IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received May 27, 2020, accepted June 20, 2020, date of publication July 3, 2020, date of current version July 16, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3007019

flexHH: A Flexible Hardware Library for Hodgkin-Huxley-Based Neural Simulations

RENE MIEDEMA[1], GEORGIOS SMARAGDOS[1], (Member, IEEE), MARIO NEGRELLO[1], ZAID AL-ARS[2], (Member, IEEE), MATTHIAS MÖLLER[3], AND CHRISTOS STRYDIS[1], (Senior Member, IEEE)

[1]Neuroscience Department, Erasmus Medical Center, 3000 Rotterdam, The Netherlands
[2]Quantum and Computer Engineering Department, Delft University of Technology, 2628 Delft, The Netherlands
[3]Applied Mathematics Department, Delft University of Technology, 2628 Delft, The Netherlands

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

This work was supported by the European Union Horizon 2020 Projects, in part by VINEYARD under Grant 687628, and in part by EuroEXA under Grant 754337.

ABSTRACT The Hodgkin-Huxley (HH) neuron is one of the most biophysically-meaningful models used in computational neuroscience today. Ironically, the model's high experimental value is offset by its disproportional computational complexity. To such an extent that neuroscientists have either resorted to simpler models, losing precious neuron detail, or to using high-performance computing systems, to gain acceleration, for complex models. However, multicore/multinode CPU-based systems have proven too slow while FPGA-based ones have proven too time-consuming to (re)deploy to. Clearly, a solution that bridges user friendliness and high speedups is necessary. This paper presents flexHH, a flexible FPGA library implementing five popular, highly parameterizable variants of the HH neuron model. flexHH is the first crucial step towards making FPGA-based simulations of compute-intensive neural models available to neuroscientists without the debilitating penalty of re-engineering and re-synthesis. Through flexHH, the user can instantiate custom models and immediately take advantage of the acceleration without the mediation of an engineer, which has proven to be a major inhibitor to full adoption of FPGAs in neuroscience labs. In terms of performance, flexHH achieves speedups between 8×–20× compared to sequential-C implementations, while only a small drop in real-time capabilities is observed when compared to hardcoded FPGA-based versions of the models tested.

INDEX TERMS Hodgkin-Huxley, data-flow computing, neural network.

I. INTRODUCTION

The field of computational neuroscience focuses on explaining and predicting experimental neuroscientific data. A method to understand how biological-brain systems organize and process information.

Neuroscientists use hypotheses formulated using in-silico experimentation that can subsequently be validated by more informed and guided biological tests, to answer such questions. Among the most popular realistic models for such purposes are Spiking-Neural-Network (SNN) models [1], [2] of the Hodgkin-Huxley (HH) variety [3] (other formalisms exist as well, such as Izhikevich and Integrate-and-Fire models). The choice of SNN model depends on the studied problem [4]. When exploring the electrochemical properties that faithfully reproduce neuronal response, a biophysically-meaningful neuron model is required, such as the HH model. These models capture closely the electrochemical behavior that produces the neuron activity by modeling the various cell-membrane ion channels. The ultra-high computational complexity of the standard HH model and its extensions is what makes such models challenging to simulate using traditional computing approaches.

Owing to the high computational requirements of HH models, neuroscientists have explored FPGA solutions in the past. The community has embraced the speed gains of FPGAs only to soon become frustrated by the steep programming curve and the design rigidity they also entail. Hardcoded models been especially limiting given the ''trial-and-error'' mentality of researchers who are constantly modifying their models

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

trying to fit them to ever-changing biological data. This moving target, combined with recurring coding efforts, have led FPGA supporters to grudgingly move to more flexible and programming-friendly platforms, namely CPUs and GPUs, to be discussed in Section II.

However, we are convinced that fast *and* flexible models using FPGAs are possible: First off, a careful combing of the online ModelDB repository [5] and of literature at large reveals that the vast majority of available models comprises a few common systems of Ordinary Differential Equations (ODEs) which are highly parallelizable as workloads (subject to the solver used, of course); as a matter of fact, neuron models often are *dataflow* workloads. Secondly, modern FPGAs have made large strides in High-Level Synthesis (HLS) improving programmability. For the special case of dataflow applications in particular, there is a unique offering by Maxeler Technologies: a Dataflow-Engine (DFE). DFEs are FPGAs programmed through the MaxJ programming language; essentially Java generating HDL dataflow code [6]. DFEs capitalize on the simplified control requirements of dataflow applications and allow for at least an order of magnitude better-performing DFE applications compared to established HLS solutions due to utilizing the entirety of the FPGA resources for implementing useful functional units rather than complex control flow.

The above observations provide a strong hint towards designing one or a few *reusable DFE kernels* that can be used to efficiently simulate more than a single neuron-model configuration. Besides, FPGAs have outright benefits compared to other acceleration platforms like GPUs, such as low-latency on-chip memory (BRAM) which minimizes data moves making low-latency, real-time simulations possible [7]. Therefore, in this paper we develop, validate against a reference design and extensively evaluate *flexHH*, a unique, flexible, synthesis-free, DFE-based library of five HH-model variants, which are among the most compute-intensive neuron models in the field.

This work builds upon BrainFrame [7], a Cloud-based High-Performance Computing (HPC) framework for accelerating computational-neuroscience experiments by employing multiple acceleration technologies (Intel Xeon-Phi CPU, Nvidia GPU, Maxeler DFE). By controlling a mix of heterogeneous accelerators, BrainFrame assigns the best-suited accelerator to each new simulation request by matching optimal accelerator to particular model properties. BrainFrame has been validated with hardcoded models so far but, for the Cloud service to be useful in practice, it must allow neuroscientists to develop their own models using flexible model libraries, such as flexHH. The contributions of this work are as follows:

- flexHH, a synthesis-free, scalable and high-performing FPGA library of parameterizable and NeuroML-compliant [8] HH models. It is available online.[1]
- A detailed performance and power analysis of the flexHH kernels and their potential.
- All in all, practical proof that modern FPGAs can indeed facilitate fast and versatile neuro-simulations for realistic experimentation.

The paper structure is as follows: Section II discusses related work on high-performance SNN simulators. Section III provides background information about Hodgkin-Huxley modeling. In Section IV, the implementation of the new flexHH library is detailed. Validation of flexHH, evaluation of its performance and its power usage, and a comparison of its efficiency are discussed in Section V. Finally, Section VI concludes this paper.

II. RELATED WORK

In literature, various efforts have been made to provide parameterizable and high-performing neuron models. A major challenge resides in accelerating this particular type of HH models, while still keeping modeling flexibility high. In this context, we present a concise overview of the current art.

On the GPU front, Beyeler *et al.* [9] have developed a large-scale SNN simulator based on C/C++, called CarlSim. Currently in its 4th version, the simulator provides a variety of features alongside GPU acceleration support. With suitable hardware, the simulator can support networks of hundreds of thousands of neurons. However, at the moment it does not support complex models, such as conductance models like HH models. A more complete GPU/CPU accelerated simulator is NCS6 [10]. It provides similar features to CarlSim but also includes HH modeling. Even though HH models are supported, the performance of the simulator when running HH models is unclear and no extra updates have been reported in recent years.

On the FPGA front, there are a few simulator proposals that are quite notable, as well. SNAVA, by Sripad *et al.* [11], is a multi-FPGA SNN simulator focused on large-scale neuron simulations. The simulator supports a variety of models using 16-bit fixed-point arithmetic operations. The architecture, though, does not support more complex modeling such as HH models. The most promising solution, both in terms of usability and computational capabilities, was proposed by Cheung *et al.* [12] with NeuroFlow. In this work, the researchers integrated PyNN to their multi-DFE-based simulator. NeuroFlow also provides a very complete library of IPs in the back-end. The performance and efficiency analysis is presented for a single use case of a generally simpler model (Izhikevich) with relatively low connectivity density (about 10%), showing impressive results. The behavior and performance of the system for the rest of the supported features, on the other hand, is not self-evident and is expected to be significantly reduced, especially for more demanding modeling [7]. Furthermore, the proven applicability of NeuroFlow in the case of HH models is very limited due to relying on event-driven simulations only, which can often be

[1] https://gitlab.com/neurocomputing-lab/Inferior_OliveEMC/flexhh.

impractical for HH models. From the above, it becomes apparent that flexHH is an essential approach for successfully tackling biophysically-meaningful models at high performances and high modeling flexibility with virtually no programming effort. To the best of our knowledge, no such work has been published before.

III. THE HODGKIN-HUXLEY NEURAL MODEL

The HH neural networks described in this paper are represented by a set of ODEs. This means that a suitable ODE solver is needed to 'solve' (i.e. simulate) these models. A typical ODE solver is the forward-Euler method which is described by (1), where u^n is a vector holding the (approximated) state variables at step n, Δt is the time-step size, and f a vector describing the derivatives of each state variable:

$$u^{n+1} = u^n + \Delta t \cdot f(u^n) \tag{1}$$

flexHH has been initially built to support the forward-Euler method as a straightforward method of tackling the typically stiff HH equations. However, the flexHH library has been designed to be modular so as to support different types of solvers in the future, as the need arises. This modularity is underpinned by the high-level language used to code the flexHH library, which is Maxeler Java (MaxJ), to be discussed in the next section.

To provide a general HH-model description required for our accelerated-library kernels, the models implemented are formulated based on standards compatible with a NeuroML description [8]. NeuroML is a popular format for representing computational neural models with a hierarchy similar to biological neurons and has been largely adopted by the neuroscientific community. In this way, flexHH can be seamlessly accepted and used by its targeted audience. From top to bottom, the *hierarchy* is as follows: network, cell, compartment, ion channel (comprising one or more ion-channel gates). A schematic overview is given in Fig. 1. The classical HH model is a single-cell model consisting of a single compartment with three ion gates which can be described by the three standard HH equations [3].

For defining the flexHH-library properties and also as a heavily researched model in and of itself, we checked here – without loss of generality – against the Inferior Olive (IO)-model [13], previously used to also evaluate the BrainFrame proof-of-concept system [7]. The IO-model captures many modern extensions to the basic HH model, then called extended Hodgkin-Huxley (eHH). These extensions are: intercellular connections (gap junctions), multiple cell compartments per cell (here: three; Dendrite, Soma, Axon), and additionally to the standard gate equations, the IO model contains additional, user-defined ion gates.

A DFE library supporting the required features in a modular and flexible way provides a basis for covering a vast variety of possible eHH models.

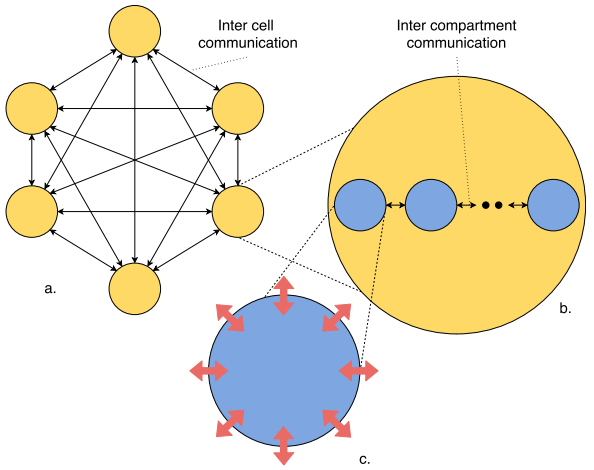

FIGURE 1. Schematic overview of neural network of 6 cells (a). Visible is also a single neuron cell (b), and a single cell compartment with gates in red (c).

TABLE 1. Supported eHH-model features per flexHH kernel.

	Custom ion gates	Gap junctions	Multiple cell compartments
HH	✗	✗	✗
HH+gap	✗	✓	✗
HH+custom	✓	✗	✗
HH+custom+multi	✓	✗	✓
HH fully featured	✓	✓	✓

IV. flexHH-LIBRARY IMPLEMENTATION

A. OVERVIEW

This first version of flexHH targets a single DFE node. Besides the standard HH model, three crucial extensions were added: User-defined ion gates, gap-junction interconnectivity, and support for multi-compartmental cells. Each feature incurs a hardware-resource overhead that is subsequently translated into a performance overhead on the DFE technology. In order to keep performance as high as possible, flexHH has been built to provide *five* different instances (or kernels), each incorporating more or less a superset of features compared to its predecessor. That way, the neuroscientist can opt for using a library instance as closely matched to the problem at hand as possible for achieving the highest speed or the maximum network capacity possible; see Table 1: The simplest flexHH kernel (*HH*) supports the basic HH model. The *HH fully featured (HH+custom+multi+gap)* kernel supports all eHH features and can, for instance, simulate the complete IO model. New model characteristics which cannot be described by the current set of features, such as synaptic plasticity, can be implemented similarly to the ones already implemented. flexHH is designed to the be as extendable as required.

In case of single-compartmental cells, the terms cell and compartment are interchangeable. The ODE systems implemented are represented by state variables comprising membrane potentials of the compartments (V_i) and gate-activation

variables (y_i),[2] where i is the index of the variable. The index can be a combination of multiple integers; e.g. to represent gate h of compartment k of cell j, the index (j, k, h) can be used. Those state variables – which are single-precision, floating-point variables – are updated as described in Algorithm 1.

Algorithm 1 HH-Model Evaluation

```
1:  for 0 ≤ i < N_steps do
2:      for 0 ≤ j < N_cells do
3:          for 0 ≤ k < N_comps,j do
4:              for 0 ≤ h < N_gates,j,k do
5:                  Y_i,j,k,h ← updateY(gateConsts, Y, dt)
6:              end for
7:              V_i,j,k ← updateV(gateConsts, compConsts,
8:                                cellConsts, V, dt)
9:          end for
10:     end for
11: end for
```

For each simulation, the solver is invoked for updating the neural network for a predefined number of steps N_{steps} and with a time step dt. For each gate (in N_{gates}) of each compartment (in N_{comps}) of each cell (in N_{cells}) – across simulation steps –, an `updateY` function is called which iteratively updates the values of the gate-activation variables y_i. For each compartment, a second function `updateV` updates the compartment's membrane-potential value V_i. Before going over the implementation details of Algorithm 1 on a DFE, the dataflow execution model employed in flexHH will be briefly introduced.

B. HH-MODEL DATAFLOW-COMPUTING PARADIGM

In a dataflow application, the traditional control logic is absent since compute dependencies are solved statically, at compile-time. Control reduces to counters that simply advance data through execution units in the datapath. The dataflow paradigm allows, thus, for most FPGA resources to be used for computations instead of control logic. It, further, allows for applications to be implemented in a deeply pipelined fashion leading to a high computational throughput. The performance benefits due to the dataflow paradigm, when compared to the control-flow paradigm, are shown in [14]. Moreover, by programming with the MaxJ toolflow, the programming complexity is significantly reduced in comparison to using low-level (e.g. VHDL) hardware-description languages on traditional FPGA toolflows. Compared to HLS languages (e.g. Vivado C and OpenCL) MaxJ allows for finer control of the generated logic allowing for more efficient use and greater optimization of the implemented design. As a result, the Maxeler toolflow is an excellent programming environment for efficient development. For a more comprehensive coverage of the subject, the interested reader is referred to [15], [16].

To provide a flexible yet efficient as possible implementation, the following strategies were followed for flexHH:

1) GROWING-COMPLEXITY KERNELS

We provided a variety of kernels each supporting a growing set of features. Kernels with less features consume less hardware, thus allowing for larger or faster neural-network simulations. Thus, the flexHH user is offered a performance-to-feature trade-off, matching their specific needs.

2) AGGRESSIVE LOOP UNROLLING

Even though eHH models are generally compatible with dataflow execution, model features such as gap junctions disrupt the pure dataflow paradigm to varying degrees. This can lead to wasted cycles where compute elements are busy crunching invalid operations (when input data is not ready), while waiting for operations in the critical path to finish. This is an inherent consequence of the paradigm. To mitigate this effect, fully unrolling loops in hardware for the computations in the critical path (or to the extend possible by available FPGA area) was required. This led to higher area usage per implemented function but also to increased throughput. Additionally, operations with no dependency on prior values were overlapped with the critical-path operations when possible to ensure even more efficient use of the available area.

3) LOW COMMUNICATION OVERHEADS

The lack of (complex) control flow in the dataflow-computing paradigm is offset by an urgent need to feed the many dataflow functional elements on the chip with large amounts of input data so as to avoid DFE stalling. As the data between the host CPU and the DFE card is channeled over (the slow, by comparison) PCIe channel, this may introduce communication overheads. These overheads were avoided by storing data locally on the DFE (either on the on-chip BRAM or on the on-board DRAM) during the simulation and, thus, minimizing communication between host and DFE.

C. HH-MODEL EQUATION GENERALIZATION

The dataflow paradigm employed over the reconfigurable substrate is one crucial aspect of flexHH. Another is the smart way of generalizing the equations used in key functions `updateY` and `updateV`. A successful generalization will permit harnessing the maximum performance of a dataflow-driven FPGA chip while not sacrificing the crucial modeling power of flexHH.

To implement generalized (thus, reusable) kernels on the DFE (in hardware), the functions themselves are required to be generalized, by exposing the parameters used for the simulation of eHH models as user-defined input arguments. Otherwise, a new time-consuming synthesis cycle would be required each time a user changes the function parameters. Therefore, each variable, parameter and constant, other than the state variables, can be set by the user on the CPU-host.

[2] An activation variable defines the proportion of ion gates in the total population which are open.

It is interesting to note that the adopted NeuroML standardisation seriously helps guiding the generalization process. The description of the generalized eHH functions is as follows: The derivative of the voltage of a HH compartment i (note that the index can be a combination of multiple integers), is calculated through (2).

$$\frac{\mathrm{d}V_i}{\mathrm{d}t} = \frac{I_{app,i} - I_{channels,i} - I_{leak,i} - I_{mc,i} - I_{gap,i}}{C} \tag{2}$$

where C is the membrane capacitance, $I_{app,i}$ is the applied current to the respective cell or compartment, representing external input to the network/cell, $I_{channels,i}$ is the sum of all ion-channel currents, $I_{leak,i}$ the leakage current, and $I_{mc,i}$ and $I_{gap,i}$ are currents received from the (optionally modeled) inter-compartment connections and from the gap junctions, respectively. The terms $I_{mc,i}$ and/or $I_{gap,i}$ may be omitted if the model instance does not include these features.

$I_{leak,i}$ is always represented by (3), where $g_{leak,i}$ and $V_{leak,i}$ are the conductance and the reverse voltage of the leakage channel, and thus this equation does not require any generalization.

$$I_{leak,i} = g_{leak,i}(V - V_{leak,i}) \tag{3}$$

The current $I_{app,i}$ can be represented by any *arbitrary* mathematical function. However, the support for such flexibility in hardware is impossible. Therefore, it was decided to only support pulse functions for the evoked input, guided by the NeuroML standard (represented in NeuroML by a `pulseGenerator`). The pulse function has three parameters: a start step ($step_{start}$), an end step ($step_{end}$), and an amplitude (A), as can be seen in (4).

$$I_{app,i}(step) = \begin{cases} A, & \text{if } step_{start} \leq step < step_{end} \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

To calculate $I_{channels,i}$, (5) is implemented, where $N_{channels,i}$ is the number of channels in compartment i, $g_{c,j}$ and $V_{c,j}$ are the conductance and the reverse voltage of each channel j, respectively. The variable $yProd$ represents the product of the gate-activation variables (y_k) in a channel, calculated with (6), which defines the probability that all the activation variables have produced an open channel. As there can be multiple gates of the same type in a channel, p_k is an integer variable which represents the number of gates in that channel.

$$\begin{aligned} I_{channels,i} &= \sum_{j=0}^{N_{channels,i}-1} I_{channel,j} \\ &= \sum_{j=0}^{N_{channels}-1} g_{c,j}(V - V_{c,j}) yProd_j \end{aligned} \tag{5}$$

$$yProd_j = \prod_{k=0}^{M_{gates[j]}-1} y_k^{p_k} \tag{6}$$

After generalizing the HH aspects, we proceed to generalize the extended HH (eHH) features, as follows:

1) MULTIPLE CELL COMPARTMENTS

When multiple compartments are supported, currents are exchanged between two adjacent compartments in the same cell. The structure of the connections between the compartments is a tree whose morphology can differ per network model. To generate efficient hardware, in this version of flexHH, we simplify the tree structure by only assuming sequential connections among compartments. Non-sequential connections apply only to dendro-dendritic networks or when dendrites have maximally two branches, thus it is an acceptable simplification still covering a large fraction of neuroscientific simulations. To calculate the current flowing between compartments, we use a similar equation as in the seminal work [17]. The final equation used in our implementation is shown in (7):

$$I_{mc,i} = g_{int} \sum_{j=0}^{N_{comps,i}-1} \frac{V_i - V_j}{p_{i,j}} \tag{7}$$

where $N_{comps,i}$ is the number of other compartments compartment i is connected to in the cell, g_{int} is the internal conductance of the whole cell, $p_{i,j}$ is the surface ratio of the two compartments i and j and V_i, V_j are their respective membrane potentials. When this model extension is desired, current $I_{mc,i}$ is added to the sum of currents for calculating dV_i/dt, as shown in (2).

2) GAP JUNCTIONS

Gap junctions are electrical connections between cells. We have implemented a generalized version of the gap-junction function introduced in [18] by calculating $I_{gap,i}$ through (8), where c_0, c_1, and c_2 are constants and are identical for every connection, and $w_{i,j}$ is the variable-connection weight between compartments i and j, where j belongs to a different cell than i. This way, we reduce data-storage and memory-I/O requirements significantly, while maintaining model flexibility. flexHH supports both the use of standard single variables or whole functions of the form of (8) for representing the gap-junction current.

$$I_{gap,i} = \sum_{j=0}^{N_{cells}-1} (w_{i,j}(c_0 \exp(c_1 \cdot V_{i,j}^2) + c_2)V_{i,j}) \tag{8}$$

3) ION GATES

Per compartment i, each gate-activation variable (y_j) is calculated via (9) or (10). In case of models without support for custom ion gates, the gate-activation variables are always provided via (9). If custom ion gates are supported, (10) can also be used, depending on the model needs. The transition rates α_j and β_j or the target value inf_j and the time constant τ_j are calculated either via (11) (which mirrors the functions of gate equations in NeuroML) if custom ion gates are not supported, or via (12) if custom ion gates are supported. The input of both equations consists of the membrane voltage V_i of compartment i, multiple constants – 3 in case of no custom ion gates (k_0, k_1, k_2) and 9 ($k_0, k_1, .., k_8$) in case of custom ion

gates – and an extra variable f_t (to select a function branch), as parameters.

$$\frac{dy_j}{dt} = (1 - y_j) \cdot \alpha_j - y_j \cdot \beta_j, \tag{9}$$

$$\frac{dy_j}{dt} = \frac{inf_j - y_j}{tau_j} \tag{10}$$

$$f(V_i, k_0, k_1, k_2, f_t) = \begin{cases} \frac{k_0 \cdot (k_1 - V_i)}{e^{(k_1 - V_i) \cdot k_2} - 1}, & \text{if } f_t = 0 \\ k_0 \cdot e^{(k_1 - V_i) \cdot k_2}, & \text{if } f_t = 1 \\ \frac{1}{e^{(k_1 - V_i) \cdot k_2} + 1}, & \text{if } f_t = 2 \end{cases} \tag{11}$$

$$fCustom(V_i, k_0, k_1, .., k_8, f_t) = \begin{cases} \frac{k_5(k_1 - V_i)}{fExp(k_0, (k_1 - V)\, k_2, k_3)} + k_8 & \text{if } f_t = 0 \\ \frac{k_8}{[fExp(k_0, k_2(k_1 - V_i), k_3) \\ +fExp(k_0, k_5(k_6 - V_i), k_7)]} & \text{if } f_t = 1 \\ \frac{fExp(k_0, (k_1 - V_i)k_2, k_3)}{fExp(k_4, (k_6 - V_i)k_5, k_7)} + k_8 & \text{if } f_t = 2 \\ min(k_0\, v, k_1) & \text{if } f_t = 3 \end{cases}$$

where :

$$fExp(scale, x, offset) = scale \cdot exp(x) + offset \tag{12}$$

D. HARDWARE IMPLEMENTATION

Having defined the generalized functions, we can finally implement them efficiently as DFE kernels, following the previously discussed strategies, where the parameters can change without the need for resynthesising. For the implementation, firstly, the generalized equations discussed in Section IV-C are coded as computational blocks using the MaxCompiler high-level-synthesis functions. This leads to the architecture shown in Fig. 2. The system architecture of the DFE consists of a host CPU which establishes input/output data streams to the DFE board and the on-board memory. The DFE comprises (FPGA) on-chip, fast BRAM memory and on-board, slower DRAM memory alongside the reconfigurable chip. The input data consists of the initial values of the state variables and the parameters of the generalized functions (the `updateV` and `updateY` parameters). Consequently, the behaviour of the model can be modified on the host CPU without resynthesizing. The output consists of state-variable values at each time step. The amount of input/output data required to be transferred is too large to fit into the FPGA BRAMs in the general case and, therefore, those parameters are placed into the on-board DRAM. The data transfers between board and host only occur before and after a simulation run to prevent any data-transfer bottleneck. Furthermore, to adhere to the dataflow paradigm, the parameters of the generalized functions (the `updateV` and `updateY` parameters) are streamed to the DFE kernel, from the on-board DRAM, at appropriate times during the simulation. In contrast, the state variables are updated frequently and, therefore, are stored in the BRAMs to reduce transfer latencies.

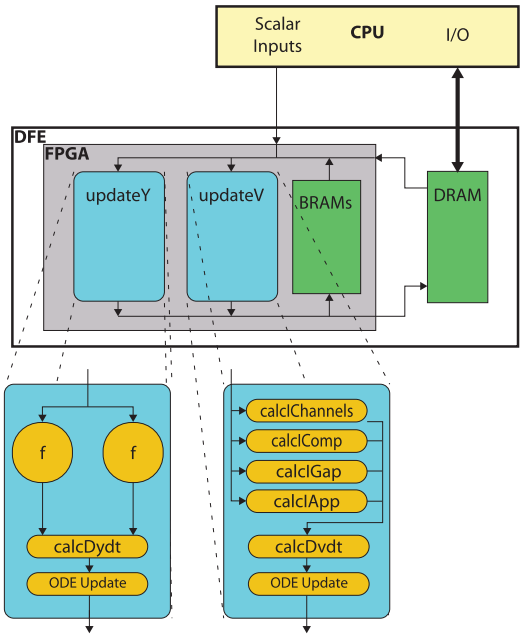

FIGURE 2. Schematic overview of implementation on the DFE.

The implementation of the DFE kernel, as shown in Fig. 2, consists of computational blocks implementing the `updateY` and `updateV` functions. Within `updateY` there are two computational blocks of `f`, each having the functionality of (11) or its more complex variant (12), depending on whether custom ion gates are supported. Additionally, in `calcDydt` the derivative dy_j/dt is calculated with either (9) or (10). Within `updateV`, the currents $I_{channels,i}$, $I_{app,i}$, $I_{mc,i}$, and $I_{gap,i}$ are calculated via the functions `calcIChannels`, `calcIApp`, `calcIComp`, and `calcIGap`, respectively. Depending on which of the five kernels is instantiated, `calcIComp` and/or `calcIGap` may be omitted. With the use of these functions, dV_i/dt is calculated in `calcDvdt`. The final step in both `updateY` and `updateV` is to update the state variables (y_j and V_i) with the forward-Euler method in the `ODE Update` blocks.

Next, we present the pseudocode and detail the implementation of each functional block shown in Fig. 2. To retain high hardware efficiency and avoid idle cycles, a number of issues had to be addressed and are also discussed next. We like to remind the reader that, in the following, the variables are processed in a streaming fashion and that the variable indices can also be a combination of multiple integers.

1) VARIABLE INNER LOOP

The iterations of the computation loops of Algorithm 1 are used to control execution flow, as enforced by the dataflow programming paradigm, implemented with hardware counters. The maximum value of the innermost loop is received as input stream. Therefore, a buffer is used to hide the input latency of the stream, as inspired by the *dfesnippets*

library [19], to allow for an efficient dataflow implementation. Consequently, the number of total operation ticks is increased by 4 (the input latency), however, this is a negligible increase compared to the overall operation ticks and the benefit of buffering.

2) ION GATES

To update the gate variables (y_j) without custom ion gates, the derivative $\mathrm{d}y_j/\mathrm{d}t$ from (9) is used. For this equation, the calculations of both α_i and β_i are done using (11). For the implementation of (11), the amount of divisions and exponentiations is minimized, as Algorithm 2 shows, to reduce the hardware usage of this function. This optimization is done so that – independent of the Maxeler tools – the hardware-usage is minimized. Both α_i and β_i employ this function, consequently, this algorithm is generated twice for the implementation.

Algorithm 2 Pseudocode of `f`, a Generalized Function to Calculate α and β

function f(V_i, k_0, k_1, k_2, f_t)
  $V_{diff} \leftarrow k_1 - V_i$ ▷ Define variable which is used multiple times
  if $f_t == 0$ **then**
    $num \leftarrow k_0 \times V_{diff}$
    $c \leftarrow -1$
  else if $f_t == 1$ **then**
    $num \leftarrow k_0$
    $c \leftarrow 0$
  else if $f_t == 2$ **then**
    $num \leftarrow 1$
    $c \leftarrow 1$
  end if
  $denum \leftarrow \exp(V_{diff} \times k_2) + c$
  return $\frac{num}{denum}$
end function

For the kernels which support custom ion gates, instead of Algorithm 2, Algorithm 3 is implemented. From this function, two instances are created to either calculate α_j and β_j or inf_j and τ_j, depending on whether y_j is updated with equation (9) or (10), respectively. The final choice of equation for updating y_j is, then, selected by an extra if-else statement (which translates to a hardware multiplexer) for which the variable f_t is automatically used as selector.

3) calcIApp

The function `calcIApp` implements (4), as shown in Algorithm 4.

4) calcIChannels

To calculate $I_{channels}$, $yProd$ is required which is calculated through an exponentiation. In hardware, an exponentiation cannot be generated with a variable exponent of power p_j. Therefore, we multiply the variable y_j with itself (to be able to support the IO model powers up to and including $p_j = 4$ are supported), combined with a multiplexer to select either y_j multiplied with itself or the unmodified y_j. This is shown in the pseudocode of Algorithm 5. Furthermore, following the dataflow paradigm, the loop used for the sum of (5) is completely unrolled in hardware preventing any stalls and, thus, wasted operations cycles. The pseudocode to calculate $I_{channels}$ is shown in Algorithm 6.

Algorithm 3 Pseudocode for `fCustom`, a Generalized Function to Calculate α, β, inf and τ for Kernels Which Support Custom Ion Gates, and the Function `fExp` Which Is Used in `fCustom`

function fCustom($V, k_0, k_1, .., k_8, f_t$)
  $V_{diff} \leftarrow k_1 - V$
  $V_{diff\,2} \leftarrow k_6 - V$
  if $(f_t\ AND\ 11_b) == 0$ **then** ▷ Use mask on f_t for selector
    $z_{12} \leftarrow V_{diff}$
  else
    $z_{12} \leftarrow V_{diff\,2}$
  end if
  $z_1 \leftarrow k_2 \cdot V_{diff}$
  $z_2 \leftarrow k_5 \cdot z_{12}$
  $exp1 \leftarrow fExp(k_0, z1, k_3)$
  $exp2 \leftarrow fExp(k_4, z2, k_7)$
  if $f_t == 0$ **then**
    $num \leftarrow z2$
    $denum \leftarrow exp1$
  else if $f_t == 1$ **then**
    $num \leftarrow k_8$
    $denum \leftarrow exp1 + exp2$
  else if $f_t == 2$ **then**
    $num \leftarrow exp1$
    $denum \leftarrow exp2$
  else if $f_t == 3$ **then** ▷ Unused variables
    $num \leftarrow 0$
    $denum \leftarrow V_{Diff}$
  end if
  $y \leftarrow \frac{num}{denum}$

  if $f_t != 1$ **then** ▷ Function branches 0 and 2 require extra addition
    $y \leftarrow y + k_8$
  end if
  $y_1 \leftarrow min(z1, k_0)$
  if $f_t == 3$ **then** ▷ Use min function instead of division in case of function branch 3
    $y \leftarrow y_1$
  end if
  return y
end function

function fExp($scale, x, offset$)
  return $scale \times \exp(x) + offset$
end function

Algorithm 4 Pseudocode of `calcIApp`

```
function calcIApp(step_start, step_end, A, step)
    if (step ≥ step_start) ∧ (step < step_end) then
        I_app ← A
    else
        I_app ← 0
    end if
return I_app
end function
```

Algorithm 5 Pseudocode for `calcYProd`. The `stream.offset` Function Selects a Previous Value of the Stream

```
function calcYProd(y, p, gGate)
    yProd ← y
    if p > 1 then
        yProd ← yProd × y
    end if
    if p > 2 then
        yProd ← yProd × y
    end if
    if p > 3 then
        yProd ← yProd × y
    end if
    gOld ← stream.offset(gGate, −1)
    if gOld == 0 then  ▷ Check if channel consists out of multiple gates
        yProdOld ← stream.offset(yProd, −1)
        yProd ← yProd × yProdOld
    end if
    return yProd
end function
```

Algorithm 6 Pseudocode for `calcIChannels`. The `stream.offset` Function Selects a Previous Value of the Stream

```
function calcIChannels(N_gates, V_c, yProd, g_c, V, N_gates,max)
    iChannels ← 0
    for 0 ≤ i ≤ N_gates,max pardo ▷ Unrolled in hardware
        G_offset ← stream.offset(g_c, −i)
        V_c,offset ← stream.offset(V_c, −i)
        YProd_offset ← stream.offset(yProd, −i)
        iChannel ← YProd_offset × G_offset(V − V_c,offset)
        if i < N_gates then
            iChannels ← iChannels + iChannel
        else
            iChannels ← iChannels
        end if
    end for
return iChannels
end function
```

5) calcIComp

Because of the sequential structure between multiple compartments, compartment k will only receive currents (which is calculated by (7)) from compartments $k-1$ and $k+1$, when $k-1$ and $k+1$ are within the limits of the cell. Because of the supported structure, for the outgoing current to other compartments of a single compartment $I_{comp,i}$, there are three positions a compartment can be in:

- *The starting position*
  In this case, the compartment only exchanges current with the compartment next in line as there is no compartment before. This results in (13) for the calculation of the current:

$$I_{comp,i} = \frac{V_i - V_{i+1}}{1 - p_{i,i+1}} g_{int} \tag{13}$$

- *In between other compartments*
  When a compartment i is in between other compartments it means that compartment i is connect to two other compartments. Therefore, it exchanges current with both neighbouring compartments. This results in (14) for the calculation of the current:

$$I_{comp,i} = \left(\frac{V_i - V_{i+1}}{1 - p_{i,i+1}} + \frac{V_i - V_{i-1}}{p_{i-1,i}}\right) g_{int} \tag{14}$$

- *The ending position*
  In this case the compartment only exchanges current with the compartment which lays before in the line. This results in (15):

$$I_{comp,i} = \frac{V_i - V_{i-1}}{p_{i-1,i}} g_{int} \tag{15}$$

As follows from (13) to (15), (14) (the current when compartment i is between other compartments) is the sum of (13) (the current at the starting position) and (15) (the current at the ending position). Consequently, (13) is stored in $I_{comp,next}$ and (15) is stored in $I_{comp,prev}$ and based on the position one, of these currents or the sum of these current is chosen for $I_{comp,i}$. Additionally, a current of zero could be chosen which will allow single-compartmental cells in the network. The implementation of `calcIChannels` is shown in Algorithm 7.

6) calcIGap

The calculation of $I_{gap,i}$ is calculated through (8). As this function reveals, for each cell a summation of N_{cells} is required. The summation is impossible to completely unroll in hardware due to the limited hardware resources. Therefore, the summation can be implemented in two ways: Either the summation per cell can be done continuously, or the summations of different cells can be done alternatingly. In the first way, the gap-junction current of one cell ($I_{gap,i}$) is calculated before the summation of the next cell ($I_{gap,i+1}$) is calculated. In this way, the latency of one addition operation is required to be one, to prevent injecting bubbles in the pipeline. However, the latency of an addition with floating-point variables on the

Algorithm 7 Pseudocode of `calcIComp`

```
1: function calcIComp(i, N_comps, Vs, ps, g_int)
2:     iCompNext ← (V_i − V_{i+1}) g_int / (1 − p_{i,i+1})
3:     iCompPrev ← (V_i − V_{i−1}) g_int / p_{i−1,i}
4:     iCompAll ← iCompNext + iCompPrev
5:     if N_comps == 1 then
6:         iComp ← 0
7:     else if i == 0 then
8:         iComp ← iCompNext
9:     else if i == (N_comps − 1) then
10:        iComp ← iCompPrev
11:    else
12:        iComp ← iCompAll
13:    end if
14: return iComp
15: end function
```

Algorithm 8 Pseudocode of `calcIGap`

```
1: procedure calcIGap(N_cells, iGapMem, vMem, cs)
2:     for i ∈ {0, uf, 2 uf, ..., N_cells} do
3:         for 0 ≤ j < N_cells do
4:             if i == 0 then
5:                 iGapOld ← 0
6:             else
7:                 iGapOld ← iGapMem.read(j)        ▷ Read BRAMs
8:             end if
9:             V_own ← vMem.read(j)                 ▷ Read BRAMs
10:            for 0 ≤ k < uf pardo                 ▷ Unrolled in hardware
11:                V_other ← vMem.read(i + k)       ▷ Read BRAMs
12:                V_diff ← V_own − V_other
13:                iGapTemp[k] ← w_{j,i+k}(cs[0] exp(cs[1] V_diff^2) + cs[2]) V_diff)
14:            end for
15:            iGapNew ← iGapOld + sum(iGapTemp)
16:            iGapMem.write(j, iGapNew)
17:        end for
18:    end for
19: end procedure
```

targeted DFE requires more than one cycle. Consequently, pipeline bubbles cannot be avoided in this case. On the other hand, by alternating the additions of different summations of the different cells, the updated values are only required after each cell has completed its addition. This prevents the bubbles from occurring, resulting in an efficiently used pipeline. Thus, we opted to the alternating implementation, which can be seen in Algorithm 8. This implementation requires that the intermediate results are stored in memory, which is the reason for using `iGapMem` in Algorithm 8.

7) LOOP UNROLLING

To increase the performance of the kernels, loop unrolling is applied which leads to more operations being executed concurrently. How large this unroll factor *uf* is depends on the problem size; i.e., supported model features and neural-network size. The larger the unroll factor or the problem size, the greater the resource usage. Additionally, an increase in the unroll factor leads to a higher required I/O bandwidth and, therefore, the performance gain of increasing the unroll factor can be limited by the available bandwidth between the on-board DRAM and the FPGA.

The part of overall Algorithm 1 to be unrolled depends on the presence of gap junctions in the model. If there are no gap junctions instantiated, then the performance complexity of both `updateY` and `updateV` is equal to $\Theta(1)$, in which case we opt for unrolling the `updateY` loop (lines 4 – 6 of Algorithm 1). However, when gap junctions are supported, the complexity of `updateV` increases to $\Theta(N_{cells})$. Since it is expected that there are more cells than gates per compartment, in such a model instance, we opt for unrolling part of the `updateV` function (lines 10 – 14 of Algorithm 8), which refers to the gap-junction current calculations of the cell. What is more, parts of `updateV` can be parallelized at a finer granularity that `updateY`, therefore resulting in a overall higher achievable loop-unroll factor in the case of gap junctions.

V. EVALUATION

A. FUNCTIONAL VALIDATION

To guarantee functional correctness of our kernel implementations, we validate flexHH-based models against the established (by the community) versions of the same models.

The basic HH model [3] was initially available to us in NEURON but was rewritten in C for various analysis purposes. This model is used to validate the basic *HH* kernel (see Table 1). The simulation output for the validation of the particular DFE kernel is shown in Fig. 3(a) and the absolute error (the difference between the C and DFE output) can be seen in Fig. 3(b). The error trace shows negligible and bounded error, which is expected due to the difference of the arithmetic implementation between the x86 CPU and the FPGA libraries, and is small enough to not influence the correctness of the output.

To validate the additional features provided by flexHH, the *HH fully featured* kernel is validated against the original IO reference code, which was verified by the developers of the model [20]. The simulation output of the C code is shown in Fig. 3(c) for a single axon (the axon of cell *0*) and the error is shown in Fig. 3(d). Similar results can be drawn here as with the HH model: The error, that can be attributed to the rounding error by moving to the different platform, is negligible and does not affect output correctness.

Although single-cell runs are functionally correct, this is not enough to ensure general stability of the network model. Therefore, network-wide validation runs are also performed: The error map of the full IO network can be seen in Fig. 3(e). The maximum output error of individual cells in the network

does not surpass the order of 10^{-3}, a difference that cannot affect cell behavior.

Additionally to the previously discussed validation, we also compared the phase portraits of different gate variables, for both the HH and IO model, which give a qualitative description of the dynamics [21]. The results, plotted in Fig. 4, show equivalent results. Consequently, we conclude that flexHH produces correct qualitative descriptions of the dynamics of the evaluated models. For a more comprehensive discussion on accuracy, the reader is referred to [22].

B. EXPERIMENTAL SETUP

The next step is to assess the performance (scalability) of flexHH. The accelerated kernels have been implemented on a Maxeler Maia DFE. The Maia specifications are shown in Table 2. The accelerated kernels are compared against sequential-C implementations, optimized with the `O3` flag using the GCC 8.3.0 compiler. The C reference code is executed on an 2.5-GHz Intel Core i7-4870HQ CPU. which is of a similar lineage to the Maia DFE, and provides a simple baseline against which to compare the flexHH kernels among them. Furthermore, for a more qualitative performance comparison, the *HH full featured* kernel is compared against the BrainFrame platform [7]. The BrainFrame platform implements a hardcoded version of the IO model on the same Maia DFE, a Xeon Phi 5110P CPU and a Nvidia Titan X GPU. A more detailed description of the BrainFrame implementations is given in [7].

TABLE 2. Specifications of the hardware used for performance measurements.

Specification	Maia DFE	Intel Core i7-4870H
On-board DRAM (GB)	48	16
DRAM b/w (GB/s)	76.8	25.6
On-chip memory	6 MB (FPGA BRAMs)	256 KB (L2 Cache)
Chip frequency (GHz)	Implem.- specific	2.5
Chip architecture	Stratix V (5SGSD8)	Crystal Well
IC process	65nm	22nm

The main parameters affecting resource usage are $N_{Comps,max}$, $N_{gates,max}$, and uf. Since the maximum values $N_{Comps,max}$, $N_{gates,max}$, and uf are interdependent, there is a performance trade-off between maximally achievable network size and performance of the kernel based on the values of these design parameters. Besides conferring with experts, we also polled 10% of all 660 realistic single-neuron models available in ModelDB [5] at this moment (November 2019). We found that 10 channels per compartment cover 89% of all cases and have, thus, chosen to restrict the maximum number of gates ($N_{gates,max}$) to 10 per compartment, as a reasonable ceiling for modeling custom ion gates. Keeping this constant at compile time bounds the DFE-resource requirements while still allowing for a wide variety of experiments. We have explored (but do not include here for brevity) and derived the most viable pairs of these parameters for each one of the five flexHH kernels, also taking into account the memory I/O-bandwidth restrictions of the DFE. The objective was to increase performance, by taking advantage of as much FPGA area as possible, while still providing support for experiments with network sizes of at least 20K compartments. The kernel configurations that resulted from this exploration are summarized in Table 3. These configurations are used for performance evaluation, discussed next.

TABLE 3. Optimized flexHH-kernel configurations, used for evaluation and the speedup against the CPU.

Model	uf	$N_{comps,max}$	$N_{gates,max}$	Speedup vs. CPU
HH	4	53,248	10	20.05
HH+gap	24	24,576	10	17.82
HH+custom	3	53,248	10	11.18
HH+custom+multi	4	28,672	10	8.63
HH fully featured	16	24,576	10	14.11

The evaluation measurements have been done using simple neuron-model experiments of several-thousand simulation steps. The *HH* and *HH+gap* kernels were tested using the standard HH model. The *HH+custom* kernel test simulates soma compartments from the IO model. Finally, both the *HH fully featured* and *HH+custom+multi* kernel are tested using IO cells, with each cell consisting of three compartments as in the original model description. Kernel times include both compute and on-board DRAM communication latency.

C. PERFORMANCE, POWER AND SCALABILITY

To derive performance speedups compared to the single-threaded version of the kernels, we simulate runs of 23,040 compartments (the reason for using this number will be explained below). Using the same problem size for each kernel gives a good basis for comparing the overheads of the different flexHH features. Additionally, for the *HH fully featured* kernel, that is tested by simulating IO neurons, 23,040 compartments account for 7,680 cells, as a single IO cell consists of 3 compartments. This gives us a basis for comparing the performance of our reusable library (simulating 23.040/3=7,680 cells) with the previously reported hardcoded DFE version of the IO in [7], which maximally supports 7,680 cells.

In Table 3 (right-most column), we can see the *speedup* results of our five DFE instances compared to single-threaded C versions. The observed speedup is between 8× and 20×. It must be noted that the C version already provides significant performance benefits compared to the established NEURON simulation environment, resulting in a cumulative DFE speedup of 1,065× for the simple *HH* kernel compared to NEURON. An interesting observation is that the most complex kernel in the flexHH library actually exhibits higher speedups compared to the simpler *HH+custom+multi* kernel (when also simulating an IO cell). This can be attributed to the instantiation of gap junctions: The higher speedup of the *HH fully featured* kernel is a direct effect of the higher unroll

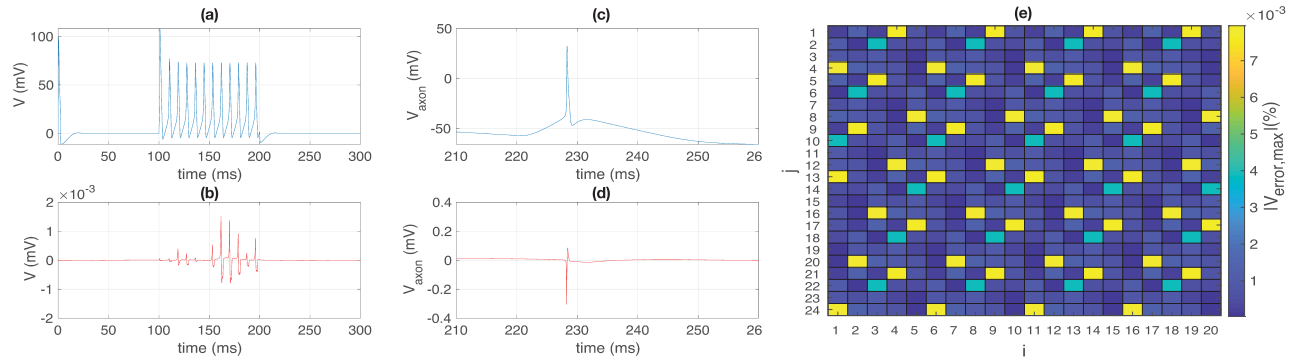

FIGURE 3. (a) Output voltage for a simulation in NEURON of a single HH cell. (b) Error between C and our DFE implementation of a single HH cell. (c) Axonal output voltage for a simulation in C of cell 0 of an IO network (fwd-Euler). (d) Error between C and the *HH fully featured* implementation on the DFE. (e) Maximum absolute error of the axonal voltage per cell of the *HH fully featured* network implementation (i and j are indices of a cell).

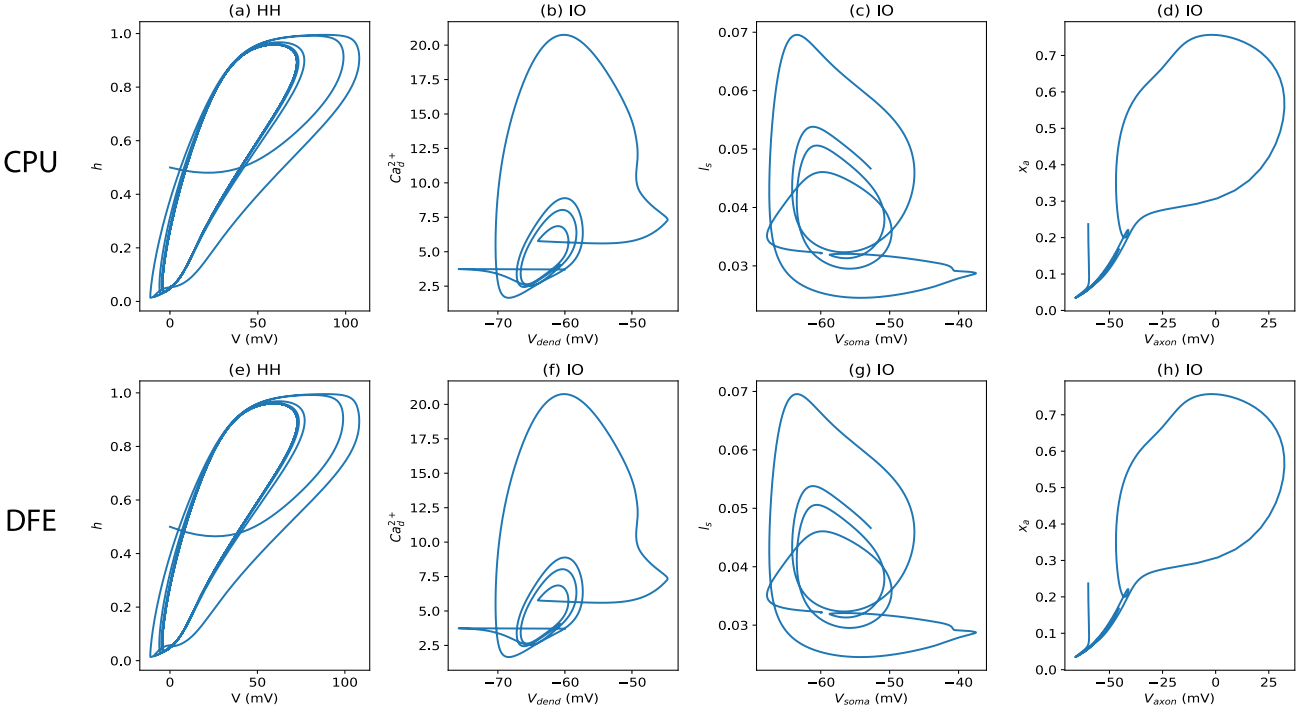

FIGURE 4. Phase portraits for the HH model (a and e) and the IO model (b, c, d, f, g, h) of both the reference, run on a CPU, code (a, b, c, d) and the flexHH library, run on a DFE (e, f, g, h). *a* and *e*: Phase portrait of the gate variable associated with the HH-model potassium channel. *b* and *f*: Phase portrait of the calcium concentration of the IO-cell dendrite. *c* and *g*: Phase portrait of the gate variable associated with the IO-cell soma low-threshold calcium channel. *d* and *h*: Phase portrait of the gate variable associated with the IO-cell axon potassium channel.

factor achievable when gap junctions are present, as discussed in Section IV-D7. Furthermore, as shown in [7], including gap junctions in the model can dominate computational requirements. Consequently, the kernels with gap junctions are less restricted by I/O bandwidth restrictions, compared to the models without gap junctions. However, the presence of the gap junctions severely limits the maximum compartment count.

Fig. 5 compares the performance per simulation step of the IO model, when simulated via the flexHH *HH fully featured* kernel and via three preexisting HPC kernels on the BrainFrame platform: flexHH performs better than all three BrainFrame implementations. Because larger networks require more resources on a FPGA, resulting in a lower unroll factor and thus performance, it must be noted that the Xeon-Phi and GPU platforms are better-suited for very large networks. Still, up to the scale tested within BrainFrame, flexHH is shown to provide the greater benefit: The flexHH speedup (on DFE) over the hardcoded-IO implementation of BrainFrame is 1.36×. This performance gain is only in part due to a higher operating frequency (180 MHz vs 150 MHz); this accounts for a mere 20% of the speedup. Gain is observed mostly due to the fact that the flexHH implementation does not require pipeline flushing between simulation steps, unlike

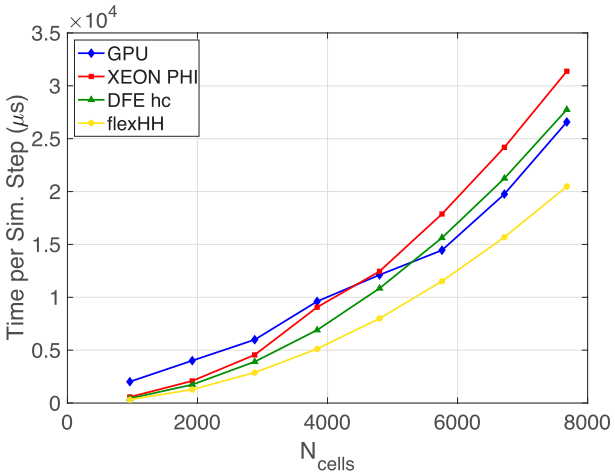

FIGURE 5. Execution time per simulation step for the various BrainFrame implementations of the IO model. *flexHH* is the *HH fully featured* kernel instantiated for the IO model; *DFE hc* is the BrainFrame hardcoded version.

TABLE 4. Maximum achievable real-time NN size of flexHH and BrainFrame kernels for various IO-model interconnectivity densities. The *HH fully featured* kernel is used in all except for the 0% case where the *HH+custom+multi* kernel used. Empty entries imply *no* real-time performance.

NN density	flexHH	DFE hc	Xeon Phi	GPU
100%	166	310	-	-
75%	166	310	-	-
50%	166	310	-	-
25%	166	310	-	-
0%	672	7,680	96	500

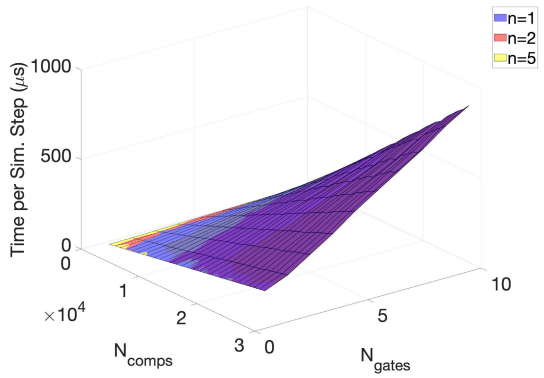

FIGURE 6. Execution time per step for the *HH+custom+multi*.

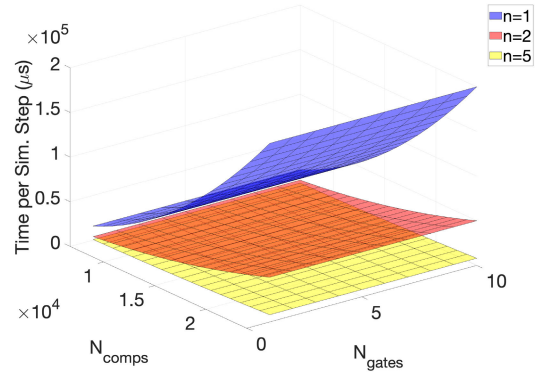

FIGURE 7. Execution time per step for the *HH fully featured*.

the hardcoded version which did not adopt the `calcIGap` optimization of using alternating cell summations.

By comparing the *real-time* (RT) capabilities of the flexHH compared to the BrainFrame kernels (Table 4), we can see that even though the flexHH kernel can support real-time simulations considerably better than the Xeon-Phi and GPU platforms, attainable RT networks are smaller than the hardcoded-DFE ones when full model detail is required. Even though – per simulation step – the flexHH version is faster than the BrainFrame version, the model time step (which defines the real-time constraint) for the flexHH case is much stricter. As a result, for the same simulated brain time, flexHH must execute more simulations steps, impacting RT performance in this specific case. We will analyze this aspect further in the following section when comparing flexHH on DFE with other FPGA designs.

Another disadvantage of the flexHH kernels compared to hardcoded kernels, that can impact real-time performance, is the higher *data-transfer* requirements. For flexHH, besides the initial values of the state variables, the parameters of the equations are sent to the DFE; a necessary trade-off for supporting user-defined HH equations. Additionally, during the execution of the kernel, all equation parameters are perpetually transferred from the on-board DRAM to the FPGA chip as opposed to the hardcoded version that can fit all (hardcoded) parameters to the on-chip memory (BRAMs) during execution. These higher data-transfer requirements increase the length of the pipeline. However, this does not negatively influence throughput, as the higher frequency and higher unroll factor of flexHH shows.

In terms of *performance scalability*, execution time scales linearly with the gate count, for all cases. In terms of compartment count, on the other hand, instantiating gap junctions significantly changes things: Even though execution time scales linearly with compartment count in the absence of gap junctions, the relation becomes quadratic when gap junctions are included; see Figs. 6 and 7. Additionally, for the *HH fully featured* kernel, the amount of compartments per cell has a direct effect on performance scalability. As gap junctions dominate execution for larger problem sizes, the more compartments per cell an experiment instantiates, the less gap-junction connections will be present for the same amount of total compartments. This allows for the execution time to scale more gracefully the more compartments are included within a cell (Fig. 7).

The *power consumption* of the kernels is measured by using the Maxeler tools during runtime of the kernels. The DFE power consumption for each kernel instance ranges between 40.1 and 46.8 Watts. It must be noted that the kernels that include gap junctions exhibit a slightly lower power

TABLE 5. Overview of competitive FPGA-based HH implementations.

Kernel	flexHH *HH*	flexHH *HH fully featured*	BrainFrame DFE [7] *(hardcoded)*	Beuler et al. [23] *(hardcoded)*	Zjajo et al. [24] *(hardcoded)*
Model	HH	IO (extended HH)	IO (extended HH)	simplified HH	IO (extended HH)
Time-step size (ms)	0.06	0.01	0.05	0.10	0.05
NN-interconnectivity density	0%	$\leq$100%	100%	0 %	2%
NN size @ RT	10,368	166	310	1,600	768
OPs/neuron in 1 ms	1,083	285,100	91,580	1,010	18,660
OPs/neuron*NN-size @ RT (GFLOPS)	11.22	44.07	28.38	1.6	14.33
CPU reference	Intel Core i7-4870H, 2.5GHz (16GB RAM)			Intel Core i7-6500U 2.5 GHz (2 cores used)	Intel Xeon E5-1620 3.6GHz (32GB RAM)
Speed-up vs. CPU	35.49 (C lang.)	23.98 (C lang.)	17.69 (C lang.)	2.49 (C++ lang.)	>3,500 (SystemC lang.)
FPGA chip	Stratix V 5SGSD8	Stratix V 5SGSD8	Stratix V 5SGSD8	Virtex 6 XC6VLX240T	Virtex 7 XC7VX550
Device capacity (LUTs* or ALMs)	262,400 ALMs	262,400 ALMs	262,400 ALMs	150,720 LUTs	846,400 LUTs
Perform. density (FLOPS/LUT**)	21,395	90,180	54,096	10,721	16,931

* 6-input LUTs. | ** 2 ALMs $\approx$ 4 6-input LUTs.

consumption. This can be attributed to the fact that part of the hardware allocated for compartment computation is generally idle until the computation of the demanding gap junctions is finished (as they are required to conclude before the rest of the compartment computations). In order to compare with the power consumption of the other acceleration platforms of BrainFrame, we would need to perform similar power measurements. Unfortunately, the BrainFrame paper [7] does not report actual power numbers but resorts to comparing the Thermal Design Power (TDP) of those platforms, which is 225 Watts (Intel Xeon-Phi CPU) and 250 Watts (Nvidia Titan X GPU), respectively. Given that both accelerators are shown to be utilized at peak performance, actual power consumption is known to be higher than the reported TDP. As such, it is safe to conclude that the DFE consumes only a small fraction of the power consumption of either the CPU or GPU acceleration platform of BrainFrame. Consequently, despite the higher acquisition costs of a DFE, it is the best option from a long-term economic standpoint.

There is a final, interesting by-product of flexHH. Since it does not require constant resynthesis in between simulation runs, it allows for accurate predictions of execution time and power consumption for a given problem. This *performance-predictability* aspect is key to combating multi-tenancy stochasticity in HPC clusters and Cloud deployments alike, and can help to offer better quality of service and more precisely estimate operational costs.

D. COMPARISON TO OTHER HH FPGA DESIGNS

To fairly assess the computational capabilities the flexHH library, it is useful to compare against other FPGA-based implementations of HH models with similar complexity. Necessarily, we had to compare with works that are one-off model implementations providing very limited model flexibility, yet serve as relevant design points for assessing computational capacity. Also, all works chosen use *single-precision floating-point* arithmetic. Moreover, to provide a common ground for comparison, only *real-time* simulations are considered; i.e. machine execution time is equal or smaller than the simulated brain time. The network sizes reported, thus, are maximized for respecting real-time performance; results are shown in Table 5.

A main characteristic to note is the time step of each implemented model. Different time-step sizes can significantly affect the computational requirements of a model as it then requires more computations per second in order to provide real-time performance. Time-step size is model-dependant but gap-junctioned HH network models tend to be generally stiff and need small time steps for correct results. Besides, the step size of the BrainFrame *hardcoded* kernel [7] is quite relaxed compared to the flexHH *HH fully featured* one, which needs to accommodate a broad range of different simulations, thus impacting its own step size. The hardcoded implementations, as they are not meant for general use beyond the specific models, can afford to use non-standard integration methods that allow for the use of more relaxed time steps. flexHH, though, must adhere to standard ODE methods to retain its generality. Simpler models like the basic *HH* kernel also exhibit a relaxed step size compared to their more complex counterparts.

Looking at the real-time network support for each implementation, we see that the flexHH *HH* kernel outperforms the HH implementation of Beuler *et al.* [23] despite having a stricter time-step size. In the case of the IO model, on the other hand, the hardcoded IO implementation provides a larger real-time network. Although per time step the flexHH is faster than the hardcoded version as described before, having such a stricter time-step size than the BrainFrame kernel [7], forces the flexHH kernel to conduct far greater computations for the same simulated brain time. The implementation of Zjajo *et al.* [24] can support a larger network size under real-time conditions as it would be expected with the reduced connectivity density.

To fairly analyse the computational capacity of the different kernels, performance is given in FLOPS (FP arithmetic operations per second) and is measured in largest possible networks so long as real-time simulation speeds are maintained. For the flexHH kernels,the BrainFrame kernel, and

the Zjajo kernel, FP operations are derived by the C reference code. For the Beuler kernel, FP operations are derived from the model description provided in their work.

In terms of FLOPS, the flexHH kernels outperform their hardcoded counterparts: for the simple HH model about 10× higher compared to the corresponding Beuler kernel and for the IO model almost 2× higher compared to the BrainFrame kernel (which is the hardcoded eHH kernel with the most FLOPS). Compared to the Zjajo kernel, the flexHH *fully-featured* kernel provides almost 3.5× more FLOPS, highlighting the potential that the dataflow programming has in using reconfigurable substrates efficiently.

Since not all compared implementations use the same FPGA hardware, it is also fair to compare the performance density of the various designs; namely, the FLOPS per processing element (in this case the 6-input LUT). The computational capability can, thus, be calculated for each kernel while also accounting for device-capacity differences. The MAX4 DFE used in the DFE-based related work as well as for the flexHH is a Stratix V 5SGSD8. We adopt the assumption also used in [25] that 2 ALMs ≈ 4 6-input LUTs to transform ALM count to LUT count for the comparison between designs. Here, the flexHH library provides more than twice the performance density for the simple HH case and about 65% higher FLOPS/LUT for the IO implementation compared to the BrainFrame hardcoded design. Compared to the traditional FPGA-based platform of Zjajo *et al.* [24], flexHH provides more than 5× higher performance density when simulating at real-time speeds. Consequently, dataflow programming also favors performance efficiency.

VI. CONCLUSION

In this paper, we presented a flexible, scalable and high-performing HH-model library called flexHH and implemented it on a (FPGA-based) DFE platform. flexHH enables synthesis-free neuro-simulations while providing clear performance benefits compared to C-based, single-threaded execution and traditional NEURON-based simulations. It also performs uniformly better per simulation step than prior hardcoded hardware implementation of models of the same category, while maintaining high simulation flexibility. Furthermore, the most feature-rich instance of the library is performing better than respective Xeon-Phi and GPU implementations. Lastly, flexHH exhibits a high performance density compared to related works, showing high efficiency in logic-resource usage.

The high flexibility of flexHH is achieved at the cost of potentially higher compute overheads over the same simulated brain time. The need to adhere to standard HH-model formalism and ODE solutions gives less space for model-specific optimizations and precludes solver hacks. This can affect performance especially in cases of real-time experimentation. Whether such a cost needs be paid, though, is highly model-dependent and relies ultimately on how modelers will use flexHH during experiment design. The flexibility and compliance with scientific standards, on the other hand, make flexHH unique and immediately useful to computational neuroscientists, giving it a major advantage towards community adoption compared to traditional FPGA-based models. Besides, flexHH is the first work to demonstrate the benefits of the dataflow-computing paradigm for accelerating brain simulations.

Last but not least, since all model features are defined at build time and resources are allocated statically, both power and execution time can be predicted very accurately based on problem size. A predictable application profile is an important property when deploying on HPC clusters, which also makes flexHH a pivotal addition to heterogeneous HPC simulation platforms like BrainFrame.

ACKNOWLEDGMENT

The authors also wish to thank Milos Puzovic for his critical help in facilitating our Maxeler-DFE experiments in the STFC Hartree Centre, U.K. The authors gratefully acknowledge the continuous support provided by Maxeler Technologies throughout the research effort.

REFERENCES

[1] W. Maass, "Noisy spiking neurons with temporal coding have more computational power than sigmoidal neurons," in *Proc. Neural Inf. Process. Syst.*, 1996, pp. 211–217.

[2] W. Maass, "Networks of spiking neurons: The third generation of neural network models," *Neural Netw.*, vol. 10, no. 9, pp. 1659–1671, Dec. 1997.

[3] A. L. Hodgkin and A. F. Huxley, "A quantitative description of membrane current and its application to conduction and excitation in nerve," *J. Physiol.*, vol. 117, no. 4, pp. 500–544, Aug. 1952. [Online]. Available: https://physoc.onlinelibrary.wiley.com/doi/abs/10.1113/jphysiol.1952.sp004764

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

[5] Y. U. Shepherd Lab. (2019). *Modeldb, Models That Contain the Model Type : Neuron or Other Electrically Excitable Cell*. [Online]. Available: https://senselab.med.yale.edu/ModelDB/ModelList.cshtml?id=3537

[6] *Whitepaper: MaxCompiler*, Maxeler Technologies, London, U.K., Feb. 2011. [Online]. Available: https://www.maxeler.com/media/documents/MaxelerWhitePaperMaxCompiler.pdf

[7] G. Smaragdos, G. Chatzikonstantis, R. Kukreja, H. Sidiropoulos, D. Rodopoulos, I. Sourdis, Z. Al-Ars, C. Kachris, D. Soudris, C. I. De Zeeuw, and C. Strydis, "BrainFrame: A node-level heterogeneous accelerator platform for neuron simulations," *J. Neural Eng.*, vol. 14, no. 6, Dec. 2017, Art. no. 066008. [Online]. Available: http://stacks.iop.org/1741-2552/14/i=6/a=066008

[8] R. C. Cannon, P. Gleeson, S. Crook, G. Ganapathy, B. Marin, E. Piasini, and R. A. Silver, "LEMS: A language for expressing complex biological models in concise and hierarchical form and its use in underpinning NeuroML 2," *Frontiers Neuroinform.*, vol. 8, p. 79, Sep. 2014. [Online]. Available: https://www.frontiersin.org/article/10.3389/fninf.2014.00079

[9] M. Beyeler, K. D. Carlson, T.-S. Chou, N. Dutt, and J. L. Krichmar, "CARLsim 3: A user-friendly and highly optimized library for the creation of neurobiologically detailed spiking neural networks," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, Jul. 2015, pp. 1–8, doi: 10.1109/IJCNN.2015.7280424.

[10] R. V. Hoang, D. Tanna, L. C. J. Bray, S. M. Dascalu, and F. C. Harris, Jr., "A novel CPU/GPU simulation environment for large-scale biologically realistic neural modeling," *Frontiers Neuroinformat.*, vol. 7, p. 19, Oct. 2013. [Online]. Available: https://www.frontiersin.org/article/10.3389/fninf.2013.00019

[11] A. Sripad, G. Sanchez, M. Zapata, V. Pirrone, T. Dorta, S. Cambria, A. Marti, K. Krishnamourthy, and J. Madrenas, "SNAVA—A real-time multi-FPGA multi-model spiking neural network simulation architecture," *Neural Netw.*, vol. 97, pp. 28–45, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0893608017302150

[12] K. Cheung, S. R. Schultz, and W. Luk, "NeuroFlow: A general purpose spiking neural network simulation platform using customizable processors," *Frontiers Neurosci.*, vol. 9, p. 516, Jan. 2016. [Online]. Available: http://journal.frontiersin.org/article/10.3389/fnins.2015.00516
[13] J. R. De Gruijl, P. Bazzigaluppi, M. T. 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, 2012, Art. no. e1002814.
[14] G. Smaragdos, C. Davies, C. Strydis, I. Sourdis, C. Ciobanu, O. Mencer, and C. I. De Zeeuw, "Real-time olivary neuron simulations on dataflow computing machines," in *Proc. Int. Supercomputing Conf.* Cham, Switzerland: Springer, 2014, pp. 487–497.
[15] O. Pell and V. Averbukh, "Maximum performance computing with dataflow engines," *Comput. Sci. Eng.*, vol. 14, no. 4, pp. 98–103, Jul. 2012.
[16] T. Becker, P. Burovskiy, A. M. Nestorov, H. Palikareva, E. Reggiani, and G. Gaydadjiev, "From exaflop to exaflow," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2017, pp. 404–409.
[17] N. Schweighofer, K. Doya, and M. Kawato, "Electrophysiological properties of inferior olive neurons: A compartmental model," *J. Neurophysiol.*, vol. 82, no. 2, pp. 804–817, Aug. 1999.
[18] N. Schweighofer, K. Doya, H. Fukai, J. V. Chiron, T. Furukawa, and M. Kawato, "Chaos may enhance information transmission in the inferior olive," *Proc. Nat. Acad. Sci. USA*, vol. 101, no. 13, pp. 4655–4660, Mar. 2004.
[19] P. Grigoras, P. Burovskiy, J. Arram, X. Niu, K. Cheung, J. Xie, and W. Luk, "Dfesnippets: An open-source library for dataflow acceleration on FPGAs," in *Applied Reconfigurable Computing*, S. Wong, A. C. Beck, K. Bertels, and L. Carro, Eds. Cham, Switzerland: Springer, 2017, pp. 299–310.
[20] J. R. De Gruijl, T. M. Hoogland, and C. I. De Zeeuw, "Behavioral correlates of complex spike synchrony in cerebellar microzones," *J. Neurosci.*, vol. 34, no. 27, pp. 8937–8947, Jul. 2014. [Online]. Available: http://www.jneurosci.org/content/34/27/8937
[21] E. M. Izhikevich, *Dynamical Systems in Neuroscience*. Cambridge, MA, USA: MIT Press, 2007.
[22] R. Miedema, "Flexhh: A flexible hardware library for hodgkin-huxley-based neural simulations," M.S. thesis, Comput. Eng., Delft Univ. Technol., Delft, The Netherlands, 2019.
[23] M. Beuler, A. Krum, W. Bonath, and H. Hillmer, "Nepteron processor for real-time computation of conductance-based neuronal networks," in *Proc. Euromicro Conf. Digit. Syst. Design (DSD)*, Aug. 2017, pp. 78–85.
[24] A. Zjajo, J. Hofmann, G. J. Christiaanse, M. van Eijk, G. Smaragdos, C. Strydis, A. de Graaf, C. Galuzzi, and R. van Leuken, "A real-time reconfigurable multichip architecture for large-scale biophysically accurate neuron simulation," *IEEE Trans. Biomed. Circuits Syst.*, vol. 12, no. 2, pp. 326–337, Apr. 2018.
[25] G. Smaragdos, S. Isaza, M. F. van Eijk, I. Sourdis, and C. Strydis, "FPGA-based biophysically-meaningful modeling of olivocerebellar neurons," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, New York, NY, USA, 2014, pp. 89–98, doi: 10.1145/2554688.2554790.

RENE MIEDEMA was born in Spijkenisse, The Netherlands, in 1992. He received the B.Sc. degree in electrical engineering and the M.Sc. degree in computer engineering from the Delft University of Technology, in 2015 and 2019, respectively. He is currently a Research Analyst with the Neuroscience Department, Erasmus Medical Center. His research interests include high-performance computing, numerical methods, and brain modeling.

GEORGIOS SMARAGDOS (Member, IEEE) received the Diploma (Engineering) degree in electronics and computer engineering from the Technical University of Crete (TUC), Chania, and the M.Sc. degree in computer engineering from the Delft University of Technology (TUD), The Netherlands, in July 2012. He is currently pursuing the Ph.D. degree with the Neuroscience Department (Brain Modeling HPC Acceleration), Erasmus Medical Center, under the supervision of C. I. de Zeeuw and the Co-Supervision of C. Strydis and Y. Sourdis.

MARIO NEGRELLO received the Ph.D. degree *(summa cum laude)* in cognitive science from the University of Osnabrück, Germany. He is currently an Assistant Professor in computational neuroscience with the Erasmus Medical Center, Rotterdam. He combines empirical research and computational models to uncover the principles of unsupervised motor learning from biological neural networks. He has published in the fields of machine learning, cognitive robotics, artificial life, evolutionary robotics, neuroethology and neuroscience, and a monograph, published by Springer, U.S., in the Series Cognitive and Neural systems entitled Invariants of Behavior, in 2012.

ZAID AL-ARS (Member, IEEE) held various roles with a number of tech industry heavyweights, such as Siemens and IBM. He is currently an Associate Professor with the Quantum and Computer Engineering Department, Delft University of Technology, where he leads the Accelerated Big Data Systems Group. His work focuses on developing computing infrastructures for efficient processing of big data analytics applications. He is a Co-Founder of a couple of big data companies specialized in high performance analytics solutions and AI. He is also on the advisory board of a number of high-tech start-ups. He has published more than 100 peer-reviewed publications. He holds two patents.

MATTHIAS MÖLLER received the Diploma and Ph.D. degrees in mathematics from the Faculty of Mathematics, TU Dortmund University, Germany, in 2003 and 2008, respectively. He is currently an Assistant Professor with the Numerical Analysis Group, Department of Applied Mathematics, Delft University of Technology, The Netherlands. His research interest includes numerical methods for solving partial differential equations and their efficient implementation on heterogeneous high-performance computing platforms, including GPUs and FPGAs. He is also active in the emerging field of quantum-accelerated numerical linear algebra applications.

CHRISTOS STRYDIS (Senior Member, IEEE) received the M.Sc. *(magna cum laude)* and the Ph.D. degrees in computer engineering from the Delft University of Technology. He is currently a tenured Assistant Professor in computer engineering and the Head of the NeuroComputing Laboratory, Neuroscience Department, Erasmus Medical Center, The Netherlands. He has published work in well-known international conferences and journals. He has delivered invited talks in various venues. His current research interests include brain simulations, high-performance computing, low-power embedded (implantable) systems, and functional ultrasound imaging.