SimHH: A Versatile, Multi-GPU Simulator for Extended Hodgkin-Huxley Networks

Engelen, Max; Betting, River; Strydis, Christos

DOI
10.1109/ACCESS.2025.3550444

Publication date
2025

Document Version
Final published version

Published in
IEEE Access

IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 13 February 2025, accepted 1 March 2025, date of publication 11 March 2025, date of current version 21 March 2025.

Digital Object Identifier 10.1109/ACCESS.2025.3550444

RESEARCH ARTICLE

SimHH: A Versatile, Multi-GPU Simulator for Extended Hodgkin-Huxley Networks

MAX C. W. ENGELEN[1], RIVER BETTING[1], AND CHRISTOS STRYDIS[1,2], (Senior Member, IEEE)
[1]Department of Neuroscience, Erasmus Medical Center, 3015 GD Rotterdam, The Netherlands
[2]Quantum and Computer Engineering Department, Delft University of Technology, 2628 CD Delft, The Netherlands

Corresponding author: Christos Strydis (c.strydis@erasmusmc.nl)

ABSTRACT Computational neuroscience relies on complex mathematical models to simulate brain activity and decipher underlying biological processes. However, these simulations are computationally intensive, prompting the exploration of high-performance computing systems as a viable solution to enhance efficiency. In this work, we introduce SimHH, an extended-Hodgkin-Huxley simulator designed for versatility and high performance. Leveraging the OpenMPI library, SimHH exhibits exceptional scalability, catering to a wide spectrum of computing environments. Scalability is optimized through two distinct configurations: one that distributes all possible cell-compartment potentials among network nodes and another that shares compartment potentials only among relevant nodes, employing MPI Allgather and Alltoall. Seamless support for CUDA, CUDA-aware MPI, and NVLink further enhances performance, with communication overhead minimized through concurrent execution of compute kernels. Benchmarking against various neuron models, including the challenging Inferior-Olivary Nucleus, demonstrates SimHH's potential, achieving remarkable results on up to 256 compute nodes. Notably, large-scale GPU clusters enable the simulation of highly biologically plausible networks exceeding 10 million cells. Comparative analyses against CPU- and FPGA-based solutions underscore SimHH's superiority, boasting a speedup of approximately 150× over single-threaded CPU implementations, 10× over single-FPGA setups, and 10× over multi-threaded CPU configurations with 128 threads, all for a fully connected network of approximately 7,000 IO cells. Additionally, a 7× speedup is attained compared to the established NEST neurosimulator running on 32 nodes, simulating a network of 94,720 Hodgkin-Huxley neurons with gap junctions. These findings underscore SimHH's efficacy in advancing computational-neuroscience research by facilitating efficient and scalable simulation of complex neuronal networks.

INDEX TERMS Computational neuroscience, general-purpose GPU computing, high-performance computing, neural simulation.

I. INTRODUCTION

Computational neuroscience is the subdiscipline of neuroscience concerned with the explanation and prediction of experimental neuroscientific data. In-vivo and in-vitro experiments are used systematically and are common tools in this field. Even though such experiments are effective, they are often laborious and cumbersome; in certain cases – for example, when investigating deep brain structures – they are all but impossible to mount, because of the complexity of these structures and the high amounts of computational resources required to simulate them.

Computational neuroscientists construct predictive simulations that test scientific theories and predictions by incorporating spiking neural network (SNN) models of varying complexity. These models must be simulated in computing systems that run specialized simulators. This work presents a novel, high-performance, scalable neurosimulator called SimHH, which focuses on the scalability and parallelization of extended Hodgkin-Huxley (eHH) simulations. These simulations are known for their high biological plausibility and high computational intensity. SimHH achieves a high degree

The associate editor coordinating the review of this manuscript and approving it for publication was Shadi Alawneh.

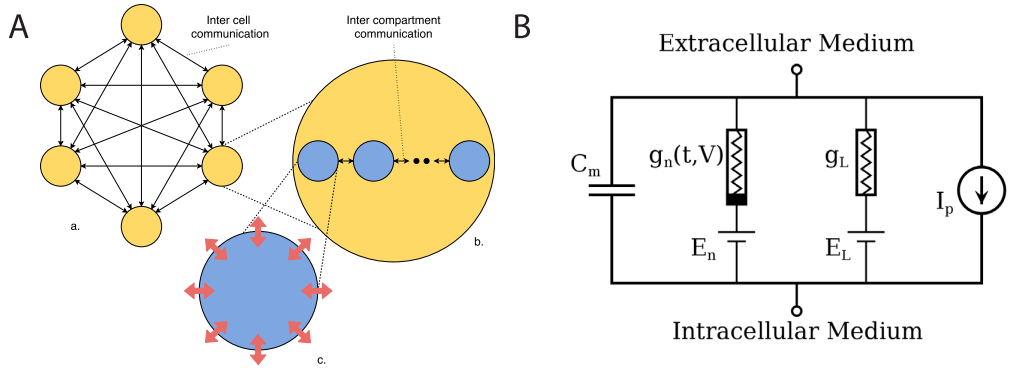

FIGURE 1. A: Schematic overview of a neural network of 6 cells (a). Visible is also a single neuron cell (b), and a single cell compartment with gates in red (c) [1] B: The components, representing the cell-membrane's biophysical characteristics, of the Hodgkin–Huxley-type models. The capacitance (C_M) represents the lipid bi-layer. The ion channels are modeled by linear (g_L) and nonlinear (g_n) conductances. The voltage sources (E_n and E_l) represent the electrochemical gradients driving the flow of ions. The current source (I_p) represents the ion pumps and exchangers.

of parallelization by utilizing multiple graphics-processing units (GPUs) and scalability through deployment on distributed computing systems utilizing Message Passing Interface (MPI), while still being able to use only the central processing unit (CPU) in systems without a GPU. SimHH was used to observe the impact of scalability on actual simulation speeds. Bottlenecks present in eHH simulations are explored. Furthermore, and crucially, SimHH supports extended Hodgkin-Huxley (eHH) models with continuous-time gap-junction support – which are notorious for the high computational and communication requirements – and with highly customizable multicompartmental and ion-channel support per each neuron. With these features, SimHH can simulate eHH networks that were previously out of the reach for neuroscientists. The contributions of this work, thus, are as follows:

- A new versatile CPU- and GPU-aimed multi-node neurosimulator for the distributed simulation of eHH models, with (i) native support for single- and multi-GPU-optimized continuous-time gap-junction calculations; (ii) automatic hiding of communication latency; and (iii) optimized memory use via model-parameter randomization and model-description reuse, increasing performance and creating biologically accurate simulations.
- Valuable insights into the differences in performance between MPI-communication strategies and how to optimize MPI communication for large-scale neurosimulations.
- A publicly available code-base, made immediately available for download.[1]

The remainder of the paper is organized as follows. Section II provides the background for this work. Related works are discussed in Section III. Section IV provides an overview of the SimHH design and implementation, while Section V evaluates SimHH and compares it with other simulators. A discussion of the results and concluding remarks are provided in Section VI and VII, respectively.

[1]https://gitlab.com/neurocomputing-lab, under 'BrainFrame'

II. BACKGROUND

On average, the human brain consists of roughly 8.6×10^{10} (eighty-six billion) neurons [2], and the average synaptic-connection count among the neurons is 7,000 [3]. If each graph edge can be represented by 8 bytes of memory, a sparsely saved graph to capture all of these connections would require ∼5,000 terabytes of memory. In the current standard computing systems, this is unrealistic. Fugaku, the fourth supercomputer on the TOP500 list of the most powerful computer systems in the world [4], would come short of that figure by more than 1,800 terabytes [5].

Nowadays, neuroscientists increasingly rely on Spiking Neural Networks (SNNs) to simulate biological networks. SNNs closely mimic natural neural networks. The concept of time is integrated into the network, together with neuronal and synaptic states. When a neuron fires, the spike travels along the synaptic connections and influences connected-neuron potentials. Neural behavior is bounded by rules with fixed amplitudes and durations for specific spiking patterns, and computational models have been developed to simulate these patterns. Spiking behavior models range from very basic to complex. A computationally challenging neuron model is the Hodgkin-Huxley model. For a detailed discussion of the differences between neural models, refer to Izhikevich [6]. Fig. 1A shows a schematic overview of the hierarchical structure of a neural network.

A. HODGKIN-HUXLEY MODELS

Conductance-based models represent a neuron as an electrical circuit, where protein-molecule ion channels are represented by conductances, and the lipid bilayer is represented by a capacitor. As such, neuronal behavior can be represented by a set of ordinary differential equations

TABLE 1. Overview of the characteristics of the reviewed subset of neurosimulators that are still active.

Simulator	Neuron models						Frontend				Characteristics					
												Multi-GPU				
	LIF	IZH	HH	Multi Compartmental	Chemical synapses	Gap junctions	Python	C/C++	PyNN	NeuroML	Single-GPU	Multi-node	GPUDirect	Gap-junction optimization	Gate-level parallelism	Open Source
Arbor [9]	x	x	x	x	x	x	x	x		x	x	x	x			x
NEST [10], [11]	x	x	x		x	x	x		x	x		x				x
CoreNeuron / NetPyNE [12]	x	x	x	x	x	x			x	x	x	†				x
Brian [13], [14]	x	x	x	x	x	x	x	x	x	x	x					x
GeNN 3 [15]	x	x	x		x		x	x			x					x
CARLsim [16]	x	x		x				x	x		x					x
SpiNNaker [17]	x	x	x		x				x			x				x
flexHH/GenEHH [1], [18]			x		x	x				‡						x
SimHH (this work)			x	x		x		x		‡	x	x	x	x	x	x

† CoreNeuron does not automate the setup for the distribution of the network across nodes, leaving it to the user. NetPyNE provides an additional script to facilitate this in specific cases.
‡ Uses a NeuroML-compliant interface.

(ODEs). Conductance-based models offer biophysically highly meaningful and measurable results, and are therefore of great use to computational neuroscientists. However, high biophysical meaning comes at the cost of high computational complexity. This complexity poses interesting computational challenges.

The original conductance-based model was developed by Hodgkin and Huxley [19], and the equivalent electrical circuit of the model is presented in Fig. 1B. The basic Hodgkin-Huxley (HH) model has been extended in various ways to better reflect the behavior of different types of neurons. One such extension concerns the neurons of the inferior-olivary nucleus (IO), which is located in the ventral brain stem and is essential for the functioning of the cerebellum [20], [21]. IO neurons are densely coupled through intercellular connections known as *gap junctions*, which influence spiking behavior. Therefore, to simulate IO spiking patterns, gap junctions must also be represented.

The Inferior-Olive model presented by De Gruijl et al. is an *extended Hodgkin-Huxley* (eHH) model that represents IO neurons in a biophysically meaningful way by extending the original HH model with gap junctions, custom user-defined gates, and multiple compartments per cell [20]. The De-Gruijl IO model is widely used by neuroscientists specializing in cerebellar functioning, and is therefore an excellent choice for implementation in a High-Performance Computing (HPC) simulator.

B. NUMERICAL SOLVERS

As stated before, extended Hodgkin-Huxley models are described as systems of ODEs, which call for the use of numerical methods to solve them [22]. Various solving methods (or *solvers*) can be used. Christoph et al. [23] compared different solvers and explored their time-step requirements – subject to model complexity. Specifically, the use of exponential-time differencing appears to be suitable for eHH models. However, it also adds complexity with respect to the standard explicit time-stepping schemes, which may be unnecessary in specific cases.

To compare standard time-stepping schemes, Miedema et al. [1] implemented forward-Euler, 2nd-order Runge-Kutta, and 3rd-order Runge-Kutta, and showed that forward-Euler solvers have the lowest execution times for the edge cases of stability. If one wants to optimize for both speed and accuracy, there may be more than one Pareto-optimal solution.

III. RELATED WORK

The landscape of neurosimulators is diverse, with many tools developed to cater to different neuronal models and computing architectures. However, there remains a gap in simulators that offer both scalability across multi-node GPU systems and efficient handling of biophysically detailed models, such as HH. The table of existing neurosimulators (Table 1) highlights the limitations and scope of some of the most prominent tools, providing a context for the development of SimHH.

Several neurosimulators, such as CARLsim [16] and GeNN [15], offer single-GPU-accelerated simulations but fall short in supporting HH models and multicompartmental neurons. GeNN, for example, lacks support for multicompartmental models, restricting its utility for complex models such as the Inferior Olive (IO), whereas CARLsim focuses

exclusively on spiking neural networks (SNNs) without support for HH dynamics.

General-purpose simulators such as NEURON [12], NEST [10], [11], and Brian [13], [14] provide extensive *flexibility* by supporting a wide range of neuron models, including HH and multicompartmental models (except NEST), and offering compatibility with standard neural description languages such as NeuroML [24]. However, this flexibility often comes at the cost of performance. For instance, although NEURON and NEST can be extended to multi-node systems, their performance does not scale well for large-scale HH simulations. Furthermore, most lack full optimization for GPU platforms or efficient handling of dense interconnections, such as gap junctions.

Arbor [9] and CoreNeuron [12], both of which support multicompartmental models and are capable of using *multi-node GPUs*, offer some optimizations, including integration with NeuroML. However, CoreNeuron does not fully automate the distribution of the network across nodes, leaving this task to the user, which can hinder its adoption and scalability, particularly in large networks. Arbor is a new general-purpose simulator with full support of most popular model classes. However, its wide scope comes at the cost of missed opportunities for parallelization and scalability. SimHH exhibits two key novelties that address these issues. First, SimHH supports gate-level parallelism, which can significantly improve the performance for small- and medium-sized networks. In comparison, Arbor parallelizes execution per model mechanism (not per ion gate), which is less flexible. Second, Arbor can only map gap-junctioned (sub)networks inside a single compute node (with or without GPU support), in order to eschew performance problems. In contrast, SimHH attempts to partition arbitrarily shaped gap-junctioned (sub)networks across multiple nodes, leading to improved performance (as illustrated in Section V).

Scalability is a critical issue in neural simulations, particularly when leveraging multi-GPU platforms. Simulators such as SpiNNaker [17] and flexHH/GenEHH [1], [18] have explored specialized hardware platforms. flexHH and GenEHH focus on HH models, with flexHH leveraging Field Programmable Gate Array (FPGA) acceleration for smaller networks, whereas SpiNNaker (no HH-model support) and GenEHH target CPU-based systems for larger network sizes. However, both fall short in supporting multi-node distributed computing, limiting their utility for larger, more complex simulations.

For large-scale simulations (i.e., networks larger than 1,000 neurons), multi-node GPU-based approaches are essential. Pioneering work by Van der Vlag et al. [25] introduced a multi-GPU solution for the De-Gruijl IO model, marking a crucial step in the field. However, this early implementation was hardcoded, inflexible, and prone to significant communication overheads, especially in networks with dense gap-junctions. This communication bottleneck remains a key challenge for scaling HH-type models on multi-node platforms. To address these limitations, Jordan et al. [26] explored optimized communication strategies for high-performance computing (HPC) systems. Their study demonstrated that sparse neural networks benefit from more tailored communication methods, such as the `MPI_Alltoallv` API, which can outperform broader techniques such as `MPI_AllGather`. While Van der Vlag et al. employed both, they did not offer a comparative analysis of these approaches, thereby limiting insights into performance benefits. Despite these advancements, the challenge of optimizing the communication for dense neural networks remains largely unresolved. This is particularly critical for large-scale, multi-node GPU simulations, where dense connectivity significantly increases communication demands, thereby hindering scalability. Thus, while the initial solutions laid the groundwork, future work must focus on overcoming these communication barriers to enable efficient, scalable simulations on modern HPC architectures.

In summary, while existing neurosimulators offer various features and optimizations, there is no current solution that combines support for HH-class models, multicompartmental neurons, and dense gap-junction networks with scalability across multi-node GPU platforms. SimHH aims to address these gaps by providing a neurosimulator specifically designed for large-scale, multi-node, GPU-based simulations, optimizing both inter-process communication and computational efficiency for dense networks such as the De-Gruijl IO model. The table of existing simulators (Table 1) highlights that none fully meets these needs, positioning SimHH as a critical advancement in the field.

IV. DESIGN AND IMPLEMENTATION

SimHH, a new simulator for HH-class neurons, improves over the current state of the art within the distributed, multi-GPU design space. SimHH uses Compute Unified Device Architecture (CUDA) and optimizes both intranode processing and internode communication patterns, without compromising accuracy. For intranode improvements, we laid out memory to support CUDA-warp alignment (see Section IV-C1). Gate and compartment updates within a single timestep update do not depend on each other, and can therefore be fully parallellized (see Section IV-C2). Furthermore, by studying the manner in which gap-junction calculations are performed, we exploit the GPU low-level architecture in a better way in order to perform the required sparse matrix-vector calculations (see Section IV-C3).

As for inter-node communications, we observed that there was much to gain from NVLink (GPU to GPU) communication. The scalability design explores MPI communications and intranode-GPU communication. We gained the insight that using CUDA's Unified Virtual Addressing (UVA) does not lead to any benefits, and decided to manually tune this intranode GPU communication with direct transfers. MPI is utilized for inter-node communication, which hooks into GPUDirect Remote Direct Memory Access (RDMA) for communication. In addition, this work addresses the

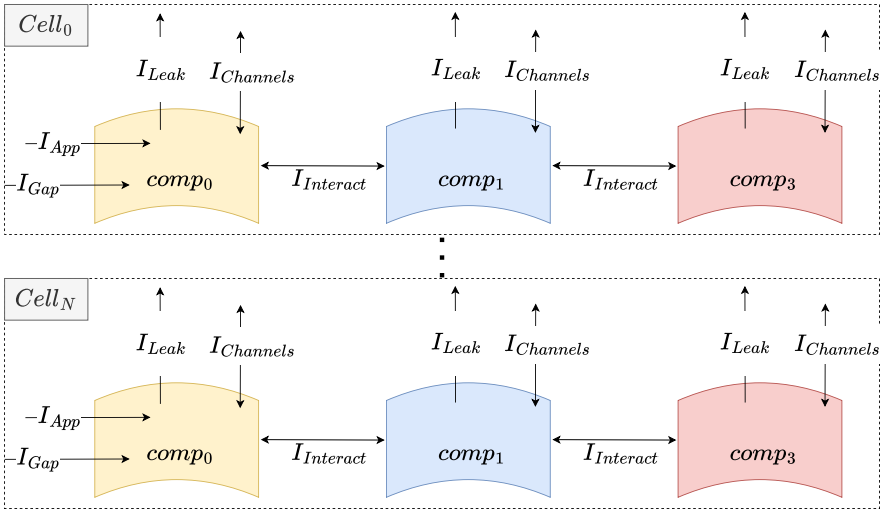

FIGURE 2. An overview of the neural-network structures supported by this work. In the illustration, two neurons are shown, each consisting of three compartments. I_{app} and I_{gap} can only act on the first compartment of the chain-style compartment list. The dotted lines represent the expandability of the neural network.

broader research question regarding the optimization of communication for large-scale, multi-node GPU simulations.

From bottom to top, the neural model implemented in SimHH consists of the following five-level hierarchical structure: *gate* > *channel* > *compartment* > *cell* > *network*. As shown in Fig. 1A, any element can encapsulate multiple elements of the level directly below it; for instance, a cell can contain multiple compartments, and a network can contain multiple cells. This hierarchical structure forms a strong basis for a flexible and scalable simulator that can efficiently utilize parallelism.

However, the presence of gap junctions in the De-Gruijl IO model complicates the design of such a simulator and underlines our initial reason for using this model in our experiments. After all, allowing gap junctions to act on the different compartments converts the dependency problem from a cell-dependency to a compartment-dependency problem, since gap junctions allow for interaction between compartments of different cells. In our design, this problem was addressed by introducing the following two limitations on inter-compartment interaction:

1) Externally applied or gap-junction currents can only act on the first compartment of a cell description.
2) Cell compartments are ordered in a chain following the order of the cell description.

As explained in Miedema et al. [1], despite the first limitation, the design remains capable of addressing numerous research questions while fulfilling the requirements of most HH models. Even though the formulation of flexHH's interaction-current supports arbitrary compartment ordering, its implementation does not. Many HH models, including the De-Gruijl IO model, do not require such functionality. GenEHH supports only chain-style compartment ordering. Our work here followed these design choices.

A. BASIC SIMULATOR STRUCTURE AND LIMITATIONS

Having discussed the design limitations of the simulator, we can now describe the equations that it solves. A conceptual illustration of SimHH is shown in Fig. 2. Essentially, SimHH is a transient simulator that iteratively calculates the membrane voltage of any number of (compartments of) arbitrarily connected HH neurons. The derivative of the membrane voltage of a single compartment i following the HH formalism can be calculated as shown in Equation (1). Multiple coinciding currents contribute to this voltage derivative: (i) an optional, externally applied perturbation current I_{App}, (ii) a cumulative current I_{Channels} due to the essential sodium and potassium (and any other modeled) ion channels in the cell, (iii) the aggregate current I_{Gap} contributed by other cells or compartments via gap-junction connections in contact with the compartment (if any), (iv) a current $I_{\text{Multicomp}}$ contributed by neighboring compartments $i+1$ and $i-1$ in the multicompartmental chain, and finally (v) the leakage current of the membrane I_{Leak}, in line with the HH formalism. The full set of equations is as follows:

$$\frac{\mathrm{d}V_i}{\mathrm{d}t} = \frac{I_{\text{App},i} - I_{\text{Channels},i} - I_{\text{Gap},i} - I_{\text{Multicomp},i} - I_{\text{Leak},i}}{C_{\text{m}}} \tag{1}$$

$$I_{\text{App},i} = \begin{cases} a & \text{if step}_{\text{start}} \leq \text{step} < \text{step}_{\text{end}} \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

$$I_{\text{channels},i} = \sum_{n=0}^{N_{\text{channels}}-1} \left(I_{g_{channel}[j]}(V - V_{channel}[j])\right) \times \prod_{n=0}^{M_{\text{gates}[j]}-1} y_{\text{j,i}}{}^{\text{p}_{\text{j,i}}} \tag{3}$$

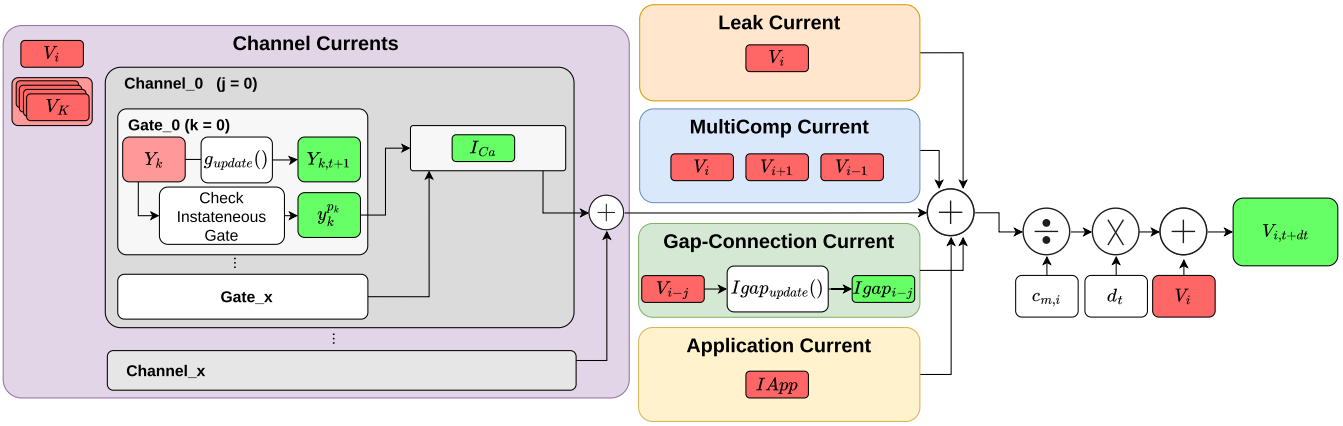

FIGURE 3. Schematic of the way Equations (2)–(6) need to be combined to yield Equation (1) as well as all intermediate calculations, for a single solver step of the simulation of a single compartment. All upper-case variables are timestep-dependent. All lower-case variables are constants that are defined in a JSON configuration file. Inputs are represented with red and outputs with green color. Dotted lines represent the ability to add multiple channels and/or gates to a compartment.

$$I_{\text{multicomp},i} = g_{mc,c} * \left(\frac{V_i - V_{i+1}}{p_{i,i+1}} - \frac{V_i - V_{i-1}}{p_{i,i-1}}\right) \tag{4}$$

$$I_{\text{gap},i} = \sum_{j=0}^{n_{\text{connectedcells}}-1} (w_{i,j}(c_0 \exp(c_1 \times (V_i - V_j)^2)(V_i - V_j)) \tag{5}$$

$$I_{\text{leak},i} = g_{\text{leak},i} \times (V_i - v_{\text{leak},i}) \tag{6}$$

where g_* are activation variables and c_* are constants; C_{m} is the capacitance of the cell membrane for compartment i; p_i signifies the surface ratio between compartments; $w_{i,j}$ is the strength of the contribution between compartments i and j whose respective voltages are V_i and V_j; finally, $v_{\text{leak},i}$ is the leakage voltage.

The calculation of the individual channel currents depends on the individual model and is therefore beyond the scope of this paper. In this work, we use the De-Gruijl IO model for benchmarking, for which the full set of channel current equations can be found in the supplement to [20] and in [1].

Gap-junction currents are dependent on a connectivity list, which can be supplied as an input file by the neuroscientist. An alternative option is to use an algorithm to generate the currents; the neuroscientist then provides the parameters for this algorithm. The simulator supports two algorithms: a random-binary (uniformly distributed) and a random-Gaussian generation of currents. Both were assessed during the SimHH evaluation.

The elementary function *UpdateComp()* contains all the value-update operations described in the aforementioned equations. The workings of this function are illustrated in Fig. 3. The equations correspond to those used by flexHH in [1], apart from the following adaptation: the compartment now has the option of adding a dedicated calcium ion channel, instead of needing to model it as another generic channel, which flexHH and GenEHH require. This choice makes the channel updates independent of each other and generalizes the support for calcium ion channels, which leads to better parallelism.

B. USER INPUT/OUTPUT

JavaScript Object Notation (JSON) was selected as the input file format for accepting user input as well as the overall simulator behavior. This decision was made because some of the required groundwork had already been completed for GenEHH. JSON is also a widely accepted data format that can be used without coding knowledge, and there is no limit to what can be expressed as structured data other than the efficiency of the American Standard Code for Information Interchange (ASCII) encoding. In practice, most SNN models use populations of replicated, ‘‘template’’ neuron models. There are only a few different neuron models per network, and hence neuron modeling is manageable with JSON. Connection matrices are currently either generated from statistical distributions or loaded explicitly from a separate, raw file. Being able to simulate the full eHH class of neurons with the configuration is a primary requirement. This is possible due to the fact that all possible models in the eHH class of neurons can be described in the flexHH formulations. Python bindings can also be used to input network parameters and to control simulator behavior.

When describing neurons, one can replicate the description of a single cell by adding a multiplier entry to the cell description. However, biological systems are never exact replicas of one another. Therefore, we allow for a randomization factor to be added to the conductance and reversal potential of each channel. The randomization strategy was devised based on experimental values from peer works in neuroscience and was sufficient for realistic simulation runs. Of course, the user still has the freedom to describe each cell explicitly with different parameters.

The simulator output can be adjusted according to the user’s requirements. The compartment potentials and calcium

concentrations, channel currents, and gate-activation states are available to the user on each time step in formatted ASCII format written to a text file. However, the float-to-ASCII conversions lead to a higher execution time. To counter this, the design includes the option of outputting a raw binary file. The user can set a time-step interval between each output generation. Higher performance can be achieved by outputting binary data, particularly with non-local storage locations. Writing the output to a file can slow down simulation runs.

C. DESIGN CHOICES FOR PARALLELIZATION

Parallelization refers to design choices that allow for multiple processes to be executed simultaneously. This greatly reduces execution time, but can only be achieved when processes do not have to wait for each other. Memory dependencies heavily limit parallelization. In our model, the output of each time step depends on the output of the previous time step. Therefore, it is evident that different time steps, regardless of the type or method, must be processed sequentially. However, within a single time step, neurons that do not depend on each other can be processed independently. This offers opportunities for parallelization.

Parallelization at the cell or compartment level yields the same blocks of calculations per element. Compartment-wise, parallelization is preferred because it splits up the calculations into more parallel blocks and does not lose performance when compartment counts differ across the network cells. However, there are further opportunities for parallelization within the compartments: the gap-junction calculations could benefit from having multiple threads available to calculate the gap-junction current to further parallelize the accumulation of single connections. Furthermore, the gate updates can be processed in parallel.

Because of the diversity of calculations and accumulations, naively creating a separate kernel for each contributing current in the compartment-update function would not benefit the performance. Instead, we split the *UpdateComp*() function into separate gap-calculation, gate-update, and compartment-update functions. This division allows for *overlapping* communication and calculations when running on a distributed computing system, which is further explained in Fig. 4.

SimHH is compatible with CPU-only systems, and makes use of parallelization on CPU-only systems too. However, SimHH only reaches its full potential in a multi-GPU setting. Therefore, the reported performance on CPU-only systems will serve only as a baseline for comparison.

1) MEMORY ARRANGEMENT TO SUPPORT WARP EQUALITY

CUDA works in a single-instruction-multiple-data (SIMD) fashion at the streaming-multiprocessor (SM) level. Each SM executes a Warp of Threads, typically 32. A CUDA block typically consists of many of these warps to allow the SM to context switch between warps to hide memory access times. Because a single instruction is executed for all threads in the warp, it is crucial to group the same compute in SM-aligned tasks, to achieve the best parallelism and full SM occupancy. Mapping this to SimHH, it becomes clear that compartments or gates should be grouped. The gap-junction calculations are not affected by this because of the already similar tasks for each CUDA thread. Combining multiple memory accesses into a single transaction is referred to as *coalesced memory access* or *memory coalescing*. For example, in a single transaction, any successive memory of 128 bytes can be accessed by a warp. However, uncoalesced loads can result in memory access becoming serialized, which is detrimental to memory performance. Therefore, it is vital to be mindful of memory coalescing when implementing CUDA kernels. For SimHH this means getting the memory layout right. As we parallelize over the gate level, it becomes clear that striding all neuron parameters into memory should be done with the same stride as we parallelize the compute. CPU support will also benefit from this arrangement, as it improves cache performance.

2) GATE AND COMPARTMENT UPDATES

The gate- and compartment-update kernels are implemented in a straightforward manner. The gate-update kernel has the same number of threads as there are gates in the neurons. Each thread is responsible for updating its gate with the $Y_{update}()$ function (see Fig. 3). The compartment-update kernel does the same at the compartment level, implementing the complete scheme shown in the figure (with the exception of the gate updates and gap-junction-connection current calculation). The gap-junction-connection current is provided as a direct input by the gap-junction-calculation kernel. Then, the Y_k value is updated to $Y_{k,t+1}$, which is handled by the gate-update kernel. This approach allows for full parallelization of the compute. We then grouped the same compute paths, resulting in full SM occupancy.

3) GAP-JUNCTION CALCULATIONS

For the gap-junction calculation on the GPU backend, we took a slightly different approach. For each cell, an adjustable number N of CUDA threads will be instantiated. All these N threads work together to fetch from memory and accumulate to a shared result: the gap-junction current of that specific cell. The accumulations rely on warp-level primitives, limiting the maximum number of threads to 32 per cell (one warp). We hypothesized that the system could benefit from even more threads by utilizing shared memory for the additions, making it possible to use a full CUDA thread block. However, we briefly tested this and observed that, in fact, it did *not* offer any improvements. Since the focus is on large-scale network experiments, it is estimated that the GPU(s) will be fully utilized regardless, and further division would likely decrease, rather than enhance, the performance. After all, more threads require more management in terms of communication and memory. Sparse gather-scatter operations, and more specifically, the

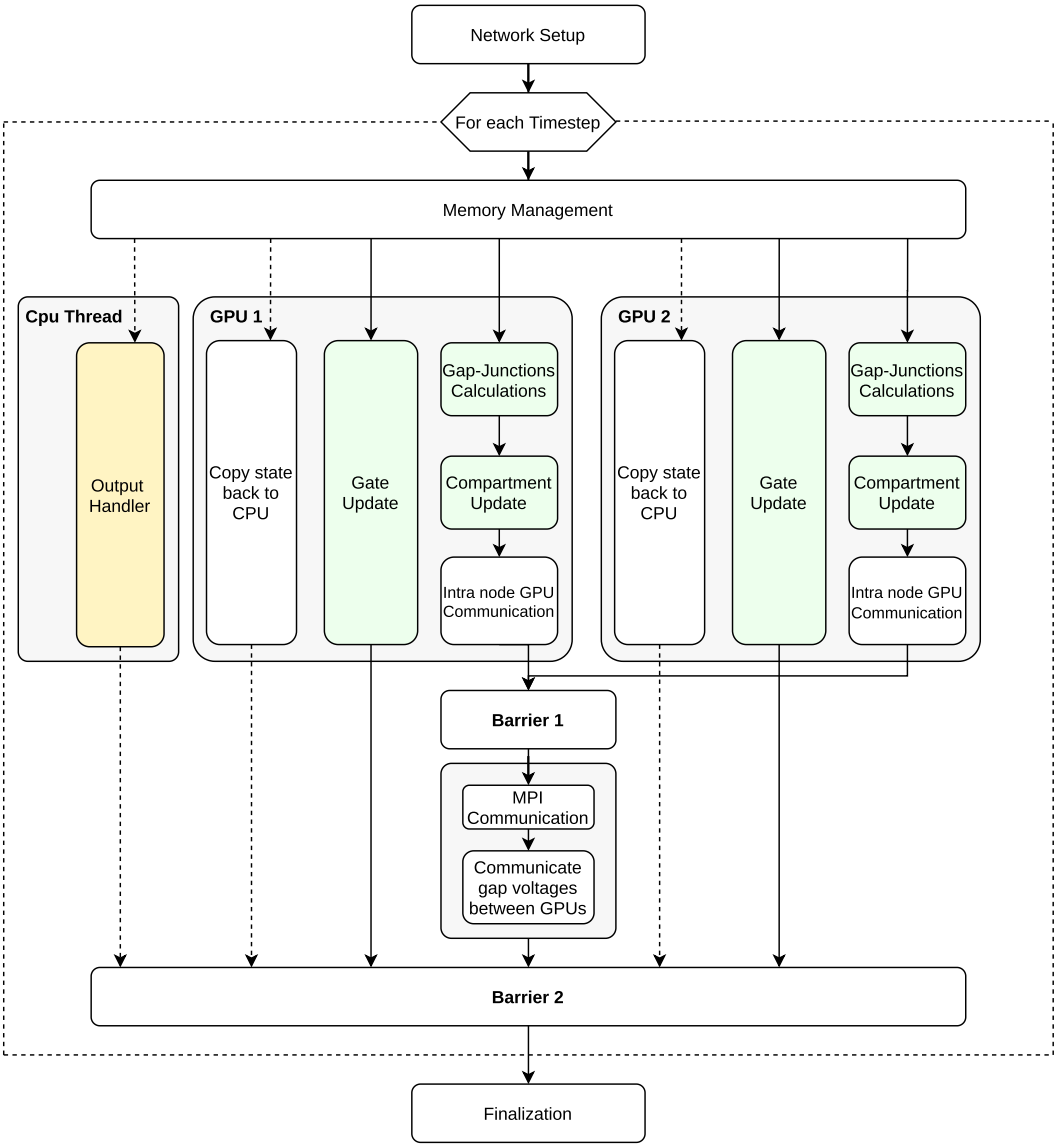

FIGURE 4. Functional overview of the SimHH deployment and execution. Green-colored blocks represent CUDA kernels on the GPU backend (or CPU functions when the CPU backend is selected), while yellow-colored blocks represent CPU tasks. White blocks are specific to data management and control flow. All calculations in Fig. 3 are included in the 'Compartment update' block, except for gap-junction calculations and gate updates, which have dedicated kernels represented by the 'Gap-Junction Calculations' and 'Gate Update' blocks, respectively. This overview is for a single process with two GPUs connected to one compute node. MPI communication occurs inter-node across multiple compute nodes in a cluster, with each node running a similar process. For single-GPU nodes, the 'Inter-node GPU communication' block can be seen as a NULL function. Similarly, the 'MPI communication' block is a NULL function in single-process execution.

sparse nonlinear coupling described above, can be efficiently computed within the CUDA framework.

4) GAP-JUNCTION GENERATION

Setting up the network mostly pertains to allocating and initializing the memory correctly based on the configuration file. However, if the gap-junction network is generated according to an algorithm, it is incorporated into the simulator. Alternatively, when a neuroscientist designs a connection list herself, she can link a file in the configuration, and SimHH will read out the connection list.

The generation of the connection graph is important for the performance experiments of the network. Most end-users will probably provide their custom-tailored connection lists that suit their specific needs. Therefore, even though we optimized the generation for GPU execution, we chose not to focus on implementing every optimization possible. As stated before, two schemes are implemented in SimHH: Random-Binary, shown in Algorithm 1 and 1D-Gaussian, shown in Algorithm 2. The RandomBinary generation is considered to be the most challenging interconnection network because of its unpredictable memory-access patterns.

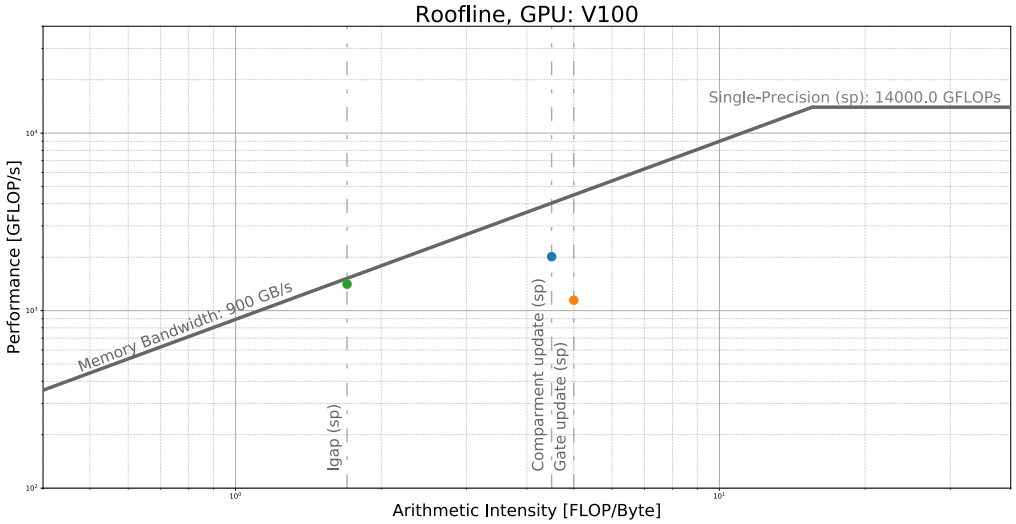

FIGURE 5. Roofline model for the SimHH GPU kernels. Single precision (sp) kernels are presented in this figure. The exact configuration for the presented results is: 131072 IO neurons, 1% Density for a RandomBinary generated synaptic-connection network, and the De-Gruijl IO model for each cell in the network. The dots represent performances, which were measured manually. The created roofline is for a NVIDIA V100 GPU.

Algorithm 1 RandomBinary Generation (Density, NetworkSize)

```
for 0 ≤ NeighborCell < NetworkSize do
    if rand(0, 1) ≤ Density/NetworkSize then
        -> Add NeighborCell to TargetCell
    end if
    if ConnectionCount ≥ Density then
        break()
    end if
end for
```

Algorithm 2 1D-Gaussian Generation (Mean, Variance, NetworkSize, Density)

```
xd = 1 / (variance * sqrt(2 * PI))
yd = -(1 / (2 * pow(variance, 2)))
for 0 ≤ Distance < NetworkSize do
    if (TargetCell − Distance) < NetworkSize then
        if rand(0, 1) ≤ xd∗exp(pow(distance−mean, 2)∗
            yd) then
            -> Add NeighborCell (TargetCell - Distance)
                to connection graph
        end if
    end if
    if TargetCell > Distance then
        if rand(0, 1) ≤ xd∗exp(pow(distance−mean, 2)∗
            yd) then
            -> Add NeighborCell (TargetCell + Distance)
                to connection graph
        end if
    end if
    if ConnectionCount ≥ Density then
        break()
    end if
end for
```

D. DESIGN CHOICES FOR SCALABILITY

Scalability is of great importance for the SimHH implementation. A highly scalable application can utilize anything from a computer at home up to HPC platforms such as supercomputers from the TOP500 list, while making optimal use of the available resources. Expanding to multi-node or, in other words, a multi-process program requires communication between processes. The analysis of the communication boils down to the communication of compartment potentials when gap junctions are utilized. Everything else is known in each separate process, either by the process ID or by the input handling that each process performs at startup. It would be beneficial to perform communication and computing in parallel. Unfortunately, communication regarding compartment updates is necessarily sequential, simply because each new time step requires the compartment voltages of the connected cells. These are available only at the end of the previous time step. However, the gate updates do *not* have a sequential dependency within a time step, and can therefore be processed simultaneously with communication.

Memory-wise, it is necessary to save all compartment potentials required for the gap-junction calculations, for every local cell-connection list. When utilizing GPUs, this needs to be in GPU-accessible memory. This can be memory directly located on a GPU card, but it can also be located in the Unified Virtual Addressing (UVA) space of a compute node. Using UVA saves memory on multiple GPU compute nodes, but leads to worse performance compared to GPU-specific memory allocation. In general, there is a trade-off between lower memory use and higher performance. However, for SimHH, the choice is easy: using the unified memory space

leads to such high data latencies compared to the GPU-specific memory, that using UVA memory will *always* lead to worse performance. Distributing these potentials to remote nodes instead is much simpler, as the remote nodes treat these values as read-only and do not update them.

Open Multi-Processing (OpenMP) and Open Message Passing Interface (OpenMPI) were designed to allow developers to optimize memory access and inter-process communication, and were therefore selected as support libraries to create this feature. To communicate between MPI processes, two strategies are available:

1) Communicate everything to everyone (`MPI_Allgather`).
2) Communicate the necessary data only to specific nodes (`MPI_Alltoallv`). Each process may send a different amount of data and provide displacements for the input and output data.

Measuring the performance using these two strategies across different types and levels of connectivity will provide valuable insights for HPC neural simulations of any kind. We hypothesize that the more localized the connections, the greater is the performance benefit from the second strategy. After all, the number of gap junctions that span MPI processes is reduced, which lowers the communication overhead. MPI can be used in a CUDA-aware fashion, enabling direct access to GPU memory, which enhances the performance by minimizing data movement. Inter-GPU communication is handled through peer-to-peer copying, utilizing GPUDirect with or without NVLink, depending on platform support. The performance results for each MPI communication strategy are presented in detail in Section V.

E. ANALYSIS OF THE PROPOSED DESIGN

1) FUNCTIONAL FLOW

The functional flow of SimHH is broken down into a functional-level description for a particular system configuration, as shown in Fig. 4. The neuronal-network setup consists of initializing CPU and, when applicable, GPU memory. Memory management must ensure data correctness because the system works with separate kernels to perform gate and compartment updates. A double-buffering system is necessary to prevent the compartment potentials from the previous time step from being overwritten by the values of the current time step. Such a system ensures that all operations dependent on the old values have a chance to read them before they are updated: it employs two memory allocations, where one holds the old values and one holds the updated values. Therefore, each state of the network, compartment potentials and calcium concentrations, channel currents, and gate-activation states are double-buffered. An added benefit is that we can copy the old values back to system memory from the first buffer, *while simultaneously calculating the new values*. In other words, because of the double buffering, the GPU-to-CPU data transfers can be performed entirely concurrently with the execution of the compute kernels. The double-buffering system is an important design optimization that is useful not only for neural simulations, but for any type of simulation where dependencies exist between timesteps, but not within a single timestep.

2) ROOFLINE-MODEL GPU KERNELS

For the case where a GPU backend is available (i.e., one or more nodes), we constructed a roofline model to gain further insight into the specific GPU kernels (see Fig. 5). It is clear that, in this case, the gap-junction calculation kernel is heavily memory-bound. The kernel often stalls because it has to wait for memory accesses. This suggests that there is still room for performance improvement in the compartment- and gate-update kernels. Both are, in principle, memory-bound tasks, but it can be observed that the device does not achieve peak performance within this bound. This is mainly because GPU warps are stalling, waiting for dependencies. The `Stall Long Scoreboard` is the most frequently occurring stall, which indicates that memory-access patterns are not optimal for these kernels. The compartment-update kernel does not achieve maximum warp occupancy, because the number of available registers is lower than the optimal number of registers required per thread, for this specific neural model.

Possible improvements for the compartment- and gap-update kernels include better memory-access patterns by rearranging the memory used for the configuration parameters, differently, or better shared-memory usage. The gap-junction calculation kernel is bottlenecked by memory bandwidth. Efforts can be made to better hide this latency by adding more compute complexity. The gap-junction calculations also lack concurrency in the memory accesses, due to their random nature, which limits the maximum achievable performance.

V. RESULTS

In this section, we evaluate SimHH in terms of performance, scalability and various design decisions. We will also provide a performance comparison to other competitive, state-of-the-art neurosimulators, in order to demonstrate the merits of SimHH. Several experiments have been conducted over a range of neuronal networks based on the IO model to establish the scalability characteristics. The resources for the experiments were allocated on Piz Daint, a hybrid Cray XC40/XC50 system at the Swiss National Supercomputing Centre (CSCS). Functional validation of the simulator was conducted using the original De-Gruijl IO model, which is also the neuron description used for the performance evaluation. The Gaussian distributed gap-junction network is biologically realistic, whereas the Random-Binary distributed network was mostly added for stress-testing purposes.

A. EXPERIMENTAL SETUP

Piz Daint is the computing platform that was used for this evaluation. This work exclusively uses the XC50 nodes which contain an Intel Xeon E5-2690 v3 @ 2.60GHz

TABLE 2. Benchmark exploration space. All experiments are performed using IO neurons. .

Parameter	Range
Number of Nodes	1, 2, 4, 8, 16, 32, 64, 128, 256
Selected Back-end	GPU, CPU
MPI communication type	Alltoallv, Allgather
Gap-junction distribution type	Random Binary, Gaussian
Density	10, 100, 1000
Number of neurons	8192, 16384, 32768, 65536, 131072, 262144, 524288,
(only for* ≥ 64 *nodes)*	*1048576, *2097152, *4194304, *8388608
Gaussian mean	0
Gaussian variance	Number of neurons * 2

(12 cores, 64GB RAM) CPU and a single NVIDIA Tesla P100 16GB GPU. The De-Gruijl IO model was selected for benchmarking, which provides a fair comparison with previous works. The full benchmarking configuration is presented in Table 2.

B. CPU- VS GPU-BACKEND PERFORMANCE

The design of SimHH had a strong focus on the GPU backend, which is expected to significantly outperform the OpenMP multi-threaded CPU backend. The results presented in Fig. 6 confirm this. Fig. 6 also shows the strong linear-scaling properties of SimHH, as long as the problem size is sufficient to fully saturate the capabilities of the compute node(s).

C. GAP-JUNCTION DISTRIBUTION

As can be seen in Fig. 7, the Gaussian versus Random-Binary gap-junction results clearly show that the more local the gap junctions are (relative to the connected neuron), the better the performance becomes. Clearly, a higher degree of cache hits contributes to faster simulation times. Conversely, a higher number of nodes leads to higher overhead costs. Therefore, as the number of nodes increases, the achievable gains in simulation time per step decrease. The presented results are for a GPU backend with the `MPI_Allgather` communication option.

D. FUNCTIONAL-FLOW PERFORMANCE

The bottom bar chart in Fig. 8 displays the simulation time (in seconds) for different numbers of nodes, broken down into their respective tasks along the critical path, as detailed in the functional flow diagram in Fig. 4. The top bar chart shows the execution times for the three GPU kernels. As the node count increases, the execution times for the gap-junction calculation and compartment updates (represented by `stall barrier 1`) decrease proportionally due to the distributed computing tasks. Kernel launch times also decrease with smaller local network sizes, as higher node counts lead to smaller grid dimensions for each launch. However, MPI communication times increase with more nodes due to the additional management overhead of each extra node. The gate updates (represented by `stall barrier 2`) are executed almost entirely concurrently with other tasks and do not contribute to the critical path, indicating a well-balanced design for this network configuration.

E. MPI-COMMUNICATION PERFORMANCE

Communication overhead is a crucial metric for distributed systems, but it is also notoriously difficult to measure and optimize. Unlike computational resources, users typically have limited control over the network infrastructure that connects different processing nodes. This lack of transparency makes it challenging to identify and address communication bottlenecks.

When the interconnection architecture is unknown, it is impossible to ensure that any two nodes are physically close to each other. Since the cluster network may be used by other jobs simultaneously, obtaining noise-free performance results for comparison is very difficult. However, the Piz Daint network architecture uses a dragonfly topology with 384 nodes per electrical group, thereby eliminating any interconnection uncertainties within a group. Therefore, the results obtained by running SimHH on this network are suitable for comparison.

We hypothesized that locality in the gap-junction network would result in lower MPI communication times when using the `MPI_alltoallv` strategy, while a more randomly distributed network would experience added overhead, making the `MPI_allgather` strategy preferable. Fig. 9 confirms our hypothesis: neither strategy is optimal for all networks, and the optimal choice depends on the locality of the gap-junction network. Since it is the key factor explaining the performance differences between strategies, the locality of the gap-junction network should be the guiding metric when it comes to the choice of strategy.

F. VERSATILITY OF SIMHH

As mentioned previously, SimHH is a versatile eHH simulator. Fig. 10 illustrates the performance of three different network configurations: the original HH model (without any gap junctions), the De-Gruijl IO model with three compartments and gap junctions, and an eHH description with 1,000 compartments, featuring exponential gap junctions (1000Comp). The ability of SimHH to simulate such a broad range of neuronal networks demonstrates its versatility.

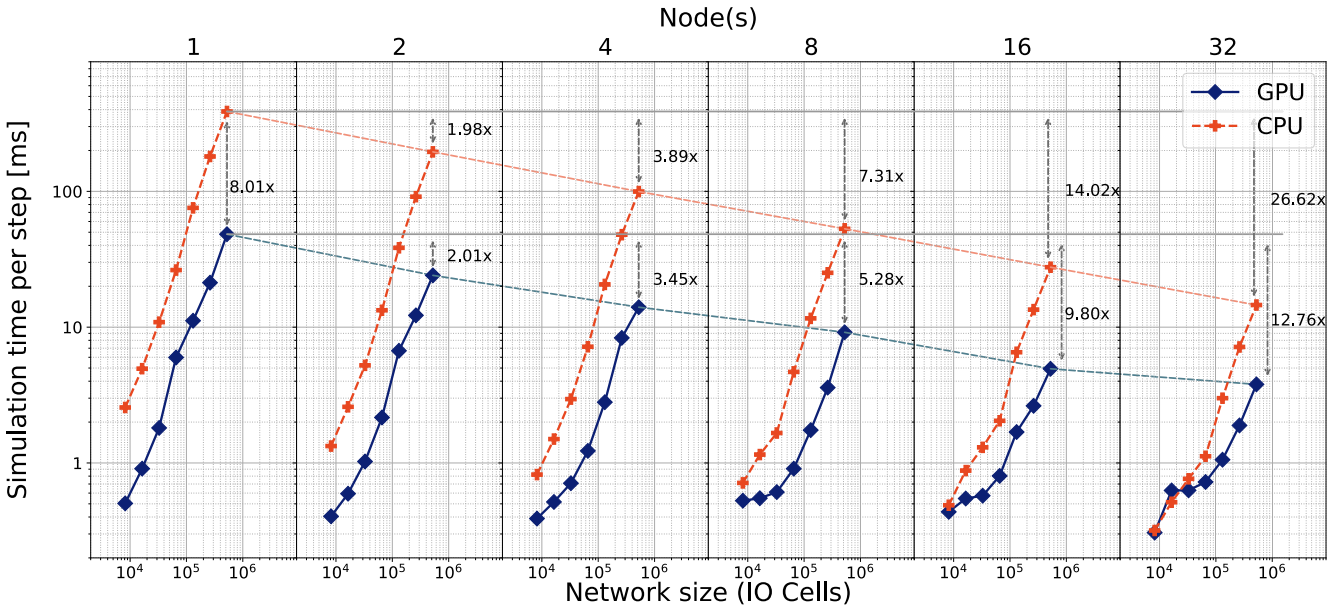

FIGURE 6. Performance-scalability results of the SimHH GPU and CPU; the GPU benefits are obvious. Results are displayed up to 32 compute nodes. For the network topology, we used a Random-Binary distribution with a density of 1,000 gap junctions generation with a density of 1,000. Visual aiding lines are added to show excellent weak-scaling characteristics.

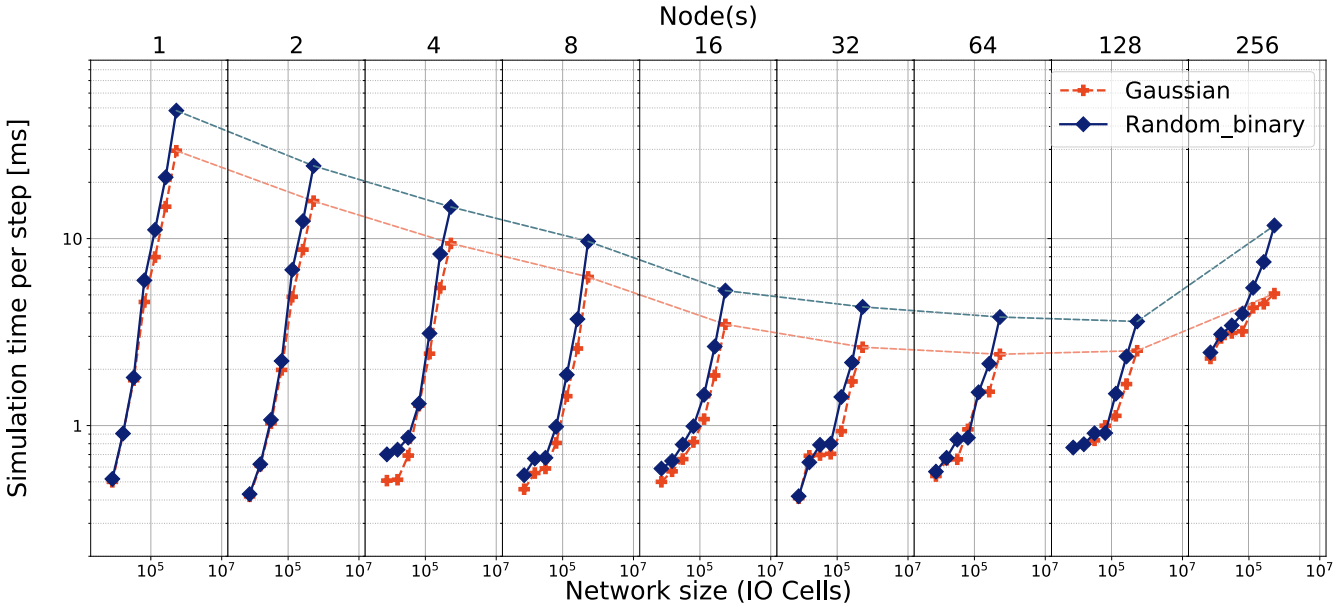

FIGURE 7. Experimental results to determine the impact of a RandomBinary versus 1D-Gaussian distributed gap junction network. The network configuration has a density of 1000 and the GPU backend is utilized.

It can handle anything from a single compartment with N gates to M compartments with N gates, as long as it falls within the subset of supported network configurations for SimHH.

1000Comp represents a large and complex model. Fig. 10 shows that larger models require more compute resources and their simulation is therefore slower. However, the simulation of the *1000Comp* model was *less* than $\frac{1000}{3}\times$ slower than that of the De-Gruijl IO model with three compartments. This is because kernel initialization and cache performance do not scale linearly with the network parameters.

As expected, the *1000Comp* model has a considerably larger memory footprint than the De-Gruijl IO model. Unfortunately, the results for simulations of this model with large network sizes on the 2- and 4-node configurations are not plotted, as these simulations failed to complete due to insufficient GPU memory. However, they successfully

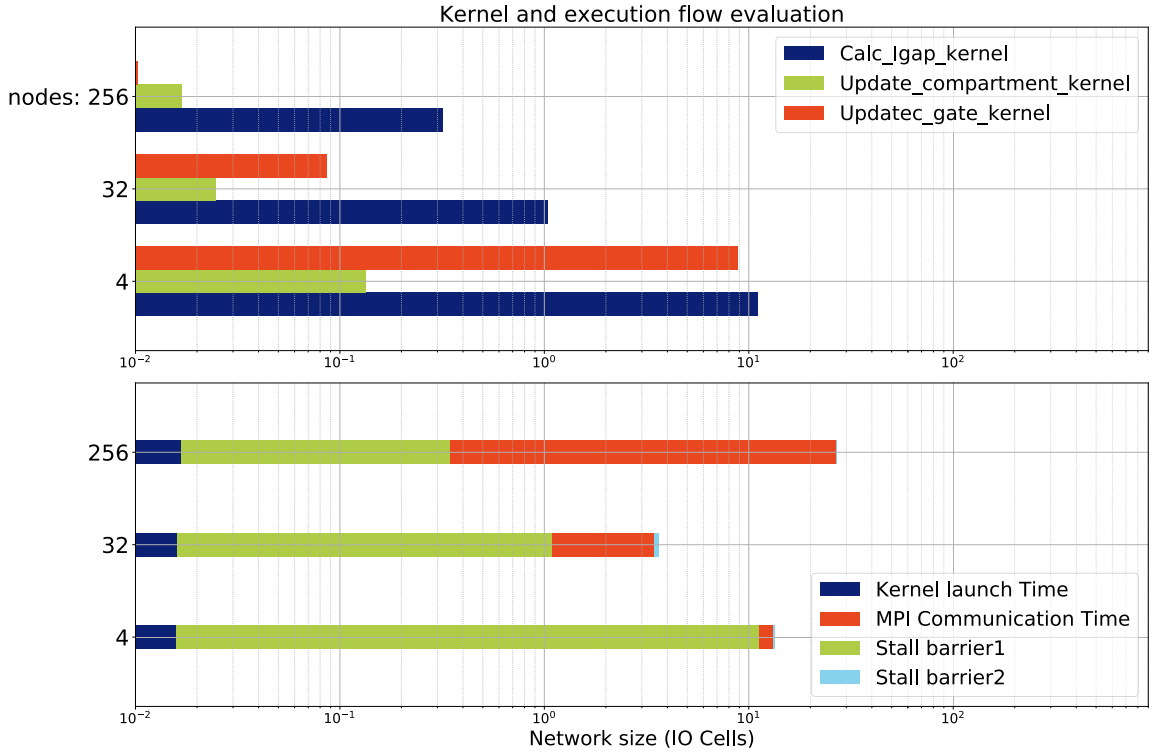

FIGURE 8. Functional-flow performance, in line with Fig. 4. The network was configured to simulations with 524,288 IO neurons and a density of 1,000 with a Random-Binary distribution. MPI communication utilizes the `MPI_AllGather` strategy. Results are shown for 4, 32 and 256 nodes at the Piz Daint compute cluster.

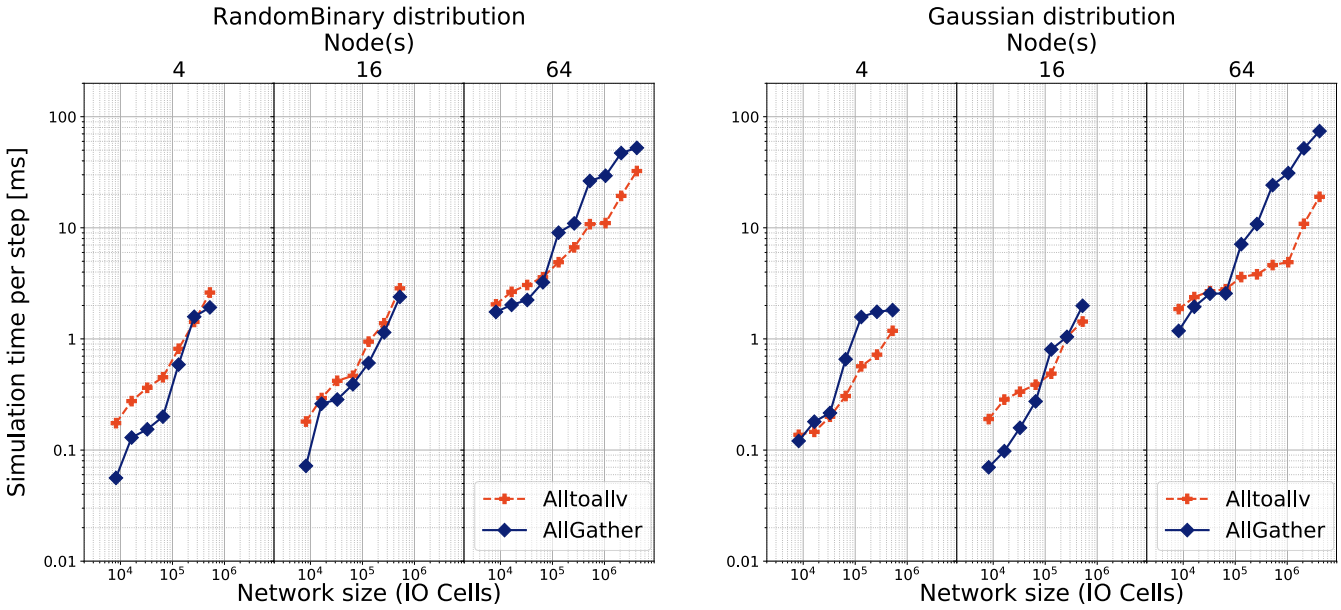

FIGURE 9. MPI communication results, at the Piz Daint compute cluster. The results illustrate the `MPI_allgather` method versus the `MPI_alltoallv` method. It becomes clear that, with a higher rate of locality (Gaussian), the latter becomes favorable.

completed on the 8-node configuration. This underscores that memory is a scarce commodity and highlights how scaling across multiple nodes enables the simulation of larger, more intricate SNNs.

G. COMPARISON OF SIMHH WITH RELATED WORK

Given the abundance of available neurosimulators, a comprehensive comparison with all related works is infeasible. However, Table 3 provides an overview of the landscape,

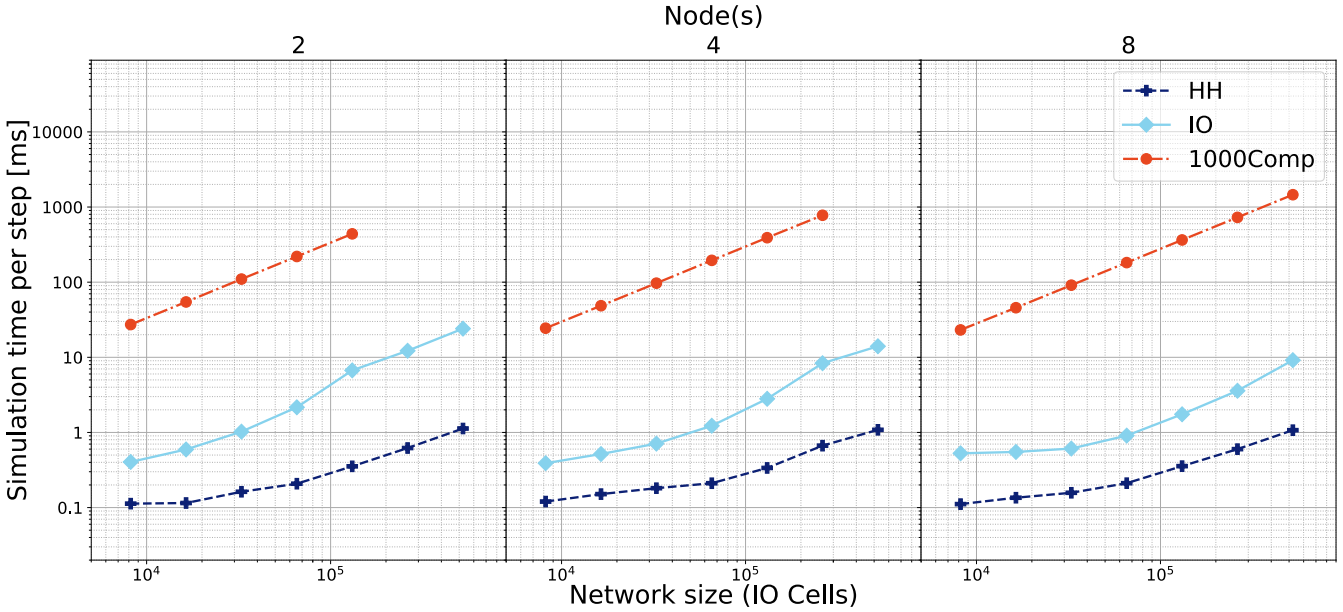

FIGURE 10. Performance, measured as simulation time per step, for the HH model, the De-Gruijl IO model, and a non-biologically meaningful 1000-compartment model. Simulation time varies with increasing neuron complexities in terms of size. Notably, results for the 1000-compartment model are absent for larger network sizes on the 2- and 4-node configurations due to memory limitations.

TABLE 3. Performance comparison with related works.

Reference	Platform	Network size	Connection/neuron	Biological time or simulation steps	Execution time (s)
Florimbi et al. [27]	NVIDIA Tesla K40	400,000	8	40,000 steps	5,193
SimHH	1x NVIDIA Tesla V100	400,000	8	40,000 steps	7.8
G. Chatzikonstantis et al. [28]	Multi-node Phi KNL (8 nodes)	1 million	1,000	50 ms	2,000
SimHH	Piz Daint (8x1 NVIDIA Tesla P100)	1 million	1,000	50 ms	0.36
v/d Vlag et al. [25]	Surfsara (16*2x NVIDIA Tesla K40m)	4 million	1,000	25 ms	800
SimHH	Piz Daint (32*1x NVIDIA Tesla P100)	4 million	1,000	25 ms	60
Miedema et al. (flexHH) [1]	Maxeler Maia Data Flow Engine	7,808	7,808	2.5 ms	2.05
SimHH	1x NVIDIA Tesla V100	7,808	7,808	2.5 ms	0.1951
Brain2 (cython)	AMD EPYC 7551 32-Core Processor	7,808	7,808	2.5 ms	404.63
SimHH	AMD EPYC 7551 (1thread)	7,808	7,808	2.5 ms	33.60
SimHH	AMD EPYC 7551 (128thread)	7,808	7,808	2.5 ms	2.02
Panagiotou et al. (genEHH) [18]	Intel Xeon Platinum 8124M	16,000	4,000	10,000 steps	200
SimHH	1x NVIDIA Tesla V100	16,000	4,000	10,000 steps	17.54
NEST [11]	JURECA HPC 32 nodes	94,720	60	500 ms	∼100
SimHH	Piz Daint (32*1x NVIDIA Tesla P100)	94,720	60	500 ms	14

including the work presented here, offering insight into the performance improvements achieved compared to earlier work. Multiple entries of SimHH were included to allow for performance comparison on comparable platforms. [27] To ensure a fair comparison, we utilized the same solver (forward-Euler) wherever possible, and attempted one-to-one comparisons of simulation steps. In cases where this was not feasible, biological time was used as a metric, with SimHH employing a time step of 0.025 ms.

Each comparison in Table 3 models SimHH's network configuration to closely resemble the referenced work. However, achieving a perfectly fair comparison is not always possible, because published numbers often indicate differences in orders of magnitude between various simulators. It is essential to note that SimHH is a dedicated eHH simulator, enabling a higher degree of optimization compared to more general simulators such as Neural Simulation Tool (NEST) and BRIAN. Furthermore, the superior performance of SimHH compared to hard-coded IO simulators can be attributed to better code and kernel designs. The significant speedup over previous multi-node implementations (Chatzikonstantis et al. [28] and v/d Vlag et al. [25]) is largely due to SimHH's overlapping-kernel design and a generally more performance-optimized codebase.

VI. DISCUSSION

As described in Section IV-E2, the performance in the compartment- and gate-update kernels is suboptimal. The GPU does not achieve peak performance within the memory bounds defined in the roofline model. Potential improvements include optimizing memory-access patterns, shared-

memory usage or adapting memory allocation based on configuration parameters. However, additional algorithms can incur overhead, and full resource usage does not always equate optimal performance. Large simulations are expected to get more predictable memory accesses for the local neural update and the gap-junction updates. This potentially creates more random lookups, even though – following biology – the number of connections stays more or less the same and does not grow with the amount of neurons. So, following this reasoning, larger models become more predictive in terms memory accesses.

Section V-G highlights the challenges of achieving a perfectly fair comparison with previous works due to varying simulation setups and reported metrics. It should also be noted that different neurosimulators have different strengths. While SimHH excels in performance for eHH models when compared to NEST and BRIAN, these latter simulators offer broader model support. SimHH's specialization limits its applicability to other neuron models. However, SimHH remains a highly flexible tool for eHH models, offering superior performance and scalability compared to existing solutions.

Moving forward, several promising directions emerge. Investigating the impact of gap-junction network locality on communication overhead offers insights into the preferred communication methods. Ongoing CUDA design enhancements hold potential for further performance gains. Expanding the simulator's features to support a wider range of experiments and incorporating higher-order solvers can enhance the versatility. In addition, collaborating with neuroscientists to simulate human-scale IO models could offer insights into complex neural dynamics. Exploring strategies for optimized neuron grouping and memory allocation should minimize communication overhead. Grouping could be done with respect to the compute-power availability for each grouping. Furthermore, an analysis of energy and hardware resource utilization could provide valuable insights regarding the cost-effectiveness of SimHH. Finally, implementing parsers to convert NeuroML descriptions would streamline the simulation-setup process and enhance usability.

VII. CONCLUSION

This paper introduces a novel neurosimulator, SimHH, capable of simulating a wide range of eHH neural networks. Building upon the work of Miedema et al. and Panagiotou et al., who developed flexible eHH simulators for specialized hardware, SimHH offers a distributed-memory implementation with GPU and CPU backends. This design achieves outstanding performance for large-scale eHH simulations. SimHH exhibits high scalability due to its efficient use of parallelism. GPU-to-CPU data transfers are done entirely concurrently with the execution of the compute kernels. SimHH uses OpenMP and OpenMPI to optimize memory access and inter-process communication, avoiding the high data latencies that would come with UVA memory. It can handle simulations of millions of cells with high connectivity densities by concurrently performing gap-junction communication alongside cell computations whenever the network topology allows. The memory management of SimHH was optimized to minimize storage requirements. Unnecessary data are offloaded to the file system during simulation, allowing for very long biological time simulations without memory limitations. We demonstrated SimHH's effectiveness in simulating a challenging inferior-olivary-nucleus model by De Gruijl et al. Our experiments involved up to 256 nodes, showcasing linear weak and strong scaling as long as hardware resources are fully utilized. Two communication schemes were compared for inter-process communication: `MPI_allgather` and `MPI_alltoallv`. `MPI_alltoallv` outperforms `MPI_allgather` for networks with high locality in the gap-junction graph. Conversely, `MPI_allgather` is more efficient for completely random networks because of the lower overhead. This finding helps to optimize communication for multi-GPU neural simulations. In conclusion, SimHH breaks new ground when it comes to performance, while also providing important insights for future neurosimulators.

ACKNOWLEDGMENT

This paper is partially supported by the European-Union Horizon Europe R&I program through projects SEPTON (no. 101094901) and SECURED (no. 101095717) and through the NWO - Gravitation Programme DBI2 (no. 024.005.022). We would also like to thank Sotirios Panagiotou and Lennart Landsmeer for their valuable help and input to this work.

REFERENCES

[1] R. Miedema, G. Smaragdos, M. Negrello, Z. Al-Ars, M. Möller, and C. Strydis, "FlexHH: A flexible hardware library for Hodgkin–Huxley-based neural simulations," *IEEE Access*, vol. 8, pp. 121905–121919, 2020.

[2] S. Herculano-Houzel, "The human brain in numbers: A linearly scaled-up primate brain," *Frontiers Human Neurosci.*, vol. 3, p. 31, Nov. 2009.

[3] D. A. Drachman, "Do we have brain to spare?" *Neurology*, vol. 64, no. 12, pp. 2004–2005, Jun. 2005. [Online]. Available: https://n.neurology.org/content/64/12/2004

[4] *TOP500*. Accessed: Jun. 1, 2024. [Online]. Available: https://www.top500.org/lists/top500/2024/06/

[5] *Fugaku*. Accessed: Jun. 1, 2024. [Online]. Available: https://www.r-ccs.riken.jp/en/fugaku/about

[6] E. M. Izhikevich, "Which model to use for cortical spiking neurons?" *IEEE Trans. Neural Netw.*, vol. 15, no. 5, pp. 1063–1070, Sep. 2004.

[7] S. Noor, S. A. AlQahtani, and S. Khan, "Chronic liver disease detection using ranking and projection-based feature optimization with deep learning," *AIMS Bioeng.*, vol. 12, no. 1, pp. 50–68, 2025. [Online]. Available: https://www.aimspress.com/article/doi/10.3934/bioeng.2025003

[8] S. Khan, S. Noor, T. Javed, A. Naseem, F. Aslam, S. A. AlQahtani, and N. Ahmad, "XGBoost-enhanced ensemble model using discriminative hybrid features for the prediction of sumoylation sites," *BioData Mining*, vol. 18, no. 1, p. 12, Feb. 2025, doi: 10.1186/s13040-024-00415-8.

[9] N. A. Akar, B. Cumming, V. Karakasis, A. Kusters, W. Klijn, A. Peyser, and S. Yates, "Arbor—A morphologically-detailed neural network simulation library for contemporary high-performance computing architectures," in *Proc. 27th Euromicro Int. Conf. Parallel, Distrib. Netw.-Based Process. (PDP)*, Feb. 2019, pp. 274–282, doi: 10.1109/empdp.2019.8671560.

[10] M.-O. Gewaltig and M. Diesmann, "NEST (NEural simulation tool)," *Scholarpedia*, vol. 2, no. 4, p. 1430, 2007.

[11] J. Jordan, M. Helias, M. Diesmann, and S. Kunkel, "Efficient communication in distributed simulations of spiking neuronal networks with gap junctions," *Frontiers Neuroinform.*, vol. 14, p. 12, May 2020. [Online]. Available: https://www.frontiersin.org/article/10.3389/fninf.2020.00012

[12] P. Kumbhar, M. Hines, J. Fouriaux, A. Ovcharenko, J. King, F. Delalondre, and F. Schurmann, "CoreNEURON : An optimized compute engine for the NEURON simulator," *Frontiers Neuroinform.*, vol. 13, p. 63, Sep. 2019. [Online]. Available: https://www.frontiersin.org/article/10.3389/fninf.2019.00063
[13] D. Goodman, "Brian: A simulator for spiking neural networks in Python," *Frontiers Neuroinform.*, vol. 2, p. 350, Nov. 2008. [Online]. Available: https://www.frontiersin.org/articles/10.3389/neuro.11.005.2008
[14] M. Stimberg, D. F. M. Goodman, V. Benichoux, and R. Brette, "Brian 2—The second coming: Spiking neural network simulation in Python with code generation," *BMC Neurosci.*, vol. 14, no. 1, p. 38, 2013, doi: 10.1186/1471-2202-14-S1-P38.
[15] E. Yavuz, J. Turner, and T. Nowotny, "GeNN: A code generation framework for accelerated brain simulations," *Sci. Rep.*, vol. 6, no. 1, p. 18854, Jan. 2016, doi: 10.1038/srep18854.
[16] L. Niedermeier, K. Chen, J. Xing, A. Das, J. Kopsick, E. Scott, N. Sutton, K. Weber, N. Dutt, and J. L. Krichmar, "CARLsim 6: An open source library for large-scale, biologically detailed spiking neural network simulation," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, Jul. 2022, pp. 1–10.
[17] M. Ward and O. Rhodes, "Beyond LIF neurons on neuromorphic hardware," *Frontiers Neurosci.*, vol. 16, Jul. 2022, Art. no. 881598. [Online]. Available: https://www.frontiersin.org/articles/10.3389/fnins.2022.881598
[18] S. Panagiotou, R. Miedema, H. Sidiropoulos, G. Smaragdos, C. Strydis, and D. Soudris, "A novel simulator for extended Hodgkin–Huxley neural networks," in *Proc. IEEE 20th Int. Conf. Bioinf. Bioeng. (BIBE)*, Oct. 2020, pp. 395–402, doi: 10.1109/BIBE50027.2020.00071.
[19] A. Hodgkin and A. Huxley, "A quantitative description of membrane current and its application to conduction and excitation in nerve," *Bull. Math. Biol.*, vol. 52, no. 4, pp. 500–544, Aug. 1952. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/pmid/12991237/?tool=EBI
[20] J. R. De Gruijl, P. Bazzigaluppi, M. T. G. de Jeu, and C. I. De Zeeuw, "Climbing fiber burst size and olivary sub-threshold oscillations in a network setting," *PLoS Comput. Biol.*, vol. 8, no. 12, Dec. 2012, Art. no. e1002814.
[21] N. Schweighofer, E. J. Lang, and M. Kawato, "Role of the olivo-cerebellar complex in motor learning and control," *Frontiers Neural Circuits*, vol. 7, p. 94, May 2013. [Online]. Available: https://www.frontiersin.org/article/10.3389/fncir.2013.00094
[22] M. V. Mascagni, "Numerical methods for neuronal modeling," *Methods Neuronal Model.*, vol. 2, pp. 439–484, Aug. 1989.
[23] C. Börgers and A. R. Nectow, "Exponential time differencing for Hodgkin–Huxley-like ODEs," *SIAM J. Scientific Comput.*, vol. 35, no. 3, pp. B623–B643, Jan. 2013.
[24] P. Gleeson, S. Crook, R. C. Cannon, M. L. Hines, G. O. Billings, M. Farinella, T. M. Morse, A. P. Davison, S. Ray, U. S. Bhalla, S. R. Barnes, Y. D. Dimitrova, and R. A. Silver, "NeuroML: A language for describing data driven models of neurons and networks with a high degree of biological detail," *PLoS Comput. Biol.*, vol. 6, no. 6, Jun. 2010, Art. no. e1000815.
[25] M. A. V. D. Vlag, G. Smaragdos, Z. Al-Ars, and C. Strydis, "Exploring complex brain-simulation workloads on multi-GPU deployments," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 4, pp. 1–25, Dec. 2019, doi: 10.1145/3371235.
[26] J. Jordan, T. Ippen, M. Helias, I. Kitayama, M. Sato, J. Igarashi, M. Diesmann, and S. Kunkel, "Extremely scalable spiking neuronal network simulation code: From laptops to exascale computers," *Frontiers Neuroinform.*, vol. 12, p. 2, Feb. 2018.
[27] G. Florimbi, E. Torti, S. Masoli, E. D'Angelo, G. Danese, and F. Leporati, "The human brain project: Parallel technologies for biologically accurate simulation of granule cells," *Microprocessors Microsyst.*, vol. 47, pp. 303–313, Nov. 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0141933116300515
[28] G. Chatzikonstantis, H. Sidiropoulos, C. Strydis, M. Negrello, G. Smaragdos, C. I. De Zeeuw, and D. J. Soudris, "Multinode implementation of an extended Hodgkin–Huxley simulator," *Neurocomputing*, vol. 329, pp. 370–383, Feb. 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231218312906

MAX C. W. ENGELEN received the M.Sc. degree in computer engineering from Delft University of Technology, in 2021. He is currently employed with the Maxeler IoT Laboratories. He is also employed as a Research Analyst with the Neuroscience Department, Erasmus Medical Center, The Netherlands.

RIVER BETTING received the B.Sc. degree in mechanical engineering and the M.Sc. degree in computer engineering from Delft University of Technology, in 2015 and 2018, respectively, and the B.A. degree in history and the M.A. degree in Russian and Eurasian studies from Leiden University. They are currently a Research Analyst with the Neuroscience Department, Erasmus Medical Center. Their current research interests include artificial intelligence, computer vision, and algorithm development.

CHRISTOS STRYDIS (Senior Member, IEEE) is studied Electronics and Computer Engineering from the Technical University of Crete, Greece. He received the bachelor's diploma (magna cum laude), in 2003, the M.Sc. degree (magna cum laude) in computer engineering from Delft University of Technology, The Netherlands, in 2005, with a minor in biomedical engineering, and the Ph.D. degree in computer engineering from Delft University of Technology, in 2011. He holds a Joint Associate-Professor with the Neuroscience Department, Erasmus Medical Center, Rotterdam (NL), and with the Quantum and Computer Engineering Department, Delft University of Technology. He funding from the ICT Delft Research Centre (DRC-ICT) and Google Inc. He has supervised multiple B.Sc., M.Sc., and Ph.D. students, and teaches various bachelor- and master-level courses. His current research interests span the fields of biologically plausible brain simulations, next-generation neural implants, and ultrasound-based brain imaging. He has acted as program-committee member in various international conferences. He has also peer-reviewed for as well as published manuscripts in well-known international conferences and journals, and delivered invited talks in various venues. He has been awarded many national- and EU-level research projects.