt, 2018), N-Body systems (Del Sozzo et al., 2018; Menzel et al., 2021; Huthmann et al., 2019),

and much more, demonstrating advantages over alternative solutions.

In this work, we use of HLS to design simulators for the Potjans-Diesmann cortical microcircuit (Potjans and Diesmann, 2014). While there is ample prior work on FPGA-based neuromorphic systems (see section 5.4 for related work), *our system is (to the best of our knowledge) the most energy-efficient simulator of the Potjans-Diesmann circuit in existence (25 nJ/event), while reaching a faster-than-realtime ($\approx 1.2x$) simulation speed on a single FPGA*. We use the Intel’s OpenCL SDK for FPGA HLS toolchain (Czajkowski et al., 2012) to design our simulators, but our designs are modular enough to easily be ported to other HLS-based systems (e.g., Vivado (O’Loughlin et al., 2014) or OneAPI), and other FPGAs. Our contributions are:

- The first simulators of the previously mentioned circuit on a single FPGA, running faster than real-time.
- The most energy-efficient simulators for the circuit when measured by energy per synaptic event.
- The presentation and analysis of the algorithms, thought processes, trade-offs, and lessons learned, while designing these simulators.
- An empirically motivated analysis on what hardware features are required to simulate the circuit even faster than what we are capable of.

The rest of this article is structured as follows. In section 2, we discuss SNNs in general and the microcircuit in particular. We explain how FPGAs work and we briefly introduce the HLS design methodology. In section 3 we discuss SNN simulation and present the algorithms and ideas underlying our simulators. The utility of many of the ideas have already been demonstrated in other parts of Computer Science, but not in connection with SNN simulation. Hence, we believe they deserve a thorough treatment. We evaluate many different variants and parametrizations of our simulators in section 4. Finally, in section 5 we put our results in perspective and compare them with the state-of-the-art.

2 Material and Methods

We begin with an overview of spiking neural networks (SNNs) before discussing the Potjans-Diesmann cortical microcircuit, an SNN for simulating a small part (microcircuit) of the mammalian brain. In section 2.3 we explain how FPGAs work and what makes them different from conventional hardware. In sections 2.4 to 2.6 we introduce HLS and how we use OpenCL for HLS.

2.1 Spiking Neural Networks

An SNN is an artificial neural network (ANN) that transfers signals in time-dependent bursts, i.e. spikes. Unlike other ANNs, SNNs are designed with biological plausibility in mind, making them useful for neuroscience. SNNs are usually modelled as directed (multi-)graphs, where vertices represent neurons and edges synaptic connections between neurons. Neurons have a membrane potential that varies over time. When the potential exceeds a threshold the neuron discharges – spikes – and sends current via its synapses to its neighbours which they receive after a synapse-specific delay. The amount of current as well as the transfer time is synapse-specific (Han et al., 2020). The neuron’s statefulness and the non-differentiable, discontinuous signal transfer function are two fundamental aspects distinguishing SNNs from other ANNs. While these basic operating principles are enough to describe most SNNs, SNNs vary in neuron model and other parameters. In this work, we use the basic leaky integrate-and-fire (LIF) neuron model, defined by

$$RC \frac{du}{dt} = u_{\text{rest}} - u(t) + RI(t). \quad (1)$$

The equation describes the membrane potential over time. The variables R and C are the resistance and capacitance of the membrane, $u(t)$ its potential at time t , $I(t)$ the amount of current it receives at time t from its neighbours, and u_{rest} its resting potential. With $\tau_m = RC$, and $u(0) = u_{\text{rest}} = 0$ the solution to the equation is

$$u(t) = RI(t) - RI(t)e^{-\frac{t}{\tau_m}}. \quad (2)$$

Solving the equation using forward Euler produces a recurrent, discrete representation of the neuron’s potential over time. With $\Delta t \rightarrow 0$,

$$\tau_m \frac{u(t + \Delta t) - u(t)}{\Delta t} = -u(t) + RI(t) \quad (3)$$

$$\implies u(t + \Delta t) = u(t) + \frac{\Delta t}{\tau_m} (-u(t) + RI(t)) \quad (4)$$

$$\implies u(t + \Delta t) = \left(1 - \frac{\Delta t}{\tau_m}\right) u(t) + \frac{\Delta t R}{\tau_m} I(t). \quad (5)$$

The solution forms the basis of step-wise simulation of LIF neurons.

After a neuron spikes it enters its refractory state. Its potential becomes fixed at u_{reset} and it ceases to respond to stimuli for a duration controlled by the τ_{ref} parameter. Usually, the refractory period is in the order of milliseconds and for simplicity one sets $u_{\text{reset}} = u_{\text{rest}} = 0$. With $r(t)$ denoting how long the neuron will stay refractory at time t , u_{thr} the neuron’s spiking threshold, and Δt arbitrarily set to one, we can incorporate refractoriness into the LIF model:

$$r(t + 1) = \begin{cases} \tau_{\text{ref}} & \text{if } u(t + 1) \geq u_{\text{thr}} \\ r(t) - 1 & \text{elseif } r(t) > 0 \\ 0 & \text{otherwise} \end{cases} \quad (6)$$

$$u(t + 1) = \begin{cases} \left(1 - \frac{1}{\tau_m}\right) u(t) + \frac{R}{\tau_m} I(t) & \text{if } r(t) = 0 \\ 0 & \text{otherwise} \end{cases} \quad (7)$$

2.1.1 Network topology

A major difference between conventional ANNs and SNNs is that the former often are layered; all neurons in one layer only receive inputs from neurons in the previous layer and only send outputs to neurons in the following layer. If all neurons in a layer send output only to all neurons in the following layer the layer is said to be fully-connected and its signal-transfer can be represented as a matrix-vector multiplication. SNNs can similarly be layered and it has been found to be a good approach for classification (Zheng et al., 2021). But it is not suitable for simulation as the neurons in real brains are not organized into fully-connected layers. Instead, their topology is

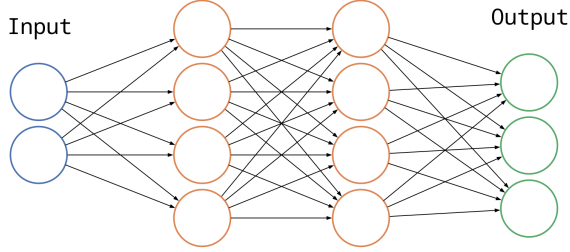


Figure 1: A fully-connected neural network with two hidden layers, two input neurons, and three output neurons

Name	Value	Description (Unit)
Δt	0.1	Time step duration (ms)
C_m	250	Membrane capacity (pF)
τ_m	10	Membrane time constant (ms)
τ_{ref}	2	Refractory period (ms)
τ_{syn}	0.5	Postsyn. current time constant (ms)
u_{rest}	-65	Resting and reset potential (mV)
u_{thr}	-50	Spiking threshold (mV)
v_{th}	8	Thl. neurons' mean spiking rate (Hz)
ω_{ext}	0.15	Thl. spikes amplitude (mV)

Table 1: Microcircuit’s general and simulation parameters

“chaotic” and full of recurrent connections, self-loops (autapses), and multiple edges (multapses). This has far-reaching consequences for what data structures are appropriate for SNNs. An adjacency matrix, for example, is not enough to represent their topological richness.

2.2 Potjans-Diesmann’s Microcircuit

In 2014 Potjans and Diesmann (2014) compiled the results of a dozen empirical studies to create a full-scale spiking neural network microcircuit of one cortical column in the mammalian early sensory cortex. The microcircuit covers 1 mm^2 of the cerebral cortex and consists of 77,169 LIF neurons grouped into four

neocortical layers;¹ L23, L4, L5, and L6.² Each layer is subdivided into one excitatory population that increases neural activity and one inhibitory population that decreases it. Around 300 million synapses connect the neurons.

The microcircuit is a balanced random network so that neural activity is balanced by excitatory neurons that increases activity and inhibitory ones that dampen it (Brunel, 2000). Connectivity and features are sampled from parametric probability distributions, rather than set explicitly. Table 2 and table 3 specify these distributions’ parameters. For example, the initial potential of every inhibitory neuron in L23 is set by sampling a Gaussian with mean -63.16 mV and standard deviation 4.57 mV and the expected number of synapses from population L23/inh to population L5/exc is $5,834 \cdot 4,850 \cdot 0.0755 \approx 2$ million. Synapses are sampled with replacement – multiple synapses can connect the same neuron pair. While the neurons are arranged in terms of neocortical layers, the layers’ connection probabilities show that the network’s *topology* is not layered; neurons in most populations can connect to neurons in any of the other populations.

In addition to the synapses within the cortical column, the circuit receives spikes from external neurons – thalamic input. Column K in table 2 specifies the number of thalamic neurons a given population’s neurons receive spikes from, and parameter v_{th} in table 1 how frequently thalamic neurons spike.³ The expected number of thalamic spikes received per second by all neurons in a population is $v_{\text{th}}K$. For example, neurons in population L23/exc receive about $8 \cdot 1,600 = 12,800$ thalamic spikes per second. The amplitude of all thalamic synapses is fixed at $\omega_{\text{exh}} = 0.15$ mV. As thalamic spikes can be computationally expensive to simulate, Potjans and Diesmann (2014) suggest approximating them with constant direct current injected at a rate of $v_{\text{th}}K\omega_{\text{exh}}\tau_{\text{syn}}$ mV per second. In our simulator we

¹These layers are *not* analogous to layers in conventional ANNs.

²Layer L2 and L3 merged and L1 omitted.

³Potjans and Diesmann (2014) set the parameter to 15 Hz, but we use the NEST model as a baseline, where it is set to 8 Hz.

i	Pop.	N_i	K	u_{init}	ω_i	δ_i
1	L23/exh	20 683	1 600	$\mathcal{N}(-68.28, 5.36)$	$\mathcal{N}(0.15, 0.015)$	$\mathcal{N}(1.5, 0.75)$
2	L23/inh	5 834	1 500	$\mathcal{N}(-63.16, 4.57)$	$\mathcal{N}(-0.6, 0.06)$	$\mathcal{N}(0.75, 0.325)$
3	L4/exh	21 915	2 100	$\mathcal{N}(-63.33, 4.74)$	$\mathcal{N}(0.15, 0.015)$	$\mathcal{N}(1.5, 0.75)$
4	L4/inh	5 479	1 900	$\mathcal{N}(-63.45, 4.94)$	$\mathcal{N}(-0.6, 0.06)$	$\mathcal{N}(0.75, 0.325)$
5	L5/exh	4 850	2 000	$\mathcal{N}(-63.11, 4.94)$	$\mathcal{N}(0.15, 0.015)$	$\mathcal{N}(1.5, 0.75)$
6	L5/inh	1 065	1 900	$\mathcal{N}(-61.66, 4.55)$	$\mathcal{N}(-0.6, 0.06)$	$\mathcal{N}(0.75, 0.325)$
7	L6/exh	14 395	2 900	$\mathcal{N}(-66.72, 5.46)$	$\mathcal{N}(0.15, 0.015)$	$\mathcal{N}(1.5, 0.75)$
8	L6/inh	2 948	2 100	$\mathcal{N}(-61.43, 4.48)$	$\mathcal{N}(-0.6, 0.06)$	$\mathcal{N}(0.75, 0.325)$

Table 2: Population-specific parameters. The first four columns denote the index, i , the name, the size, N_i , and the number of thalamic connections, K_i , of the eight populations. The last three columns denote the Gaussians from which the neurons’ initial potential (mV), the neurons’ synapses amplitudes (mV), and delays (ms) of excitatory postsynaptic potential are sampled from. However, synapse amplitudes from population L23/exh to L3/exh are sampled from $\mathcal{N}(0.3, 0.03)$ and not $\mathcal{N}(0.15, 0.015)$.

	L23/exc	L23/inh	L4/exc	L4/inh	L5/exc	L5/inh	L6/exc	L6/inh
L23/exc	0.1009	0.1689	0.0437	0.0818	0.0323	0.0000	0.0076	0.0000
L23/inh	0.1346	0.1371	0.0316	0.0515	0.0755	0.0000	0.0042	0.0000
L4/exc	0.0077	0.0059	0.0497	0.1350	0.0067	0.0003	0.0453	0.0000
L4/inh	0.0691	0.0029	0.0794	0.1597	0.0033	0.0000	0.1057	0.0000
L5/exc	0.1004	0.0622	0.0505	0.0057	0.0831	0.3726	0.0204	0.0000
L5/inh	0.0548	0.0269	0.0257	0.0022	0.0600	0.3158	0.0086	0.0000
L6/exc	0.0156	0.0066	0.0211	0.0166	0.0572	0.0197	0.0396	0.2252
L6/inh	0.0364	0.0010	0.0034	0.0005	0.0277	0.0080	0.0658	0.1443

Table 3: Probability that a random neuron in the population specified by the rows is connected to a random neuron in the population specified by the columns.

model thalamic spikes, however.

rent:⁴

$$d = \tau_{\text{syn}} - \tau_m \quad (8)$$

$$p = \tau_{\text{syn}} \tau_m \quad (9)$$

$$q = \tau_m / \tau_{\text{syn}} \quad (10)$$

$$w_f = \frac{C_m d}{p(q^{\tau_m/d} - q^{\tau_{\text{syn}}/d})} \approx 585 \quad (11)$$

The constants p_{22} and p_{11} define the membranes’ and presynaptic currents’ decay rate:

$$p_{11} = \exp(-\Delta t / \tau_{\text{syn}}) \approx 0.82 \quad (12)$$

$$p_{22} = \exp(-\Delta t / \tau_m) \approx 0.99 \quad (13)$$

The synaptic parameters are also scaled. The synapse amplitude by w_f which is a function of the membrane time constant, τ_m , membrane capacity, C_m , and the postsynaptic time constant, τ_{syn} , that maps postsynaptic potential to postsynaptic cur-

⁴See Hanuschkin et al. (2010) for the derivation.

The injection of the presynaptic current is scaled by p_{21} :

$$\beta = \tau_{\text{syn}}\tau_m/(\tau_m - \tau_{\text{syn}}) \quad (14)$$

$$\gamma = \beta/C_m \quad (15)$$

$$p_{21} = p_{11}\gamma(\exp(\Delta t/\beta) - 1) \approx 0.00036 \quad (16)$$

The subscripts match those found in the source code for the NEST simulator (Plesser et al., 2015) and have no deeper meaning here.⁵ Taken together, this gives us the following discrete recurrences for the step-wise update of membrane potential, u_t :

$$u_{t+1} = \begin{cases} u_{\text{reset}} & \text{if } r_t > 0 \\ p_{22}u_t + I_t p_{21} & \text{otherwise} \end{cases}, \quad (17)$$

presynaptic current, I_t :

$$I_{t+1} = p_{11}I_t + T_t w_f \omega_{\text{exh}}, \quad (18)$$

and refractoriness, r_t :

$$r_{t+1} = \begin{cases} \tau_{\text{ref}} & \text{if } u_{t+1} \geq u_{\text{thr}} \\ r_t - 1 & \text{elseif } r_t > 0 \\ 0 & \text{otherwise} \end{cases}. \quad (19)$$

The variable T_t denotes the number of thalamic spikes received by the presynapse at time t and is modelled as a Poisson distributed random variable with mean $v_{\text{th}}K\Delta t$.

Having presented the theoretical foundations for LIF SNNs and the specifics of the Potjans-Diesmann microcircuit, we now present the technologies implement our simulator with. We return to SNN simulation in section 3 where we both delve deep into simulation methods and present our simulators.

2.3 Field-Programmable Gate Arrays

An FPGA is a type of reprogrammable integrated circuit. First marketed by Altera and Xilinx in the 1980's, FPGAs have found uses in many niches of the electronics industry; in avionics, in telecommunications, and in VLSI design because of their unique

blend of performance, flexibility, and development costs (Trimberger, 2015). FPGAs are not as performant as Application-Specific Integrated Circuits (ASICs), but ASICs are extremely expensive to develop which makes them cost-prohibitive unless the number of units produced runs in the tens of millions. Furthermore, ASICs are not reprogrammable and thus suitable only for the specific tasks they were developed for. Central Processing Units (CPUs), on the other hand, are flexible and inexpensive, but have low performance. Graphics Processing Units (GPUs) sit between CPUs and ASICs on the flexibility-performance spectrum. While GPUs can run any computation, they generally only excel at highly regular, numerically intensive computations. In particular, they do not handle divergent control flow well.

FPGAs trade performance for flexibility in a different manner than CPUs, GPUs, and ASICs. They consist of configurable blocks organized in a grid-like fabric and each block can be configured to compute a small and specific function. Like the logical and of two one-bit signals or comparing two eight-bit numbers. The number of configurable blocks varies enormously from one FPGA to another; high-end FPGAs can contain hundreds of thousands or even millions of blocks. For performance-sensitive workloads, FPGAs' advantage is the lack instruction processing overhead. CPUs and GPUs load programs from memory and decode and process their instructions one after the other, while FPGAs merely pass data through a pre-configured, fixed-function circuit, similar to how ASICs operate.⁶ However, the reprogrammability of FPGAs comes at a great cost and their raw performance cannot rival that of ASICs or GPUs unless algorithms are specifically designed for them. They also run at lower clock frequencies than comparable CPUs from the same generation. For example, our simulators run at around 600 MHz,⁷ while Intel Core i9 processors operate at over 3 GHz.

⁶It is of course true that superscalar and pipelined processors can handle many instructions in parallel. But the point stands; the overhead caused by the need to decode and dispatch instructions is large.

⁷However, Langhammer and Constantinides (2023) ran a softcore microprocessor at over 770 MHz on the same FPGA which is close to the theoretical limit.

⁵https://github.com/nest/nest-simulator/blob/aa9907a5d5a7916eeeca00c2e6584202702eb2a/models/iaf_psc_exp.cpp

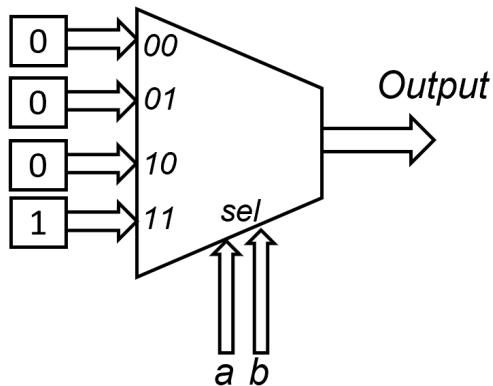


Figure 2: A LUT for computing any two-variable boolean function implemented as one 4:1 multiplexer and four memory bits. The bit values determines which function the LUT computes.

To stay competitive, FPGA implementations need to carry out much more work per clock cycle than equivalent CPU implementations. These potential drawbacks aside, researchers have deployed FPGAs for performance sensitive workloads, with promising results (?).

Bitstreams are files that configure FPGA blocks and specialized tools write them to the FPGA’s fabric. Bitstreams are to FPGAs what machine code are to microprocessors. Tools that take descriptions of the digital circuit the FPGA should implement and produce bitstreams are called *synthesizers*. The descriptions are usually written in hardware description languages (HDLs), but many tools support higher-level languages as well.

A configurable block is a logic block that functions as a small combinational circuit that computes the boolean function it was configured for. Depending on configuration, the same logic block can serve as an and-gate, or-gate, xor-gate, etc. Memory embedded adjacent to the block stores its configuration and the block’s output is retrieved from this memory. Because the blocks look up their results from memory they are called look-up-tables (LUTs). Figure 2 shows a LUT for a two-input and-gate built with a multiplexer and four memory bits. Flip-flops (FFs)

are the FPGA’s basic memory blocks and work as small and fast RAMs. Each FF stores one bit, but multiple FFs can be combined into registers to store larger datums.

In addition to configuring the FPGA’s logic and memory blocks, bitstreams define how the blocks should be connected through the FPGA’s interconnect network. This is called *routing* and is one of the synthesizer’s most important tasks. Routing is critical to the design as the interconnect occupies most of the FPGA’s fabric (Betz and Rose, 1999). Good routing should minimize the total length of the interconnect, the number of blocks required, and the length of the longest path connecting two blocks as the operating frequency of the design is bounded by this length. Routing is a challenging problem in computer science and finding high-quality routing of large designs is both difficult and time-consuming.

Due to their reconfigurability, digital logic implemented in LUTs is slower and requires more components than equivalent logic implemented in non-reconfigurable ASIC gates. Memories built using FFs have lots of overhead because FFs only store one bit. Therefore, modern FPGAs come with non-reconfigurable blocks for arithmetic and storage. One can view these blocks as small ASICs embedded in the FPGA that the designer connects via the configurable blocks. For example, our Agilx 7 FPGA have five different types of blocks; 487 200 Adaptive Logic Modules (ALMs) which functions as LUTs, 1 948 800 flip-flops, 7 110 M20K RAM blocks, two 18.432 Mb eSRAM blocks, and 4 510 DSPs. We describe these blocks in the following sections.

2.3.1 Adaptive Logic Module

The *Adaptive Logic Module* (ALM) in figure 3 is the basic building block of Intel’s family of FPGAs. Consisting of one fractured eight-input-LUT, four FFs, and two full-adders, it is a versatile multi-purpose block and can – depending on its configuration – compute two four-input boolean functions, one six-input boolean function, or perform four-bit addition with carry. It can also serve as a four-bit memory.

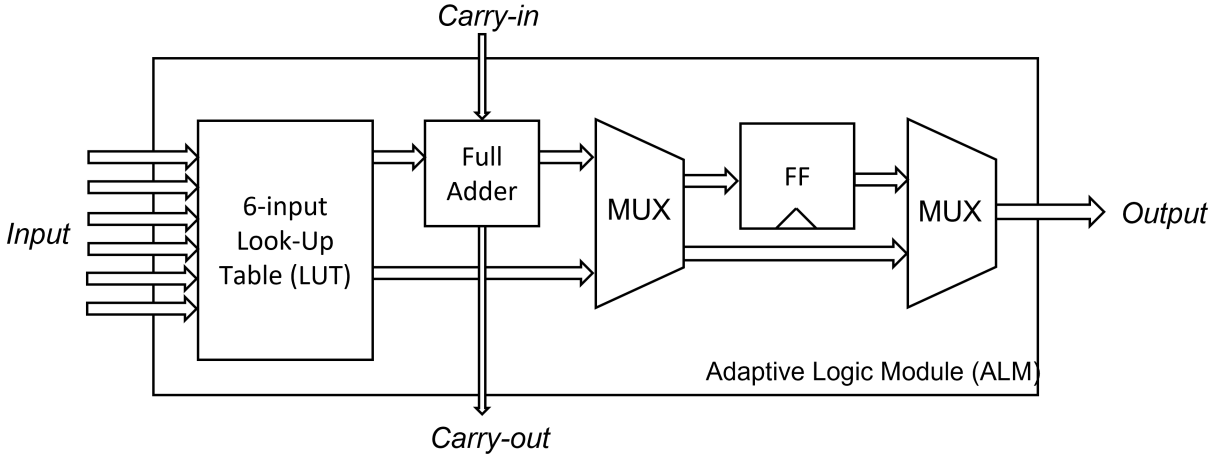


Figure 3: Simplified schematic of the Agilex 7 ALM. The multiplexers’ (trapezoids) control signals (not shown) and the contents of the LUT configures the ALM. The ALM can serve as – among other things – a four-bit adder, a four-bit memory, or as combinational logic of six inputs depending on configuration.

2.3.2 Digital Signal Processing

The *Digital Signal Processing* (DSP) block contains functions for multiplication, addition, subtraction, and accumulation. The block functions as the FPGA’s arithmetic logic unit (ALU). Agilex 7 has two types of DSPs; one for integer arithmetic and one for IEEE 754 floating-point arithmetic in single- and half-precision modes. Pipeline registers organized into three stages are contained within the DSP. Data can be routed through one or more of the stages or bypassed completely to achieve a given latency. This can be useful if one operand is available multiple cycles before the other.

2.3.3 Memory Hierarchy

In processor-based hardware, registers, caches, and main memory is organized into a fixed hierarchy. Applications must be structured around the memory hierarchy to optimally use it. Typically, data transfer between the hierarchy’s levels is implicit and not directly under the programmer’s control. On an FPGA the designer constructs the memory hierarchy from the available memory blocks and has fine-grained control over exactly where data is stored. This allows

the memory system to be tailored to the application rather than the other way around.

The FPGA’s memory is *on-chip* if it is embedded in the FPGA fabric itself and *off-chip* otherwise. On-chip memory is much faster and smaller than off-chip memory. The Agilex 7 has three types of on-chip memory; Memory Logic Array Blocks (MLABs), M20Ks, and embedded SRAM (eSRAM). Scalars and shallow FIFOs are often stored in MLABs, small arrays and caches in M20Ks, and larger buffers in eSRAM. MLABs are not memory blocks per se, but instead a technique for using ALMs as memory. The Agilex 7 can combine ten unused ALMs into one 640-bit register. As many FPGA designs do not use all logic blocks, repurposing them can be very useful. MLABs have very low latencies since they are close to the logic that use them. Xilinx FPGAs implement the same concept under the name *distributed RAM* or LUTRAM. On-chip M20Ks are also known as block RAM (BRAM) and are much larger than MLABs. The Agilex 7 contains around seven thousand M20K each of which can hold 20 kbit of data, as their name suggests. In total, about 139 Mbit. The board also has two 36 Mbit embedded SRAM blocks. These have high bandwidth and high random trans-

action rates and compliment the other on-chip memory blocks. Unfortunately, the Intel OpenCL FPGA compiler cannot infer eSRAM so they are unused in this work. On the Agilex 7 four 8 GB DRAM sticks constitute the board’s off-chip memory. Accessing off-chip memory takes much longer than accessing on-chip memory, with access times measured in the hundreds of cycles.

In addition to controlling the types of memory used, the designer can also configure individual memory blocks. Multipumping, for example, can sometimes double memory throughput at the expense of significantly lowering the design’s operating frequency. Banking can increase expected memory access concurrency, but also increase stalls due to conflicts so the technique is best reserved for evenly distributed data.

2.4 High-Level Synthesis

Designers traditionally design circuits in *Hardware Description Languages* (HDLs) such as VHDL and (System) Verilog. With these the designer specifies the behaviour of all the circuit’s logic gates and flip-flops. The result is a *Register-Transfer Level* (RTL) design, so called because it models the register-to-register transfer the circuit’s signals. A synthesizer takes the RTL design and produces a low-level representation (netlist) of it which can be lowered further to create an ASIC or transformed into an FPGA bitstream. The synthesizer’s job is far from straightforward and it must – among other things – place every component of the circuit on a two-dimensional grid and ensure that it can operate with the desired clock frequency.

While HDLs offer a great deal of control over the resulting circuit, they are low-level and lack support for many high-level programming constructs so using them can be tedious and error-prone. It often makes sense to work in a higher-level language instead. That workflow is called High-Level Synthesis (HLS) and is supported by many tools. For example, Intel’s FPGA software generates synthesizable Verilog code from designs coded in the imperative C-like language OpenCL.

The main worry of using HLS is that the hard-

```

1  __kernel void mul_sd(
2      __global float *A,
3      __global float *B,
4      __global float *C,
5      int N) {
6      for (uint i = 0; i < N; i++)
7          C[i] = A[i] * B[i];
8  }
9  __kernel void mul_nd(
10     __global float *A,
11     __global float *B,
12     __global float *C,
13     int N) {
14     uint i = get_global_id(0);
15     if (i < N)
16         C[i] = A[i] * B[i];
17 }
```

Figure 4: Single and multiple work item OpenCL kernels for element-wise vector product

ware will not be as efficient as if an HDL had been used. As the designer works in an imperative, high-level language their view of the hardware may be obscured and not easy to visualize. Furthermore, the machine-generated HDL generated by HLS tools is often opaque and near impossible to understand. These fears may be unfounded, however, as many studies have found that HLS does not degrade performance and sometimes even improve it (Lahti et al., 2019).

2.5 OpenCL

In 2009 the Kronos Group published the first version of the OpenCL (Open Computing Language) standard (Munshi, 2009). The goal was to replace all vendor-specific languages and application programming interfaces (APIs) with a common and portable standard for writing high-performance code across all kinds of devices. An OpenCL application (unlike one written in a competing technology such as CUDA) can run *unmodified* on any device with a compliant OpenCL implementation. Today OpenCL now runs on millions of CPUs, GPUs, DSPs, and other com-

puting devices.

The OpenCL standard consists of three parts; a specification for a programming language, a host API, and a device API. The host API runs on the user’s main computing device (e.g., PC) and controls their accelerator (e.g., GPU), whose functionality is accessible through the device API. Thus, OpenCL prescribes both how the programmer should program the accelerator and how to manage it from an external device.

The OpenCL language closely resembles C. It supports functions, loops, multiple variable scopes, aggregate data types, and many other programming constructs familiar to C programmers. A big difference from C is that OpenCL comes with three explicit pointer spaces; global, local, and private. These grant the programmer fine-grained control over data storage. What constitutes global, local, and private memory is device-specific. Generally the largest and slowest memory space is global, while the smallest and fastest is private. Global memory may be sized in gigabytes, while private memory may only be a few kilobytes. OpenCL has native support for SIMD types to make it easier to write algorithms for parallel hardware. Unlike C, OpenCL specifies the bit width of most builtin types – an `int` is always 32 bits and a `long` 64 bits.

OpenCL was designed with massively parallel architectures in mind (e.g., GPUs) and has builtin support for concurrency in the form of *work items*. A work item is a small indivisible unit of work with, ideally, no dependencies to other work items. The OpenCL runtime can schedule independent work items to maximally exploit the hardware’s potential. Consider the kernels in listing 4 for computing the element-wise vector product. Launching the first kernel – `mul_sd` – causes OpenCL to instantiate one kernel on one core which runs N iterations of the for loop. Launching the second kernel, however, causes OpenCL to instantiate up to N kernels, each of which runs on an available core and computes one iteration of the for loop. Since the second kernel runs more computations in parallel, it likely runs much faster. Sometimes the algorithm’s work cannot feasibly be separated into completely independent work items. For such situations, work items that must be syn-

```
1 for (uint i = 0; i < N; i++) {
2     B[i] = h(g(f(A[i])));
3 }
```

Figure 5: A loop-nest that could benefit from pipeline parallelism. The functions `f`, `g`, and `h` are assumed to be short and inlineable.

```
1 channel uint jobs
2 __attribute__((depth(512)));
3 __kernel void consumer() {
4     while (true) {
5         uint job =
6             read_channel_intel(jobs);
7         if (!msg)
8             break;
9         process_job(job);
10    }
11 }
12 __kernel void producer(uint N)
13 for (uint i = 0; i < N; i++) {
14     uint job = create_job(i);
15     send_channel_intel(job);
16 }
17 send_channel_intel(0);
18 }
```

Figure 6: Producer-consumer kernels communicating via a channel

chronized can be organized into *work groups* which share local memory. However, synchronization is impossible between work groups.

2.6 OpenCL on Intel FPGAs

As FPGAs work very differently from processor-based hardware, Intel’s OpenCL implementation for FPGAs differs in important ways from other OpenCL implementations. One major difference concerns parallelism. CPUs and GPUs have multiple cores to run multiple computations in parallel. Thus, the workload of an OpenCL program organized into multiple

work items can be mapped onto multiple cores. FPGAs do not have any cores in the usual sense and it is difficult for Intel’s OpenCL FPGA compiler to synthesize code structured around work items into efficient FPGA bitstreams. Instead, Intel recommends designers to structure OpenCL code as single-work item kernels, and to derive most of the parallelism from executing multiple loop iterations concurrently; a type of concurrency known as “pipeline parallelism”. For example, assume the three functions `f`, `g`, and `h` in listing 5 are short, side-effect free, and represent computations that can be performed in less than one clock cycle. The FPGA compiler can create a three-stage pipeline for the loop, where combinational circuits for `f`, `g`, and `h` constitute the pipelines three stages. This pipeline can process three iterations of the loop in parallel; while the `h`-circuit processes data for the first, the `g`-circuit processes data for the second, and the `f`-circuit for the third iteration, and so on. While the latency of every iteration is three cycles, the throughput (initiation interval) is only one cycle and the latency for the loop as a whole is $N + 2$ since it takes 2 cycles to fill the pipeline. Though, this assumes that the latency of every function is fixed and predictable. If one function executes operations with variable latencies, such as memory accesses, the pipeline may need to stall. Moreover, pipeline parallelism cannot increase the throughput of the loop beyond one cycle – for that one has to use other techniques.

The FPGA compiler supports many compiler directives (“pragmas”) that can help it better optimize difficult loops or loops containing invariants it can not prove on its own. Two important directives are `#pragma ivdep` and `#pragma unroll N`. The first tells the compiler that the loop contains no loop-carried dependencies and therefore it can reorder the loop iterations as it pleases. The second tells it to duplicate the loop body N times. This means that the resulting circuit will use N times as many gates, but, potentially, run N times faster. Other compilers ignore these directives.

The FPGA compiler extends OpenCL with syntax for declaring channels for inter-kernel communication. Channels resemble the pipes feature in OpenCL 2.0 which the FPGA compiler does not sup-

```

1 for t in range(n_ticks):
2     for i in range(n_neurons):
3         N[i] = update_neuron(N[i])
4         if spikes(N[i]):
5             Q = enqueue(Q, i, t)
6     for i in range(n_neurons):
7         c = collect(Q, i, t)
8         N[i] = update_psc(Q, c)
9         Q = dequeue(Q, i, t)

```

Figure 7: Synchronous SNN simulation

port. Channels are implemented in on-chip memory as first in first out (FIFO) buffers of the desired depths and are very fast. Figure 6 shows a simple producer-consumer example, where one kernel calls `send_channel_intel` to put jobs on the FIFO and the other calls `read_channel_intel` to remove them. Both functions block if the FIFO is full or empty.

3 SNN Simulation

Having reviewed SNNs, the Potjans-Diesmann microcircuit and our implementation tools, we now discuss methods for simulating SNNs efficiently. We explore SNN simulation in general before introducing the ideas and algorithms we use to optimize our simulators. The end result is a taxonomy of twelve simulator families, grouped by their algorithms and implementation styles. We illustrate most of our points with Python-esque pseudo-code, extended with two keywords; `parfor` and `atomic`. The former for loops whose iterations are independent and therefore can be executed in parallel. The latter for operations that must execute as one indivisible unit. Text in `{brackets}` explains what the pseudo-code should do at that point.

Broadly speaking, SNN simulation can be categorized on whether it is synchronous (time-stepping) or asynchronous (event-driven). Synchronous simulation updates the state of every simulated object at every tick of a clock, regardless of whether it is necessary or not (Brette et al., 2007). Asynchronous

simulation only updates simulated objects when they receive external stimuli, i.e., events. In an asynchronous SNN simulation, spike emission and reception constitute the events, since the state of a neuron at a time between two events can be calculated easily (and generally is unimportant). Hybrid strategies, with asynchronous updates for some parts of the SNN and synchronous updates for other parts, are possible.

For SNN simulation, asynchronicity offers precision advantages. The neuron’s membrane potential only has to be recomputed when it receives spikes. If this happens only rarely the simulator can afford to use more sophisticated methods than (repeated) forward Euler to solve the LIF equation (equation (1)). Also, spikes can be sent and received at any time and do not have to be confined to a discrete grid. For example, a synchronous simulator with a 0.1 ms time steps may not be able to represent spikes sent at times that are not multiples of 0.1 ms. Asynchronicity may also have scalability advantages as the neurons’ states do not have to be synchronized to a global clock. However, asynchronicity entails irregular computation and irregular memory accesses – traits that are extremely undesirable on modern hardware. Furthermore, dense SNNs have many more synapses than neurons which results in cascading effects. One spiking neuron “wakes up” thousands of its neighbours, causing them to spike and in turn wake up thousands of their neighbours. On the whole, the event processing overhead may dominate over whatever computational savings not being bound by a global clock brings. For example, Pimpini et al. (2022) presents a sophisticated asynchronous CPU-based SNN simulator that supports speculative execution so that future neuron states can be computed in advance and then rolled back if received spikes (“stragglers”) invalidate their predicted state. While the authors measured accuracy improvements, the performance was lackluster. For these reasons, most high-performance SNN simulators, including ours, are synchronous.

Synchronous simulation splits the simulation task into two phases; one for updating neurons and one for transferring spikes. Listing 7 shows the basic algorithm. It uses two data structures; an indexable data structure, N , to store the state of all neurons

and a queue, Q , to keep track of spikes in flight. For every neuron the algorithm calls `update_neuron` to update its membrane potential, presynaptic current, and refractoriness in accordance with equations (17) to (19). If `spikes` indicates that the neuron spikes, it enqueues the neuron’s index i and the current time step t in Q . In the next phase (line 6 to 9) the algorithm calls `collect` on the queue to aggregate current destined to the i :th neuron at time step t . The call to `update_psc` adds the aggregated current to the neuron’s presynaptic current. Finally, the algorithm removes the current it just handled from the queue. In this listing we include the `for t in range(n_ticks)` loop which shows that the algorithm repeats the two phases `n_ticks` times. However, for brevity’s sake, we omit this outer loop in the following listings.

The work required for updating the neurons’ state for one tick is in the order of $O(N)$, where N is the number of neurons – i.e. linear. However, for transferring spikes it is $O(f\Delta tpN^2)$, where p is the probability that two randomly chosen neurons are connected by one or more synapses, f the average spiking rate, and Δt the tick duration. So the transfer time is proportional to the network’s density and quadratic in N – i.e. for large N it dominates. Furthermore, updating the membranes and presynapses require a handful of multiplications per neuron – cheap on modern hardware – whereas spike transfer requires expensive reads and writes to and from non-contiguous memory. Hence, we will focus on the transfer phase which is where synchronous simulators spend most of their time in the remainder of this section.

3.1 Pushing and Pulling

Researchers have identified “pushing” and “pulling” as two general strategies for designing algorithms for graph problems (Besta et al. (2017); Grossman et al. (2018); Ahangari et al. (2023)). A push strategy transfers signals *from a node to its neighbours* by writing to the neighbours incoming signal buffers. In contrast, a pull strategy transfers signals *to a node from its neighbours* by sweeping through the node’s neighbours and checking whether they have any sig-

```

1 parfor i in range(n_neurons):
2     spikes = False
3     if R[i] == 0:
4         x = p22*U[i] + p21*I[i]
5         spikes = x >= U_thresh
6         if spikes:
7             x = 0
8             R[i] = t_ref_tics
9             U[i] = x
10    else:
11        U[i] = 0
12        R[i] -= 1
13    if spikes:
14        parfor j, d, w in syns_from(i):
15            atomic W[t + d, j] += w
16    I[i] = p11*I[i] + T[t, i]*wpsn
17    I[i] += W[t, i]

```

Figure 8: One time step of push-based spike transfer

```

1 parfor i in range(n_neurons):
2     A[t, i] = False
3     if R[i] == 0:
4         x = p22*U[i] + p21*I[i]
5         if x >= U_thresh:
6             A[t, i] = True
7             x = 0
8             R[i] = t_ref_tics
9     else:
10        x = 0
11        R[i] -= 1
12    U[i] = x
13    s = 0
14    for j, d, w in syns_to(i):
15        if A[t - d, j]:
16            s += w
17    I[i] = p11*I[i] + T[t, i]*wpsn + s

```

Figure 9: One time step of pull-based spike transmission

nals to be delivered to the node. I.e., the receiver node has to “go and ask” its neighbours whether they have signals for them. SNN simulation is a graph problem involving node-to-node transfer of signals and it too can be characterized in terms of pushing and pulling. Listing 8 and 9 illustrate the two strategies. In both listings, the values of the scalars `p11`, `p21`, and `p22` come from equations (13) and (16). The arrays `U`, `I`, and `R` contain the neurons’ membrane potentials, presynaptic potentials, and refractory counters. When the neuron’s membrane potential exceeds `U_thresh` the neuron spikes and becomes refractory for `t_ref_tics` tics. The expression `T[t, i]` denote the number of thalamic spikes received by the i :th neuron at the t :th time step.

The strategies differ in how they transfer spikes. The push strategy transfers them when `spikes` indicates that a neuron spikes. It calls `syns_from` to fetch an iterator of all synapses *originating* from the i :th neuron. The three-tuples (j, d, w) it retrieves represent the synapses; j the index of the destination neuron, d the delay in time steps, and w the current. For every three-tuple it writes to an element of `W`, a two-dimensional array buffering current to be delivered. The element `W[t, i]` is the amount of presynaptic current the i :th neuron receives at the t :th time step. The `syns_to` call in listing 9 works exactly like the `syns_from` call, except it returns all synapses *terminating* at the i :th neuron and j in the three-tuples (j, d, w) denotes the *originating* neuron. Array elements `A[t, i]` indicate whether the i :th neuron spikes at time t or not. The size of the longest synapse delay is in practice bounded and small so both `A` and `W` can be implemented as statically sized wrap-around arrays – a technique we cover in section 3.2. Note that every neuron can be handled independently so we use the `parfor` construct for both algorithms’ outer loops.

Pushing of synaptic current to their neighbours happens on line 16 and 17 of the push strategy’s listing. Neurons spike infrequently, but when they do, the algorithm must retrieve all their synapses and write their current to the array `W`. This part of the algorithm is performance-critical. First, the algorithm accesses all the neuron’s synapses which can be expensive, even if they are stored in contiguous memory.


```

1 for i in range(n_neurons):
2     {Update neuron state as before}
3     if spiked:
4         Q = enqueue(Q, i)
5 for i in contents(Q):
6     parfor j, d, w in syns_from(i):
7         W[t + d, j] += w
8 Q = clear(Q)

```

Figure 10: Deferred push-based spike transfer

Second, the algorithm gathers and scatters data from and to uncorrelated indices of W . These are costly operations since the accesses to W cannot be coalesced or cached.⁸ To make matters worse, at one time step multiple neurons can write to the same indices of W . I.e., $W[t + d, j] += w$ has to be executed atomically to avoid data races. As the model is densely connected, races are not uncommon and should be accounted for. Partition-awareness, as suggested by Besta et al. (2017), is not an option because of the density and randomness of the connections, making most of them remote and not local.

The pull-based algorithm instead has the receiver neurons responsible or “pulling in” current. Spiking merely sets an element in A to true and does not trigger current transfer. On subsequent time steps, neurons connected to that neuron checks if it spiked at time $t - d$ and, if so, adds its synaptic current. The upside of this algorithm is that it is synchronization-free; neurons do not write to shared memory. The major drawback is that every neuron at every time step must check all its incoming synapses to see from which of them it receives current. Additionally, the algorithm reads from scattered memory on line 18. The large amount of data it reads probably makes it inefficient, unless the number of computational units is large and memory reads are substantially cheaper than writes. Neither of which is true for our FPGA. Consequently, we focus on push-based spike transfer in this work.

Listing 10 shows a variant of push-based spike

⁸The memory addresses are too far apart for any coalescing or caching scheme to be efficient.

transfer that first collects spiking neurons and then transfers their synapses current in a dedicated phase. Collection can either be done with a marking array, at the expensive of wasting memory, or with a queue (as in the listing), at the expense of making loop iterations dependent. The choice depends on the target platform. Though, splitting the update and spike transfer into two phases reduces the number of cache conflicts which is advantageous. Lines 13 to 15 of the basic push algorithm can flush prefetched and cached parts of the U , I , and R arrays.

3.2 Buffer Sizing and Wrapping

To reduce the size of the W array we use “wrap-around indexing”. The indices of the rows in W written to in one time step lies within the interval $t + 1$ to $t + d_{\max} - 1$, where $d_{\max} - 1$ is the network’s largest synaptic delay and is small. Moreover, at time step t rows 0 to $t - 1$ will not be read again so that space can be reused. We do that by setting the number of rows in W to $d_{\max}$ and use modular arithmetic to index rows. The expressions for accessing W on line 15 and 17 in listing 8 become $W[(t + d) \% D_{\max}, j]$ and $W[t \% D_{\max}, i]$, respectively. We choose a large enough value for $d_{\max}$ by evaluating the cumulative distribution function of the Gaussians we sample the slower excitatory synapses delays from, $\mathcal{N}(1.5, 0.75)$:

$$P(\mathcal{N}(1.5, 0.75) \leq 6.4) \approx 0.999999999968 : \quad (20)$$

This shows that with $\Delta t = 0.1$, 64 rows is more than enough since the probability of sampling a synaptic delay longer than 6.4 ms is astronomically low. It is also a power of two so we use masking to realize the modular arithmetic. With this scheme we also have to clear $W[t, i]$ after reading it to avoid double reads.

Even with only 64 rows and assuming four-byte floats, the W array still consumes $64 \cdot 4 \cdot 77169 \approx 20$ megabytes which is more than we can fit in on-chip memory. We could store the array in off-chip memory – which is plentiful – or use half-precision two-byte floats instead. Neither solution is satisfactory. As we argued in section 3.1, we need fast reads and writes to uncorrelated addresses of W which off-chip memory doesn’t give us. We also prefer not to lower the

```

1 parfor i in range(n_neurons):
2     {Update U[i], R[i] as before}
3     spiked[i] = {True if neuron spiked}
4     I[i] = p11*I[i] + T[t, i]*wpsn
5 for rt in range(D_MAX):
6     delay = (t - rt) % D_MAX
7     if delay < D_MAX - 1:
8         for n in enqueue_at(Q, rt):
9             syns = syns_from(n, delay)
10            parfor j, d, w in syns:
11                I[j] += w
12     else:
13         Q = dequeue(Q, rt)
14 rt = t % D_MAX
15 for i in range(n_neurons):
16     if spiked[i]:
17         Q = enqueue(Q, rt, i)

```

Figure 11: Three-phase just-in-time spike transfer

numeric the precision as that makes the simulation less accurate. A third option is to store all spiking neurons in a queue and only activate a subset of their synapses at a given time step.

3.3 Just-In-Time Spike Transfer

By activating all the neuron’s synapses at once, the push algorithm works harder than necessary. Obviously, all synapses must *eventually* be activated, but *right now* only synapses with a delay of one time step must be activated since the current they transfer will be read at the next time step. This observation leads us to a “lazy” just-in-time algorithm which keeps all spiking neurons in a queue for a fixed number of time steps. At every time step it activates those synapses whose current is read at the next time step. Thus, a neuron that spikes at time t will have its one-tick-delay synapses activated at time $t + 1$, its two-tick-delay synapses at time $t + 2$, and so on. Listing 11 sketches a three-phase algorithm built on this idea. The first phase (line 1 to 4) updates U and R as before and marks spiking neurons in `spiked`. The third phase (lines 14 to 17) enqueues the marked

neurons with a “relative timestamp”, `rt`. The second phase (lines 5 to 13) scans the queue and calls `enqueue_at` to fetch previously enqueued neurons. The `syns_from` function works as before, but now has a second parameter to select synapses with the given delay. Suppose that the simulator simulates time step 100 and that `rt` is 10 in one iteration of the loop. Since $26 \equiv (100 - 10) \bmod d_{\max}$ ($d_{\max} = 64$) all synapses with 26 time steps of delay of queued neurons with a relative timestamp of 10 will be activated. These neurons ought to have been stored at time step $t = 100 - 26 = 76$ which is indeed the case since $10 \equiv 74 \bmod d_{\max}$. After the neurons have stayed in the queue for $D_{\max} - 1$ time steps, the `dequeue` call evicts them. Though with a sensible FIFO implementation eviction is a no-op so we omit it in future pseudo-code.

How much memory do the just-in-time algorithm save? Through experimentation we found that the average number of spiking neurons per time step is 23, meaning that the expected number of elements in Q is $23 \cdot d_{\max} = 1472$. We generously round it up to 4096 and, as neuron indices take four-bytes to store, the total size of the queue is 16 kilobytes, which comfortably fits in on-chip memory.⁹

To improve the just-in-time algorithm we use “lanes”. Essentially, we duplicate the W array so that synapses of multiple spiking neurons can be activated simultaneously. The algorithm writes the synaptic current of the first neuron it handles in the first lane, of the second neuron in the second lane, and so on. Synaptic current of neurons in different lanes does not interfere. The update phase has to be adjusted accordingly and must sum the incoming current from all lanes. As each lane consumes about 320 kb of memory we can fit at most 16 lanes.

3.4 Horizon-Based Spike Transfer

The just-in-time algorithm requires a fair amount of bookkeeping. It activates the spiking neuron’s synapses $d_{\max}$ times instead of just once, and the loop on line 10 and 11 iterates many fewer times than

⁹As in section 3.2, we could use the cumulative distribution function to show that 4096 elements is enough to make the risk of overflow virtually zero.


```

1 parfor i in range(n_neurons):
2     I[i] += W[t % H, i]
3     W[t % H, i] = 0
4     {update U[i] and R[i] as before}
5     I[i] = p11*I[i] + wspn*T[t, i]
6 for i in range(D_MAX / H):
7     rt = (t - H*i - 1) % D_MAX
8     d_from = H*i + 1
9     d_to = d_from + H
10    for n in enqueued_at(Q, rt):
11        syns = syns_from(
12            n, d_from, d_to)
13        parfor j, d, w in syns:
14            W[(d + t) % H, j] += w
15 rt = t % D_MAX
16 for i in range(n_neurons):
17     if spiked[i]:
18         Q = enqueue(Q, rt, i)

```

Figure 12: Three-phase spike transfer with configurable horizon

the corresponding loop on line 14 and 15 of the basic push algorithm. This loop is an important source of parallelism and running it many times with fewer iterations is much worse for performance than running it fewer times with many iterations. Our solution is to reintroduce a smaller version of the spike buffer whose number of rows, h is a factor of $d_{\max}$ so that when a neuron spikes we write to the buffer $d_{\max}/h$ times. Listing 12 shows the concept. The `syns_from` function now retrieves synapses of the neuron whose delay is within the range d_{from} to $d_{\text{to}} - 1$. Suppose $d_{\max} = 64$, $t = 100$ and $h = 16$. The loop on lines 5 to 13 iterates four times, the relative timestamps assumes the values 35, 19, 3, and 51, and the half-open intervals the values [1, 17), [17, 33), [33, 49), and [49, 65). Thus, the inner loop on line 10 to 14 activates all synapses with the given delays of the neurons stored at the given relative timestamps. With this scheme we trade-off on-chip memory for better concurrency.

Note that we add the current from the spike buffer to the presynaptic potential on line 2, before we update the membrane’s potential and we add one to

```

1 def syns_from(n, d_from, d_to):
2     start = X[n, d_from]
3     end = X[n, d_to]
4     for i in range(start, end):
5         yield S[i]

```

Figure 13: Indexed access to synapse data

d_{from} on line 7. This is necessary since the algorithm transfers spikes one time step later than the basic push algorithm.

3.5 Storing Synapses

Previous sections’ pseudo-codes imply that it is very important that synapses can be queried by their sender neuron quickly. In particular, synapses should be stored so that one `syns_from` call only accesses memory in one contiguous chunk. We fulfill these goals by storing the synapses as an array sorted on sender neuron, receiver neuron, and delay that we query with a prebuilt index keyed on sender neuron and delay. This means that finding all synapses for a particular neuron or neuron-delay combination requires only two index lookups; one for the first synapse and one for the last synapse. Moreover, as the index implicitly stores the sender neuron, we only need eight bytes to represent a synapse; four for the single-precision weight (32 bits), three for the destination neuron’s id (17 bits), and one for the synaptic delay (6 bits).¹⁰ Listing 13 shows how `syns_from` performs index look-ups. The two-dimensional array X represents the index and S the synapse array so that $X[n, d_{\text{from}}]$ contains the index in S where the first synapse of neuron n with delay d_{from} is stored. Ideally, the synapses should also be prefetched.

3.6 Disjoint Synapses

A headache for push-based spike transfer is the data race caused by multiple synapses delivering current received by the same neuron at the same time step.

¹⁰With just-in-time transfer the delay is also stored implicitly.

```

1 parfor c in range(N_CLS):
2     for i in contents(Q):
3         syns = syns_from(i, c)
4         for j, d, w in syns:
5             W[t + d, j] += w
6 Q = clear(Q)

```

Figure 14: Spike transfer with partitioned synapses

```

1 o0 = X[i][0]
2 o1 = X[i][D_MAX]
3 parfor c in range(N_CLS):
4     for o in range(o0 + c, o1, N_CLS):
5         j, d, w = S[o]
6         W[(t + d) % D_MAX, j] += w

```

Figure 15: Spike transfer over interleaved synaptic storage.

This is why we can’t use `parfor` on line 5 of listing 10, for example. We cannot completely solve this problem, but we can alleviate it by partitioning the synapses into disjoint *classes*, so that synapses of different classes never trigger writes to the same memory addresses at the same time. Then every class can be handled by a separate thread. The pseudocode in listing 14 modifies the transfer phase of the deferred push algorithm from 10 to exploit of this idea.¹¹ The constant `N_CLS` denotes the number of synapse classes and the extra argument to `syns_from` which synapse class to query. Many ways of partitioning the synapses into disjoint classes are possible. For example, by delay so that one thread writes one-time step synapses, the next thread two-time steps synapses, and so on. Another by destination neuron so that one thread handles synapses going to neurons whose index is between 0 and 199, another those between 200 and 399, and so on.

We choose to partition by the destination neuron’s congruence class. The method has low overhead and evenly distributes the synapses over the classes since

¹¹We can of course use the same technique to improve all other algorithms we have discussed.

the least significant bits of the neuron index is almost random. To retain the contiguous storage we interleave the synapses. That is, if a neuron’s synapses are found at indexes o_0 to $o_1 - 1$, then all synapses of class c are stored at indices $o_0 + n_c i + c$, where n_c is the number of classes and i is a non-negative integer. Interleaving synergizes with banked memory common on many GPUs. On our FPGA, it means that we can have conflict-free dedicated memory ports for every synapse class. The method causes some memory waste however. For example, if there are four classes and all destination neuron indices of all synapses of some neuron happen to be congruent with $2 \bmod 4$, then all indices other than $o_0 + 4i + 2$ will be vacant. I.e., 75% of the space will go to waste. In general, the memory consumption for storing a neuron’s synapses grows from $n_s s$, where n_s is its number of synapses and s the synapse size to $n_c l s$, where l is the number of synapses in the neuron’s largest class. In practice, the memory waste is manageable; in the order of 5-30% depending on the number of classes and horizon. The more the classes and the shorter the horizon the more uneven the classes become and the more waste. We do not mark vacancies and instead fill them with synapses carrying no current and terminating at idempotent neurons. This way, the code in listing 15 does not need to check whether the index is vacant.

3.7 More on Parallelism

Our algorithms’ source of parallelism is the `parfor` keyword. Iterations of such loops are independent and can be executed concurrently in duplicated hardware. The replication is realized differently on different targets. On CPUs we use SIMD and on GPUs the single-instruction multiple threads (SIMT) execution model “automatically” parallelizes the computation. With Intel’s OpenCL SDK we use the `#pragma unroll N` and `#pragma ivdep` compiler directives to instruct the compiler to replicate loop hardware. Essentially, this “widens” data paths, allowing more data to be processed in parallel. But FPGAs can also run more data paths in parallel, akin to how multiple CPU cores can run multiple threads. We implement this parallelism by dividing the algo-

<pre> 1 parfor i in range(n_neurons): 2 {Update neuron state as before} 3 if spikes: 4 write(to_transfer, i) 5 write(to_update, DONE) 6 read(to_update) </pre>	<pre> 1 while True: 2 i = read(to_transfer) 3 if i == DONE: 4 break 5 parfor j, d, w in syns_from(i): 6 W[t + d, j] += w 7 write(to_update, True) </pre>
(a) Update kernel	(b) Transfer kernel

Figure 16: Basic spike transfer using two kernels

rithms over multiple concurrent communicating kernels.

Listing 16 restructures the deferred push-based spike transfer algorithm in 10 as two kernels. It uses two blocking FIFOs, `to_transfer` and `to_update`, that the kernels can `read` and `write` to. When a neuron spikes, the update kernel sends its index to the spiking kernel which activates that neuron’s synapses. Hence, the neuron update and spike transfer phases run in parallel. When the update kernel has updated all neurons it sends a `DONE` message and waits for a reply from the spike transfer kernel. When it receives one, it knows that the spike transfer kernel has transferred all spikes and it proceeds to the next time step. As inter-kernel communication is not well-supported in OpenCL, we implement it using the low-latency Intel-specific channel extension.

Listing 17 similarly restructures the horizon-based algorithm as two kernels. Since the `W` array is stored in local memory, the multi-kernel approach requires an explicit synchronization step (lines 1 and 2, and 1 to 3) for sending presynaptic current from the transfer kernel to the update kernel over the `to_update` channel. After synchronization, the transfer kernel can freely write synaptic current from queued neurons to `W`. While updating neurons, the update kernel keeps track of which of them spiked and sends them to the transfer kernel. The transfer kernel adds them to its queue `Q`.

3.8 Implementations

Given the algorithms and data structures presented, we have created twelve simulator families arranged into a taxonomy in figure 18. The taxonomy’s second level groups the simulators on their spike transfer algorithm. It can either be the basic push-algorithm from listing 8 (`gmem`), the just-in-time algorithm from listing 11 (`jit`), or the horizon-based algorithm from listing 12 (`horiz`). These three algorithms differ in how eagerly they activate synapses. The `gmem` algorithm is maximally eager and activates all the spiking neuron’s synapses at once. Thus, it has to use global off-chip memory to transfer spikes, hence its name. The `jit` algorithm is maximally lazy and only activates synapses whose current should arrive at the next time step. The horizon algorithm is a mix of the two and activates as many synapses as its horizon allows. The taxonomy’s third level shows whether the simulator uses multiple communicating kernels (`multi`) or a single, monolithic kernel (`mono`). The last level specifies whether the simulator uses double (`d`) or single precision (`s`) floating point values for neuron state. This applies to the membrane potentials, presynaptic currents, and their related coefficients, but not to the spikes and spike transfer buffers which always are in single-precision. In the following, we use a notation based on their taxonomic grouping to refer to the simulators. For example, “`horiz/multi/s`” refers to the horizon-based multi-kernel single-precision simulators.

All simulators also have parameters for tuning their performance characteristics. Some are unique to cer-

```

1 for i in range(n_neurons):
2     I[i] += read(to_update)
3 A = []
4 for i in range(n_neurons):
5     {Update neuron state as before}
6     if spiked[i]:
7         A.append(i)
8 write(to_transfer, A)

```

(a) Update kernel

```

1 for i in range(n_neurons):
2     write(to_update, W[t % H, i])
3     W[t % H, i] = 0
4 for i in range(D_MAX / H):
5     rt = (t - H*i - 1) % D_MAX
6     for n in enqueue_at(Q, rt):
7         syns = syns_from(n, rt + H)
8         parfor j, d, w in syns:
9             W[(d + t) % H, j] += w
10 for n in read(to_transfer):
11     Q = enqueue(Q, t % D_MAX, n)

```

(b) Transfer kernel

Figure 17: Horizon-based spike transfer using two kernels

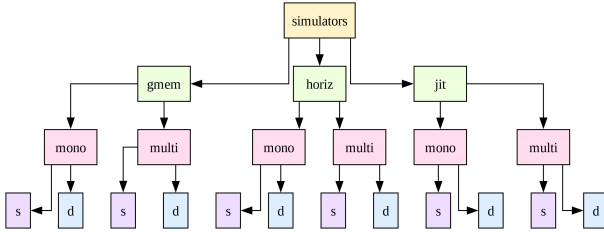


Figure 18: Taxonomy of our simulator implementations

tain families while others are shared. The *update width* is the most fundamental one and controls how many neurons the simulator updates at once. We implement it by unrolling loops such as the loop on line 1 to 4 of listing 11. More loop-unrolling is not always better as it causes the compiler to duplicate the hardware used to synthesize those loops. The *gnem*/*mono* and *gnem*/*multi* simulators have a *synapse unroll* (SU) parameter which controls how many times their spike transfer loops are unrolled (line 6 and 7 of listing 10 and lines 5 and 6 of listing 16). More unrolling allows more synapses to be fetched from memory simultaneously. The *synapse classes* (SC) parameter determines the number of congruence classes (see section 3.6). More classes allows more synapses to be activated in parallel, but also increases memory usage. The *horiz* simulators have a *horizon length* (H)

parameter which determines how many time steps worth of synapses the simulator activates at once. The *jit* simulators have a *lane count* (LC) parameter that controls how many lanes they use to transfer current. For convenience and efficiency, all parameters are positive powers of two.

4 Results

We evaluate our simulators on three axes – correctness, speed, and energy consumption – using the same experimental regimen for each. We synthesize every simulator for our FPGA with Intel FPGA SDK for OpenCL version 21.2 and use it to simulate ten seconds of biological time with a time step of 0.1 ms (i.e. 100 000 ticks) on ten different random microcircuit instances. From these 100 runs we compute mean values. We run all simulators on the full version (scale 1.0) of the microcircuit. Due to time constraints, we only synthesize a handful of simulators for every family and parameter combination. Some simulators could achieve better performance with a more exhaustive search.

4.1 Correctness

We verify our simulators' correctness using the same methods Knight and Nowotny (2018); van Albada

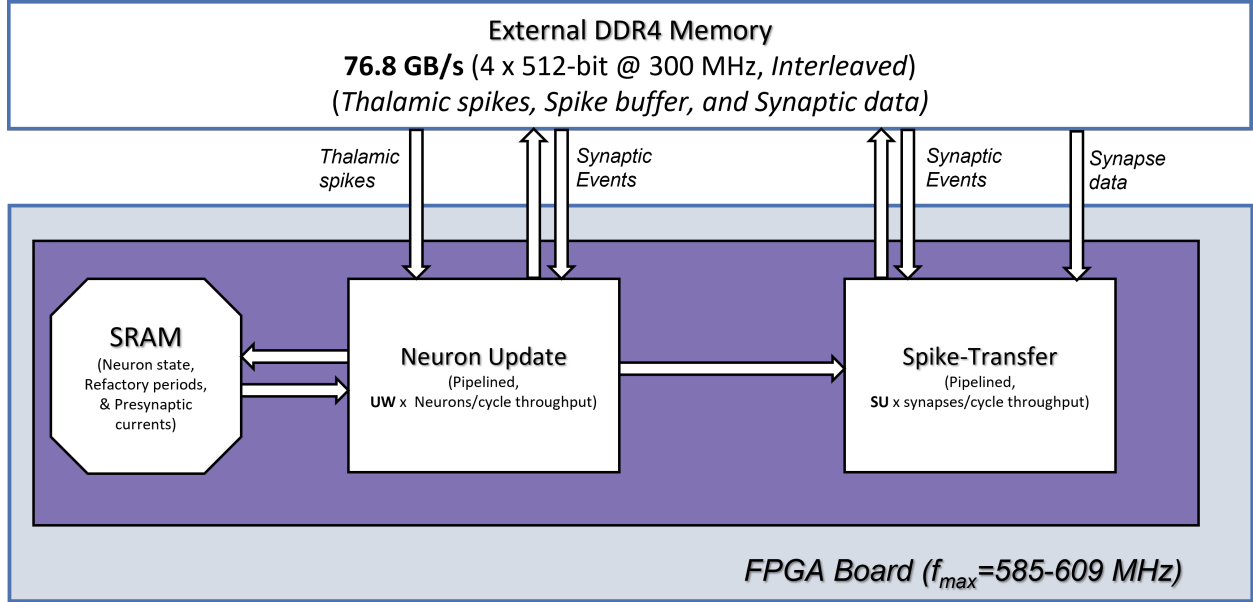


Figure 19: Schematic of the monolithic gmem implementations

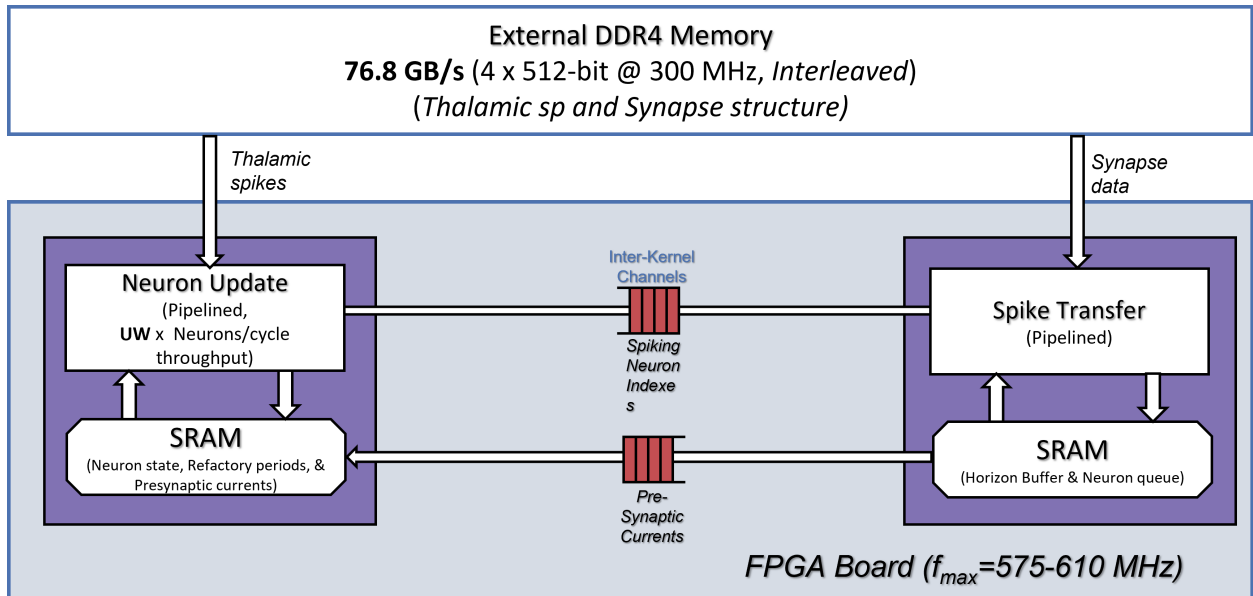


Figure 20: Schematic of the multi-kernel horiz implementations

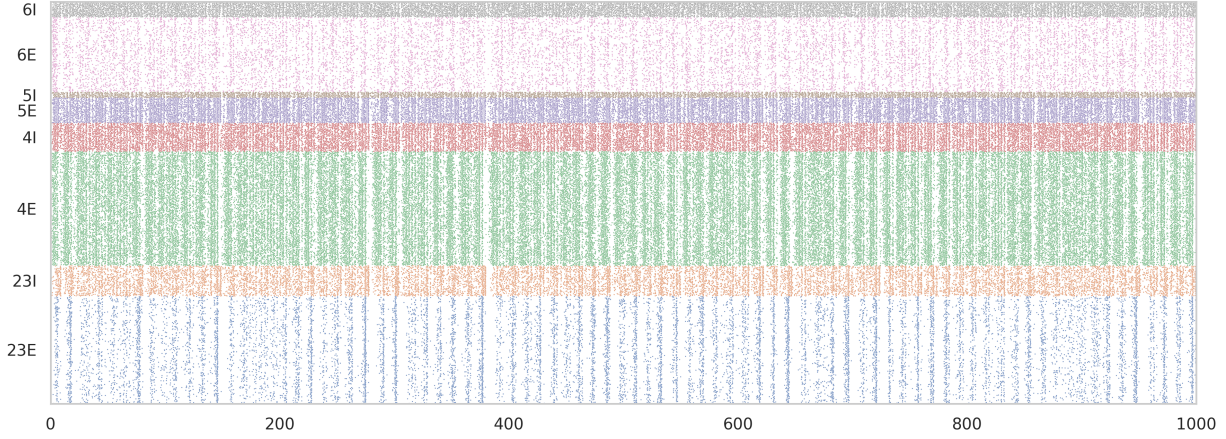


Figure 21: Spike plot of the first 1000 ms of simulation. The rhythmic nature of the microcircuit’s spiking pattern is apparent.

et al. (2018); (Potjans and Diesmann, 2014); and others used. I.e., we simulate the microcircuit initialized with the same random seed with our simulators and with grid-based (double-precision) NEST which we treat as our reference. We discard spikes during the first second and compute the following statistics over the remaining nine seconds over the eight populations; rate of spikes per second, covariant of interspike intervals, and Pearson correlation coefficient over binned spike trains. We smooth each distribution with Gaussian kernel density estimation with bandwidth selected using Scott’s Rule.

Figure 22 shows the distributions plotted for NEST in blue, the gmem/mono/s family in gold, and the jit/multi/s family in green. The plots indicate that the simulators produce distributions that are very similar to NEST’s which implies that the accuracy loss caused by the reduced numerical precision is negligible. We do not plot the distributions for our other families of single-precision simulators as they are even closer to NEST’s distributions. Neither do we plot distributions for our double-precision simulators as they produce results that are spike-for-spike identical to NEST. The Kullback-Leibler (KL) divergence between the distributions, shown in figure 23, quantifies the apparent similarities. The figure’s blue,

gold, and green bars show the KL divergences between two NEST simulations initialized with different random seeds, between NEST and gmem/mono/s, and between NEST and jit/mono/s. The latter two initialized with identical seeds. The divergence between two random seeds are much larger than between NEST and our simulators, indicating that they are accurate.

The non-associativity of IEEE 754 floating-point is the main reason for the small differences. The order presynaptic current is added depends on the spike transfer algorithm, which affects rounding. The differences are stochastic and do not bias the result.

4.2 Simulation speed

Table 4 presents the performance and resource usage of our fastest simulator configurations. The first column shows the simulator’s family in slash-notation (see section 3.8). The next five its parameters; *update width* (UW), *synapse unroll* (SU), *horizon length* (H), *synapse classes* (SC), and *lane count* (LC). The following column shows its real-time factor (RTF), defined as the time taken to run the simulation – wall-clock time – divided by the duration of the simulated biological time (10 seconds). We measure the wall-clock time as the time from the first

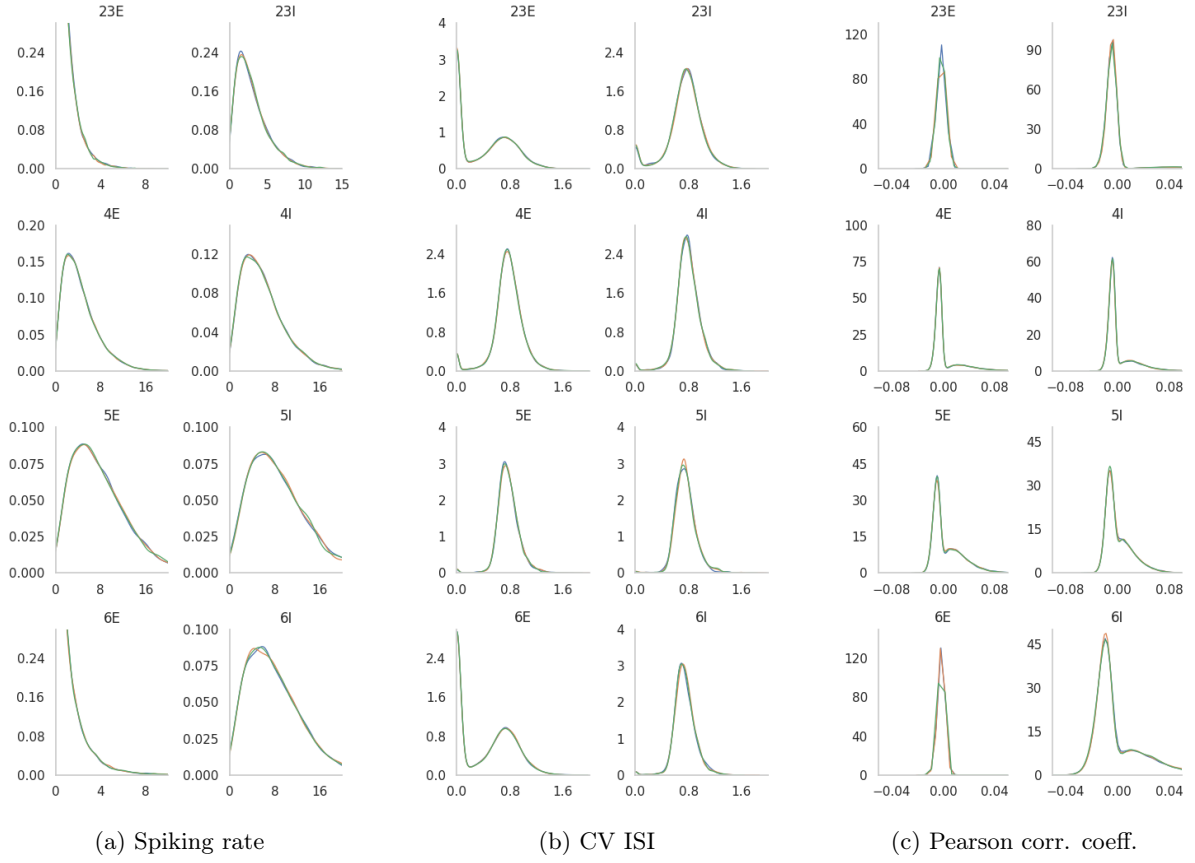


Figure 22: Kernel density estimates of **a)** spikes per second, **b)** covariance of interspike intervals, and **c)** Pearson correlation coefficient between binned spike trains for neuron samples. Blue lines for NEST, gold for gmem/mono/s, and green for jit/mono/s.

Simulator	UW	SU	H	SC	LC	RTF	Freq.	ALUT	Reg.	ALM	M20K	DSP
gmem/mono/s	8	2	n/a	n/a	n/a	4.79	601	126k	313k	21%	23%	1%
"	64	2	n/a	n/a	n/a	4.93	585	155k	391k	26%	24%	2%
gmem/mono/d	8	1	n/a	n/a	n/a	5.38	608	141k	371k	24%	27%	2%
"	8	2	n/a	n/a	n/a	4.71	608	147k	394k	25%	28%	2%
"	16	2	n/a	n/a	n/a	4.61	605	182k	477k	30%	28%	4%
gmem/multi/s	4	4	n/a	n/a	n/a	4.91	600	122k	294k	20%	22%	0%
"	64	1	n/a	n/a	n/a	5.65	585	128k	340k	22%	22%	6%
gmem/multi/d	8	1	n/a	n/a	n/a	5.52	609	139k	365k	23%	26%	2%
"	8	2	n/a	n/a	n/a	4.86	605	145k	360k	24%	27%	2%
horiz/mono/s	32	n/a	16	32	n/a	0.81	608	127k	329k	22%	79%	4%
"	64	n/a	16	16	n/a	0.85	604	143k	375k	25%	80%	7%
horiz/mono/d	4	n/a	8	4	n/a	1.74	609	122k	333k	21%	55%	1%
"	32	n/a	16	32	n/a	0.82	601	258k	653k	42%	85%	9%
horiz/multi/s	4	n/a	8	4	n/a	1.73	610	108k	287k	19%	51%	1%
"	16	n/a	16	16	n/a	0.81	605	123k	330k	22%	80%	2%
"	32	n/a	16	32	n/a	0.79	601	133k	333k	23%	81%	4%
horiz/multi/d	4	n/a	8	4	n/a	1.79	583	126k	319k	22%	57%	1%
"	16	n/a	16	16	n/a	0.80	607	187k	492k	32%	85%	5%
"	32	n/a	16	32	n/a	0.79	600	265k	664k	43%	86%	9%
jit/mono/s	16	n/a	n/a	16	16	1.47	590	116k	319k	22%	79%	7%
jit/mono/d	4	n/a	n/a	4	8	2.23	611	122k	317k	21%	56%	2%
"	16	n/a	n/a	16	16	1.44	597	182k	478k	32%	85%	10%
"	32	n/a	n/a	32	16	1.50	584	268k	704k	44%	85%	20%
jit/multi/s	16	n/a	n/a	16	16	1.43	593	120k	325k	22%	79%	7%
jit/multi/d	4	n/a	n/a	4	8	2.21	600	125k	317k	22%	55%	2%
"	8	n/a	n/a	8	16	1.56	602	147k	393k	26%	84%	5%
"	16	n/a	n/a	16	16	1.34	606	186k	492k	33%	85%	10%

Table 4: Speed and resource usage for some simulators. The table includes each family’s fastest simulator and some others for comparison purposes.

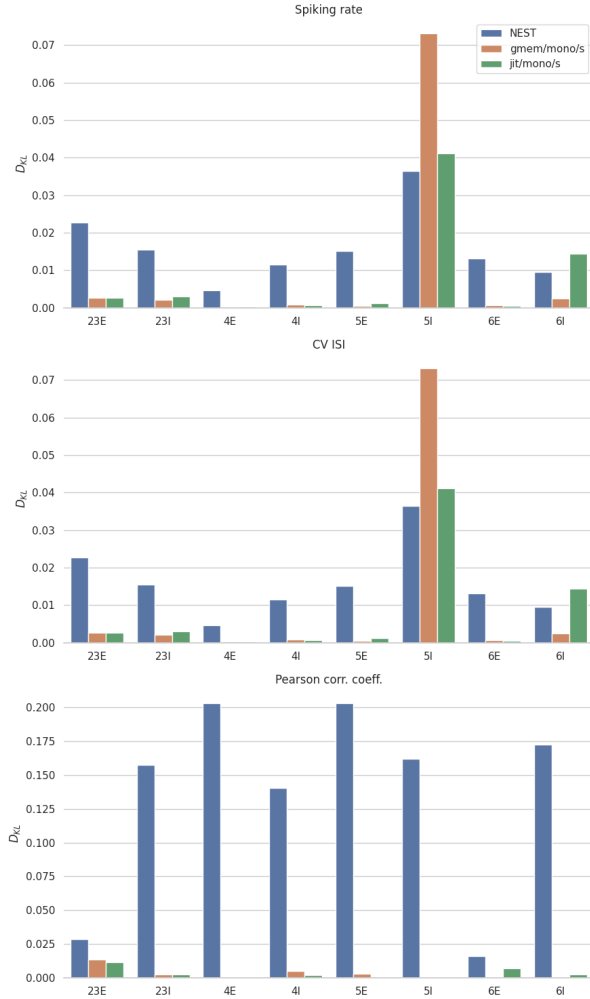


Figure 23: Kullback-Leibler (KL) divergence between distributions of single-neuron firing rates, CV ISIs, and Pearson correlation coefficients. Blue bars show KL divergences between NEST simulations run on different random seeds, gold and green bars between NEST and gmem/mono/s and jit/mono/s run on the same random seeds.

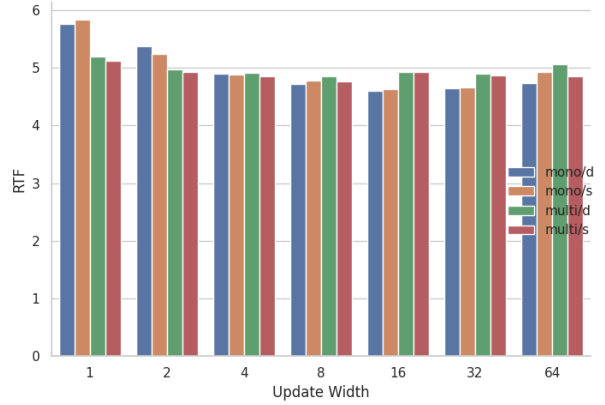


Figure 24: RTF as a function of update width for the fastest gmem/mono/s (blue), gmem/mono/d (gold), gmem/multi/s (green), and gmem/multi/d (red) simulators. Update widths larger than four does not decrease RTF.

`clEnqueueNDRangeKernel` call until the final simulation result can be read from the FPGA's memory. As the RTF does not vary beyond two decimal places even on runs on networks initialized with different random seeds, we just report its mean. The next column shows the simulator's operating frequency in MHz and FPGA resource usage. Figure 26 plots the RTF of the fastest simulators from every family as a bar chart.

A startling finding is that there is no significant performance difference between double and single precision neuron state and between multi- and single-kernel implementations. The fastest horiz/mono/s and horiz/multi/d simulators RTF is 0.81 and 0.79 which is very close to each other. Presumably, the spike transfer phase is much more expensive than the neuron update phase so making the latter run faster, either by using single-precision or by overlapping the update and transfer phases, does not improve performance. For the same reason, increasing the gmem simulator's *update width* is ineffective, as figure 24 shows. Doubling the *update width* roughly halves the number of cycles spent updating neuron state, but even that is insignificant. The performance even deteriorates for *update width* larger than 16. Likely

because the FPGA’s DDR interface is 512 bits wide.

Neither the multi- nor single-kernel gmem simulators benefit from more *synapse unroll*. The reason could be because the unrolled versions of the loop contains a false memory dependency. There is no way of letting the OpenCL compiler know that two iterations of the loop body – $W[t + d, j] += w -$ writes to distinct memory locations so the compiler refuses to schedule multiple writes per clock cycle.

The jit and horiz simulators are markedly faster than the gmem simulators because they transfer spikes in on-chip memory. The best gmem simulator has an RTF of 4.61 while it is 1.50 for the best jit simulator and 0.79 for the best horiz simulator. The latter simulators also uses up to 90% of the on-chip memory for storing the horizon and lane buffers. The results suggest that the larger these buffers are the better the performance. Unfortunately, our FPGA can not fit horizons longer than 16 time steps or more than 16 lanes. The horiz simulators’ performance edge over the jit simulators is due to them running the spike transfer loop fewer times. The jit simulators run it $d_{\max}$ times (i.e. 64) for every spiking neuron, while the horizon simulators only runs it $d_{\max}/h$ times (i.e., $64/16=4$ times). The horiz simulators also does not have to sum the incoming current from multiple lanes when updating the neurons state.

Figure 25 plots RTF as a function of the number of *synapse classes* for the horiz and jit simulators. For both, performance improves until the parameter reaches 32. The more classes, the more synapses can be activated in parallel which, clearly, is important for performance. The drawback of increasing the number of classes is memory waste. Especially for the jit simulators which iterate many fewer times per spike transfer loop. With 16 classes the average occupancy is only about 74%. It decreases to 47% with 32 classes, meaning that the simulator accesses more than twice as much data than it needs. Interestingly, there is no performance penalty. The benefits of handling many synapses in parallel is worth lots of extra off-chip memory reads. Perhaps since we store the synapses in contiguous memory reading them is quite cheap.

All simulators run at around 600 MHz, considered very good for HLS. In our experience, operating fre-

quencies much lower than that were caused by inefficiently banked on-chip memory, unaccounted for loop-carried dependencies, or similar issues.

4.3 Energy Usage

We use the Terasic Dashboard GUI to measure the energy usage of our fastest simulator – horiz/multi/s with the parameters $H=16$, $UW=32$, and $SC=32$. The dashboard connects to the Agilex 7 via a MAX 10 device that continuously monitors the voltage and current rails going into the board and the FPGA itself (Intel, 2023), allowing us to measure both a “pessimistic” power consumption for the whole board (which includes unused peripherals that draw power) and an “optimistic” one for only the FPGA fabric. Due to the device’s low sampling rate we compute the total energy usage as the product of the maximum power draw and the simulation time. According to our measurements, the simulator pessimistically requires $44.9 \text{ W} \cdot 8.13 \text{ s} = 101 \text{ mWh}$ and optimistically $16.3 \text{ W} \cdot 8.13 \text{ s} = 37 \text{ mWh}$ to simulate 10 seconds of biological time. The pessimistic energy per synaptic event (metric defined in van Albada et al. (2018)) is 21 nJ and the optimistic one 9 nJ. These values are upper bounds and the actual energy usage may be lower.

5 Discussion

Our main contribution is the presentation and analysis of methods for creating FPGA-based simulators competitive in speed and energy usage with the state-of-the-art in SNN simulation. We hope that others will create even faster simulators by adopting and refining our algorithms and implementation techniques for their hardware. Some methods are endogenous to our hardware. For example, by running multiple kernels in parallel that communicate with each other via channels, we overlap different phases of the simulation algorithm. This technique has no direct GPU-equivalent as different kernel types cannot communicate. It also relies on the Intel-specific channel extension and may not work well even on other vendors’ FPGAs. On the other hand, the technique for inter-

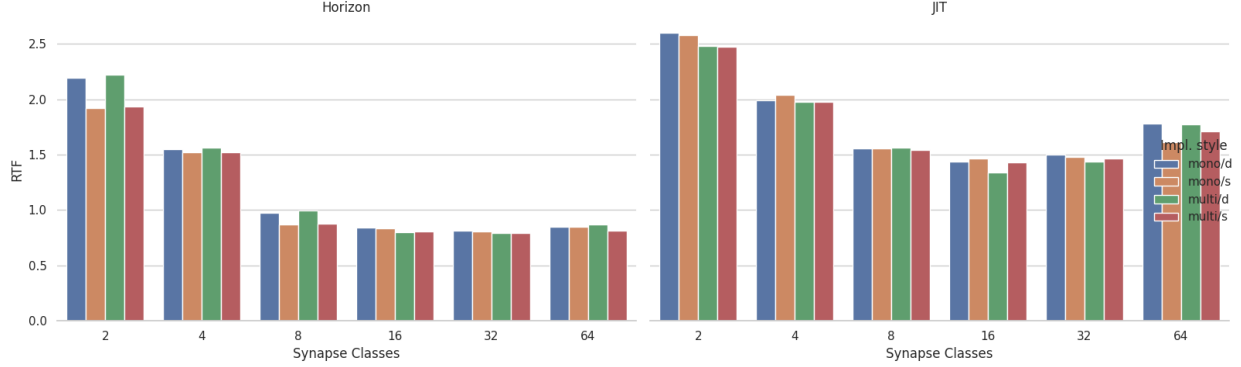


Figure 25: The horizon and JIT algorithms’ RTF as a function of the number of *synapse classes*. The *horizon length* and *lane count* parameters are both 16. The four colors represent the four implementation styles.

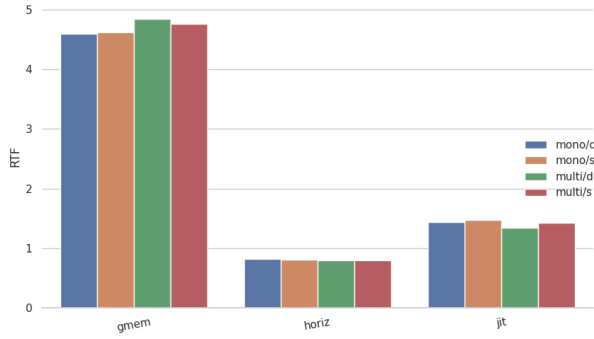


Figure 26: RTF of the fastest simulator of each type. The four colors represent the four implementation styles.

leaving synapses to improve banking and reduce conflicts (section 3.6) is adaptable to non-FPGA hardware. The same goes for the horizon-based transfer algorithm which should suit any device with enough fast local memory.

Push-based SNN simulation mandates irregular memory accesses and has low computational intensity; qualities that it shares with many other graph processing problems. This is evidenced by the fact that most of our simulators use less than 10% of the FPGA’s DSP resources, while many consume over 80% of its on-chip memory. More or even faster arithmetic resources would be useless. However, more on-chip memory would be very beneficial because we could use the basic spike transfer algorithm and would not need to bother with the more complex horizon algorithm. And higher bandwidth and lower latency memories would allow the simulators to transmit spikes faster. This finding implies that simulators would probably achieve excellent performance with neuron models more complex than LIF – as long as the interactions between neurons remain the same and as long as their memory consumption does not grow.

5.1 FPGAs for HPC

We believe that FPGAs have unique advantages for HPC; they can be tailored for the problem at hand

thanks to their malleability. However, FPGAs and particularly *tools for designing for FPGAs* have many glaring weaknesses. Unlike compilers for software, which emit the same machine code for the same source code, synthesizers use optimization algorithms based on randomness so good performance is contingent on choosing lucky random seeds. Even if the differences between lucky and unlucky seeds are small, the randomness makes squeezing out the last few percent of performance frustrating. An algorithmic change or parameter tuning causing a small performance improvement may be due to chance. Moreover, a single synthesis can take several hours even on top-of-the-line hardware, compounding this problem. Writing performance-critical code is an experimental process, wherein one needs to test hundreds or thousands of ideas to see what yields the best performance. Long turn-around times slows the process down. Simulators do not help as their performance does not reflect the performance of real hardware. For these reasons, FPGAs are decidedly more complex to develop for than GPUs.

What is a good distribution of FPGA resources for HPC? For us, our board’s distribution is far from perfect. Our fastest designs used almost all on-chip memory, with plenty of ALUTs, FFs, and DSPs to spare. If our use-case and designs are close to the norm then it would be wise for FPGA vendors to trade-off logic resources for more on-chip memory. And trends in the HPC field indicate that memory – not arithmetic – very much is a limiting factor for many problems. However, inevitably, whatever resource distribution the vendor chooses, it will not be ideal for some designs. Optimizing an FPGA so that as many designs as possible can take advantage of as much of its resources as possible seems exceedingly difficult.

5.2 HLS for HPC

In this work we choose HLS over traditional design methods because of its purported productivity advantages. How much performance did HLS cost us and how much longer would it have taken us to implement the simulators in an HDL? The question is an instance of the classic dichotomy between perfor-

mance and productivity that appears in many corners of computer science. The literature suggests that in general the productivity gains of HLS are large and the performance losses are either non-existent or low (Lahti et al., 2019; Pelcat et al., 2016). However, one can ask whether this holds true for performance-critical design?

HLS definitely allowed us to explore algorithmic ideas at a rapid pace. In particular, scheduling stallable loops and interfacing with DDR would have been at least an order of magnitude more work to implement (and debug!) in an HDL. It also helped that most – but not all – of the OpenCL code we wrote were runnable verbatim on non-FPGA targets. However, for performance judicious use of compiler directives is essential, something we struggled with. For example, adding `#pragma disable_loop_pipelining` on a loop in a gmem simulator increased its operating frequency by almost 200 MHz since the compiler could deduce that a memory port would not be shared across loop iterations. On the other hand, removing the directive on a loop in a horiz simulator more than doubled its performance! On several occasions, misplaced `#pragma ivdep` directives caused bugs that were difficult to troubleshoot. Having to “nudge” the compiler via directives to make the right decisions made us feel like we were not fully in control and was at times frustrating.

As we have no baseline to compare with, estimating the performance cost of HLS is difficult. Our designs run at over 600 MHz which – while a fair bit lower than the theoretical limit – is in the upper range of what typical Agilex 7 HDL designs runs at. If we are correct in that performance is mostly bounded by memory resources, then non-optimal performance is due to algorithmic choices and not due to the choice of implementation language.

5.3 Future Research

Time constraints forced us to leave many ideas for better performance unexplored. We list some of them here.

Our results demonstrate that single-precision floating-point is sufficient. Half-precision or some

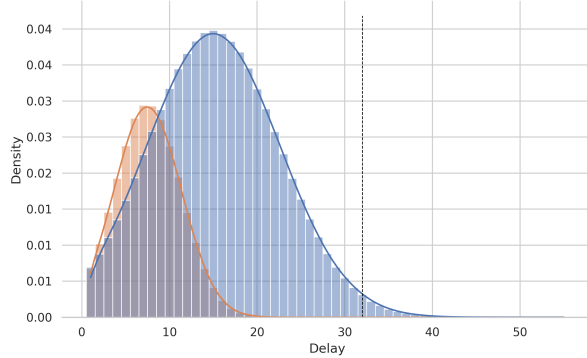


Figure 27: Synaptic delay distributions for excitatory (blue) and inhibitory synapses (gold). Almost all probability mass lies to the left of 32 (dashed line).

16-bit fixed-point representation could also be sufficient. If so, four bytes would be enough to represent synapses which would – more or less – cut our simulator’s memory demand in half.¹² And, as the synapses current is sampled from Gaussians with known means, the representation’s size could perhaps be reduced further by storing the current as *deviations from the mean*. Presumably, fewer bits are needed to represent the deviations accurately since they are rather small.

Along the same lines, one could exploit the fact that, as figure 27 shows, synaptic delays are not uniformly distributed. Something our algorithms are oblivious to. It means, for example, that the gmem simulators need to use off-chip memory for the spike transfer array. One could perhaps retain the spike transfer array for “fast” synapses with delays shorter than, say, 32 and another more space-efficient format for “slow” synapses with longer delays.

As we have emphasized, we cannot safely transfer spikes from two neurons simultaneously as they may write to the same memory locations. But this is only true if they have synapses with the same destination neuron and delay. We can apply the idea from section 3.6 on the “neuron level” and partition the neurons into disjoint classes so that synapses of

¹²Assuming we store some bits of the delay and destination neuron index implicitly in the index.

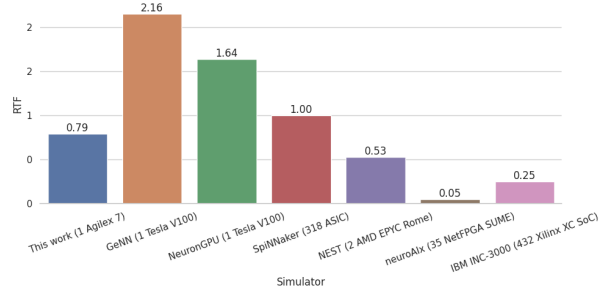


Figure 28: RTFs of some SNN simulators

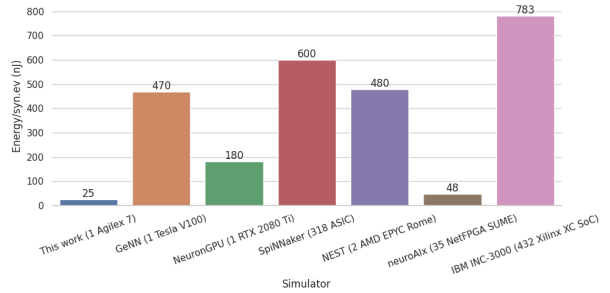


Figure 29: Energy per synaptic event of some SNN simulators

neurons of different classes never trigger writes to the same memory locations.

Our FPGA has four DDR memories each equipped with a 512 bit read/write port. We are likely not utilizing all bandwidth properly since we use the compiler’s default off-chip memory organization. Plausibly, we could see a substantial performance improvement by carefully laying out off-chip memory manually. For example, by distributing the synaptic data (by far the largest data structure) over the four memories.

Another line of research could be to deploy a cluster of FPGAs for SNN simulation. As the communication between the devices likely becomes the bottleneck, it is a much different problem from using one device.

Work	Simulator	Hardware	Node	RTF	Syn.Ev	En.
This work	horiz/mono/s	1 Agilx 7 FPGA	10	0.81		25
Heittmann et al. (2022)	IBM INC-3000	432 Xilinx XC Z7045 SoC	28	0.25		783
Kauth et al. (2023b)	neuroAIx	35 NetFPGA SUME	28	0.05		48
Golosio et al. (2021)	NeuronGPU	1 GeForce RTX 2080 Ti	12	1.06		180
Golosio et al. (2021)	NeuronGPU	1 Tesla V100	12	1.64		-
Knight and Nowotny (2018)	GeNN	1 GeForce RTX 2080 Ti	12	1.40		-
Knight and Nowotny (2018)	GeNN	1 Tesla V100	12	2.16		470
van Albada et al. (2018)	SpiNNaker	217 ASIC	130	20.00		5900
Rhodes et al. (2020)	SpiNNaker	318 ASIC	130	1.00		600
Kurth et al. (2022)	NEST	2 AMD EPYC Rome	14	0.53		480

Table 5: State-of-the-art in hardware-accelerated microcircuit simulation. Node refers to the technology node in nanometers, RTF how much slower than realtime the simulator runs (lower is better), and Syn.Ev En. estimated energy consumption per synaptic event in nano-Joule.

5.4 State-of-the-art

While we are far from the first to have investigated FPGA-based SNN simulation, our investigation is qualitatively different from most others. First, our focus is accurate SNN simulation and not solving a specific task. Gupta et al. (2020); Han et al. (2020); Li et al. (2021); Zheng et al. (2021); Carpegna et al. (2022); Liu et al. (2023); Carpegna et al. (2024) all implement layered SNNs that excel at classifying images from the MNIST dataset. For them classification accuracy is much more important than simulation speed or energy consumption. Second, many FPGA simulators simulate SNNs much smaller than the Potjans-Diesmann microcircuit or uses non-LIF neurons, making their results incomparable to ours. For example, Pani et al. (2017) simulates up to 1,440 Izhikevich neurons in real-time and Shama et al. (2020) simulates 150 Hodgkin-Huxley neurons on a Virtex-2 FPGA. In contrast, NeuroFlow by Cheung et al. (2016) simulates up to 600,000 neurons on a cluster of six FPGAs four times slower than real-time. However, it uses a 1 ms timestep (ten times larger than the *de facto* standard 0.1 ms), organizes neurons into two-dimensional grids, and spatially constrains synapses. That is, most synapses connect to the neuron’s nearest neighbours. Trench and Morrison (2022) proposes a simulator for a Zynq-7000 SoC and measure its performance on a network of 800 excitatory and 200 inhibitory Izhikevich neurons. Their architecture divides the FPGA into 16 identical pro-

cessing blocks, each supporting 64 neurons or 1,024 neurons in total per chip. They argue that if a cluster of their devices were deployed, it could simulate the Potjans-Diesmann microcircuit seven times faster than real-time. Though they did not test their claim.

The simulators we think are the most similar to ours are: IBM INC-3000, neuroAIx, NeuronGPU, GeNN, SpiNNaker, and NEST. IBM-INC 3000 and neuroAIx are both FPGA clusters, consisting of 432 Xilinx XC boards and 35 NetFPGA SUME boards respectively. IBM-INC 3000 is four times faster than real-time and neuroAIx twenty time faster than real-time (Kauth et al., 2023a; Heittmann et al., 2022). GeNN and NeuronGPU are both GPU-based simulators written in CUDA and runs on NVIDIA’s line of GPUs (Knight and Nowotny, 2018; Golosio et al., 2021). On a Tesla V100 GPU, GeNN simulate the microcircuit at about half the speed of real-time and NeuronGPU 5% slower than real-time. SpiNNaker is an older dedicated neuromorphic hardware system built out of hundreds of ASICs (Furber et al., 2013). NEST is a CPU-based simulator which runs twice the speed of real-time on two AMD EPYC Rome nodes (Kurth et al., 2022).

Table 5 and figures 28 and 29 show how our best simulator stack up against the competition. The energy usage of our fastest simulator compares favorably to other results. Likely partially due to the Agilx 7’s smaller technology node and to us confining our implementation to one FPGA. Kauth et al.

(2023b) report a much lower RTF than us, but use 35 boards, each drawing 26.54 W on average, resulting in a higher total energy usage for the same amount of biological time. They, like us, report an “all-inclusive” value for the energy usage so a dedicated platform – without any unused peripherals – could consume much less energy. It goes without saying that fairly comparing performance of systems implemented on different architectures, with different design trade-offs, and accuracy constraints is tricky. One system with excellent performance may be inadequate in other regards. For example, less configurability may improve a system’s performance, but make it unusable for certain applications. It should be noted that most simulators, unlike ours, replace thalamic spikes with DC input which limits their applicability.

Author Contributions

BAL and AP designed the study. BAL implemented the SNN framework and performed the experiments. BAL and AP analyzed the results and co-wrote the paper.

Funding

This work was supported by the Swedish Research Council through the project “Building Digital Brains” (grant reference 2021-04579).

Acknowledgments

TODO

Supplemental Data

[Supplementary Material](#) should be uploaded separately on submission, if there are Supplementary Figures, please include the caption in the same file as the figure. LaTeX Supplementary Material templates can be found in the Frontiers LaTeX folder.

Data Availability Statement

All source code will eventually be available under a permissible Open Source license.

References

- Ahangari, H., Özdal, M. M., and Öztürk, O. (2023). Hls-based high-throughput and work-efficient synthesizable graph processing template pipeline. *ACM Trans. Embed. Comput. Syst.* 22. doi:10.1145/3529256
- Aminian, M., Saeedi, M., Zamani, M. S., and Sedighi, M. (2008). Fpga-based circuit model emulation of quantum algorithms. In *2008 IEEE Computer Society Annual Symposium on VLSI* (IEEE), 399–404
- Besta, M., Podstawski, M., Groner, L., Solomonik, E., and Hoefler, T. (2017). To push or to pull: On reducing communication and synchronization in graph computations. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*. 93–104
- Betz, V. and Rose, J. (1999). Fpga routing architecture: Segmentation and buffering to optimize speed and density. In *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*. 59–68
- Bohr, M. (2007). A 30 year retrospective on denard’s mosfet scaling paper. *IEEE Solid-State Circuits Society Newsletter* 12, 11–13
- Boku, T., Fujita, N., Kobayashi, R., and Tatebe, O. (2022). Cygnus-world first multihybrid accelerated cluster with gpu and fpga coupling. In *Workshop Proceedings of the 51st International Conference on Parallel Processing*. 1–8
- Brette, R., Rudolph, M., Carnevale, T., Hines, M., Beeman, D., Bower, J. M., et al. (2007). Simulation of networks of spiking neurons: a review of tools and strategies. *Journal of computational neuroscience* 23, 349–398

- Brunel, N. (2000). Persistent activity and the single-cell frequency–current curve in a cortical network model. *Network: Computation in Neural Systems* 11, 261–280
- Carpegna, A., Savino, A., and Di Carlo, S. (2022). Spiker: an fpga-optimized hardware accelerator for spiking neural networks. In *2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)* (IEEE), 14–19
- Carpegna, A., Savino, A., and Di Carlo, S. (2024). Spiker+: a framework for the generation of efficient spiking neural networks fpga accelerators for inference at the edge. *arXiv preprint arXiv:2401.01141*
- Cheung, K., Schultz, S. R., and Luk, W. (2016). Neuroflow: a general purpose spiking neural network simulation platform using customizable processors. *Frontiers in neuroscience* 9, 516
- Czajkowski, T. S., Aydonat, U., Denisenko, D., Freeman, J., Kinsner, M., Neto, D., et al. (2012). From opencl to high-performance hardware on fpgas. In *22nd international conference on field programmable logic and applications (FPL)* (IEEE), 531–534
- Del Sozzo, E., Rabozzi, M., Di Tucci, L., Scito, D., and Santambrogio, M. D. (2018). A scalable fpga design for cloud n-body simulation. In *2018 IEEE 29th international conference on application-specific systems, architectures and processors (ASAP)* (IEEE), 1–8
- Efnusheva, D., Cholakoska, A., and Tentov, A. (2017). A survey of different approaches for overcoming the processor-memory bottleneck. *International Journal of Computer Science and Information Technology* 9, 151–163
- Faj, J., Kenter, T., Faghih-Naini, S., Plessl, C., and Aizinger, V. (2023). Scalable multi-fpga design of a discontinuous galerkin shallow-water model on unstructured meshes. In *Proceedings of the Platform for Advanced Scientific Computing Conference*. 1–12
- Furber, S. B., Lester, D. R., Plana, L. A., Garside, J. D., Painkras, E., Temple, S., et al. (2013). Overview of the spinnaker system architecture. *IEEE Transactions on Computers* 62, 2454–2467. doi:10.1109/TC.2012.142
- Ghosh-Dastidar, S. and Adeli, H. (2009). Third generation neural networks: Spiking neural networks. In *Advances in computational intelligence* (Springer). 167–178
- Golosio, B., Tiddia, G., De Luca, C., Pastorelli, E., Simula, F., and Paolucci, P. S. (2021). Fast simulations of highly-connected spiking cortical models using gpus. *Frontiers in Computational Neuroscience* 15. doi:10.3389/fncom.2021.627620
- Grossman, S., Litz, H., and Kozyrakis, C. (2018). Making pull-based graph processing performant. *SIGPLAN Not.* 53, 246–260. doi:10.1145/3200691.3178506
- Gupta, S., Vyas, A., and Trivedi, G. (2020). Fpga implementation of simplified spiking neural network. In *2020 27th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. 1–4. doi:10.1109/ICECS49266.2020.9294790
- Han, J., Li, Z., Zheng, W., and Zhang, Y. (2020). Hardware implementation of spiking neural networks on fpga. *Tsinghua Science and Technology* 25, 479–486
- Hanuschkin, A., Kunkel, S., Helias, M., Morrison, A., and Diesmann, M. (2010). A general and efficient method for incorporating precise spike times in globally time-driven simulations. *Frontiers in Neuroinformatics* 4. doi:10.3389/fninf.2010.00113
- Heitmann, A., Psychou, G., Trensche, G., Cox, C. E., Wilcke, W. W., Diesmann, M., et al. (2022). Simulating the cortical microcircuit significantly faster than real time on the ibm inc-3000 neural supercomputer. *Frontiers in Neuroscience* 15. doi:10.3389/fnins.2021.728460
- Huthmann, J., Shin, A., Podobas, A., Sano, K., and Takizawa, H. (2019). Scaling performance for n-body stream computation with a ring of fpgas. In

- Proceedings of the 10th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*. 1–6
- Intel (2023). *Intel Agilex® 7 FPGA F-Series Development Kit User Guide*. Intel Corporation, 2023-06-14 edn. Available at <https://www.intel.com/content/www/us/en/docs/programmable/683024/current/overview.html>
- Jouppi, N., Young, C., Patil, N., and Patterson, D. (2018). Motivation for and evaluation of the first tensor processing unit. *IEEE Micro* 38, 10–19
- Karp, M., Podobas, A., Jansson, N., Kenter, T., Plessl, C., Schlatter, P., et al. (2021). High-performance spectral element methods on field-programmable gate arrays: implementation, evaluation, and future projection. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (IEEE), 1077–1086
- Kauth, K., Stadtmann, T., Sobhani, V., and Gemeke, T. (2023a). neuroaix: Fpga cluster for reproducible and accelerated neuroscience simulations of snns. In *2023 IEEE Nordic Circuits and Systems Conference (NorCAS)*. 1–7. doi:10.1109/NorCAS58970.2023.10305473
- Kauth, K., Stadtmann, T., Sobhani, V., and Gemeke, T. (2023b). neuroaix-framework: design of future neuroscience simulation systems exhibiting execution of the cortical microcircuit model 20× faster than biological real-time. *Frontiers in Computational Neuroscience* 17. doi:10.3389/fncom.2023.1144143
- Knight, J. C. and Nowotny, T. (2018). Gpus outperform current hpc and neuromorphic solutions in terms of speed and energy when simulating a highly-connected cortical model. *Frontiers in Neuroscience* 12. doi:10.3389/fnins.2018.00941
- Kuon, I., Tessier, R., Rose, J., et al. (2008). Fpga architecture: Survey and challenges. *Foundations and Trends® in Electronic Design Automation* 2, 135–253
- Kurth, A. C., Senk, J., Terhorst, D., Finnerty, J., and Diesmann, M. (2022). Sub-realtime simulation of a neuronal network of natural density. *Neuromorphic Computing and Engineering* 2, 021001
- Lahti, S., Sjövall, P., Vanne, J., and Hämäläinen, T. D. (2019). Are we there yet? a study on the state of high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 898–911. doi:10.1109/TCAD.2018.2834439
- [Dataset] Langhammer, M. and Constantinides, G. (2023). egpu: A 750 mhz class soft gpgpu for fpga
- Langhammer, M., Nurvitadhi, E., Pasca, B., and Gribov, S. (2021). Stratix 10 nx architecture and applications. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 57–67
- Li, S., Zhang, Z., Mao, R., Xiao, J., Chang, L., and Zhou, J. (2021). A fast and energy-efficient snn processor with adaptive clock/event-driven computation scheme and online learning. *IEEE Transactions on Circuits and Systems I: Regular Papers* 68, 1543–1552. doi:10.1109/TCSI.2021.3052885
- Liu, H., Chen, Y., Zeng, Z., Zhang, M., and Qu, H. (2023). A low power and low latency fpga-based spiking neural network accelerator. In *2023 International Joint Conference on Neural Networks (IJCNN)*. 1–8. doi:10.1109/IJCNN54540.2023.10191153
- Makin, S. (2019). The four biggest challenges in brain simulation (<https://www.nature.com/articles/d41586-019-02209-z>). *Nature* 571
- Menzel, J., Plessl, C., and Kenter, T. (2021). The strong scaling advantage of fpgas in hpc for n-body simulations. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 15, 1–30
- Meyer, M., Kenter, T., and Plessl, C. (2023). Multi-fpga designs and scaling of hpc challenge benchmarks via mpi and circuit-switched inter-fpga networks. *ACM Transactions on Reconfigurable Technology and Systems* 16, 1–27

- Munshi, A. (2009). The opencl specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)* (IEEE), 1–314
- Murphy, C. and Fu, Y. (2017). Xilinx all programmable devices: A superior platform for compute-intensive systems. *Xilinx White Paper*
- Nane, R., Sima, V.-M., Pilato, C., Choi, J., Fort, B., Canis, A., et al. (2015). A survey and evaluation of fpga high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 1591–1604
- O’Loughlin, D., Coffey, A., Callaly, F., Lyons, D., and Morgan, F. (2014). Xilinx vivado high level synthesis: Case studies (IET)
- Pani, D., Meloni, P., Tuveri, G., Palumbo, F., Mas-sobrio, P., and Raffo, L. (2017). An fpga platform for real-time simulation of spiking neuronal networks. *Frontiers in Neuroscience* 11. doi: 10.3389/fnins.2017.00090
- Pelcat, M., Bourrasset, C., Maggiani, L., and Berry, F. (2016). Design productivity of a high level synthesis compiler versus hdl. In *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*. 140–147. doi:10.1109/SAMOS.2016.7818341
- Perry, D. L. (2002). *VHDL: programming by example* (McGraw-Hill Education)
- Pimpini, A., Piccione, A., Ciciani, B., and Pellegrini, A. (2022). Speculative distributed simulation of very large spiking neural networks. In *Proceedings of the 2022 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (New York, NY, USA: Association for Computing Machinery), SIGSIM-PADS ’22, 93–104. doi:10.1145/3518997.3531027
- Plessner, H. E., Diesmann, M., Gewaltig, M.-O., and Morrison, A. (2015). *NEST: the Neural Simulation Tool* (New York, NY: Springer New York). 1849–1852. doi:10.1007/978-1-4614-6675-8_258
- Podobas, A. (2023). Q2logic: A coarse-grained fpga overlay targeting schrödinger quantum circuit simulations. In *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (IEEE), 460–467
- Podobas, A., Sano, K., and Matsuoka, S. (2020). A survey on coarse-grained reconfigurable architectures from a performance perspective. *IEEE Access* 8, 146719–146743
- Podobas, A., Zohouri, H. R., Maruyama, N., and Matsuoka, S. (2017). Evaluating high-level design strategies on fpgas for high-performance computing. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)* (IEEE), 1–4
- Potjans, T. C. and Diesmann, M. (2014). The cell-type specific cortical microcircuit: relating structure and activity in a full-scale spiking network model. *Cerebral cortex* 24, 785–806
- Rhodes, O., Peres, L., Rowley, A. G. D., Gait, A., Plana, L. A., Brenninkmeijer, C., et al. (2020). Real-time cortical simulation on neuromorphic hardware. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 378, 20190160. doi:10.1098/rsta.2019.0160
- Sanaullah, A. and Herbordt, M. C. (2018). Unlocking performance-programmability by penetrating the intel fpga opencl toolflow. In *2018 IEEE High Performance Extreme Computing Conference (HPEC)* (IEEE), 1–8
- Sano, K., Koshihara, A., Miyajima, T., and Ueno, T. (2023). Essper: Elastic and scalable fpga-cluster system for high-performance reconfigurable computing with supercomputer fugaku. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*. 140–150
- Shama, F., Haghiri, S., and Imani, M. A. (2020). Fpga realization of hodgkin-huxley neuronal model. *IEEE Transactions on Neural Systems and Rehabilitation Engineering* 28, 1059–1068

- Theis, T. N. and Wong, H.-S. P. (2017). The end of moore’s law: A new beginning for information technology. *Computing in science & engineering* 19, 41–50
- Trensch, G. and Morrison, A. (2022). A system-on-chip based hybrid neuromorphic compute node architecture for reproducible hyper-real-time simulations of spiking neural networks. *Frontiers in Neuroinformatics* 16. doi:10.3389/fninf.2022.884033
- Trimberger, S. M. (2015). Three ages of fpgas: A retrospective on the first thirty years of fpga technology. *Proceedings of the IEEE* 103, 318–331. doi: 10.1109/JPROC.2015.2392104
- van Albada, S. J., Rowley, A. G., Senk, J., Hopkins, M., Schmidt, M., Stokes, A. B., et al. (2018). Performance comparison of the digital neuromorphic hardware spinnaker and the neural network simulation software nest for a full-scale cortical micro-circuit model. *Frontiers in Neuroscience* 12. doi: 10.3389/fnins.2018.00291
- Versace, M. and Chandler, B. (2010). The brain of a new machine. *IEEE spectrum* 47, 30–37
- Vipin, K. and Fahmy, S. A. (2018). Fpga dynamic and partial reconfiguration: A survey of architectures, methods, and applications. *ACM Computing Surveys (CSUR)* 51, 1–39
- Zheng, H., Guo, Y., Yang, X., Xiao, S., and Yu, Z. (2021). Balancing the cost and performance trade-offs in snn processors. *IEEE Transactions on Circuits and Systems II: Express Briefs* 68, 3172–3176. doi:10.1109/TCSII.2021.3090422
- Zohouri, H. R., Maruyama, N., Smith, A., Matsuda, M., and Matsuoka, S. (2016). Evaluating and optimizing opencl kernels for high performance computing with fpgas. In *SC’16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (IEEE), 409–420