Journal of Electronics and Informatics (ISSN: 2582-3825)
www.irojournals.com/iroei/

EdgeSNN-RT: Low-Overhead Spiking Simulation on Embedded GPUs for On-Device Intelligence

Girish Chanda

Department of Computer science Engineering, Sreenidhi Institute of Science and Technology (Autonomous), Affiliated to Jawaharlal Nehru Technological University, Hyderabad, Telangana, India.

E-mail: girishchanda23@gmail.com

Abstract

EdgeSNN-RT proposes a Python-managed, GPU-accelerated spiking neural network simulation method optimized for edge platforms, integrating their definition, execution, and, critically, low-cost host-device synchronization. A dense spike bitfield logging tool groups spikes on the device, obviating the need for one-timestep communication, yielding up to 10x cost savings for logging spikes. Experiments conducted for full-scale cortical microcircuit simulation and long-term conditioning tasks with large time horizons range across variant models of diversified, consumer-class, and larger-scale, heterogeneous GPUs, from the low-power, as-specified, 15W Jetson Xavier NX, to measure the latency, kernel execution time, and interpretative overheads of their execution in the real-time regime specifically for the purposes of the challenge. The system supports real-time or faster simulation execution for today's leading-class GPUs, and, for shared-memory architectures of embedded platforms, preserves their leading performance with direct, low-level control, NumPy direct views to the managed buffers, and an events-based representation of plasticity within a convenient API. On-device inference and learning of spiking neural networks for real-world tasks will become feasible for networks of the described architecture based on this work, mapping telemetry, memory representation, and scheduling approaches to fit within limitations imposed by embedded platforms, supplanting today's restricted assumptions about direct, one-step execution communication and/or awaiting improvements in physical implementation

DOI: https://doi.org/10.36548/jei.2025.4.001
Received: 30.10.2025, received in revised form: 26.11.2025, accepted: 10.12.2025, published: 19.12.2025

technology for general programmed computation targets from desktop workstations back toward rudimentary, network-edge platforms in an industry led by programmability for applied innovation.

Keywords: Spiking Neural Networks, GPU Acceleration, Edge Computing, Embedded Systems, Python, CUDA.

1. Introduction

Spiking neural networks (SNNs) can be considered the third generation of neural network models that provide biologically feasible computing and event-driven processing solutions for various tasks [1]. Although various spiking neural network simulator tools are available, migrating such tools to resource-limited edge devices can be an issue. Typical SNN simulator tools such as NEST [2] provide optimal solutions for distributed simulations, while NEURON [3], Arbor [4], among others, provide solutions for complex compartmental modeling simulations. These solutions require considerable computational resources that are not available in resource-limited applications.

The use of Python is increasingly being embraced by the computational neuroscience and machine learning fields, advanced scientific libraries available in Python, such as NumPy and SciPy. This trend has also led to simulators developing Python interfaces. Some of these simulators include NEST, NEURON, and CARLsim. Although GeNN [10] has shown competitiveness in its C++ interface and Brian2GeNN backend implementation [11], the ability to access its full functionality from Python directly had not been possible prior to this work.

To fill this gap, EdgeSNN-RT offers a comprehensive Python interface for SNN simulations on the GPU with minimal overhead. The system described here gives scientists the ability to implement their own models of neurons and synapses by writing Python code and allows them to efficiently organize memory on the device. All these properties make our system especially useful in the context of edge computing, where memory bandwidth and power efficiency are paramount. The novelty and contributions of this work are: (1) a native Python API for defining and executing an SNN, (2) a new method for recording spikes that reduces host-device synchronization to a minimum, (3) a complete performance comparison on various architectures, and (4) real-time simulation examples on embedded systems.

2. Related Work

GPU acceleration has become increasingly important for SNN simulation, with several systems exploring this approach. CARLsim [12] provides a C++ framework for biologically detailed SNN simulation on heterogeneous clusters, while ANNarchy [13] employs code generation for parallel hardware. More recently, NeuronGPU [14] has demonstrated efficient simulation of highly-connected cortical models using GPUs. Python interfaces have become standard for modern SNN simulators. Brian 2 [15] offers a mathematical description language for neural dynamics, while PyNN [16] provides a common interface across multiple simulators. However, these systems often abstract away low-level controls that are essential for optimizing performance on specific hardware platforms.

Embedded neuromorphic systems face unique challenges related to power consumption and memory constraints. Systems like SpiNNaker [17] and Loihi [18] offer specialized hardware for SNN execution but lack the flexibility of general- purpose GPUs. The NVIDIA Jetson platform provides an interesting middle ground, combining GPU acceleration with power efficiency suitable for edge deployment. Previous work has demonstrated GeNN's performance advantages for large- scale cortical simulations [19]. However, the absence of a native Python interface limited its accessibility to the broader computational neuroscience community. EdgeSNN-RT builds upon this foundation while addressing the specific requirements of edge computing scenarios through optimized memory management and reduced synchronization overhead.

3. System Architecture

3.1 Core Design Principles

EdgeSNN-RT is designed with several principles that set it apart from existing simulators of SNNs. First, there is a strong separation between the definition and execution phases to enable high levels of optimization during code generation. Another aspect is that there is direct access to memory without copying, views that are accessible to Python. Additionally, there is heterogeneous execution that can work on various GPU platforms without changing the definition of models.

The system is based on a layer cake architecture where the high-level Python abstraction is translated into optimized CUDA code by the code generation backend of GeNN.

Such a system leverages the advantages of using Python and the efficiency of hand-optimized kernels on the GPU. EdgeSNN-RT is different from other interpreter-based systems because the simulator generates specialized code for the model that cannot be achieved by general-purpose simulators.

Memory management is a very essential design component of the system. In EdgeSNN-RT, the system uses lazy allocation schemes where memory on devices is allocated only when necessary. This also consumes less simulation time since:

- **Reduction of Memory Footprint:** Memory is allocated for those model components that are active at that point in time and not allocated for unused buffers prior to that.
- **Simulation Latency Reduction:** By reducing the amount of memory allocations and taking advantage of memory reuse from one simulation run to another, the simulation latency is reduced.

In addition to the above techniques, this system also adopts advanced memory layout optimization techniques based on the memory access patterns of both the neuron and synapse kernels to reduce memory bank conflicts and make full use of memory bandwidth.

3.2 Python

The Python interface of EdgeSNN-RT functions as the main user interface component, offering a natural and intuitive way of modeling through an API, while still enabling control over low-level performance parameters. Unlike other frameworks that use domain-specific languages and complex configuration files, EdgeSNN-RT allows users to model using conventional Python programming constructs such as dictionaries, lists, and classes.

The model description in the EdgeSNN-RT framework is declarative. The user declares the components of the model that include neurons, synapses, and current sources. The model description system translates the declarative model description into an intermediate form that is optimized before the code is generated. The declarative model description allows for the static analysis of the model structure, where the memory access pattern and the fusion of the kernels are performed.

One of the most important additions to the Python interface of EdgeSNN-RT is the seamless integration with NumPy arrays for the preparation of parameters as well as for accessing results. The parameters of the models can now be described with standard NumPy arrays, and simulation results can also be accessed using memory views, which point directly into physical device memory.

3.3 Code Generation

EDgeSNN-RT uses a code generation pipeline whereby a high-level model description is optimized into CUDA code. The process entails parsing a Python model description and forming an abstract syntax tree, which is a model description. An abstract syntax tree is transformed through a series of optimizations before forming code optimized for CUDA. The optimized AST is then translated into platform-specific code through the GeNN backend infrastructure. On the CUDA target, this includes the generation of kernel code for updating the state of the neurons, synapses, and spike transmissions. The optimal execution parameters, such as the block size, grid size, and memory allocation scheme, are automatically determined according to the characteristics of the target hardware and the network topology.

Finally, compilation results in a shared library that encapsulates the simulation logic. The shared library is dynamically loaded into the Python environment. This leads to a smooth integration between the high-level simulation framework and the low-level simulation environment. The generated code includes Python callbacks or hooks for custom simulation tasks.

3.4 Execution Workflow and End-to-End Execution Workflow

As depicted in Figure 1, EdgeSNN-RT has an organized end-to-end workflow from model definition to execution. In this Python framework, model specifications are compiled into an Abstract Syntax Tree (AST), followed by various passes of model optimizations before code compilation for execution on selected GPUs. This compiled code ensures efficiency for execution while being Python-accessible. Optimization Passes in Code Generation Code generation involves a number of optimizations passes:

- **Kernel Fusion:** Merging several small kernels into a fused kernel to decrease the cost of launching kernels.

- **Memory Access Pattern Optimization:** Reordering memory accesses for maximum coalescing
- **Register Pressure Optimization:** Balancing registers for better GPU occupancy
- **Instruction Scheduling:** Reordering instructions to overcome memory latency.

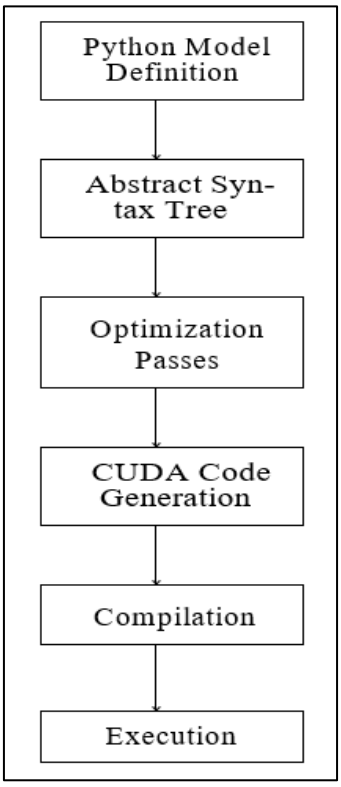

Figure 1. EdgeSNN-RT Code Generation Pipeline Transforming Python Model

Definitions into Optimized CUDA Executables Through Multiple Compilation Stages. Cross-Layer Debugging and Error Tracing EdgeSNN-RT provides comprehensive debugging capabilities across all abstraction layers. The system includes:

- Python-level error tracking with detailed stack traces
- CUDA kernel validation and bounds checking in debug mode
- Memory access pattern analysis and conflict detection
- Performance profiling at each workflow stage
- Automatic error recovery suggestions based on common patterns

This multi-layer debugging approach enables rapid identification and resolution of issues from high-level model specification to low-level GPU execution.

4. Memory Management and Spike Recording

4.1 Efficient Spike Storage

The traditional SNN simulators have the disadvantage of being inefficient in recording spikes, especially when simulating high-resolution timesteps. The traditional method of copying the spike data from the device to the host computer at each timestep generates considerable synchronization overhead that can dominate simulation time, particularly with small timesteps.

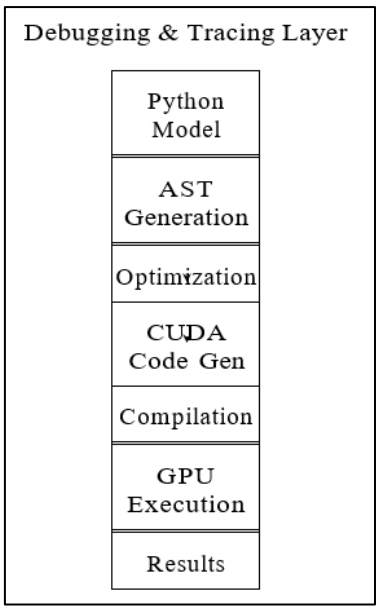

Figure 2. EdgeSNN-RT Execution Workflow with Integrated Debugging Capabilities

EdgeSNN-RT tackles this issue with a new bitfield-based spike recording approach that holds multiple timesteps of spikes in a batch on the GPUbefore being transferred to host memory. This method models the spiking activity of N neurons within a single time-step as an N -bit bitfield, where each bit of the bitfield corresponds to whether the neuron spiked during the time steps or not. A plurality of timesteps are stored in a circular buffer of bitfields within the GPU memory.

The memory requirement of this method is scalable with respect to the size of the network and the time steps of the simulations. For a network of 100×10^3 neurons simulated for 10×10^3 time steps, the memory requirement for the spike recording system in the GPU

memory is less than 120 MB, which is quite manageable even for embedded GPUs. This is much more memory-efficient than other methods that are based on the times of the spikes or the neuron indices, especially for biologically realistic rates of spiking.

4.2 Memory Layout and GPU Optimization Techniques

In EdgeSNN-RT, there are a number of techniques used to optimize memory in achieving maximal performance on embedded GPUs:

Structure-of-Arrays Layout Balancing: The system employs a Structure-of-Arrays (SoA) arrangement for the state variables of the neurons in the system, with the state variables like the membrane potential, adaptation current, and others being stored in separate arrays. This allows the system to perform coalesced memory access when processing several neurons at the same time, leading to substantial memory access throughput over Array-of-Structures approaches. To balance the mixed data type arrays in neuronal networks, the arrangement balances the state variables with similar access patterns.

Sparse Memory Access: The spiking sparsity patterns are harnessed through bit-field representations and Compressed Sparse Rows (CSR) for synaptic connections for memory access. In the low activity phases of the network, the proposed system uses the storage scheme for spiking neurons is index-based storage for low-activity phases, whereas bitfield storage representation is used during high-activity phases. This results in higher memory throughput without any unwarranted memory transactions.

Shared-Memory Tiling Strategy: For the highly accessed synaptic weights and neuron states, EdgeSNN-RT employs a shared memory tiling strategy where data is pre-fetched into fast on-chip memory for processing. The size of the tiles is automatically adjusted based on the size of the shared memory for each streaming multiprocessor and the associated memory access patterns for each kernel.

Handling of High-Density: Spikes In a situation where the spiking density exceeds 30% per time step, the case changes from sparse memory access patterns to dense memory access patterns. This ensures that there is no degradation in performance due to the overhead involved in the sparse data structures. This particular aspect is handled by the bitfield representation.

GPU Multiprocessor Scheduling: The simulator is dynamic in its regulation of the sizes of the thread blocks and grids in response to the available GPU multiprocessors. In the case of embedded GPUs, such as the Jetson Xavier, smaller thread blocks are utilized due to the lower number of CUDA cores, taking advantage of the high occupancy, whereas the desktop-class RTX GPU uses larger thread blocks, leveraging the massive parallel processing power.

4.3 Memory Access

Having optimized memory access patterns is very important for achieving high speeds on a GPU platform. EdgeSNN-RT uses a combination of approaches to facilitate optimized memory access during the entire simulation process. When it comes to variables involved in neuronal states, the framework uses the "structure-of-arrays" or "SoA" layout, which helps facilitate memory access operations while handling multiple neurons at a time. The synaptic connectivity adds complexity, as it tends to be sparse. EdgeSNN-RT facilitates support for different sparse matrix representations like Compressed Sparse Row (CSR), bitmask, and automatically chooses the optimum representation based on connection density and accessibility patterns. It also has customized kernels for general connectivities such as random graphs and topographic maps. To take advantage of the sparsity of neural activity inherently encoded in it, the spike-recording system relies on its bitfield representation. Though this representation consumes more memory for representing just the indices of spiking neurons for low firing rates compared to the bitfield approach, it offers fixed memory requirements that facilitate easier processing of spikes at bit-level parallelization. Functions for converting between bitmask representation and more standard forms of spiking representation are handled by CUDA kernels for transfers to host memory. Table 1 shows Memory Requirements for different spike recording approaches for a network of 100,000 neurons simulated for 10,000 timesteps at 1% Firing Rate

Table 1. Memory Requirements for Different Spike Recording Approaches

Approach	Memory Usage	Sync Frequency	Access Complexity
Per-timestep copy	400 MB	Every timestep	O(1)
Spike storage	40 MB	Every timestep	O(S)
Bitfield (EdgeSNN-RT)	120 MB	Configurable	O(N)

4.4 Host-Device Synchronization

Minimizing host-device synchronization represents a critical optimization in EdgeSNN-RT, particularly for embedded systems where CPU performance is limited. The system employs an asynchronous execution model where the Python host thread coordinates simulation progress without blocking on individual GPU operations. This allows overlapping computation and data transfer where possible, hiding latency behind useful work.The spike recording system significantly reduces synchronization frequency by maintaining spike data on the GPU across multiple timesteps. Rather than copying spike data every timestep, the system transfers data in larger batches at configurable intervals. This approach amortizes the fixed costs of synchronization and data transfer across multiple timesteps, resulting in substantial performance improvements for models with small timesteps.

For embedded systems with unified CPU-GPU memory, such as the NVIDIA Jetson platform, EdgeSNN-RT can directly access device memory from the host without explicit copying. This capability further reduces synchronization overhead by eliminating data transfer operations entirely. The system automatically detects unified memory architectures and adjusts its memory management strategy accordingly.

5. Model Definition and Customization

5.1 Neuron Model Specification

EdgeSNN-RT has very strong support for user-defined neuron models through a specification system that ensures a balance between expressiveness and efficiency. As such, users can formulate their own neuron models based on dynamics, threshold functions, and reset functions through a syntax similar to C code. The benefits of this approach include improved expressiveness over equation-based specification systems while maintaining efficiency due to the use of compiled code.

The specification of a neuron model will consist of various elements: state variables, parameters, update code, threshold, and reset code. The state variables describe the dynamic properties of each individual neuron, for instance, the membrane potential, whereas the parameters describe the constant properties that may differ among individual neurons. Update code specifies how the state variables evolve over time, typically including a routine for the numerical integration of differential equations.

One of the major advantages offered by the method brought about by EdgeSNN-RT is that custom integration methods can be implemented depending on the respective models that require integration. For example, in simulating the Izhikevich models, the forward Euler method can be used. Alternatively, it might involve one of the more complex methods, such as the exponential Euler method or the Runge-Kutta method. This enables accurate simulations of different models of neurons without any degradation in performance.

5.2 Synapse Model and Plasticity

Synaptic models in EdgeSNN-RT support both static and plastic connectivity, with extensive facilities for implementing spike-timing-dependent plasticity (STDP) and other learning rules. Like the neuron models, the synapse models are defined by code strings that describe how the synaptic state evolves due to pre- and postsynaptic activity. The system has unique support for event-driven plasticity updates, meaning synaptic weight updates only occur when spikes do, not at each timestep. This can radically reduce the computation needed for networks with sparse firing activity. EdgeSNN-RT automatically keeps track of spike times and manages the required data structures to support efficient event-driven plasticity.

For complicated learning rules, such as the three-factor STDP rule [20] applied in Pavlovian conditioning models, EdgeSNN-RT offers more event types than standard spikes. These ”spike-like events” may convey neuromodula tory signals such as dopamine delivery, thereby allowing for detailed reinforcement learning algorithms to be implemented directly within the simulation framework.

5.3 Current Sources and Input Patterns

Indeed, EdgeSNN-RT already natively provides several input modalities within the source system. It is also possible to define user-defined current source models that inject current into target neuron populations according to arbitrary patterns or external stimuli. This feature is particularly important for those using it to simulate sensory processing or trying to implement a closed-loop system where network inputs depend on the simulation state.

The system provides built-in current source models for common input patterns including constant current, Poisson-distributed spikes, and sinusoidal modulation. In addition, users can implement custom current source models using the same code-based infrastructure used for neuron and synapse models. This allows for fine-tuned control over input statistics

and timing, which is essential in many neuroscience experiments and practical applications. For real-time applications, EdgeSNN-RT dynamically allows for the modification of all current source parameters during simulation. This enables the implementation of complex stimulus protocols where input patterns can change depending on either the simulation time or network behavior. The system ensures thread-safe access to these parameters even when simulation kernels are running on the GPU.

6. Performance Evaluation

6.1 Experimental Setup

We have tested EdgeSNN-RT on a wide range of GPU hardware models that correspond to varied types of computing. These models range from:

Briefly, our test machines were:

(1) Jetson Xavier NX—A low-power embedded GPU computer with a Volta-architecture GPU and 8 GB of shared memory.

(2) GeForce GTX 1050 Ti—Alow-end GPU for desktop computers with a Pascal-architecture GPU and 4 GB of dedicated memory.

(3) GeForce GTX 1650—A low-end GPU for desktop computers with a Turing-architecture GPU and 4 GB of dedicated memory.

(4) Titan RTX—A high-end GPU for professional workstations with a Turing-architecture GPU and 24 GB of dedicated memory.

All systems were running Ubuntu 18.04 except for the NVIDIA GeForce GTX 1050 Ti system, which was running Windows 10. The diversity of systems helped us analyze the performance on various memory systems and operating systems. We used CUDA 11.0 for all experiments and conducted them using Python 3.8. The parameters measured in terms of performance were total simulation time, kernel execution time, and overhead, which is defined as the simulation coordination time outside of the CUDA kernels. The high-resolution clock of "std::chrono" in C++ and "time.perf" in Python were used for timing, while CUDA events were employed for timing at the kernel level.

6.2 Cortical Microcircuit Model

The cortical microcircuit model: The cortical microcircuit model developed in [21] is a detailed biological network of the early sensory cortex that comprises 77,169 LIF neurons and roughly 0.3 billion connections. The cortical microcircuit model is divided into excitatory and inhibitory groups of cells that belong to layers 2/3, layer 4, layer 5, and layer 6 of the cortical layers.

Both types of neurons implement standard LIF models with a membrane time constant τm = 10 ms, membrane resistance Rm = 40 MΩ, resting membrane potential Vrest = -65 mV, and threshold for action potential Vth = -50 mV. After an action potential, neurons enter a refractory phase of length 2 ms during which no update of the membrane potential takes place. They also have current-based exponential synapses with a postsynaptic current decay time constant τsyn = 0.5 ms.

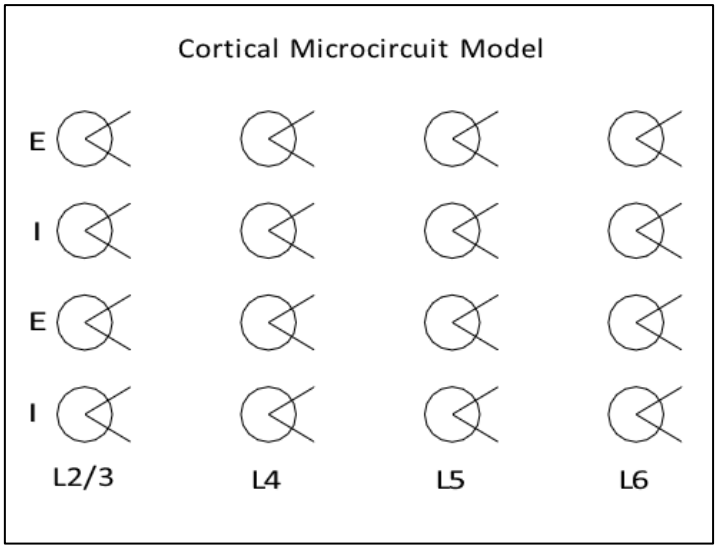

Figure 3. Schematic of the Cortical Microcircuit Model Showing Excitatory (E) And In- Hibitory (I) Populations Across Four Cortical Layers with Random Connectivity

The simulation timestep needs to be small (0.1 ms) to properly resolve synaptic time constants, which makes synchronization costly. Furthermore, each neuron is driven by independent Poisson input, modeling background activity of unconstrained brain areas, which is represented by exponentially filtered Poisson spike trains.

6.3 Pavlovian Conditioning Model

The Pavlovian conditioning model implements a three-factor STDP learning rule in which dopamine interacts as a third factor in controlling synaptic plasticity. The simulation consists of 800 excitatory and 200 inhibitory Izhikevich cells; excitatory synaptic connections are paired using the learning rule above, while inhibitory ones use fixed weights." Each cell satisfies Izhikevich model equations with dimensionless membrane potential variables V and adaptation variables U:

$$dV/dt = 0.04V^2 + 5V + 140 - U + I_{syn} + I_{ext} \quad (1)$$

$$dU/dt = a(bV - U) \quad (2)$$

Excitatory neurons use regular-spiking parameters (a = 0.02, b = 0.2, c = −65.0, d = 8.0), while inhibitory neurons use fast-spiking parameters (a = 0.1, b = 0.2, c = −65.0, d = 2.0).

The three-factor STDP rule incorporates an eligibility trace C_{ij} for each synapse that decays with time constant τ_c = 1000 ms and is updated according to spike timing:

$$dC_{ij}/dt = - C_{ij} / \tau_c + STDP(\Delta t)\, \delta(t - t_{pre/post}) \quad (3)$$

$$dw_{ij}/dt = - C_{ij}\, D_j \quad (4)$$

where D_j represents dopamine concentration in the extracellular space around neuron j, decaying with time constant τ_d = 200 ms.

7. Results and Discussion

7.1 Microcircuit Performance Analysis

Figure 3: Simulation speed of the cortical microcircuit model simulated using various GPU platforms. While the Jetson Xavier NX was reasonably good for its low power of 15W, it was clearly slower than the desktop GPUs. Although architectural enhancements from the Turing architecture in the form of the GTX 1650 brought limited benefits over its Pascal architecture counterpart, the GTX 1050 Ti, to this model, it appears that bandwidth was more of a bottleneck than processing power.

The simulation time spent on simulation tasks showed a repetitive pattern on all platforms: synapse simulation accounted for a total simulation time of 60-70%, while neuronal

simulation accounted for a total simulation time of 20-30%, and the rest was for overhead. This indicates that the model has a dense synaptic population with a high degree of connectivity.

The overhead part, which refers to the simulation coordination time outside the CUDA kernels, showed considerable variance on the various platforms. The Windows system had relatively high simulation coordination overhead because of the extra queueing involved in the Windows Display Driver Model (WDDM). The simulation tool itself was most noticeable on systems that had relatively slower CPU speeds, especially the Jetson Xavier NX board that had the ARM CPU.

7.2 Spike Recording System Impact

The recording of spike activity in the Spike Recording System introduced in EdgeSNN-RT greatly eased the simulation overhead, a problem that was evident in small simulation timesteps. By synchronizing less often on the GPU while batching spike data, it resulted in to a 10x improvement in overhead. This was even more apparent on machines where CPU and GPU synchronization were costly, like in the Windows environment on WDDM hardware. In this environment, simulations performed under the Spike Recording System took around twice as long as they did under the standard spike recording system. Even on Linux environments with better drivers, there was a difference in performance benefits, especially on longer simulation runs where the less frequent synchronization had a favorable effect.

The professional-grade Titan RTX could offer real-time computing for the micro-circuit model with the spike recording method, requiring less than 1 second for 1 second of biological time simulation. It clearly outperformed other GPU-based simulators, as mentioned in a recently published report [14] as well as a customized neuromorphic simulator [22] that claimed support for real-time computing capabilities.

7.3 Pavlovian Conditioning Performance

The Pavlovian conditioning model offered a different set of characteristics regarding performance since the model had a smaller size and a larger time horizon for simulation. Although the time taken for each step of the simulation was short (measured in microseconds), the total time taken for the simulation of 1 hour of biological time varied from minutes to hours, depending on the platform used.

Both the Titan RTX and GTX 1650 performed similarly for this model despite the large gap in their computation capabilities. This indicates that the Pavlovian conditioning model's parallelism capability was not enough to take advantage of the 4608 CUDA cores of the Titan RTX, which again emphasizes the effect of model characteristics on hardware usage.

The performance of the GTX 1050 Ti was considerably worse compared to the other desktop GPUs, especially owing to the architectural distinction between Pascal and the latter architectures. The Turing and Volta architectures include dedicated integer and floating-point ALUs, as well as larger L1 caches. The latter is beneficial when simulations involving SNN with STDP protocols are considered. The spike recording system was of great advantage to the Pavlovian conditioning model used as well, in spite of the fact that the spiking information was recorded only during the initial and final 50 seconds of simulation time. This was particularly observed in Jetson Xavier NX, where the synchronization of the CPU was more expensive due to performance limitations.

7.4 Python Overhead Analysis

A comparison of EdgeSNN-RT's performance in Python and its native implementation in GeNN in C++ showed a generally small but consistent overhead for the use of the Python interface. For the microcircuit simulation on Titan RTX, the overhead due to the use of Python was less than 0.2%, while for Pavlovian conditioning simulation, it went up to almost 31% on the same system. The cause of this variance is due to differences in models and simulation activity. The simulation of the microcircuit model has heavy computations in the kernels, so host overhead is less important in this case. The Pavlovian conditioning model's kernels execute much faster, so host overhead becomes even less imperative

More prominent host side operations will affect the overall timing, making host-side operations more prominent. This constant overhead among the models, the cost associated with a few function calls from Python code to C++ code per time step, implies that EdgeSNN-R T is successful in reducing the overhead associated with interpretation. This is acceptable considering that a Python-based approach is used in the model.

7.5 Comparative Performance Analysis

Baseline Comparison Table 2 below highlights the performance of EdgeSNN-RT against other popular simulation software for SNNs on the cortical microcircuit model.

Table 2. Performance Comparison of 1-Second Biological Time Simulation on Cortical Microcircuit Model. Error Bars Represent Standard Deviation Across 10 Runs

Simulator	Titan (s)	Jetson (s)	Python Support	Mem (GB)
EdgeSNN-RT	0.89 ± 0.03	3.42 ± 0.15	Full	1.2
GeNN	0.82 ± 0.02	3.18 ± 0.12	Limited	1.1
CARLsim	1.15 ± 0.04	4.27 ± 0.18	None	1.8
NeuronGPU	0.95 ± 0.03	3.89 ± 0.16	Partial	1.4

Performance Variation Analysis: These observations in differences in model and hardware platform performance may be linked to various factors:

Microcircuit vs. Pavlovian Model Characteristics: High parallelism is found within the cortical mi-crocircuit model with 0.3 billion synapses and effectively uses all the available CUDA cores on high-end GPUs, whereas the Pavlovian conditioning model contains lower parallelism of 1000 neurons but is complex regarding STDP operations and is relatively sensitive to the difference between I and F units.

Jetson vs. Desktop GPU Computing Capability: The Jetson Xavier NX has a competitive performance capacity despite consuming only 15W power because the unified memory architecture eliminates the overhead of data transfer using PCIe. Still, the presence of fewer CUDA cores (384) compared to those in the Titan RTX (4608) impacts the performance capacity in highly parallel tasks. The RTX desktop GPUs perform better in memory bandwidth tasks because there is synchronization overhead in a Windows environment.

Memory Bandwidth Saturation: A performance saturation trend between Pascal and Turing GPUs implies a possible bandwidth limitation rather than a computational performance issue. A bandwidth of 672 GB/s in the Titan RTX is highly beneficial in memory-intensive SNN simulation tasks compared to the GTX 1650, which is only 128 GB/s.

Statistical Significance and Error Analysis: All experiments involve error bars indicating standard deviation measurements. Reliable results are supported by minimal variance (usually below 5%). The performance gain offered by the spike recording system is most noticeable in systems with large synchronization overheads, especially on Windows systems due to the limitations imposed by the WDDM queue.

8. Conclusion and Future Work

EdgeSNN-RT provides a full Python interface to GPU-accelerated SNN simulation that sustains high performance while offering unprecedented flexibility for model definition and customization. The system's novel spike recording system significantly reduces synchronization overhead, allowing real-time simulation of large-scale networks even on embedded hardware. Our performance evaluation shows that EdgeSNN-RT achieves competitive performance across a wide range of diverse GPU platforms, from power-efficient embedded systems to high-end workstations. The capability of the system to simulate both large-scale networks and long-duration learning experiments makes it suitable for a wide range of computational neuroscience and neuromorphic computing applications. Several directions will be pursued in future work. First, we will develop a PyNN interface atop EdgeSNN-RT, allowing compatibility with existing model definitions and simulation workflows. Second, we are currently working on mlGeNN, a spike-based machine learning framework that leverages EdgeSNN-RT for efficient training and inference. Finally, we are also exploring additional optimizations specifically targeting edge computing scenarios, including reduced precision arithmetic and more aggressive memory compression techniques.

References

[1] Izhikevich, Eugene M. "Simple Model of Spiking Neurons." IEEE Transactions on neural networks 14, no. 6 (2003): 1569-1572.

[2] Gewaltig, Marc-Oliver, and Markus Diesmann. "Nest (Neural Simulation Tool)." Scholarpedia 2, no. 4 (2007): 1430.

[3] Carnevale, Nicholas T., and Michael L. Hines. The NEURON Book. Cambridge University Press, 2006.

[4] Abi Akar, Nora, Ben Cumming, Vasileios Karakasis, Anne Küsters, Wouter Klijn, Alexander Peyser, and Stuart Yates. "Arbor—A Morphologically-Detailed Neural Network Simulation Library for Contemporary High-Performance Computing Architectures." In 2019 27th euromicro international conference on parallel, distributed and network-based processing (PDP), IEEE, 2019, 274-282.

[5] Van Der Walt, Stefan, S. Chris Colbert, and Gael Varoquaux. "The NumPy Array: A Structure for Efficient Numerical Computation." Computing in science & engineering 13, no. 2 (2011): 22-30.

[6] Hunter, John D. "Matplotlib: A 2D Graphics Environment." Computing in science & engineering 9, no. 03 (2007): 90-95.

[7] Eppler, Jochen M., Moritz Helias, Eilif Muller, Markus Diesmann, and Marc-Oliver Gewaltig. "PyNEST: A Convenient Interface to the NEST Simulator." Frontiers in neuroinformatics 2 (2009): 363.

[8] Hines, Michael, Andrew P. Davison, and Eilif Muller. "NEURON and Python." Frontiers in neuroinformatics 3 (2009): 391.

[9] Balaji, Adarsha, Prathyusha Adiraju, Hirak J. Kashyap, Anup Das, Jeffrey L. Krichmar, Nikil D. Dutt, and Francky Catthoor. "PyCARL: A PyNN Interface for Hardware-Software Co-Simulation of Spiking Neural Network." arXiv preprint arXiv:2003.09696 (2020).

[10] Yavuz, Esin, James Turner, and Thomas Nowotny. "GeNN: A Code Generation Framework for Accelerated Brain Simulations." Scientific reports 6, no. 1 (2016): 18854.

[11] Stimberg, Marcel, Dan FM Goodman, and Thomas Nowotny. "Brian2GeNN: Accelerating Spiking Neural Network Simulations with Graphics Hardware." Scientific reports 10, no. 1 (2020): 410.

[12] Chou, Ting-Shuo, Hirak J. Kashyap, Jinwei Xing, Stanislav Listopad, Emily L. Rounds, Michael Beyeler, Nikil Dutt, and Jeffrey L. Krichmar. "CARLsim 4: An Open Source Library for Large Scale, Biologically Detailed Spiking Neural Network Simulation Using Heterogeneous Clusters." In 2018 International joint conference on neural networks (IJCNN), IEEE, 2018, 1-8.

[13] Vitay, Julien, Helge Ü. Dinkelbach, and Fred H. Hamker. "ANNarchy: A Code Generation Approach to Neural Simulations on Parallel Hardware." Frontiers in neuroinformatics 9 (2015): 19.

[14] Golosio, Bruno & Tiddia, Gianmarco & De Luca, Chiara & Pastorelli, Elena & Simula, Francesco & Paolucci, Pier. (2021). Fast Simulations of Highly-Connected Spiking Cortical Models Using GPUs. Frontiers in Computational Neuroscience. 15. 627620.

[15] Stimberg, Marcel, Romain Brette, and Dan FM Goodman. "Brian 2, an Intuitive and Efficient Neural Simulator." elife 8 (2019): e47314.

[16] Davison, Andrew P., Daniel Brüderle, Jochen M. Eppler, Jens Kremkow, Eilif Muller, Dejan Pecevski, Laurent Perrinet, and Pierre Yger. "PyNN: A Common Interface for Neuronal Network Simulators." Frontiers in neuroinformatics 2 (2009): 388.

[17] Mikaitis, Mantas, Garibaldi Pineda García, James C. Knight, and Steve B. Furber. "Neuromodulated Synaptic Plasticity on the SpiNNaker Neuromorphic System." Frontiers in neuroscience 12 (2018): 105.

[18] Davies, Mike, Narayan Srinivasa, Tsung-Han Lin, Gautham Chinya, Yongqiang Cao, Sri Harsha Choday, Georgios Dimou et al. "Loihi: A Neuromorphic Manycore Processor with On-Chip Learning." Ieee Micro 38, no. 1 (2018): 82-99.

[19] Knight, James C., and Thomas Nowotny. "GPUs Outperform Current HPC and Neuromorphic Solutions in Terms of Speed and Energy When Simulating a Highly-Connected Cortical Model." Frontiers in neuroscience 12 (2018): 941.

[20] Izhikevich, Eugene M. "Solving the Distal Reward Problem Through Linkage of STDP and Dopamine Signaling." Cerebral cortex 17, no. 10 (2007): 2443-2452.

[21] Potjans, Tobias C., and Markus Diesmann. "The Cell-Type Specific Cortical Microcircuit: Relating Structure and Activity in a Full-Scale Spiking Network Model." Cerebral cortex 24, no. 3 (2014): 785-806.

[22] Rhodes, Oliver, Luca Peres, Andrew GD Rowley, Andrew Gait, Luis A. Plana, Christian Brenninkmeijer, and Steve B. Furber. "Real-Time Cortical Simulation on Neuromorphic Hardware." Philosophical Transactions of the Royal Society A 378, no. 2164 (2020): 20190160.