flexHH:
A flexible hardware library for Hodgkin-Huxley-based neural simulations

Rene Miedema CE-MS

Delft University of Technology

Abstract

In the field of computational neuroscience, complex mathematical models are used to replicate brain behavior with the goal of understanding the biological processes involved. The simulation of such models are computationally expensive and therefore, in recent years, high-performance computing systems have been identified as a possible solution to accelerate their execution. However, most of those implementations are model-specific and thus non-reusable for other modeling efforts, requiring a completely new development effort per model used. The challenge lies in offering high-performance and scalable libraries (so as to support the construction and simulation of large-scale brain models) while at the same time offering high degrees of modeling flexibility and parameterization. This thesis presents *flexHH*, a scalable hardware library implementing five accelerated and highly parameterizable instances of the Hodgkin-Huxley neuron model, one of the most widely used biophysically-meaningful neuron representations. As a result, the user is able to instantiate custom models using flexHH and immediately take advantage of the acceleration without the mediation of the engineer. The five flexHH implementations target the Maxeler Data-Flow Engine (DFE), an FPGA-based acceleration solution, and incrementally support a number of features such as custom ion channels, multiple cell compartments and inter-neuron gap-junction connectivity. Furthermore, for each of the five implementations it is possible to select either the forward-Euler, second, or third-order Runge-Kutta numerical method. A speedup between 14×-36× has been achieved compared to a sequential C implementation, when run on a 2.5-GHz Intel Core-i7 CPU, while no practical performance drop is observed when compared to a hard-coded version of a DFE, an Intel Xeon-Phi CPU, and an NVidia Titan X GPU. In this thesis, flexHH kernels are rigorously validated, an evaluation of the influence of the numerical methods is done, and a comprehensive resources usage, performance, and power-consumption evaluation of the various DFE implementations is presented.

TU Delft Delft University of Technology

Quantum & Computer Engineering

flexHH: A flexible hardware library for Hodgkin-Huxley-based neural simulations

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Rene Miedema
born in Spijkenisse, The Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

flexHH: A flexible hardware library for Hodgkin-Huxley-based neural simulations

by Rene Miedema

Abstract

In the field of computational neuroscience, complex mathematical models are used to replicate brain behavior with the goal of understanding the biological processes involved. The simulation of such models are computationally expensive and therefore, in recent years, high-performance computing systems have been identified as a possible solution to accelerate their execution. However, most of those implementations are model-specific and thus non-reusable for other modeling efforts, requiring a completely new development effort per model used. The challenge lies in offering high-performance and scalable libraries (so as to support the construction and simulation of large-scale brain models) while at the same time offering high degrees of modeling flexibility and parameterization. This thesis presents *flexHH*, a scalable hardware library implementing five accelerated and highly parameterizable instances of the Hodgkin-Huxley neuron model, one of the most widely used biophysically-meaningful neuron representations. As a result, the user is able to instantiate custom models using flexHH and immediately take advantage of the acceleration without the mediation of the engineer. The five flexHH implementations target the Maxeler Data-Flow Engine(DFE), an FPGA-based acceleration solution, and incrementally support a number of features such as custom ion channels, multiple cell compartments and inter-neuron gap-junction connectivity. Furthermore, for each of the five implementations it is possible to select either the forward-Euler, second, or third-order Runge-Kutta numerical method. A speedup between 14×-36× has been achieved compared to a sequential C implementation, when run on a 2.5-GHz Intel Core-i7 CPU, while no practical performance drop is observed when compared to a hard-coded version of a DFE, an Intel Xeon-Phi CPU, and a NVidia Titan X GPU. In this thesis, flexHH kernels are rigorously validated, an evaluation of the influence of the numerical methods is done, and a comprehensive resources usage, performance, and power-consumption evaluation of the various DFE implementations is presented.

Laboratory : Computer Engineering
Codenumber : Q&CE-CE-MS-2019-05

Committee Members :

Advisor: Christos Strydis, Neuroscience Dept., Erasmus MC

Chairperson: Zaid Al-Ars, Q & CE., TU Delft

Member:	Matthias Möller, DIAM., TU Delft
Member:	Georgios Smaragdos, Neuroscience Dept., Erasmus MC

Dedicated to my family and friends

Contents

List of Figures xii

List of Tables xvi

List of Acronyms xvii

Acknowledgements xix

1 Introduction 1
- 1.1 Motivation . . . 2
- 1.2 Thesis scope and contributions . . . 2
  - 1.2.1 Thesis goal . . . 3
  - 1.2.2 Thesis contributions . . . 3
- 1.3 Thesis organization . . . 4

2 Background 5
- 2.1 The biological neuron . . . 5
- 2.2 Spiking neural networks . . . 5
  - 2.2.1 The Hodgkin-Huxley model . . . 7
  - 2.2.2 The Inferior Olive model . . . 8
- 2.3 NeuroML . . . 9
- 2.4 Numerical methods . . . 9
- 2.5 Data-flow computing . . . 12
- 2.6 BrainFrame . . . 14
- 2.7 Summary . . . 15

3 Related work 17
- 3.1 Software simulators . . . 17
- 3.2 FPGA-based simulators . . . 18

4 Implementation 23
- 4.1 Preliminary work . . . 23
- 4.2 The flexHH library . . . 25
- 4.3 Model function generalization . . . 27
  - 4.3.1 HH . . . 28
  - 4.3.2 Custom ion gates . . . 30
  - 4.3.3 Multiple cell compartments . . . 31
  - 4.3.4 Gap junctions . . . 32
- 4.4 DFE implementation . . . 33
  - 4.4.1 HH . . . 33

4.4.2 Custom ion gates . . . 45
4.4.3 Multiple cell compartments . . . 49
4.4.4 Gap junctions . . . 52

5 Evaluation **59**
5.1 Validation . . . 59
5.1.1 C-code validation . . . 59
5.1.2 DFE-code validation . . . 72
5.2 Exploration of time-step-size . . . 86
5.2.1 Discussion . . . 98
5.3 Hardware-resource usage . . . 100
5.3.1 Methodology for the prediction of hardware-usage . . . 101
5.3.2 Hardware-usage prediction validation . . . 104
5.3.3 Influence of model features on hardware-usage . . . 108
5.4 Performance evaluation . . . 111
5.4.1 Performance influence of numerical methods . . . 113
5.4.2 Theoretical performance scaling . . . 114
5.4.3 LMem bandwidth evaluation . . . 115
5.4.4 Experimental performance results . . . 126
5.4.5 Performance comparison . . . 130
5.5 Energy results . . . 133

6 Conclusions **137**
6.1 Discussion . . . 137
6.2 Contributions . . . 138
6.3 Future work . . . 138

Bibliography **143**

A Mathematical description Inferior-Olive model **145**

B Simulation parameters **149**
B.1 Validation . . . 149
B.1.1 C-code validation . . . 149
B.1.2 DFE-code validation . . . 150
B.2 Exploration time step-size . . . 152
B.2.1 *HH* . . . 152
B.2.2 *HH+gap* . . . 152
B.2.3 *HH+custom* . . . 153
B.2.4 *HH+custom+multi* . . . 153
B.2.5 *HH+custom+multi+gap* . . . 154
B.3 Discussion . . . 154
B.3.1 HH . . . 154
B.3.2 IO . . . 155

C Hardware configurations used for the evaluation of the resource usage **157**

D Power results **161**

List of Figures

1.1 Graph plotting computational complexity involved versus achieved biological plausibility of various Spiking-neural-networks (SNNs) [1]. Hodgkin-Huxley (HH) models are clearly the most demanding but also most detailed ones across the board. 2

2.1 Visual representations of a single neuron, with the compartments specified. 6
2.2 Overview of supported spiking behaviour for different models [1]. . . 7
2.3 Hierarchical representation of network structure in NeuroML. 10
2.4 Architectures of different computer paradigms. (a) Data-flow. (b) Control flow 13
2.5 Maxeler data-flow architecture. 14

3.1 Simulation time for different simulators [2]. 18

4.1 Architecture of the Data-Flow Engine (DFE) and Central Processing Unit (CPU). 42
4.2 Visualization of the *HH* kernel when the forward- Euler method is used. 42
4.3 Visualization of the *HH* kernel when the second-order Runge-Kutta is used. 42
4.4 Visualization of the *HH* kernel when the third-order Runge-Kutta method is used. 43
4.5 Visualization of the "stage" block in the *HH* kernel. The red blocks contain the hardware of the algorithms with the same label. 43
4.6 Visualization of multiple cell compartments in a sequential structure. . 50
4.7 Directions of the execution order (red arrows) and the data dependencies (black arrows) for both row-wise and column-wise calculations. . . 54

5.1 Schematic overview of the validation steps. 60
5.2 (a) Output of the voltage for a simulation in NEURON of a single HH cell. (b) Error between NEURON and fwd-Euler C. (c) Error between NEURON and rk2 C. (d) Error between NEURON and rk3 C. $dt = 0.01$ ms. 63
5.3 Output of the voltage for a simulation in NEURON of a single HH cell, $dt = 0.01$ ms. 64
5.4 (a) Output of the voltage for a simulation in NEURON of a single HH cell. (b) Error between NEURON and fwd-Euler C. (c) Error between NEURON and rk2 C. (d) Error between NEURON and rk3 C. $dt = 0.01$ ms, $t_{sim} = 6000$ ms. 65
5.5 (a) Output of the voltage for a simulation in NEURON of a single HH cell. (b) Error between NEURON and fwd-Euler C. (c) Error between NEURON and rk2 C. (d) Error between NEURON and rk3 C. $dt = 0.005$ ms, $t_{sim} = 300$ ms. 66

5.6 (a) Output of the voltage for a simulation in NEURON (using Crank-Nicolson) of a single HH cell. (b) Error between NEURON and fwd-Euler C. (c) Error between NEURON and rk2 C. (d) Error between NEURON and rk3 C. $dt = 0.01$ ms, $t_{sim} = 300$ ms. 67
5.7 Output of the voltage for a simulation in NEURON (using Crank-Nicolson) of a single HH cell, $dt = 0.01$. 68
5.8 (a) Output of the voltage for a simulation in NEURON (using Crank-Nicolson) of a single HH cell. (b) Error between NEURON and fwd-Euler C. (c) Error between NEURON and rk2 C. (d) Error between NEURON and rk3 C. $dt = 0.01ms$, $t_{sim} = 30000ms$. 69
5.9 (a) Output of the exact solution using Equation (5.1). (b) Error between the exact solution and NEURON using the bwd-Euler method. (c) Error between the exact solution and NEURON using the Crank-Nicolson method. (d) Error between the exact solution and fwd-Euler C. (e) Error between the exact solution and rk2 C. (f) Error between the exact solution and rk3 C. $dt = 0.01$ ms, $t_{sim} = 25$ ms. 70
5.10 (a) Output of the axonal-voltage for a simulation with the reference code for a single cell (cell 0, in the code). (b) Error of fwd-Euler C. (c) Error of rk2 C. (d) Error of rk3 C. $dt = 0.01$ ms. 71
5.11 (a) Output of the voltage for a simulation in C of a single HH cell. (b) Error between C and the DFE-code using fwd-Euler. (c) Error between C and the DFE-code using rk2. (d) Error between C and DFE-code using rk3. $dt = 0.01$ ms. 74
5.12 Error between C and the DFE-code in simulation mode using rk3. $dt = 0.01$ ms, $t_{sim} = 300$ ms. 75
5.13 Error when using and not using generalized functions in C. $dt = 0.01$ ms, $t_{sim} = 300$ ms. 75
5.14 (a) Output of the voltage for a simulation in C of a single cell (cell 0, in the code) of a HH network with gap junctions. (b) Error between C and the DFE-code using fwd-Euler. (c) Error between C and the DFE-code using rk2. (d) Error between C and the DFE-code using rk3. $dt = 0.01$ ms. 78
5.15 Average error per simulation step using the forward-Euler method for a simulation of the *HH+gap* implementation (i and j are indices of a cell). 79
5.16 Average error per simulation step using the second-order Runge-Kutta method for a simulation of the *HH+gap* implementation (i and j are indices of a cell). 79
5.17 Average error per simulation step using the third-order Runge-Kutta method for a simulation of the *HH+gap* implementation (i and j are indices of a cell). 80
5.18 (a) Voltage trace, (b) error using the third-order Runge-Kutta method for a simulation of the *HH+gap* implementation. 81
5.19 Voltage trace of the output of the *HH+gap* rk3 implementation in C (a) Voltage trace with initialization from CPU values, (b) Voltage trace with initialization from DFE values. 82

5.20 (a) Output of the voltage for a simulation in C of a single axon. (b) Error between C and the DFE-code using fwd-Euler. (c) Error between C and the DFE-code using rk2. (d) Error between C and the DFE-code using rk3. $dt = 0.01$ ms. 83
5.21 (a) Output of the axonal-voltage for a simulation in C of a single Inferior-Olive (IO) cell. (b) Error between C and the DFE-code using fwd-Euler. (c) Error between C and the DFE-code using rk2. (d) Error between C and the DFE-code using rk3. $dt = 0.01$ ms. 84
5.22 (a) Output of the voltage for a simulation in C of a single axon (the axon from cell 0, in the code) of an IO network. (b) Error between C and the DFE-code using fwd-Euler. (c) Error between C and the DFE-code using rk2. (d) Error between C and the DFE-code using rk3. $dt = 0.01$ ms. 85
5.23 Output of the voltage for a simulation of a single HH cell using the forward-Euler solver.(a) $dt = 0.01$ ms, (b) $dt = 0.06$ ms 87
5.24 Output of the voltage for a simulation of a single HH cell using the second-order Runge-Kutta solver. (a) $dt = 0.05$ ms, (b) $dt = 0.06$ ms . . . 88
5.25 Output of the voltage for a simulation of a single HH cell using the third-order Runge-Kutta solver. 89
5.26 Output of the voltage of compartment 0 for a simulation of a *HH+gap* network using the forward-Euler method, $dt = 0.01$ ms. 90
5.27 Output of the voltage of a single compartment for a simulation of a *HH+custom* (soma) compartment when using the forward-Euler solver for different time-step sizes. 91
5.28 Output of the voltage of a single compartment for a simulation of a *HH+custom* (soma) compartment when using the second-order Runge-Kutta solver for different time-step sizes. 92
5.29 Output of the voltage of a single compartment for a simulation of a *HH+custom* (soma) compartment when using the third-order Runge-Kutta solver for different time-step sizes. 93
5.30 Output of the voltage of a single cell for a simulation of a *HH+custom+multi* (IO) cell when using the forward-Euler solver for different time-step sizes. 94
5.31 Output of the voltage of a single cell for a simulation of a *HH+custom+multi* (IO) cell when using the second-order Runge-Kutta solver for different time-step sizes. 95
5.32 Output of the voltage of a single cell for a simulation of a *HH+custom+multi* (IO) cell when using the third-order Runge-Kutta solver for different time-step sizes. 96
5.33 Output of the axonal-voltage of cell 0 for a simulation of the IO network. 97
5.34 Output of the voltage for a relaxed simulation of a single HH cell, $dt = 0.01$ ms. 99
5.35 Output voltage of the axon for a relaxed simulation of a single IO cell, $dt = 0.01$ ms. 99

5.36 LMem bandwidth (thick line) and total required throughput for multiple unroll factors (other lines) $f = 180MHz$. (a) For the *HH* kernel instance. (b) For the *HH+custom* kernel instance. 119
5.37 LMem bandwidth (red plane or thick line) and total required throughput for multiple unroll factors (other planes or other lines) for *HH+custom+multi* kernel instance, $f = 180MHz$. (a) $960 \leq N_{comps} \leq 19600$. (b) $N_{comps} = 960$. 120
5.38 LMem bandwidth (red plane) and total required throughput for multiple unroll factors (other planes) for *HH+gap* kernel instance, $f = 180MHz$. 122
5.39 LMem bandwidth (red plane) and total required throughput for multiple unroll factors (other planes) *HH+custom+multi+gap* kernel instance, $f = 180MHz$. (a) $N_{comps,cell} = 1$. (b) $N_{comps,cell} = 3$ 123
5.40 Time per simulation step for the kernel instances without gap junctions, $N_{gates} = 10$, $f = 180MHz$. (a) *HH*. (b) *HH+custom*. (c) *HH+custom+multi*. 124
5.41 Time per simulation step for the kernel instances with gap junctions, $N_{gates} = 10$, $f = 180MHz$. (a) *HH+gap*. (b) *HH+custom+multi+gap*. . . . 125
5.42 Estimation of the frequency of compartments per compartment. 126
5.43 Time per simulation step for the kernel instances without multiple cell compartments, $f = 180MHz$. (a) *HH* (b) *HH+gap* (c) *HH+custom* 129
5.44 Time per simulation step for the kernel instances with multiple cell compartments, $f = 180MHz$. (a) *HH+custom+multi* (b) *HH+custom+multi+gap* 129
5.45 Time per simulation step for the implementations of BrainFrame and the flexHH *HH+custom+multi+gap* kernel.(DFE hc is the hard-coded DFE variant from BrainFrame, DFE gen is the flexHH kernel) 132
5.46 Energy usage for the implementations without multiple cell compartments, $f = 180MHz$. (a) *HH*. (b) *HH+gap* (c) *HH+custom* 134
5.47 Energy usage for the implementations with multiple cell compartments, $f = 180MHz$. (a) *HH+custom+multi*. (b) *HH+custom+multi+gap*. 135

List of Tables

3.1 Characterization of computational efficiency and parallelization of neural network simulators [2]. . . . 19

4.1 Maximum speedup for different implementations on the DFE in comparison with the C-code on the CPU host. . . . 25
4.2 Overview of the supported features per implemented kernel in the flexHH-library. . . . 26
4.3 Parameters of the HH-model filled into Equation (4.4). . . . 29
4.4 Parameter translation from NeuroML to the implementation on the DFE. 30
4.5 Scalar variables for the *HH* implementation. . . . 39
4.6 Variables of the structure *gateConstants* for the *HH* implementation. . 39
4.7 Variables of the structure *compartmentConstants* for the *HH* implementation. . . . 40
4.8 Variables of the structure *gateConstants* for the *HH+custom* implementation. . . . 48
4.9 Variables of the structure *compConstants* for the *HH+custom* implementation. . . . 49
4.10 Constants for different ODE solvers, to be filled in Equation (4.39) . . . 56

5.1 Variables during simulation. The index indicates to which stage the variable belongs to. . . . 73
5.2 Variables which are used to calculate *dvdt*3. . . . 73
5.3 Maximum time-step-size in ms for a simulation of each kernel. . . . 98
5.4 Maximum time-step-size in ms for relaxed simulations. . . . 100
5.5 Configurations of the *HH* implementation with different frequencies. . 101
5.6 Scaling factors per kernel segment for *HH* kernel instances. . . . 102
5.7 Scaling factors per kernel segment for *HH+custom* kernel instances. . . 102
5.8 Scaling factors per kernel segment for *HH+custom+multi* kernel instances. 103
5.9 Scaling factors per kernel segment for *HH+gap* kernel instances. . . . 103
5.10 Scaling factors per kernel segment for *HH+custom+multi+gap* kernel instances. *N_{ODE} is added after inspection of hardware-usage. . . . 104
5.11 Overview of which hardware parameters influence the hardware-usage of which kernel segment. . . . 104
5.12 Reference configurations used for the hardware-usage prediction of the *HH* kernel instances. . . . 105
5.13 Reference configurations used for the hardware-usage prediction of the *HH+custom* kernel instances. . . . 105
5.14 Reference configurations used for the hardware-usage prediction of the *HH+custom+multi* kernel instances. . . . 105
5.15 Reference configurations used for the hardware-usage prediction of the *HH+gap* kernel instances. . . . 106

5.16 Reference configurations used for the hardware-usage prediction of the *HH+custom+multi+gap* kernel instances. 106
5.17 Available resources on the Maia DFE. 107
5.18 Error of the resource prediction of *HH* configurations as a percentage of the available resources on the Maia DFE. 107
5.19 Error of the resource prediction of *HH+custom* configurations as a percentage of the available resources on the Maia DFE. 107
5.20 Error of the resource prediction of *HH+custom+multi* configurations as a percentage of the available resources on the Maia DFE. 108
5.21 Error of the resource prediction of *HH+gap* configurations as a percentage of the available resources on the Maia DFE. 108
5.22 Error of the resource prediction of *HH+custom+multi+gap* configurations as a percentage of the available resources on the Maia DFE. 109
5.23 Error of the total resource prediction for all kernel instances as a percentage of the available resources on the Maia DFE. 109
5.24 BRAMs used for *HH*, *HH+custom*, and *HH+custom+multi* kernel instances with the hardware parameters, $uf = 2$, $N_{comps,max} = 10240$, $N_{gates,max} = 4$, and $N_{ODE} = 1$. 110
5.25 BRAMs used for the kernel segments of the *HH* and *HH+custom* kernel instances with the hardware parameters, $uf = 1$, $N_{comps,max} = 30720$, $N_{gates,max} = 8$, and $N_{ODE} = 1$. 111
5.26 BRAMs used for the kernel segments of the *HH+custom+multi* and *HH+custom+multi+gap* kernel instances with the hardware parameters, $uf = 1$, $N_{comps,max} = 30720$, $N_{gates,max} = 8$, and $N_{ODE} = 1$. 112
5.27 Specifications of the hardware used for the performance measurement. 112
5.28 Optimized kernel configurations, based on the resource prediction method, which support 10 channels and minimally 16384 compartments. 113
5.29 Parameters used to see influence numerical methods on performance. . 114
5.30 Parameters used to the see influence of different numerical methods on performance. 114
5.31 I/O communication with Large Memory (LMem) of the *HH* kernel instance. *vCompIn is only needed in the first simulation step. 116
5.32 I/O communication with LMem of the *HH+custom* kernel instance. . . 116
5.33 I/O communication with LMem of the *HH+custom+multi* kernel instance. *vCompIn is only needed in the first simulation step 117
5.34 I/O communication with LMem of the *HH+gap* kernel instance. *vCompIn is only needed in the first simulation step. 117
5.35 I/O communication with LMem of the *HH+custom+multi+gap* kernel instance. *vCompIn is only needed in the first simulation step. 118
5.36 Required throughput for each kernel instance. 118
5.37 Configurations used for performance measurements of the *HH* kernel instance. 121
5.38 Configurations used for performance measurements of the *HH+gap* kernel instance. 121

5.39 Configurations used for performance measurements of the *HH+custom* kernel instance. . . . 121
5.40 Configurations used for performance measurements of the *HH+custom+multi* kernel instance. . . . 121
5.41 Configurations used for performance measurements of the *HH+custom+multi+gap* kernel instance. . . . 122
5.42 Best uf possible for configurations with maximum $N_{comps,max}$, where $N_{gates,max} = 10$ and $f = 180MHz$ for the *HH* kernel instance. . . . 127
5.43 Best uf possible for configurations with maximum $N_{comps,max}$, where $N_{gates,max} = 10$ and $f = 180MHz$ for the *HH+gap* kernel instance. . . . 127
5.44 Best uf possible for configurations with maximum $N_{comps,max}$, where $N_{gates,max} = 10$ and $f = 180MHz$ for the *HH+custom* kernel instance. . . 127
5.45 Best uf possible for configurations with maximum $N_{comps,max}$, where $N_{gates,max} = 10$ and $f = 180MHz$ for the *HH+custom+multi* kernel instance. 127
5.46 Best uf possible for configurations with maximum $N_{comps,max}$, where $N_{gates,max} = 10$ and $f = 180MHz$ for the *HH+custom+multi+gap* kernel instance. . . . 128
5.47 Configurations used to get performance results for the comparison against the CPU and the speedup against the CPU with 23040 compartments. . . . 130
5.48 Fabric specialisations of the Xeon Phi and NVidia Titan X used in [3]. . 131
5.49 Maximum power consumption per kernel instance. . . . 134

C.1 Hardware configurations used for prediction of the hardware-usage of the *HH* kernel instances. . . . 157
C.2 Hardware configurations used for prediction of the hardware-usage of the *HH+custom* kernel instances. . . . 158
C.3 Hardware configurations used for prediction of the hardware-usage of the *HH+custom+multi* kernel instances. . . . 158
C.4 Hardware configurations used for prediction of the hardware-usage of the *HH+gap* kernel instances. . . . 159
C.5 Hardware configurations used for prediction of the hardware-usage of the *HH+custom+multi+gap* kernel instances. . . . 160

D.1 Power results for the *HH* kernel instance on a Maia DFE. The parameters uf, $N_{comps,max}$, and $N_{gates,max}$ are configuration parameters while N_{comps} and N_{Gates} are parameters of the simulation itself. . . . 161
D.2 Power results for the *HH+gap* kernel instance on a Maia DFE. The parameters uf, $N_{comps,max}$, and $N_{gates,max}$ are configuration parameters while N_{comps} and N_{gates} are parameters of the simulation itself. . . . 162
D.3 Power results for the *HH+custom* kernel instance on a Maia DFE. The parameters uf, $N_{comps,max}$, and $N_{gates,max}$ are configuration parameters while N_{comps} and N_{gates} are parameters of the simulation itself. . . . 163

D.4 Power results for the *HH+custom+multi* kernel instance on a Maia DFE. The parameters uf, $N_{comps,max}$, and $N_{gates,max}$ are configuration parameters while N_{comps} and N_{gates} are parameters of the simulation itself. . . 163
D.5 Power results for the *HH+custom+multi+gap* kernel instance on a Maia DFE. The parameters uf, $N_{comps,max}$, and $N_{gates,max}$ are configuration parameters while N_{comps} and N_{gates} are parameters of the simulation itself. 164

List of Acronyms

DFE Data-Flow Engine

HH Hodgkin-Huxley

eHH extended Hodgkin-Huxley

IO Inferior-Olive

ANN Artificial-neural-network

SNN Spiking-neural-network

FPGA Field Programmable Gate Array

HPC High Performance Computing

ASIC Application-Specific Integrated Circuit

SSP Strong Stability Preserving

GPU Graphics Processing Unit

CPU Central Processing Unit

ODE Ordinary Differential Equation

DRAM Dynamic Random-Access Memory

LMem Large Memory

FMem Fast Memory

SLiC Simple Live CPU

FIFO First In First Out

BRAM Block Random-Access Memory

I&F integrate-and-fire

Acknowledgements

First and foremost, I would like to thank my thesis advisor, Christos Strydis and his PhD student George Smaragdos of the Neuroscience department of Erasmus MC, for their guidance and advices during the project and for. Furthermore, I would like to thank Zaid Al-Ars, my supervisor at Delft University of Technology, for helping me in the search for an interesting thesis topic and his valuable comments on this work. Likewise, a big thank you also goes out to Matthias Möller, Mario Negrello, and Nils Voss for their advice during the implementation process. Finally, I would like to express my gratitude to my parents, family, friends and everyone who is/was part of the lab at Erasmus MC. This accomplishment would not have been possible without them. Thank you.

Rene Miedema
Spijkenisse
April 25, 2019

1 Introduction

Reverse engineering the brain is one of the grand challenges for engineering in the 21st century [4]. The first goal of reverse engineering the brain is to get a better understanding of the brain and the second goal is brain rescue and tackling brain disorders. A third, recurring goal, is the ability to build artificial intelligence. One of the fields studying the brain is computational neuroscience. The field of computational neuroscience seeks to understand the method by which biological-brain systems organize and process information. Computational neuroscience is supplementary to direct biological experiments, which are time-consuming and their results can be easily contaminated by environmental or other factors (e.g. the impact of anesthesia on the subject). Additionally, the current techniques have a limited potential in monitoring brain systems on a large enough scale to reveal the systemic properties of biological neuron networks. Yet, it is theorized that many systemic properties cannot be revealed though simple reduction of the neuronal system to its simpler parts. This is one of the issues that computational neuroscience attempts to solve by implementing biologically realistic in-silico simulations using mathematical models describing neuron behaviour (themselves derived by direct biological experiments). Hypotheses formulated using in-silico experimentation can subsequently be validated by more informed and guided biological tests and in-vivo or in-vitro experimentation.

Among the most widely used realistic models for such purposes are Spiking-neural-network (SNN) models [5, 6] of the Hodgkin-Huxley (HH) variety [7] (other formalisms exist as well such as Izhikevich, integrate-and-fire (I&F) model types [1]). The choice of SNN model depends on the subject of the study [1]. If a researcher seeks to explore the electrochemical characteristics that produce the neuron's response, a biophysically meaningful neuron model is required, such as the HH-model. HH models belong in the family of conductance-based models and capture closely the electrochemical behaviour that produces the neuron activity by modelling the various ion channels observed inside neurons. The ultra-high computational complexity of the HH-model and its variations are what makes such models challenging to simulate using traditional computing methods. What is more, these models typically form sets of Ordinary Differential Equations (ODEs) whose "solution" (i.e. simulation) forms the workloads that need to be executed.

This thesis focuses on developing a general, Data-Flow Engine (DFE) library for simulating five HH-model solver variants. Each of the five hardware ODE-solver implementations supports a different number and type of features which can be user-specified at simulation startup–i.e. not at design time–and at marginal or no performance cost to respective hard-coded designs. Furthermore, to investigate if the performance can be improved by the use of different solvers each of those five implementations has been adapted for the use with three different numerical ODE solvers.

1.1 Motivation

Computational neuroscientists currently can use simulation environments such as NEURON [8] and GENESIS [9] or frameworks such as NeuroML [10]. These environments are relatively easily to program, however, they lack high performance. Consequently, there have been developed a lot of high performance tools for simulations of neural networks (e.g. the work of M.A. Bhuiyan et al. [11]). However, with those implementations the usability for neuroscientists is low due to the required programming knowledge to use those implementations. As a solution, this work introduces a library which will deliver high performance and will be usable for neuroscientists. **The challenge lies in offering high performance and scalable libraries (so as to support the construction and simulation of large-scale brain models) while at the same time offering high degrees of modelling flexibility and parameterization.** While such flexibility is relatively easily tenable on software-based Central Processing Unit (CPU) (Phi) and Graphics Processing Unit (GPU) platforms, it is very challenging to achieve on a Field Programmable Gate Array (FPGA)-based platform, such as a DFE. Coding modelling flexibility in DFEs will allow neuroscientists to run their own simulations (and selecting their own parameters) while enjoying the high performance of an FPGA-based platform. The implementations of this thesis will address HH-type models, which are biophysically-meaningful models, with high computational complexity costs [1], as is shown in Figure 1.1.

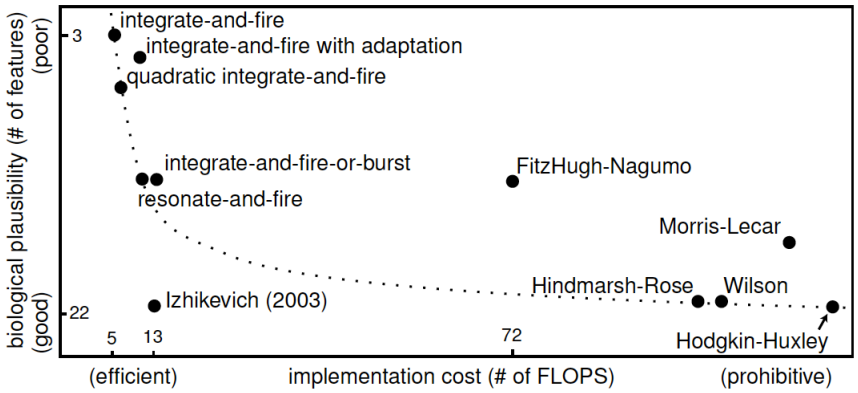


Figure 1.1: Graph plotting computational complexity involved versus achieved biological plausibility of various SNNs [1]. HH models are clearly the most demanding but also most detailed ones across the board.

1.2 Thesis scope and contributions

This work builds on top of BrainFrame [3, 12], a High Performance Computing (HPC) framework for accelerating computational-neuroscience experiments by incorporating multiple acceleration technologies (Intel Xeon-Phi CPU, NVidia GPU, Maxeler DFE). By employing a mix of heterogeneous accelerators, BrainFrame assigns the best accelerator to each provided model simulation by matching best accelerator features to model quirks. BrainFrame has been validated with hard-coded brain models in the past. But

for the framework to be useful in practice, neuroscientists must be able to develop their own models within BrainFrame using general libraries.

1.2.1 Thesis goal

The main goal of the thesis can be formulated as follows:

Develop a flexible hardware library implementing the HH neuron model in a hardware-accelerated and highly parameterizable fashion.

There is a great variety in HH-type models as different models support different features. We have decided to let the original, most popular HH-model be the simplest model the library is able to support and the most complex model to be the Inferior-Olive (IO) [13] (this model was used to evaluate the BrainFrame platform). Both models will be discussed in more detail in Chapter 2. The first challenge is how to generalize the equations of those models in such a way that the equations can be reused to capture different model instances, as per the ever-changing simulation needs. The equations need to be parametrized else a new synthesis cycle will be required for new simulations which will cancel out the performance gains by using the DFE. This fact has been the major hindrance for the pervasive use of FPGAs in the computational-neuroscience community and the pitfall of many attempts to publicize FPGA-based models. Furthermore, to increase user friendliness, it is convenient if the generalisation will be compliant with an already existing and accepted language (such as NeuroML) which is used by the neuroscientific community. The generalized equations will be used to calculate the derivatives of the so-called model state variables. The calculation of the derivatives is one part of a neural simulation. The other part is how the state variables are updated, which is done by numerical solvers. To enable a change between numerical methods, those two parts needs to be separated in the code. This results in the following subgoals:

1. Specify a general formulation of the equations in the considered HH-model variants.
2. Develop code which allows for relatively easy change of numerical ODE solvers.
3. Efficiently implement the generalized equations of the neural networks on the DFE.

1.2.2 Thesis contributions

The contributions of this thesis are as follows:

- A scalable, hardware library (called flexHH) of accelerated, parameterizable and NeuroML-compliant HH-model implementations which offer high-performance gains.
- A set of crucial model extensions: custom ion gates, gap junctions connectivity and multi-compartmental neurons.

- An in-depth functional validation and error analysis of the solvers both in reference software and in DFE hardware.
- An analysis of the difference between using three different ODE solvers for HH-type models.
- A simple prediction method for the resource usage of the different solver kernels on the DFE.
- A comprehensive performance and power/energy analysis of the library kernels implemented.

1.3 Thesis organization

The structure of the rest of this thesis is as follows: In Chapter 2, background information is given about the neural models considered in this work and the data-flow paradigm. In Chapter 3, related work other implementations on FPGAs and simulations tools used to stimulate neural networks are discussed. In Chapter 4, the implementation of the general conductance-based models on the DFE are discussed into a new library called flexHH. The evaluation of the flexHH kernel accuracy, performance, power, and resource-usage results are discussed in Chapter 5. Finally, Chapter 6 concludes this thesis with a discussion of the contributions and proposed future work.

2 Background

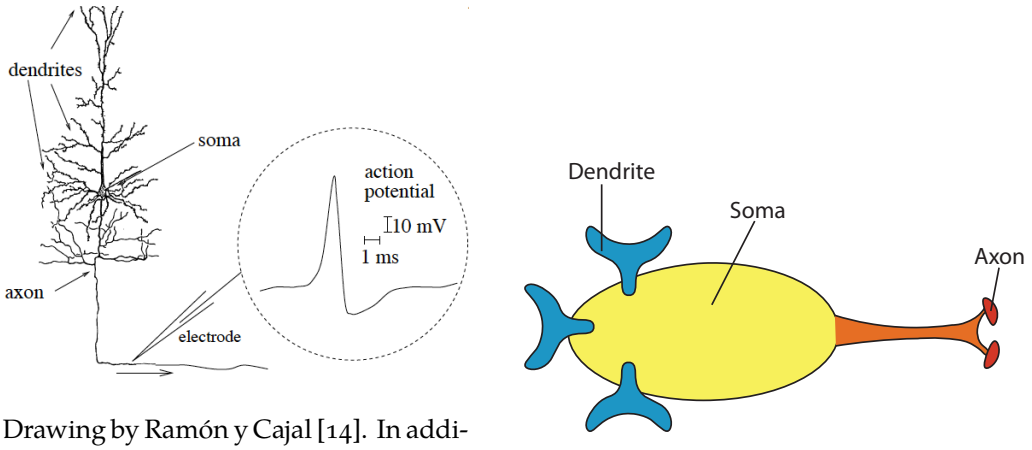
In this chapter, the background information required to understand the remainder of the thesis is presented. In section Section 2.1 the biological neuron is described. Section 2.2 discusses Spiking-neural-networks (SNNs), after which the Hodgkin-Huxley (HH) and Inferior-Olive (IO) models are discussed in more detail. In Section 2.3 a framework in which neural models can be described (NeuroML) is discussed. In Section 2.4 numerical methods, which are necessary for solving SNN models, are discussed.Section 2.5 describes data-flow computation. Finally, in Section 2.6 the BrainFrame framework is discussed.

2.1 The biological neuron

This thesis focuses on brain simulation. To get an idea what the brain looks like the biological neurons are shortly described. The brain consists of a network of neurons. There are different kind of neurons with different kinds of morphology. However, there is a classic way of describing a neuron. In this case the neuron consists of a dendrite, a soma, and an axon. In Figure 2.1 visual representations of a single neuron are shown. The dendrites can be described as the input stage of a single neuron, the soma as the processing unit, and the axon as the output device. Each of the compartments dendrite, a soma, and an axon have a membrane with ion channels. Consequently, depending if the ion channels are open or closed ions can go through those channels changing the voltage potential of the membrane. Depending, on the changes of the membrane voltage signals could be generated. The signals are electric pulses which are also called spikes. The information of the spikes is encoded in the shape and in the pattern of the spikes. The neuron receives a spike by its dendrites which propagates the spike to the soma. If the total of the incoming signals of a soma exceed the threshold, then a output signal is sent to the axon, which is in connection with other dendrites through synapses.
A synapse is the connection between two neurons. The first of two types of synapses are the chemical synapses. In a chemical synapse, bio-chemical processes lead to changing the ion influx. Subsequently, this leads to a change in the voltage of the membrane. The second type of synapses are electrical synapses or also called gap junctions [14].

2.2 Spiking neural networks

Having gotten a first idea of the neurons, the models which are simulated in this thesis are discussed. For this thesis, SNNs [15] are of interest. SNNs are typically more complex than Artificial-neural-networks (ANNs). ANNs abstractly represent neural networks while SNNs are more complex and more biologically accurate [5, 6]. Consequently, in computational neuroscience, SNNs are widely used nowadays.



(a) Drawing by Ramón y Cajal [14]. In addition, a spike (action potential) can be seen.

(b) Schematic representation

Figure 2.1: Visual representations of a single neuron, with the compartments specified.

There is a large variety of different SNN models. To see the differences between the models the paper of Izhikevich [1] is discussed. In this paper, different SNNs are classified based on 20 different characteristics which describe spiking dynamics together with the computational cost of the models. Additionally to the spiking dynamics, the way the neurons are connected in a network determines how accurate the models are. However, in the paper it is not discussed how the neurons are connected. Consequently, no conclusions can be made on how accurate the models are when modelling interconnected neurons. Nonetheless, the results of this paper can be used to indicate how biologically accurate the models are, which is an important characteristic of models targeted in this thesis. In Figure 2.2 a comparison of different models is given based on the 20 different characteristics and in Figure 1.1 a visual representation is given. From the results, it becomes clear that in general a model which is more biologically plausible is computationally more expensive, with as exception the Izhikevich model [16] which is relatively more efficient.

Although the Izhikevich model is the most efficient model regarding spiking dynamics, we are interested in the most complete model, which is the HH-model [17]. This because the Izhikevich model is not biophysically meaningful while the HH-model is. That is, the HH-model uses physics to describe the behaviour of the neuron while the Izhikevich model consists of only two equations (which are obtained by using bifurcation methodologies [18] on the HH-model) and has only one non-linear term. Additionally, there are more extensive HH models. HH models are very popular and the basis for modern computational neuroscience, however recent findings suggest that extended features need to be incorporated to the original HH-model, namely: multiple cell compartments, an ion concentration model, more complex ion channels, and gap junctions. One such in-house model that encompasses the above extra features and is also of high importance to the Neuroscience Department of the Erasmus MC is the

Models	biophysically meaningful	tonic spiking	phasic spiking	tonic bursting	phasic bursting	mixed mode	spike frequency adaptation	class 1 excitable	class 2 excitable	spike latency	subthreshold oscillations	resonator	integrator	rebound spike	rebound burst	threshold variability	bistability	DAP	accommodation	inhibition-induced spiking	inhibition-induced bursting	chaos	# of FLOPS
integrate-and-fire	-	+	-	-	-	-	-	+	-	-	-	-	+	-	-	-	-	-	-	-	-	-	5
integrate-and-fire with adapt.	-	+	-	-	-	-	+	+	-	-	-	-	+	-	-	-	-	+	-	-	-	-	10
integrate-and-fire-or-burst	-	+	+		+	-	+	+	-	-	-	-	+	+	+	-	+	+	-	-	-		13
resonate-and-fire	-	+	+	-	-	-	-	+	+	-	+	+	+	+	-	-	+	+	+	-	-	+	10
quadratic integrate-and-fire	-	+	-	-	-	-	-	+	-	+	-	-	+	-	-	+	+	-	-	-	-	-	7
Izhikevich (2003)	-	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	13
FitzHugh-Nagumo	-	+	+	-		-	-	+	-	+	+	+	-	+	-	+	+	-	+	+	-	-	72
Hindmarsh-Rose	-	+	+	+			+	+	+	+	+	+	+	+	+	+	+	+	+	+		+	120
Morris-Lecar	+	+	+	-		-	-	+	+	+	+	+	+	+		+	+	-	+	+	-	-	600
Wilson	-	+	+	+			+	+	+	+	+	+	+	+	+		+	+					180
Hodgkin-Huxley	+	+	+	+			+	+	+	+	+	+	+	+	+	+	+	+	+	+		+	1200

Figure 2.2: Overview of supported spiking behaviour for different models [1].

IO-model. As both the HH- and IO-model are used as reference models in this thesis, they will be discussed next.

2.2.1 The Hodgkin-Huxley model

The HH-model was published in 1952 and has had a significant influence on computational neuroscience. As shown in [1], it is still the most biologically accurate model for a single cell. The model gives a mathematical description of the axon of a squid of which the equations can be seen in Equations (2.1) to (2.14).

$$\frac{\mathrm{d}V}{\mathrm{d}t} = \frac{I_{app} - I_{channels} - I_{leak}}{C_M} \tag{2.1}$$

$$\frac{\mathrm{d}n}{\mathrm{d}t} = \alpha_n(1-n) - \beta_n n \tag{2.2}$$

$$\frac{\mathrm{d}m}{\mathrm{d}t} = \alpha_m(1-m) - \beta_m m \tag{2.3}$$

$$\frac{\mathrm{d}h}{\mathrm{d}t} = \alpha_h(1-h) - \beta_h h \tag{2.4}$$

where

$$\alpha_n = \frac{0.01(V+10)}{\exp(\frac{V+10}{10}) - 1} \tag{2.5}$$

$$\beta_n = 0.125\exp(V/80) \tag{2.6}$$

$$\alpha_m = \frac{0.1(V+25)}{\exp(\frac{V+25}{10}) - 1} \tag{2.7}$$

$$\beta_m = 4\exp(\frac{V}{18}) \tag{2.8}$$

$$\alpha_h = 0.07\exp(\frac{V}{20}) \tag{2.9}$$

$$\beta_h = \frac{1}{\exp(\frac{V+30}{10}) + 1} \tag{2.10}$$

$$I_{channels} = I_K + I_{Na} \tag{2.11}$$

$$I_{leak} = g_l(V - V_l) \tag{2.12}$$

where

$$I_K = g_K n^4 (V - V_K) \tag{2.13}$$

$$I_{Na} = g_{Na} m^3 h (V - V_{Na}) \tag{2.14}$$

The derivative of the voltage of the membrane (V) is the sum of currents which go through the membrane. I_{app} is the membrane current and can be modelled by any function as current clamp. I_{leak} is the leakage current and has a constant conductance. $I_{channels}$ is the sum of the currents generated by the ion channels (the Na and K channel in case of the HH-model). The conductances (conductance is the inverse of the electrical resistance) are dependent on the so-called gate-activation variables (n, m, and h in case of the HH-model). The gate-activation variables defines the proportion of ion gates in the total population which are open. Depending on the values of those variables the ionic currents $g_K n^4$ (G_K) and $g_{Na} m^3 h$ (G_{Na}) change. How the gate-activation variables change is described by Equations (2.2) to (2.4). The derivation of those equations were done so that they match the experiments and thus are biophysically meaningful.

2.2.2 The Inferior Olive model

The IO is crucial for functioning of the cerebellum. Its functionality is associated with learning, on line motor control [19], and has a role in timing independent of motor behaviour [20]. The model used to simulate IO cells is an extended HH-model and is developed by De Gruijl [13]. Because it is an extended HH-model the derivatives of the voltages are formulated by the sum of currents divided by a capacitance, however, the model also adds a few extensions.

The first extension of this model in comparison to the HH-model is the use of extra ion channels with different gates. Therefore, some of the derivatives of the gate-activation variables are described by more complex functions which can contain two exponents, instead of one in case of the HH-model. The second extension is that the derivatives of the gate variables not only vary with the voltage but also with the calcium concentration. The third extension is the use of multiple cell compartments. Where the HH-model only describes the axon, the IO-model describes a dendrite, a soma, and an axon. The adjacent compartments exchange current with each other. Meaning that there is a bi directional current between the dendrite and soma, and a current between the soma and axon. The fourth and final difference is the that the cells are connected with gap junctions. The gap junctions are the connections between cells and thus describe how the cells are connected and thus responsible for one of the two important issues of the network dynamics [1]. The gap junctions used in this model are described in [21]. For all the equations see Appendix A.

2.3 NeuroML

To simulate the previously discussed models, the models need to be implemented in code. Currently, a lot of neural models are implemented in different simulation environments (such as: MATLAB, NEURON, and C/C++). This makes it hard and time-consuming to reproduce, compare, and evaluate different neural models. Those properties are needed for neural modelling to become a greater scientific tool. NeuroML [10] is a solution as it is provides a framework in which neural models can be described independent of specific simulation environments.
NeuroML makes use of a hierarchical structure for the descriptions of the neural models. This hierarchy is visually presented in Figure 2.3. This shows that a network consists of cells. The cells consist of compartments, the compartments consists of channels, and the channels consist out of gates. Each component of this hierarchy is described by its own set of parameters which allows for the creation of heterogeneous neural networks. NeuroML offer just an XML declaration of the models. To simulate these modes, it needs to be input to one of a growing number of simulation environments, consequently making it relatively easy to compare the models. As NeuroML has been widely adopted by the neuroscience community, making our implementation NeuroML-compliant will increase the likelihood of it being used by neuroscientists.

2.4 Numerical methods

The neural models consist of Ordinary Differential Equations (ODEs), therefore, to simulate those models, numerical methods needs to be used. There are two different kinds of numerical methods, explicit and implicit methods [22]. The difference between explicit and implicit methods is that an explicit method calculates the next step in time based on the current state of the model, while an implicit method calculates the next step in time based on the current state and the next state of the model. Mathematically, an explicit method can be represented with Equation (2.15). On the other hand, to use

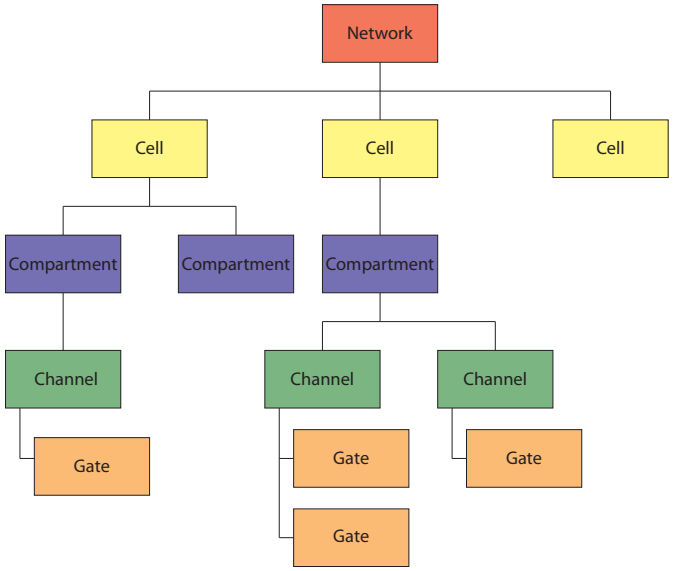


Figure 2.3: Hierarchical representation of network structure in NeuroML.

an implicit method an equation in the form of Equation (2.16) needs to be solved. The advantage of implicit methods is that the conditions on the time-step-size, which are required for a stable solution, are less severe. However, this comes at the cost of higher computational since root-finding algorithms, such as the Newton's Method [23], are required on top of the numerical method.

$$X_{n+1} = F(X_n) \tag{2.15}$$

$$G(X_n, X_{n+1}) = 0 \tag{2.16}$$

Both methods are approximations as they are a discretization of the real continued solutions. The errors of the approximate (the difference between the approximate solution and the actual outcome) are called either truncation or discretization errors. Besides the truncations error, there is a round-off error due to the finite precision of the representation of numbers on computers.

To analyse how accurate the numerical methods are, three concepts are introduced: consistency, stability, and convergence. Consistency means that the solution of a discrete problem which is solved by the numerical method is the same as the continuous problem. In other words, this means that the truncation error goes to zero if the discretization step goes to zero. A method is stable if the solution and therefore the error is bounded. If a method is both consistent and stable then the method converges, which means that when the discretization step goes to zero, the error between the real and discrete solution goes to zero.

A characteristic of an ODE system which influences how accurate a numerical method needs to be is the stiffness of the system. The stiffness of an ODE system is a measure of

how difficult the system is to solve with a specific accuracy (if a system is more difficult to solve, it is more stiff). The accuracy can be analysed though the local and global truncation error. The local truncation error is the error in a single step and the global truncation error is the error at a given time after n steps.

To keep the error small with stiff problems, in general implicit solvers are used, while with non-stiff problems explicit solvers are sufficient. The local error is dependent on both the time-step-size (Δt) and the order of the solver (p) as the error is equal to $\mathcal{O}(\Delta t^p)$. Thereby, if the solver is convergent then it holds that $\mathcal{O}(\Delta t^p)$ for the global truncation error [22]. Therefore, it may be that with a higher-order solver the same simulation can be done in fewer simulation steps in comparison to a lower-order solver. If the decrease in steps is greater than the increase of the computational cost then the use of a higher-order solver is beneficial in terms of performance.
The simplest numerical method to solve ODEs is the (first-order) forward-Euler method which is described by Equation (2.17).

$$X_{n+1} = X_n + \frac{\mathrm{d}X}{\mathrm{d}t}\Delta t \tag{2.17}$$

Gottlieb et. al. [24] introduced a method for explicit Strong Stability Preserving (SSP) higher-order Runge-Kutta methods. The equations for the second-order can be seen in Equations (2.18) to (2.19) and the equations of the third-order can be seen in Equations (2.20) to (2.22). These Runge-Kutta methods are explicit and therefore, relatively cheap to implement in hardware. The computational cost of the forward-Euler method and the Runge-Kutta methods scale linearly with the order of the methods. When the global truncation error will scale exponentially (as indicated by $\mathcal{O}(\Delta t^p)$) and thus the number of simulations steps will scale exponentially, then the Runge-Kutta methods use less computations in a simulation. However, as the HH-type models are classified as stiff problems, the Runge-Kutta methods may not have an exponential decrease in number of simulation steps compared to the forward-Euler method. As implicit models will require a root-finding algorithm which has high computational costs and consequently, is relatively expensive in hardware it has been decided not to look into implicit numerical methods.

$$X_1 = X_n + \frac{\mathrm{d}X}{\mathrm{d}t}\Delta t \tag{2.18}$$

$$X_{n+1} = \frac{1}{2}X_n + \frac{1}{2}X_1 + \frac{1}{2}\frac{\mathrm{d}X}{\mathrm{d}t}\Delta t \tag{2.19}$$

$$X_1 = X_n + \frac{\mathrm{d}X}{\mathrm{d}t}\Delta t \tag{2.20}$$

$$X_2 = \frac{3}{4}X_n + \frac{1}{4}X_1 + \frac{1}{4}\frac{\mathrm{d}X}{\mathrm{d}t}\Delta t \tag{2.21}$$

$$X_{n+1} = \frac{1}{3}X_n + \frac{2}{3}X_2 + \frac{2}{3}\frac{\mathrm{d}X}{\mathrm{d}t} \tag{2.22}$$

2.5 Data-flow computing

In the last decades, the performance of computers has increased enormously. The increase in performance was caused among other things, by the increase in frequencies of the processors. However, the increase in frequency has slowed down because of the power usage. Previously, it was possible to lower the voltage, to lower the power usage, while increasing the frequency. However, lowering the voltage, and thus power, of the current state of the are Central Processing Units (CPUs) will lead to a too high leakage current. This problem is know as the power wall.

As a solution, the change from sequential computing to parallel computing was made, in the form of multi-core CPUs and additionally Graphics Processing Units (GPUs). However, the speedup is limited by how much of a program can be executed in parallel, as is described by Ahmdal's law [25]. Thereby, even if the application can be executed completely in parallel, then the memory may become a huge bottleneck.

On the other hand there is the Multiscale Data-flow Computing paradigm. In this paradigm there are no instructions, as data is streamed into a chip, and then the functionality is dependent on the structure of the functional units. This is because, contrary to the Von-Neumann architectures, after data leaves a functional unit it goes to the next functional unit without any interaction of the memory. Therefore, data-flow computation can also be called computation in time, while the Von-Neumann architecture is an example of computation in space. The difference in the architectures of both systems can be seen Figure 2.4.
Maxeler has developed a system which uses the Multiscale Data-flow Computing paradigm. The architecture, which can be seen in Figure 2.5, consists of a CPU host with its own Dynamic Random-Access Memory (DRAM) and one or multiple Data-Flow Engines (DFEs). A DFE is an Field Programmable Gate Array (FPGA) together with on-board DRAM on which the kernels are implemented in hardware. The data transfers to the DFE are conducted in one of two methods:

- via the CPU through a PCIe interconnect.
- via the on-board DRAM which is called the Large Memory (LMem).

The DFE has another memory which is called Fast Memory (FMem) which are the Block Random-Access Memories (BRAMs) of the FPGA. Besides the connections of the DFE to the CPU and LMem, it is possible to connect to other DFEs via the so called MaxRing interconnect. This is a high bandwidth interconnect which supports full overlap of computation and communication so that that applications on multiple DFEs scale efficiently.
The programming of the DFE is done with at least one kernel and a manager. In the kernel the functional part is defined, while in the manager the data movement is set up. Both the kernel and the manager are programmed with MaxJ which is an extended version of Java. On the CPU host the Simple Live CPU (SLiC) interface is used. This interface is automatically generated from the code in the Manager and is used to activate the kernels and data transfer between the CPU and DFE.

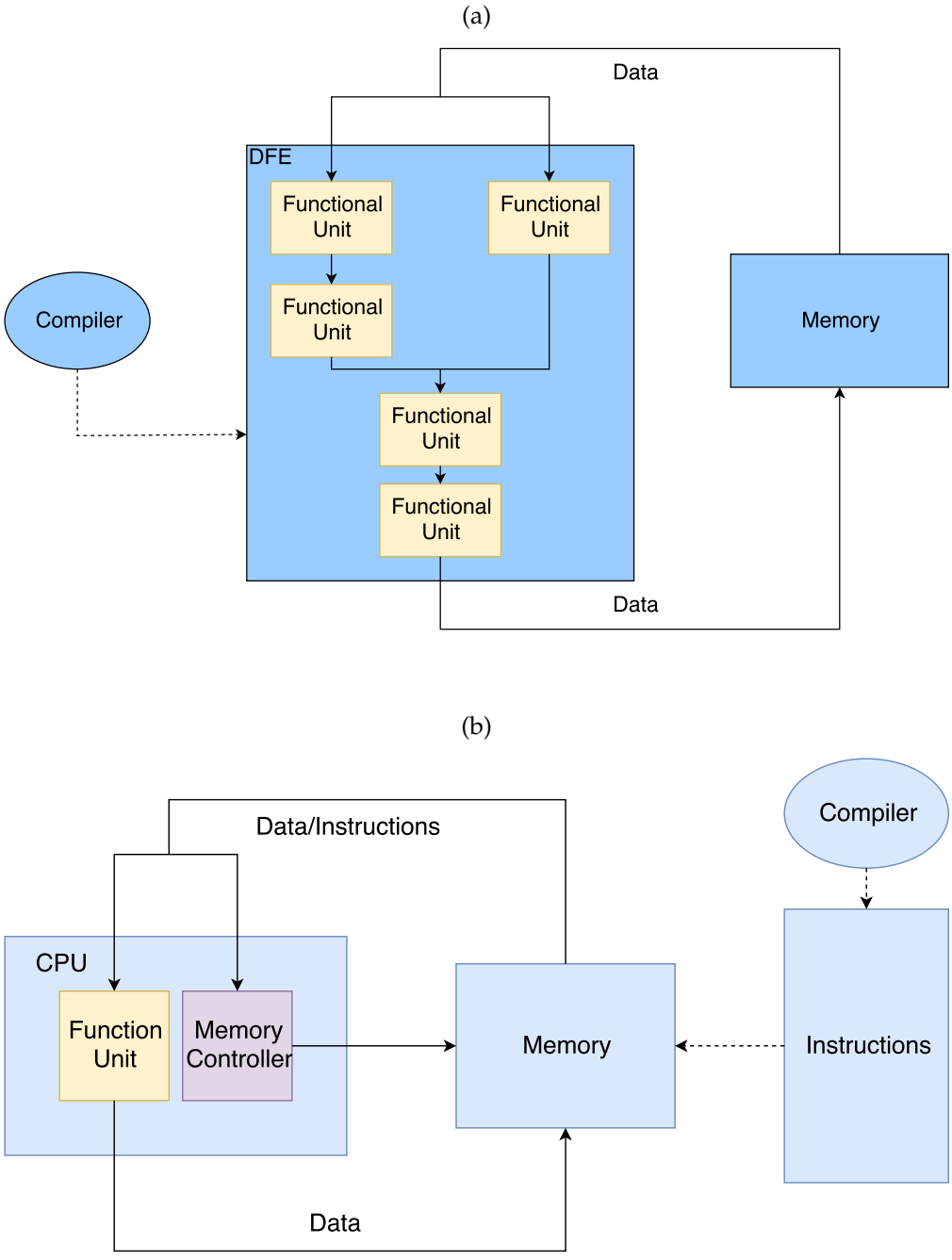


Figure 2.4: Architectures of different computer paradigms. (a) Data-flow. (b) Control flow

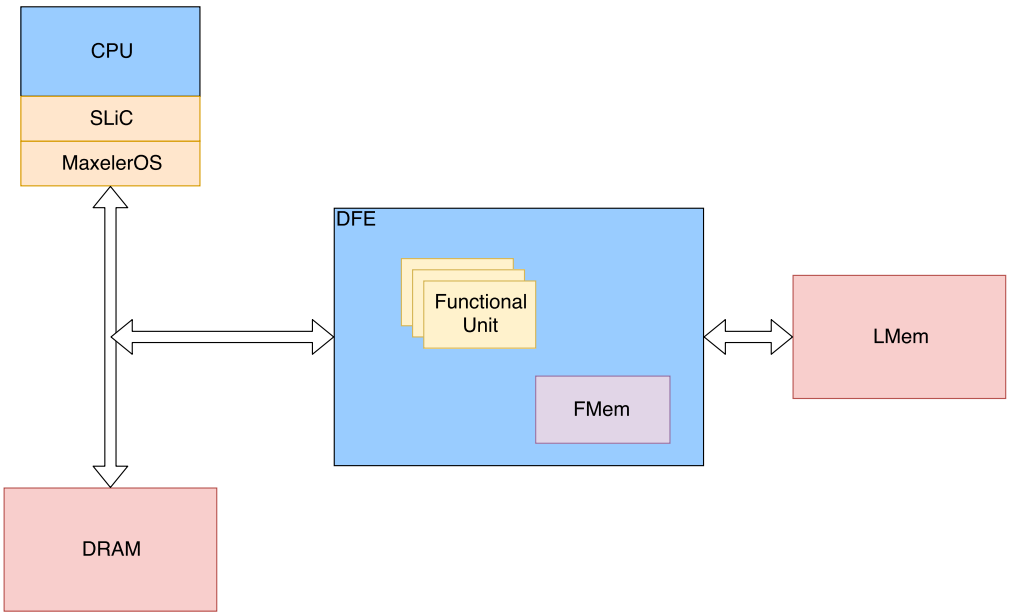


Figure 2.5: Maxeler data-flow architecture.

2.6 BrainFrame

BrainFrame [3] is a High Performance Computing (HPC) framework for accelerating computational-neuroscience simulations by incorporating multiple acceleration technologies (Intel Xeon-Phi CPU, NVidia GPU, Maxeler DFE), using PyNN [26], or since recently using NeuroML, as an interface. PyNN is an interface which uses Python scripts to use multiple simulators. Therefore, allowing for an increase in productivity of neural network modelling and an increase in reliability of modelling studies. BrainFrame assigns the best accelerator to each provided model simulation by matching best accelerator features to model quirks (network size and connectivity). BrainFrame has been validated with hard-coded brain models in the past. The hard-coded DFE implementation only supports the IO-model and was first introduced in [12] by Smaragdos et al. In this implementation all of the compartments of a single cell could be executed in parallel, as each compartment has its own pipeline. However, the gap junctions were shown to be the most computationally expensive part of the simulation due to there quadratic computational cost in relation to the number of cells in the network while the other computations scale linearly. This also explains why unrolling the loop of the gap junctions is a more efficient way to achieve speedup than duplicating the kernel on the DFE. This DFE implementation showed great performance with certain quirks in comparison to the CPU (Phi) and GPU platforms. However, for the framework to be useful in practice, neuroscientists must be able to develop their own models within BrainFrame using general libraries. As discussed previously, the challenge lies in offering high performance and scalable libraries so as to support the construction and simulation of large-scale brain models while at the same time offering high degrees of

modelling flexibility and parameterization.

2.7 Summary

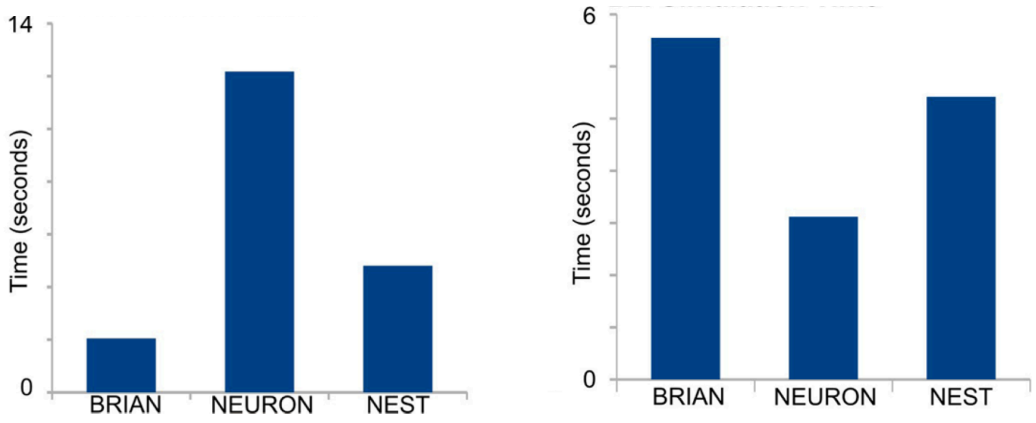
To get an idea of the simulations run in this thesis, this chapter started with a description of the biological neuron. After this SNNs, with in particular the HH- and IO-model, were discussed. Subsequently, NeuroML, a general language to describe neural models was described. Making our implementation NeuroML-complaint will increase the chance of our implementation being used. Afterwards, numerical methods were discussed. Those methods influence the accuracy of the simulations and consequently, the number of steps. The number of simulation steps have, in and of themselves, an influence on the execution time of the simulation. Furthermore, data-flow computing was discussed which is another computing paradigm to the Von-Neumann (or control-flow) paradigm, on which programs can be efficiently computed. Finally, the HPC framework for simulations of neural models BrainFrame was described. This framework showed great performance, however, the DFE implementation is not flexible. Consequently, flexHH will be a great addition to BrainFrame.

3 Related work

Currently, neuroscientists already have the choice of multiple brain simulation simulators to simulate their models. To indicate where those simulators can be improved, Section 3.1 discusses the most popular brain software simulators. Subsequently, in Section 3.2 other implementations on FPGAs are presented.

3.1 Software simulators

Tikidji-Hamburyan et al. [2] discuss the three most popular neural network simulators, based on frequency of occurrence in Model DB [27]: NEURON [8], GENESIS [9], and Brian [28]. Additionally, Nest [29] is added to the comparison. These simulation tools attempt to make the programming of the models, including using numerical methods, transparent to the user, which is useful for neuroscientists without a special expertise in programming and/or numerical methods.
To compare the computational performance of the different simulators Tikidji-Hamburyan et al. investigated the simulation time of NEURON, Brian, and Nest for two single-threaded test cases (GENESIS was not included in these simulations as Tikidji-Hamburyan et al. could not find a way to simply implement both test cases) run on a CPU (dual-core Intel Core i5 2.70 GHz, 16 Gb RAM). The first test case simulates a simplified network with simple integrate-and-fire (I&F) neurons and synapses (this network is discussed in more detail in [30] and [31]). The second test case is the simulation of a so-called PIR-ING network [32], which is based on a more complex Hodgkin-Huxley (HH)-type model. The performance of both tests cases show significantly different behaviour, as can be seen in Figure 3.1. This figure shows that Brian performs best and NEURON the worst for the first simple use case, while in the second use case Brian performs worst and NEURON best. Hence, the aforementioned simulators cannot guarantee a specific performance, which can be explained as the simulators are designed with different model types in mind. Therefore, this strengthens the decision to make the flexHH library to compatible with a language (NeuroML) which is simulator independent.
To make use of the efficient use simulator it may be required that the user has to develop low-level code (C/C++) or make use of external modules. For example, to efficiently program NEST the user has to develop in C++ (e.g. for using lookup tables instead of doing computations) and if in GENESIS an equation is not supported the user has to develop C-code. Consequently, it can be concluded that even these simulators have their limitations regarding user-friendliness. Furthermore, as some models are computationally expensive NEURON, NEST, and Brian support parallel-execution as can be seen in Table 3.1. However, this introduces limitations in functionality. For example, Brian has no MPI support and Nest does not support distributed computations for complex multi-compartmental neurons on several clusters through MPI. Consequently



(a) Test case 1

(b) Test case 2

Figure 3.1: Simulation time for different simulators [2].

we can conclude that the software simulators for neural networks have the following limitations:

- Limited exploited parallelism:
  All simulators make attempts to use parallelization/optimization features which, however, are not fully supported across all simulator functionality.
- Model-specific optimizations:
  Simulation speeds are not consistent across simulators (since each simulator has been designed with different model types in mind).
- Poor code portability:
  As discussed in Section 2.3, interoperability among different simulators is limited. Thus, effort spent on building a model in one simulator/language cannot be (easily) migrated to another simulator. This problem is nowadays being addressed by efforts like NeuroML and PyNN, which have been developed with the aim of standardizing the modelling of neuron descriptions and making model-code portable across different simulators.

We thus, see a need for High Performance Computing (HPC)-enabled simulators, with high levels of model-code portability across simulators. In the following section, we will discuss a subset (Field Programmable Gate Array (FPGA)-based simulators) of these HPC-enabled simulators.

3.2 FPGA-based simulators

As far as hardware simulators are concerned, we focus on solutions with the following properties:

Table 3.1: Characterization of computational efficiency and parallelization of neural network simulators [2].

Feature	NEURON	Nest	Brian	GENESIS
MPI support for neuron-to-neuron communication	Y	Y	N	Y
MPI support for gap junctions	Y	Y	N	N
Distributed computations for complex multi-compartmental neurons on several clusters nodes through MPI	Y	N	N	?
Multithreading support on single compute node	Y (p-threads)	Y (OpenMP)	Y (OpenMP or GPU(limited))	?

1. FPGA-based simulators:
   There is chosen for FPGA-based simulators due to the element of flexibility they offer compared to for example, Application-Specific Integrated Circuit (ASIC) solutions.
2. Simulators using floating-points:
   This because as shown in [33], the differences between a 32-bit fixed-point model and a 32-bit floating-point model can lead to a significance phase difference between the neuron spikes. As the functional behaviour of a neuron is defined by the spikes, the principles of neural information processing could be altered. Consequently, without a comprehensive study of the accuracy, the functional correctness of a fixed-point representation for the class of models we assume in this work cannot be guaranteed. However, it must be noted that the use of fixed-point arithmetic can lead to a significant reduction of hardware-usage on a FPGA and therefore, the use of fixed-point arithmetic is potentially preferable if a tolerable accuracy can be achieved.
3. Simulators using Spiking-neural-network (SNN) models:
   There is chosen to focus on SNN models due to their high potential in constructing biologically plausible neural simulations.

Blair et al. [34] introduce an FPGA platform for the simulation of I&F models. This implementation uses an XML language to describe the models and thus enables neu-

roscientists to directly translate the model on FPGAs. However, to achieve optimal performance, design parameters which are used for loop unrolling need to be modified. Those design parameters are dependent on which part of the algorithm is the bottleneck for parallel execution. This does require competence on parallel programming to accomplish, that the neuroscientist might not posses. Furthermore, this platform only simulates I&F models, which are simpler than the HH models.

Another implementation by Graas et al. [35] implements two conductance-based models, namely the original HH-model and a model of a motorneuron as a two-compartment model as is presented in Booth and Rinzel [36]. However, a relatively simple FPGA (Xilinx XC2V1000-4FG256 Virtex-II FPGA, 40 MHz) was used. Still, significant speedup, 16x for the HH-model and 72x for the motorneuron, could be achieved in comparison to a CPU (AMD Athlon processor, 1333MHz). Although Simulink was used to generate the model descriptions, understanding of the hardware of the FPGA was required to make efficient use of the pipeline. Additionally, if a new model needs to be implemented, a new synthesis cycle is required (which is time-consuming).

An implementation which can simulate large networks is presented by M.A. Bhuiyan et al. [11]. This implementation can simulate a network of HH up to 0.5 million (720x720) neurons. The hardware used for this implementation is the SRC 7 H MAP, which contains an Intel Xeon dual core processor together with two 150 MHz Altera Stratix II EP2S180 FPGAs. In the implementation, both the CPU and one FPGA are used. A maximum performance speedup of 38x against the Intel dual core Xeon was achieved. Although the implementation supports large networks, which can be simulated with a good performance, the usability for neuroscientists is again low.

The main problem with all previously discussed implementations is that they require expert acceleration knowledge to be used optimally, thus requiring the modification of an acceleration engineer, which can significantly delay the research process.

R.K.Weinstein et al. [37] introduce an architecture which describes the algorithms used in conductance-based models in a more generalized way. They created their own modelling language called DYNAMO together with their own compiler. Using this new interface incurred effort, even though this environment was functionally complete, thus the platform failed to become widely adopted.

Cheung et al. [38] introduces the most promising solution called NeuroFlow. NeuroFlow is a platform which uses PyNN as a high-level API which is understandable for the neuroscientist. Moreover, the hardware mapping, with optimizations such as the degree of parallelism, of the neural methods is done automatically. Therefore, the performance of the hardware system can be used relatively easily. The platform translates the derivatives of the neural model, which are derived from the model in PyNN, to Maxeler-code. Consequently, as one equation of the derivatives is changed, a new time-consuming (multiple hours) synthesis cycle is required. The performance and efficiency analysis for NeuroFlow is presented for a single use case of a generally simpler model (Izhikevich) with relatively low connectivity density (about 10%), showing impressive results. The behaviour 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 modelling [3]. Furthermore, the applicability of NeuroFlow in the case of (especially gap-junctioned) HH models is very limited due to

relying only on event-driven simulations, which is impractical for HH models. From the above, it becomes apparent that no prior work has tackled the problem of flexible and fast HH-model simulations on a FPGA-based platform showing significant acceleration results.

4 Implementation

This chapter discusses the first version of flexHH, targeting a single Data-Flow Engine (DFE) node. The structure of the remainder of this chapter is as follows. First, preliminary work for this thesis conducted by me is discussed. Secondly, an overview of the five kernels implemented is given. Then, in Section 4.3 the way equations of Hodgkin-Huxley (HH)-type models are generalized is discussed and in Section 4.4 the implementation of those functions on a DFE is discussed. Both sections start with the discussion of the general implementation of the basic *HH* kernel, which is used as a foundation for all the other kernels, after which the extensions are discussed.

4.1 Preliminary work

In this section, we will present some preliminary work of a DFE-based Ordinary Differential Equation (ODE) solvers, conducted by me prior to the work presented in Chapter 4. Four different ODE solvers used in this preliminary work are:

- Forward-Euler method (fwd-Euler)
- Modified-Euler method (mod-Euler)
- Optimal second-order SSP Runge-Kutta method (rk2)
- Optimal third-order SSP Runga-Kutta method (rk3)

With the use of those solvers, Equation (4.1) and Equation (4.2) were simulated with a variable number of elements ($N_{elements}$) as Algorithm 1 shows.

$$f(u, v) = 0.4u - 0.0002uv \tag{4.1}$$
$$g(u, v) = 0.3v - 0.0001uv \tag{4.2}$$

The state variables u_i and v_i (where $0 \leq i < N_{elements-1}$) were simulated, with Equation (4.1) and Equation (4.2), as can be seen in Algorithm 1.
In the preliminary work the used ODE solvers were implemented (by me) on the DFE and had the following characteristics:

1 An efficient use of the pipeline, meaning that all the stages of the pipeline where concurrently active.

2 Optimized performance of the implementations by increasing the frequency and

   (a) unrolling the loop of elements in hardware (with an unroll factor uf_e),

Algorithm 1 Pseudocode for the simulation used in the preliminary work.

```
1: procedure SIM(u, v, dt, N_steps, N_elements)
2:     for 0 ≤ j < N_elements do
3:         for 0 ≤ i < N_steps do
4:             uOld = u[j]
5:             vOld = v[j]
6:             u[j] ← updateU(uOld, dt, f(uOld, vOld))
7:             v[j] ← updateV(vOld, dt, g(uOld, vOld))
8:         end for
9:     end for
10: end procedure
```

(b) unrolling the number of simulation steps in hardware (with an unroll factor uf_s)

3 Usage of the PCI express bus for the data transfer during the simulation.

The maximum speedup, against a sequential C implementation run on a Central Processing Unit (CPU), can be seen in Table 4.1. Those results show that both the increase of frequency and unrolling the loops leads to significant performance benefits. The following conclusion can be made from these results:

- The more complex solvers have better performance if the same unroll factors are used. This is because a more complex solver has a deeper pipeline and therefore more pipeline stages are working concurrently.
- Both unrolling methods create multiple instances of `updateU` and `updateV` in hardware. Consequently, a higher unroll factor will lead to more hardware-usage. The unroll factor uf_s was limited by the amount of hardware on a DFE. On the other hand, uf_e is limited by the limited amount of data streams which can be present on the PCI express bus, while there still was unused hardware on the DFE. Therefore, the unroll factor uf_s could be increased more than uf_e.
- A simpler solver allows for a larger unroll factor, in comparison to the more complex solvers. Moreover, the larger unroll factor for the simpler solvers, provides a better speedup against the CPU than the more complex solvers with lower unroll factors.

Finally, the execution times of the DFE instances showed a large variance due to the data transfer during the simulation on the PCI express. Consequently, the kernels were re-implemented to make use of the Large Memory (LMem) (instead of the PCI express). Those execution times showed a more stable kernel performance.

Table 4.1: Maximum speedup for different implementations on the DFE in comparison with the C-code on the CPU host.

ODE solver	f (MHz)	uf_e	uf_s	max speedup
fwd	100	1	1	0.61
	350	1	1	2.04
	250	6	1	8.70
	200	1	50	57.34
mod	100	1	1	1.31
	340	1	1	4.23
	250	6	1	18.75
	200	1	15	37.29
ssp2	100	1	1	1.25
	340	1	1	4.02
	250	6	1	17.68
	200	1	20	46.72
ssp3	100	1	1	1.87
	340	1	1	6.00
	250	6	1	26.37
	200	1	12	42.04

4.2 The flexHH library

The models we want to simulate can be as simple as a network of HH cells or as complex as Inferior-Olive (IO) cells. The IO-model extends the basic HH-model with three extra features:

1. **Custom ion gates**:
   The description 'custom' follows from the fact that to support all gate equations of the IO-model, some equations are custom defined in NeuroML, this in comparison to gate equations of the HH-model, which can be described by predefined equations in NeuroML.
2. **Gap junctions**:
   Gap junctions are inter-cellular connections.
3. **Multiple cell compartments**:
   The feature "multiple cell compartments" allows a neuron cell to consist of one or more compartments (which have their own membrane voltages) allowing for a current flow between them.

The features are implemented as extensions. The extensions come at the cost of the use

Table 4.2: Overview of the supported features per implemented kernel in the flexHH-library.

	Custom ion gates	Gap junctions	Multiple cell compartments	ODE solvers
HH	✗	✗	✗	fwd-Euler rk2 rk3
HH+gap	✗	✓	✗	fwd-Euler rk2 rk3
HH+custom	✓	✗	✗	fwd-Euler rk2 rk3
HH+custom+multi	✓	✗	✓	fwd-Euler rk2 rk3
HH+custom+multi+gap	✓	✓	✓	fwd-Euler rk2 rk3

of extra hardware resources. Those resources could be used for increasing performance or network capacity. Therefore, flexHH will provide five different kernel instances of the HH representation. Each instance incorporates more or less a superset of features compared to its predecessor. Consequently, flexHH gives the option of using simpler kernels (depending on the simulation) with a benefit in performance and/or maximum network capacity. The five kernel instances together with which feature each kernel instance supports can be seen in Table 4.2. The most basic kernel supports the basic HH-model. The *HH+custom+multi+gap* kernel supports all features and is able to simulate the IO-model.

The feature "multiple cell compartments" states that there can be one or more compartments per cell. However, as can be seen in Table 4.2, not all kernel instances support this. In the case that multiple cell compartments are not supported, the terms cell and compartment are interchangeable.

The HH neuron models are generally described as ODE systems. These systems are represented by so-called state variables. In the case of the discussed neuron models, these state variables comprise the membrane potentials of the compartments (V_i) and gate-activation variables (y_i), where i is the index of the variable. The index can be a combination of multiple integers; for example, to represent gate h of compartment k of

cell j the index (j, k, h) can be used. Those state variables are updated as described in Algorithm 2 when a first-order solver is used. In this algorithm the voltages are stored in array V and the gate-activation variables in array Y. When an higher-order explicit solver is used, each (time) step contains multiple stages and therefore, another loop needs to be added. The use of different solvers is discussed in Section 4.4.1.4.

Algorithm 2 Pseudocode for the simulation of a HH-type 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, cellConsts, V, dt)
8:         end for
9:     end for
10: end for
```

The algorithm shows that, for each simulation, the solver is invoked for updating the neural network for a predefined number of simulation 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 the simulations steps, an `updateY` function is called which iteratively updates the values of the gate-activation variables y_i. For each compartment (of each cell), a second function `updateV` is called for updating the membrane potential V_i. The other parameters (`gateConsts`, `compConsts`, `cellConsts`) are constant parameters (during a single simulation) per gate, compartment, and cell, respectively. The description of `updateY`, `updateV`, and the constant parameters will be discussed in the remainder of this chapter.

4.3 Model function generalization

To implement generalized (thus, reusable) kernels on the DFE (in hardware) it is required to generalize the functions, through parameters used for the simulation of HH-like models. Otherwise, in case non-generalized functions are used, each time a new/ different equation is used for the simulation, a time-consuming synthesis cycle would be required. To decide which equations and what parameters to use, NeuroML is used as a guide. We rely on NeuroML since it has done an excellent job of hierarchically structuring neuron models; see Figure 2.3.
Making the kernels NeuroML-compatible has the additional benefit of being familiar for neuroscientists. The first thing to take from NeuroML is the hierarchal description of neural models. Furthermore, in NeuroML an effort is made to generalize the equations used in neural networks. This generalization is done using predefined functions which can be altered by changing the parameters. This suits a hardware implementation because the functionality is predefined while parameters can be changed and thus reusable for multiple simulations.

4.3.1 HH

The original HH-model is the foundation of all other HH-like models and was described in Section 2.2.1. In the equations of the HH-model (Equations (2.1) to (2.14)) it can be seen that the derivative of the voltage ($\frac{dV}{dt}$) is a summation of different currents divided by the capacitance of the membrane, where I_{app} is the applied current, $I_{channels}$ is the sum of the currents generated by the ion channels (the Na and K channel in case of the HH-model), and I_{leak} the leakage current. The equation to calculate $\frac{dV}{dt}$ does not change between the simulations and consequently, this equation does can be used directly.
The first current I_{app} can be represented by any function. However, the support for any function in hardware is impossible as it will use too many resources. Therefore, it was decided to only support pulse functions, which in NeuroML is represented by creating a `pulseGenerator`. The pulse function has three parameters: a start time (t_{start}), an end time (t_{end}), and an amplitude (A), as can be seen in Equation (4.3).

$$I_{app}(t) = \begin{cases} A, & \text{if } t_{start} \leq t < t_{end} \\ 0, & \text{otherwise} \end{cases} \tag{4.3}$$

The next current needed to calculate $\frac{dV}{dt}$ is the current relating to the channels $I_{channels}$. This current is the sum of the current flowing through all the ion channels. In the HH-model there are two ion channels, the Na and K channel. Both channel currents (I_{Na} and I_K) and every other channel current in an IO cell can be represented with Equation (4.4). In this equation, $M_{gates}[i]$ is different from the $N_{gates}[j][k]$ in Algorithm 2, as $M_{gates}[i]$ is the number of gates per channel and $N_{gates}[j][k]$ is the number of gates per compartment. Moreover, from the documentation from NeuroML (see the `ionChannelHH` and `channelDensity` objects in the documentation) it follows that Equation (4.4) is indeed a general equation for $I_{channels}$. As an explanatory example, Equation (4.4) is used to represent the equations of both channel currents of the HH-model as shown in Table 4.3. The final current needed is I_{leak}, which is simpler than $I_{channel}$ as the calculation of the current does not require an exponent and there is always only one gate (see Equation (4.5)). Therefore, the equation is always the same and thus the equation of I_{leak} does not need to be generalized. So, all the functions which are required to calculate the derivative of the voltages are now general enough to be implemented in hardware.

$$I_{channel} = g_{channel} \prod_{i=0}^{M_{gates[i]}-1} y_i^{p_i} (V - V_{channel}) \tag{4.4}$$

where

$$p_i \in \mathbb{Z}_{>0}$$
$$I_{leak} = g_{leak}(V - V_{leak}) \tag{4.5}$$

The equations for the other derivatives, used for the gate-activation variables, in the HH-model are described by Equations (2.2) to (2.4). Those equations show that each

Table 4.3: Parameters of the HH-model filled into Equation (4.4).

channel	$g_{channel}$	M_{gates}	y_1	y_2	p_1	p_2	$V_{channel}$
K	g_K	1	n		4		V_K
Na	g_{Na}	2	m	h	3	1	V_{Na}

differential equation of the gate-activation variables is in the form of Equation (4.6). To represent Equations (2.5) to (2.10), three different functions, which are presented in Equation (4.7), are needed. In this equation, x_1, x_2, and x_3 are floating-point values to represent the variables of the equation and $fType$ is an integer value to select a function branch. In NeuroML, also three different functions are implemented to represent the equations of the derivatives of the gate-activation variables. These are shown in Equations (4.8) to (4.10). Coincidentally, it is possible to present them with Equation (4.7). The way the variables relate to each other can be seen in Table 4.4. Consequently, no additional changes are required to make equations of the gate-activation derivatives of the flexHH library compatible with NeuroML.

$$\frac{\mathrm{d}y_i}{\mathrm{d}t} = \alpha_i(1 - y_i) - \beta_i y_i \tag{4.6}$$

$$f(V_i, x_1, x_2, x_3, fType) = \begin{cases} \dfrac{x_1 \cdot (x_2 - V_i)}{e^{(x_2 - V_i)\cdot x_3} - 1} & \text{if } fType = 0 \\ x_1 \cdot e^{(x_2 - V_i)\cdot x_3} & \text{if } fType = 1 \\ \dfrac{x_1}{e^{(x_2 - V_i)\cdot x_3} + 1} & \text{if } fType = 2 \end{cases} \tag{4.7}$$

$$hhSigmoidRate = \frac{rate}{1 + \exp\dfrac{midpoint - V_i}{scale}} \tag{4.8}$$

$$hhExpLinearRate = \frac{rate\dfrac{midpoint - V_i}{scale}}{\exp\dfrac{midpoint - V_i}{scale} - 1} \tag{4.9}$$

$$hhExpRate = rate \exp\frac{V_i - midpoint}{scale} \tag{4.10}$$

Table 4.4: Parameter translation from NeuroML to the implementation on the DFE.

Function	$fType$	x_1	x_2	x_3
$hhSigmoidRate$	2	$rate$	$midpoint$	$\frac{1}{scale}$
$hhExpLinearRate$	0	$\frac{rate}{scale}$	$midpoint$	$\frac{1}{scale}$
$hhExpRate$	1	$rate$	$midpoint$	$\frac{-1}{scale}$

4.3.2 Custom ion gates

As discussed before, the IO-model requires the use of custom-defined ion gates in NeuroML. The equations which require a custom definition can be seen in Equations (4.11) to (4.15). In these equations s_d, q_d, l_s, and n_s are gate variables. $Ca2Plus_d$ is the concentration of the Ca^{2+} ion concentration in the dendrite and thus adding dynamics of a calcium concentration in addition to the dynamics of gate-activation variables. The current $I_{cah,d}$ is the high-threshold calcium current in the dendrite. These equations reveal that the differences between the standard gate equations (as discussed above) and these used in the IO-model are: (a) that a equation can contain multiple exponent functions and (b) that the equation can be a `min` function. Additionally, in case of the standard gates, the voltage is always used as input for the function, while the custom gates can also use the calcium current or another gate-activation variable as input.

$$\frac{\mathrm{d}s_d}{\mathrm{d}t} = min(0.00002 \cdot Ca2Plus_d, 0.01) \cdot (1 - s_d) - 0.015 \cdot s_d \tag{4.11}$$

$$\frac{\mathrm{d}q_d}{\mathrm{d}t} = \frac{\frac{1}{1 + e^{\frac{V_{dend} + 80}{4}}} - q_d}{\frac{1}{e^{-0.086 \cdot V_{dend} - 14.6}} + e^{0.070 \cdot V_{dend} - 1.87}} \tag{4.12}$$

$$\frac{\mathrm{d}Ca2Plus_d}{\mathrm{d}t} = -3 \cdot I_{cah,d} - 0.075 \cdot Ca2Plus_d \tag{4.13}$$

$$\frac{\mathrm{d}l_s}{\mathrm{d}t} = \frac{\frac{1}{1 + e^{-\frac{V_{soma} + 85.5}{-8.5}}} - l_s}{\frac{20 \cdot e^{\frac{V_{soma} + 160}{30}}}{1 + e^{\frac{V_{soma} + 84}{7.3}}} + 35} \tag{4.14}$$

$$fCustom(fType, V, xs) = \begin{cases} \dfrac{x_5(xs[1]-V)}{x_0\exp((x_1-V)x_2)+x_3} + x_8 & \text{if } fType = 0 \\ \dfrac{x_8}{x_0\exp(x_2(x_1-V))+x_3+x_4\exp(x_5(x_6-V))+x_7} & \text{if } fType = 1 \\ \dfrac{x_0\exp((x_1-V)x_2)+x_3}{x_4\exp((x_6-V)x_5)+x_7} + x_8 & \text{if } fType = 2 \\ min(x_0 v, x_1) & \text{if } fType = 3 \end{cases} \tag{4.16}$$

$$\frac{\mathrm{d}n_s}{\mathrm{d}t} = \frac{\dfrac{1}{1+e^{-\frac{V_{soma}+3}{10}}} - n_s}{5+47\cdot e^{\frac{-(-50-V_{soma})}{900}}} \tag{4.15}$$

To generalize those functions, similarly to the standard ion gates, a predefined set of equations is used. The predefined set of equations are presented in Equation (4.16). Besides the costs of the more complex functions, nine instead of three floating-point parameters (the xs) are used in $fCustom$. With those modifications the standard gates, the custom gates from the IO-model, and the dynamics of the calcium concentration can be described using the custom gates. Note that, because the standard gates can be described, Equation (4.16) can be used instead of Equation (4.7).
Another difference in comparison with the equations for the standard gates is that, besides Equation (4.6), the gate derivative may be calculated with Equation (4.17). In this equation, similar to the standard gates, the variables α_i, β_i, inf_i, and tau_i are calculated by the predefined set of equations.

$$\frac{dy_i}{dt} = \frac{inf_i - y_i}{tau_i} \tag{4.17}$$

In addition to the custom gates, instantaneous gate-activation variables are used in the IO-model, which means that the $\frac{dy_i}{dt}$ variable is not used in case of those variables. However, those instantaneous variables can be described by the set of equations used to calculate α_i, β_i, inf_i, or tau_i and therefore, no new function needs to be implemented.
The generalization of the custom ion gates specifically aimed specifically to support the custom-defined ion gates in NeuroML of the IO-model. However, the generalized equations used for the custom ion gates are in fact useful for more extended Hodgkin-Huxley (eHH) models. That is, the here presented generalized equations are not IO-specific only but have general value. Thus, using them as guidelines in flexHH does not diminish flexHH's applicability.

4.3.3 Multiple cell compartments

When multiple compartments are supported, a current between two connected compartments in the same cell is formed. The relation between compartments is unspecified in NeuroML and therefore dependent on the simulation environment (such as NEURON).

To calculate the current flowing from compartment i to compartment j, Equation (4.18), the same equation as in the IO-model, as specified in [21], is used. This equation makes use of internal conductance of the cell (g_{int}), the surface ratio between compartments ($p_{i,j}$), and the voltage difference between compartments ($V_i - V_j$). If this extension is supported, this current is added to the sum of currents to calculate $\frac{\mathrm{d}V}{\mathrm{d}t}$, as shown in Equation (4.19).

$$I_{mc,i,j} = \frac{g_{int}}{p_{i,j}}(V_i - V_j) \tag{4.18}$$

$$\frac{\mathrm{d}V}{\mathrm{d}t} = \frac{I_{app} - I_{channels} - I_{leak} - I_{mc}}{C_M} \tag{4.19}$$

4.3.4 Gap junctions

Gap junctions are intercellular connections. In NeuroML, the current through a gap-junction is represented by Equation (4.20). In this equation, the intercellular current ($I_{gap,i,j}$) between two cells (cell i and cell j) is simply calculated as a conductance multiplied with the voltage difference. Consequently, the total current for a single cell i is calculated with Equation (4.21). In the IO-model the gap-junction current for a single cell i is calculated with Equation (4.22) ($w_{i,j}$ is a constant weight which scales the strength of the connection between two cells).

$$I_{gap,i,j} = g_{i,j}V_{ij} \tag{4.20}$$

$$I_{gap,i} = \sum_{j=0}^{N_{Cells}-1} g_{i,j}V_{i,j} \tag{4.21}$$

$$I_{gap,i} = \sum_{j=0}^{N_{Cells}-1} (w_{i,j}(0.8\exp(-0.01 \cdot V_{i,j}^2) + 0.2)V_{i,j}) \tag{4.22}$$

The current of the gap junctions in the IO-model shows that the conductance may be represented by an equation. Similar to the reasoning in the calculation of I_{app}, not all possible equations can be implemented on the DFE. Therefore, it was decided to generalise the calculation of $I_{gap,i}$ with Equation (4.23), where gx_0, gx_1, and gx_2 are floating-point variables. By generalising the current in this way, it is both possible to use the gap-junction current as described in the IO-model or use a single-floating-point variable as conductance. By adding gap junctions to a model, the current is used for the calculation of $\frac{\mathrm{d}V}{\mathrm{d}t}$, as can be seen in Equation (4.24).

$$I_{gap,i} = \sum_{j=0}^{N_{Cells}-1} (w_{i,j}(gx_0\exp(gx_1 \cdot V_{i,j}^2) + gx_2)V_{i,j}) \tag{4.23}$$

$$\frac{\mathrm{d}V}{\mathrm{d}t} = \frac{I_{app} - I_{channels} - I_{leak} - I_{gap}}{C_M} \tag{4.24}$$

4.4 DFE implementation

Now that the generalized functions are defined,they can be implemented as in DFE kernels where the parameters can change without the need of resynthesising. In what follows we describe how the algorithm to simulate HH model instances is implemented on the DFE. For the floating-point variables, single-floating-point precision was chosen as it has been proven accurate enough in the BrainFrame implementation and shifting the arithmetic precision (e.g. to fixed-point variables) will require an extensive accuracy analysis which is out of scope for this thesis. In the remainder of this section, first the implementation of the *HH* kernel is described after which the implementation of the features needed for the rest of the kernels is discussed.

4.4.1 HH

The implementation which supports the basic HH-model follows Algorithm 3. Note that, in comparison with Algorithm 2, the for-loop which iterates over the number of cells is removed. This is because, as previously discussed, it is a single-compartmental model and to prevent confusion with the other models, the cells are represented by compartments.

Algorithm 3 Pseudocode for the simulation of the HH-model.

1: **for** $0 \leq i < N_{steps}$ **do**
2: **for** $0 \leq k < N_{comps}$ **do**
3: **for** $0 \leq h < N_{gates}[k]$ **do**
4: $ys[i][k][h] \leftarrow updateY$
5: **end for**
6: $vs[i][k] \leftarrow updateV$
7: **end for**
8: **end for**

To implement Algorithm 3, the hardware required on the DFE consists of the `updateY`, `updateV`, the storage, and the control logic. The control signals are used to enable/disable the I/O streams, select the right streams of data from multiplexers, and enable memory reads/ writes. The for-loop sizes of Algorithm 3 are needed for the control signals. On the DFE, the loops are implemented with hardware counters. The input of the most (the loop of N_{gates}) receives its input from a stream (as N_{gates} is variable). Therefore, a buffer is used to hide the input latency of the stream inspired by the library dfesnippets [39] to allow for a efficient data-flow implementation. Consequently, the number of ticks is increased by 4 (the input latency), however, this is negligible in comparison to total amount of ticks needed for simulations. Furthermore, the most inner loop will be unrolled to speedup the kernel with a factor denoted as the unroll factor (uf). By unrolling the most inner loop, multiple pipelines are created. The hardware of these pipelines fit on the DFE, contrary to the hardware required for the outer loops, since this would require all the hardware required to calculate each of the more inner loop(s) to be on the DFE.

4.4.1.1 updateY

To update the y variables, the derivative $\frac{dy}{dt}$ from Equation (4.6) is used. For this equation, the calculations of both α and β are done with Equation (4.7). For the implementation of Equation (4.7), the amount of divisions and exponential functions are minimized, as Algorithm 4 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 and β employ this function, consequently, this algorithm is generated twice for the implementation. The first input argument of `f` is the voltage of compartment i (V_i). The xs, (x_1, x_2, x_3) are single-precision floating-point variables which can vary per gate. The variable $fType$ needs to be able to represent three different numbers, as there are three different equation branches. This can be done with two bits. The output of this function will be a single-precision floating-point variable.

Algorithm 4 Pseudocode of `f`, a generalized function to calculate α and β.

function F($V_i, x_1, x_2, x_3, fType$)
  $V_{diff} \leftarrow x_2 - V_i$
  if fType == 0 **then**
    $num \leftarrow x_1 \times V_{diff}$
    $c \leftarrow -1$
  else if fType == 1 **then**
    $num \leftarrow x_1$
    $c \leftarrow 0$
  else if fType == 2 **then**
    $num \leftarrow 1$
    $c \leftarrow 1$
  end if
  $denum \leftarrow \exp(V_{diff} \times x_3) + c$
  return $\frac{num}{denum}$
end function

4.4.1.2 updateV

Derivate voltage

To calculate the derivative of the voltage for a compartment, a sum of three currents is needed and a division by the conductance of the membrane of the compartment. However, instead of using a division with the conductance, a multiplication with the elastance is done as this is cheaper in terms of hardware costs. Consequently, Equation (4.25) is implemented. The elastance is set on the CPU host. When the conductance is given as variable of a model (as in NeuroML) then the elastance can be calculated, on the CPU host, with Equation (4.26).

$$\frac{dV}{dt} = (I_{app} - I_{channels} - I_{leak})S_M \tag{4.25}$$

where

$$S_M = \frac{1}{C_M} \tag{4.26}$$

Applied current I_{app}

The first current which is needed to calculate the derivative of the voltage is the applied current I_{app}. I_{app} is implemented similar to Equation (4.3). The difference is that t_{start} and t_{end} are converted to 32-bit unsigned integer step numbers ($step_{start}$, $step_{end}$), with the formula $step_{start/end} = \frac{t_{start/end}}{dt}$. This is done as the variable $step$ (representing the current step number) is already available as it is needed for the control of the implementation and therefore, can be used as input to calculate I_{app}. The other input variable A is the amplitude of the block function and is represented as a single floating-point value. The pseudocode for the function can be seen in Algorithm 5.

Algorithm 5 Pseudocode of `calcIApp`.

```
1: function CALCIAPP(step_start, step_end, A, step)
2:     if (step ≥ stepStart) ∧ (step < stepEnd) then
3:         iApp ← A
4:     else
5:         iApp ← 0
6:     end if
7: return iApp
8: end function
```

Channel current $I_{channels}$

The equation for $I_{channels}$ can be seen in Equation (4.27). The sum of all channel currents is calculated by either accumulating the additions or by unrolling the whole equation. The advantage of choosing for the accumulating option is that only one adder is needed. Furthermore, there is no need for an extra loop as `updateY` already needs to be calculated for each gate. However, an accumulation requires the old value before the new accumulated value can be calculated. The delay of a floating-point adder, in comparison to fixed-point adder, will give a loss in performance as the accumulation cannot be efficiently pipelined and therefore, creates idle ticks. Consequently, the option of unrolling the summation is chosen which has the benefit that it does not incur performance loss at the cost of using more hardware resources. There is only one issue with unrolling the summation, which is that $N_{channels}$ is a variable and unrolling a variable number is not possible in hardware. Therefore, there is a need for a maximum value. It was decided to set this maximum value to the number of gates per compartment ($N_{gates,max}$) as a channel consists out of 1 or more gates. This leads to the implementation of Algorithm 6 to calculate $I_{channels}$ on the DFE.

$$I_{channels} = \sum_{j=0}^{N_{channels}-1} I_{channel}[j]$$

$$= \sum_{j=0}^{N_{channels}-1} g_{channel}[j](V - V_{channel}[j]) \prod_{i=0}^{M_{gates}[j]-1} y_{j,i}^{p_{j,i}} \tag{4.27}$$

Algorithm 6 Pseudocode for `calcIChannels`.

```
1: function calcIChannels(N_gates, vChannel, yProd, gGate, vCompartment, N_gates,max)
2:     iChannels ← 0
3:     for 0 ≤ i ≤ N_gates,max do
4:         offsetG ← stream.offset(g, −i)
5:         offsetVChannel ← stream.offset(vChannel, −i)
6:         offsetYProd ← stream.offset(yProd, −i)
7:         iChannel ← offsetYProd × offsetG(vCompartment − offsetVChannel)
8:         if i < N_gates then
9:             iChannels ← iGates + iGate
10:        else
11:            iGates ← iGate
12:        end if
13:    end for
14: return iGates
15: end function
```

The variable $gGate$ is either equal to 0 or $g_{channel}$ depending on whether the currently processed gate is the final gate of the channel, so that $iChannels$ can only be updated once per channel. This is required as we loop over the gates instead of the channels. The loop is over the gates so as to have the flexibility of supporting a variable number of gates per channel without creating the need for extra variables. Additionally, the variable $yProd$ is needed per channel. The definition of $yProd$ can be seen in Equation (4.28) and is implemented as in Algorithm 7. As there is no generic exponential function in hardware, the variable y_j will be multiplied with itself one or more times as described by Algorithm 7. With this implementation, a p of 1 up to and including 4 is supported. The maximum of 4 is chosen as this is the maximum exponent needed to support both the basic HH and IO-models. Furthermore, $yProd$ can be the product of multiple exponential functions; see Equation (2.14) as an example. The solution is to let those gates be consecutively processed and then let $gGate$ only be non-zero for the final of the consecutive gates of a channel (which was already the case so that $iChannels$ is only updated once per channel). Then, Algorithm 7 calculates the right $yProd$ for a single pipeline. When the unroll factor is greater than one and a channel consists of more than one gates then, the calculation of $yProd$ is distributed over multiple pipelines. Subsequently, the results of the different pipelines need to be combined. However, for simplicity this is not shown in Algorithm 7.

$$yProd = \prod_{i=0}^{M_{gates}-1} y_i^{p_i} \tag{4.28}$$

Algorithm 7 Psuedocode for `calcYProd`.

```
1: function calcYProd(y, p, gGate)
2:     yProd ← y
3:     if p > 1 then
4:         yProd ← yProd * y
5:     end if
6:     if p > 2 then
7:         yProd ← yProd * y
8:     end if
9:     if p > 3 then
10:        yProd ← yProd * y
11:    end if
12:    gOld ← stream.offset(gGate, −1)
13:    if gOld == 0 then
14:        yProdOld ← stream.offset(yProd, −1)
15:        yProd ← yProd * yProdOld
16:    end ifreturn yProd
17: end function
```

Leakage current I_{leak}

The calculation of the leakage current I_{leak} (Equation (2.12)) is implemented in a straightforward way. v_l and g_l are single-floating-point variables which can change per compartment. The voltage V is the voltage of the compartment being processed.

4.4.1.3 Data-transfer needs

In the DFE, the variables can be:

- scalar variables derived from the host CPU.
- stored in Fast Memory (FMem).
- stored in LMem.
- received from a stream from the host CPU.

For large amounts of data, the data needs to be streamed from either the LMem or the host CPU. Because the LMem has a higher throughput and a more constant data transfer rate than streaming from the host CPU, the LMem will be used for large amounts of data.
The state variables of HH-type models are consisting of both the voltages (V_i) and gate-activation variables (y_j). The state variables are stored on the board of the DFE itself to prevent any performance loss of transferring the data between the CPU and DFE. Consequently, the state variables can be stored in either the LMem or FMem. In our design we store state variables in the FMem as they are updated every simulation step. It

is expected that when the state variables are stored into LMem the latency for updating the memory will be too high resulting in a decline in performance.
The memory for the voltages is called vMem and the memory of the gate-activation variables yMem. The vMem holds all the voltages, whose amount is equal to the total number of compartments ($N_{comps,total}$). To prevent the need to synthesize for a different number of compartments per simulation, a maximum number of compartments ($N_{comps,max}$) is set. This maximum is used as the size of the vMem. Consequently, as long as $N_{comps,total} \leq N_{comps,max}$ all the voltages can be stored and updated.
In yMem, in the case of a single pipeline, the total number of gate-activation variables ($N_{gates,total}$) needs to be stored. The total number of gates is the sum of the number of gates per compartment $N_{gates,total} = \sum_{i=1}^{N_{comps,total}} N_{gates}[i]$ and if each compartment has the same number of the gates the total number of gates is equal to $N_{gates,total} = N_{comps,total} \cdot N_{gates}[0]$.
Similar to vMem, the size of yMem is set to to a maximum size. This maximum is equal to the product of the maximum number of compartments and a maximum number of gates per compartment ($N_{gates,max}$). $yMem_{size} = N_{comps,max} \cdot N_{gates,max}$. When the loop of the gates is unrolled, each pipeline will have its own memory. The size of each memory of yMem is then equal to $\left\lceil \frac{N_{gates,max} \cdot N_{comps,max}}{uf} \right\rceil$.
$N_{gates,max}$ and $N_{comps,max}$ are two variables which are set at compile time. Another variable which is set at compile time is the unroll factor uf.
All the other variables are defined at runtime and can therefore be changed between each simulation, without the requirement of a new synthesis cycle. The scalar variables contain singular variables, which do not change during the simulation. The scalar variables used for the *HH* implementation are presented in Table 4.5. N_{steps} represents the number of simulation steps, N_{comps} represents the number of compartments, and *totalGatesPipe* is an array with the total number of gates per pipeline. All those variables are used for control signals. The time-step-size dt is used for the ODE solvers. The *throwAwayFactor* represents a factor which reduces the output size. The output of the state variables will be stored every *throwAwayFactor* steps into LMem. So when *throwAwayFactor* is equal to two, the state variables are stored into LMem, at every other step, instead of at every step,. Therefore, it is possible to store less data, which still may be enough for a neuroscientist to analyse, while keeping the same size of the time steps (dt) and thus the same accuracy during the simulation.
The other input variables are stored into LMem as the space required to store the variables is large. There are two constrains when using the LMem. The first constraint is that the data needs to be a multiple of 96 bytes as the data transfer happens in bursts of 96 bytes. The second constraint is that there is a limited number of LMem streams. Because of those two constraints the data is grouped together as much as possible while taken into account that a burst consists of 96 bytes. Considering that, data is needed much or less often based on whether the variables are needed to be logged per gate or per compartment. Two structures are made:

- The *gateConstants*
  For the constants which are needed as input per gate.

Table 4.5: Scalar variables for the *HH* implementation.

Variable	Type
N_{steps}	64-bit integer
N_{comps}	64-bit integer
$totalGatesPipe$	64-bit integer array of size uf
dt	32-bit single-precision-floating-point
$throwAwayFactor$	64-bit integer

Table 4.6: Variables of the structure *gateConstants* for the *HH* implementation.

Variable	Type
$aFType$	32-bit unsigned integer
$aX1$	32-bit single-precision-floating-point
$aX2$	32-bit single-precision-floating-point
$aX3$	32-bit single-precision-floating-point
$bFType$	32-bit unsigned integer
$bX1$	32-bit single-precision-floating-point
$bX2$	32-bit single-precision-floating-point
$bX3$	32-bit single-precision-floating-point
p	32-bit single-precision-floating-point
g	32-bit single-precision-floating-point
$vGate$	32-bit single-precision-floating-point
$yInit$	32-bit single-precision-floating-point

- The *compConstants*
  For the constants which are needed as input per compartment.

The structure *gateConstants* can be seen in Table 4.6. It contains the variables for the calculation of α and β, using Algorithm 9, which are needed for $\frac{\mathrm{d}y}{\mathrm{d}t}$. The xs, are of the type single-precision-floating-point. The $fType$ variable only needs 2 bits, however, due to the memory alignment it is decided to make this variable 32 bits. Additionally to the variables needed for $\frac{\mathrm{d}y}{\mathrm{d}t}$, the variables needed to calculate *calcIGates* are added to the structure. As a result, the structure contains 44 bytes so far. By adding the initial value of each y_i ($yInit$) also to the structure, finally it contains 48 bytes in total. This comes at the cost of increased data transfers between the kernel and the LMem.
The structure *compartmentConstants* is shown in Table 4.7. It contains the variables for the calculation of the $iApp$, the elastance, and the conductance and voltage of the leak

Table 4.7: Variables of the structure *compartmentConstants* for the *HH* implementation.

Variable	Type
$iAppStart$	32-bit unsigned integer
$iAppEnd$	32-bit unsigned integer
$iAppAmplitude$	32-bit single-precision-floating-point
S	32-bit single-precision-floating-point
$vLeak$	32-bit single-precision-floating-point
$gLeak$	32-bit single-precision-floating-point

gate. This structure, which contains 24 bytes, is already dividable by 96 and therefore, no initial voltage value for the compartments is padded to this structure.
The value of the initial voltage ($vCompIn$) is instead sent in its own stream. Another variable which is sent in its own stream is the number of gates per compartment N_{gates}. The type of $vCompIn$ is a single-precision-floating-point number and N_{gates} is a 32-bit unsigned integer. By knowing all the input variables it is possible to calculate the size needed to store all those variables. This is equal to: $sizeIn = 48 \cdot N_{gates,total} + (24 + 8) \cdot N_{comps,total}$ bytes.
Finally, the output variables consist of both the voltages and gate-activation variables (both of the type single-precision-floating-point) for each compartment or gate of multiple steps, where based on the *throwAwayFactor* all or just a set of all steps are stored. The total size of stored output variables is equal to $sizeOut = \frac{(N_{comps,total} + N_{gates,total}) \cdot 4 \cdot N_{steps}}{throwAwayFactor}$ bytes.

4.4.1.4 Choice of numerical solvers

How the state variables are updated is depending on which ODE solver is used. For the *HH* kernel the following three ODE explicit methods are implemented:

- Forward-Euler (see Equation (2.17).)
- Second-order Runge-Kutta (see Equations (2.18) to (2.19).)
- Third-order Runge-Kutta (see Equations (2.20) to (2.22).)

The forward-Euler method can directly be taken over from Equation (2.17). In case of the higher-order solvers there are multiple stages, where in each stage both $\frac{\mathrm{d}y}{\mathrm{d}t}$ and $\frac{\mathrm{d}V}{\mathrm{d}t}$ are calculated. Those extra stages introduce an extra loop as seen in Algorithm 8.

The first method which can be used to implement higher-order solvers is to place all stages on the DFE, by unrolling the loop in hardware. The second method is that

Algorithm 8 Psuedocode for a simulation of a HH-like model with a variable-order of ODE solver.

```
1: for 0 ≤ i < N_steps do
2:     for 0 ≤ j < N_ODE do
3:         for 0 ≤ k < N_comps do
4:             for 0 ≤ h < N_gates[k] do
5:                 ys[i][j][k][h] ← updateY
6:             end for
7:             vs[i][j][k] ← updateV
8:         end for
9:     end for
10: end for
```

an extra loop, with hardware counters, is implemented, which loops over the stages. The first method requires more hardware resources, as all the stage are implemented on the DFE while the second method requires that only one stage is implemented on the DFE. However, the first method uses less ticks for the same simulation because a longer pipeline is created. If the pipeline is filled efficiently, more computations are done simultaneously, leading to a better performance as the same number of ticks are used for different-order solvers. However, the extra hardware resources could also be used to achieve a higher unroll factor. The second method needs to store intermediate state variables and therefore, requires more Block Random-Access Memories (BRAMs) to store those variables and more ticks are required to complete the simulation and the number of ticks scales with the order of the solver. The expected better performance of the first method combined with the extra required BRAMs for the second method, which is the limiting resource type as shown in Section 5.3, is the reason why it was decided to implement the first method in the case of the *HH* kernel.

4.4.1.5 Overview

In the previously discussed implementation, the performance complexity of both `updateY` and `updateV` is equal to $\Theta(1)$. Therefore, the performance scales with the total number of gates. As discussed before, to speedup the implementation it was decided to unroll the most inner loop with an unroll factor uf. As a result, the number of ticks needed for a simulation is given by Equation (4.29).

$$N_{Ticks} = N_{steps} \cdot N_{Comps} \sum_{i=1}^{N_{comps}} \frac{N_{gates}[i]}{uf} + 4 \tag{4.29}$$

The top-level, which shows the architecture of the DFE and the CPU is shown in Figure 4.1. The connections between the CPU and the LMem, the scalar inputs which are directly fed into the kernel, and the connection between the LMem and the kernel on the DFE are visible.

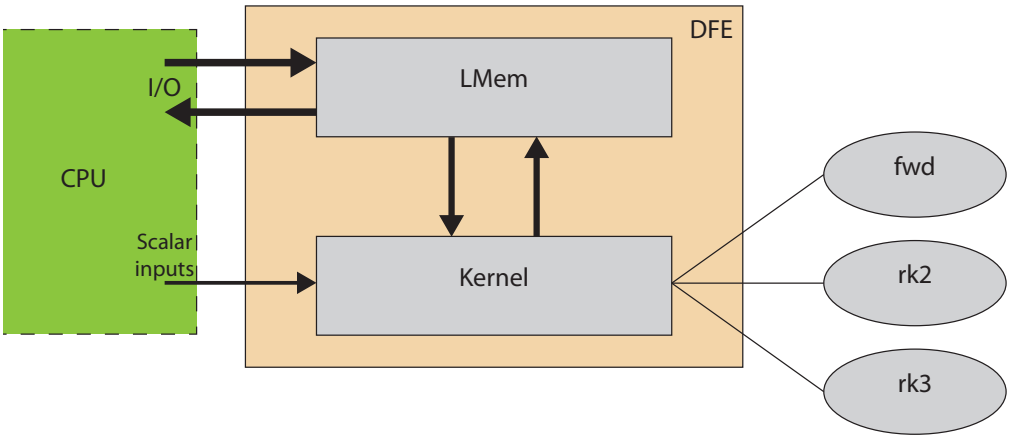


Figure 4.1: Architecture of the DFE and CPU.

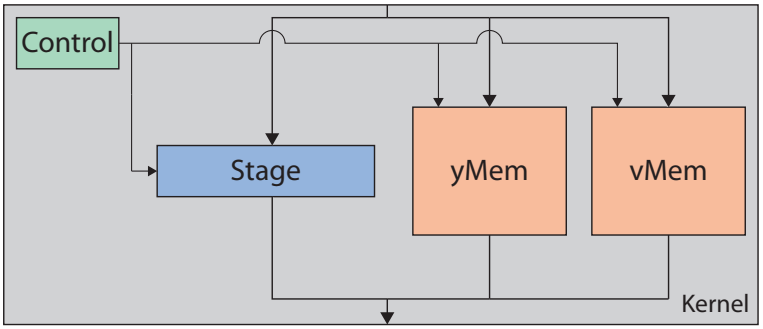


Figure 4.2: Visualization of the *HH* kernel when the forward- Euler method is used.

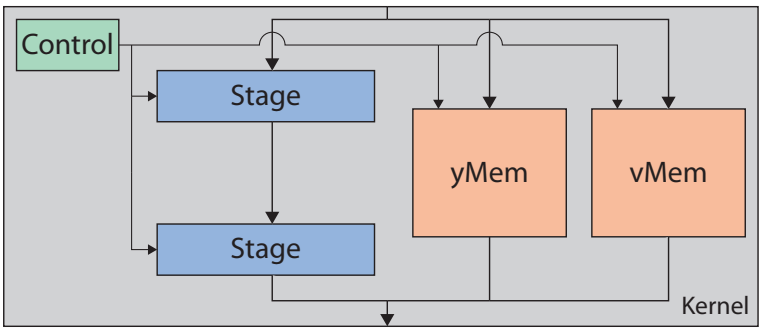


Figure 4.3: Visualization of the *HH* kernel when the second-order Runge-Kutta is used.

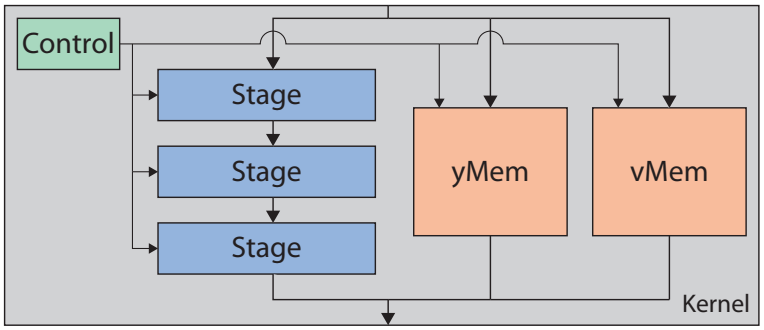


Figure 4.4: Visualization of the *HH* kernel when the third-order Runge-Kutta method is used.

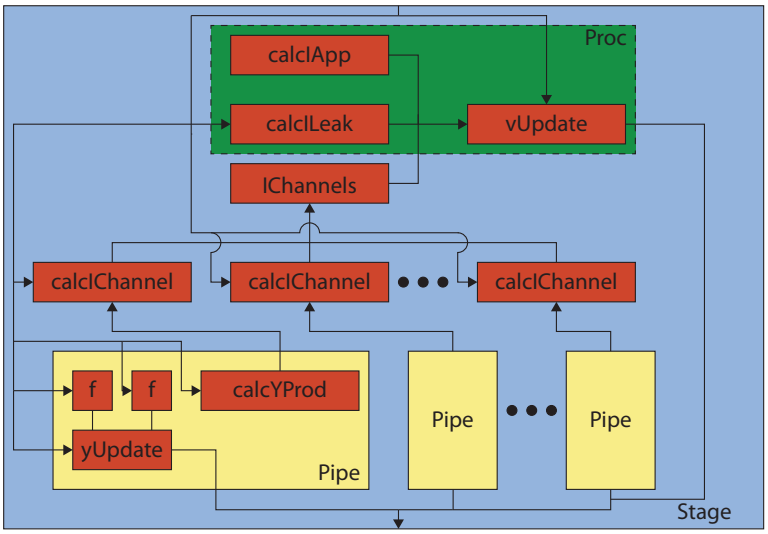


Figure 4.5: Visualization of the ”stage” block in the *HH* kernel. The red blocks contain the hardware of the algorithms with the same label.

The architecture of each solver kernel can be seen in Figures 4.2 to 4.4. These figures also show the unrolling of the stages for higher-order solvers. A "stage" contains the hardware which calculates one of the equations of the ODE solvers (Equations (2.17) to (2.22)), thus a ”stage” contains the hardware required to calculate the derivatives and one "ODE update". Note that, the "stage" design is repetitive as an extra "stage" can be added as-is per solver order for the ODE solvers implemented. A visualization of a single ”stage” can be seen in Figure 4.5. The segments of a "stage" will be discussed below. Before the segments of a single stage are discussed it is important to note that the implementation on the DFE depend on parameters $N_{comps,max}$, $N_{gates,max}$, and uf. The parameters $N_{comps,max}$ and $N_{gates,max}$ affect how big the supported network is. On the other hand, uf influences the performance. Each of these discussed parameters ($N_{comps,max}$, $N_{gates,max}$, and uf) influence the maximum value of each other as there is a limit in hardware resources available. Additionally, it depends on the order of ODE (N_{ODE}) used. To get a better understanding how the kernel is influenced buy these variables the *HH* kernel is divided into the following kernel segments:

- `vMem`
  The memory to store the membrane voltages. The size of the memory is equal to $4 \cdot N_{comps,max}$ bytes.

- `yMem`
  The memory to store the gate-activation variables. The size of the memory is equal to $4 \cdot N_{comps,max} \cdot N_{gates,max}$ bytes.

- `calcIChannel`
  The function to calculate $I_{channels}$, which is divided over uf pipeline. Therefore, this segment uses uf for loops with a size of $\frac{N_{gates,max}}{uf}$. Consequently, this segment is expected to scale with $N_{gates,max} \cdot N_{ODE}$.

- `iChannel`
  This is the summation in each ”stage” of uf outputs of `calcIChannel`. This summation takes $uf - 1$ additions and is needed in stage which gives that this segment is expected to scale with $(uf - 1)N_{ODE}$.

- `pipe`
  Each pipe consists of hardware for (a) to select the initial value of variable ($yInit$) or the value of the previous step (y) of the gate-activation variable, (b) the implementations of `f` using Algorithm 4 for both α and β, (c) the implementation of Equation (4.6), (d) the calculation of $yProd$, (e) and the ODE update of y. This segment is needed in every stage and there are uf pipes. Therefore, the segment `pipe` is expected to scale with $uf \cdot N_{ODE}$.

- `proc`
  The remaining hardware which does calculations in a stage. This consists of the implementations of Equation (2.12) and Algorithm 5 and the ODE update of V_i. This segment is expected to use the same amount of resources per stage and therefore, the hardware-usage is expected to scale with N_{ODE}.

- `control`
  The hardware needed to control the simulation consisting of the hardware counters and the control signals. This part of the implementation is expected to be roughly constant.

4.4.2 Custom ion gates

To not only support the standard HH gate equations from NeuroML but also all the gate equations from the IO-model (some are custom defined in NeuroML), some adjustments are required as discussed in Section 4.3.2. The first adjustment to support the custom gates is that both Equation (4.6) and Equation (4.17) are used. The variables of those functions can be calculated with the parameters of the other function as can be seen in Equations (4.30) to (4.33). However, choosing to rewrite the equations will not lead to a drop in hardware-usage, as each conversion requires a division. Therefore, it was decided to implement both equations and use a select signal, to select between the two as needed.

$$\alpha = \frac{inf}{tau} \tag{4.30}$$

$$\beta = \frac{1 - inf}{tau} \tag{4.31}$$

$$inf = \frac{\alpha}{\alpha + \beta} \tag{4.32}$$

$$tau = \frac{1}{\alpha + \beta} \tag{4.33}$$

The second adjustment in comparison to the basic HH-kernel instance is replacement of Algorithm 4 by Algorithm 9. With this new implementation, similar to the standard ion gates, the amount of exponentials and divisions is minimized. It must be noted that, in `fCustom`, $fType$ requires two bits instead of one bit to select the right function.

The third adjustment is that instantaneous variables are supported. Instantaneous variables are variables which change instantaneously instead of being updated by an ODE solver. The values for the instantaneous variables can be calculated with the equations from Equation (4.16). Due to the instantaneously changing variables the variables does not have to be y_i, but can also be an output of Equation (4.16). Although, the instantaneous variables are no gate-activation variables, they are treated as such. Meaning that the equation is described by the variables of $gateConstants$, the count of N_{gates} will be increased, the variables are stored in `yMem` and written into LMem. The storage of the instantaneous variables in both `yMem` and in the LMem serve no purpose. The storage space of `yMem` and in the LMem is precious for the flexHH kernels. Alternatively, the instantaneous variables could be processed detached from the gate-activation variables. This would have as an advantage that no extra storage space in both the `yMem` and in the LMem is used. However, it was decided to not make implement

Algorithm 9 Psuedocode for `fCustom`, a generalized function to calculate α, β, inf, τ, and the instantaneous variables for kernels which support custom ion gates. The functionallity of `fExp` is described in Algorithm 10.

function FCUSTOM($fType, V, xs$)
  $V_{diff} \leftarrow xs[1] - V$
  $V_{diff2} \leftarrow xs[6] - V$
  if $(fType\&3) == 0$ **then**
    $z_{12} \leftarrow V_{diff}$
  else
    $z_{12} \leftarrow V_{diff2}$
  end if
  $z_1 \leftarrow xs[2] \cdot V_{diff}$
  $z_2 \leftarrow xs[5] \cdot V_{diff2}$
  $exp1 \leftarrow fExp(xs[0], z1, xs[3])$
  $exp2 \leftarrow fExp(xs[4], z2, xs[7])$
  if fType == 0 **then**
    $num \leftarrow z2$
    $denum \leftarrow exp1$
  else if fType == 1 **then**
    $num \leftarrow xs[8]$
    $denum \leftarrow exp1 + exp2$
  else if fType == 2 **then**
    $num \leftarrow exp1$
    $denum \leftarrow exp2$
  else if fType == 3 **then**
    $num \leftarrow 0$
    $denum \leftarrow V_{Diff}$
  end if
  $y_0 \leftarrow \frac{num}{denum}$
  if fType != 1 **then**
    $y_0 \leftarrow y_0 + xs[8]$
  end if
  $y_1 \leftarrow min(z1, xs[0])$
  if fType == 3 **then**
    $y \leftarrow y_1$
  end if
  return y
end function

Algorithm 10 Psuedocode for `fExp`.

function FEXP($scale, x, offset$)
  return $scale \cdot \exp(x) + offset$
end function

this approach, as the adjustments would require serious development effort to map both memories, as it is variables which gates are instantaneous.
The fourth adjustment in comparison to the standard gates is that the input of Algorithm 9 does not have to be the voltage of a compartment but can be a current or another gate-activation variable. Consequently, between these different input variables the right variable needs to be selected. Moreover, to support custom ion gates more select signals are required. Per select signal the requirement is given after which it is discussed how this requirement is fulfilled in the implementation.

- Select the right function of Equation (4.16).
  Similar to the *HH* kernel, two bits of $fType$ (both of $aFtype$ and $bFtype$) will be used to select the right function of Equation (4.16).
- Select the right streams to get the result of either Equation (4.6) or Equation (4.17).
  The result of either Equation (4.6) or Equation (4.17) is only needed once per gate. Consequently, one bit from $aFtype$ is used.
- Select the right input for $calcYProd$.
  The input for $calcYProd$ can either be a gate-activation or an instantaneous variable. For the instantaneous variables, β is used and therefore, a bit from $bFtype$ is used to select the right input for $calcYProd$.
- Select gate-activation variable instead of voltage as input for Algorithm 9.
  There is only one function (the `min` function) which uses a gate-activation as input variable. Consequently, we decided to use the same bits of $fType$ which selects the (`min`) function to select a gate-activation as input.

The fifth and final adjustment is that a specific ion current, such as I_{Ca}, can be calculated. An ion current is the sum of the currents of the gates influenced by the specific ion (instead of all the gates in case of I_{gates}). The number of gates which contribute is the only difference between I_{Ca} and I_{gates}. Consequently, N_{Ca} is introduced to represent the number of gates which influence the specific ion current. Moreover, because the number of gates is the only difference between the currents the same hardware, to calculate the currents, can be used as is shown Algorithm 11. (Although here the current (I_{Ca}) and the number of gates (N_{Ca}) are called after the Ca ion, any other ion could be chosen). This implementation requires that the all N_{Ca} gates which are influenced by the ion are processed first (before gates which are influenced by other gates), as I_{Ca} is only updated in the first N_{Ca} ticks. Consequently, a limitation is added to this implementation, which is that only on specific ion current per compartment can be calculated.

The variable N_{Ca} can change per compartment and therefore is placed in the $compStructure$. The $gateConstants$ contains two times eight instead of two times three x's (parameters to calculate Equation (4.7) or Equation (4.16)), which can be seen in Table 4.8. This structure now contains 88 bytes instead of the 48 bytes when using the standard gates as in the *HH* kernel instance.
The number of Ca (or other specific ion) gates (N_{Ca}) is added to the $compConstants$. Without any other modifications this would lead to a structure of 28 bytes, which is non

Algorithm 11 Psuedocode of `calcIChannelsCustom`.

```
1: function calcIChannelsCustom(N_gates, nCa, vGate, yProd, g, vCompartment, N_gates,max)
2:     iGates ← 0
3:     iCa ← 0
4:     for 0 ≤ i ≤ N_gates,max do
5:         offsetG ← stream.offset(g, −i)
6:         offsetVGate ← stream.offset(g, −i)
7:         offsetYProd ← stream.offset(yProd, −i)
8:         iGate ← offsetYProd × offsetG(vCompartment − offsetVChannel)
9:         iGate ← yProd[i] × offsetG(vCompartment − offsetVChannel)
10:        if i < N_gates then
11:            iGates ← iGates + iGate
12:        else
13:            iGates ← iGates
14:        end if
15:        if i < nCa then
16:            iCa ← iCa + iGate
17:        else
18:            iCa ← iCa
19:        end if
20:    end for
21: return iGates, iCa
22: end function
```

Table 4.8: Variables of the structure *gateConstants* for the *HH+custom* implementation.

Variable	Type
aFType	32-bit unsigned integer
aXs	8 32-bit single-precision-floating-points
bFType	32-bit unsigned integer
bXs	8 32-bit single-precision-floating-points
p	32-bit single-precision-floating-point
g	32-bit single-precision-floating-point
vGate	32-bit single-precision-floating-point
yInit	32 -bit single-precision-floating-point

optimal because 96 is not dividable by 28. Therefore, the structure is padded with $vInit$, as this variable needs to be send anyway, so the structure contains 32 bytes and thus $vInit$ is removed from the remaining streams. The *compConstants* for the *HH+custom* implementation can be seen in Table 4.9. This means that the total input size will change to $88 \cdot N_{gates,total} + (32 + 4) \cdot N_{comps}$ bytes.
In comparison to the *HH* kernel the same kernel segments are used in the *HH+custom*. The significant difference in hardware is the replacement of `f` (Algorithm 4) by `fCustom`

Table 4.9: Variables of the structure *compConstants* for the *HH+custom* implementation.

Variable	Type
$iAppStart$	32-bit unsigned integer
i$AppEnd$	32-bit unsigned integer
$iAppAmplitude$	32-bit single-precision-floating-point
S	32-bit single-precision-floating-point
$vLeak$	32-bit single-precision-floating-point
$gLeak$	32-bit single-precision-floating-point
nCa	32-bit unsigned integer
$vInit$	32-bit single-precision-floating-point

(Algorithm 9), the replacement of `calcIChannels` (Algorithm 6) by `calcIChannelsCustom` (Algorithm 11), and the support of Equation (4.17). This because of the extra exponent and divisions which are used in those functions. The other changes are not expected to give significant differences as the control signals to select the right streams require relatively simple hardware.

4.4.3 Multiple cell compartments

When a single cell can have multiple cell compartments, an additional loop is added (as in Algorithm 2) to loop both over the cells and the compartments. The support of multiple cell compartments allows for a current to flow between compartments. This current, between compartments i and j, is represented by the compartments $I_{mc,i,j}$. The structure of the connections between the compartments can potentially form a tree the structure of which can differ per model. Still, to generate hardware which will be efficient, not creating empty stages in the pipeline, only sequential connections are supported in the current version of flexHH. This structure is visually represented in Figure 4.6. This simplification has been negotiated carefully with the in-house neuromodelers and has been agreed to be a reasonable compromise. A large volume of existing models can be captured by sequentially connected compartments.
Because of the sequential structure, compartment k will only receive currents (which is calculated by Equation (4.18)) 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 Equation (4.34) for the calculation of the current.

$$I_{comp,i} = \frac{V_i - V_{i+1}}{1 - p_{i,i+1}} g_{int} \tag{4.34}$$

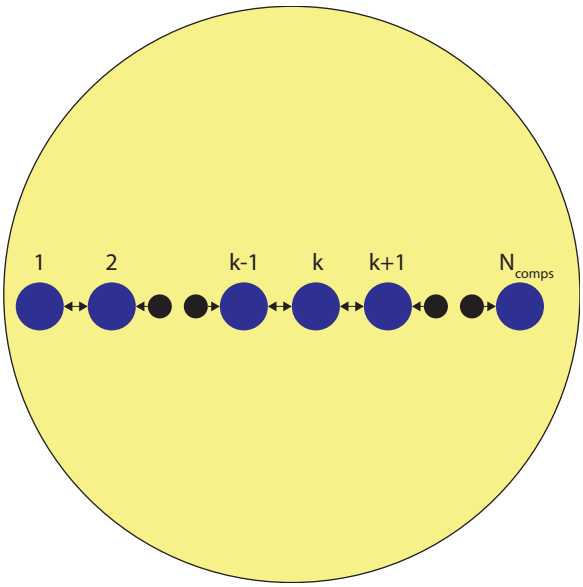


Figure 4.6: Visualization of multiple cell compartments in a sequential structure.

- 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 Equation (4.35) for the calculation of the current.

$$I_{comp,i} = (\frac{V_i - V_{i+1}}{1 - p_{i,i+1}} + \frac{V_i - V_{i-1}}{p_{i-1,i}})g_{int} \tag{4.35}$$

- The ending position.
  In this case the compartment only exchanges current with the compartment which lays before in the line. This results in Equation (4.36).

$$I_{comp,i} = \frac{V_i - V_{i-1}}{p_{i-1,i}} g_{int} \tag{4.36}$$

As follows from Equations (4.34) to (4.36), Equation (4.35) (the current when compartment i is between other compartments) is the sum of Equation (4.34) (the current at the starting position) and Equation (4.36) (the current at the ending position). Consequently, Equation (4.34) is stored in $I_{comp,next}$ and Equation (4.36) 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. This leads to the pseudocode in Algorithm 12.

In the pseudocode, i is the index of the compartment in the cell, the variable N_{comps} is equal to the number of compartments of the cell which is being process, Vs is an array

Algorithm 12 Psuedocode 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
```

containing the voltage of the previous compartment (V_{i-1}), the voltage of the current compartment (V_i), and the voltage of the next compartment (V_{i+1}), ps consist of two ratios $p_{i-1,i}$ and $p_{i,i+1}$, and finally g_{int} is the internal conductance of the cell.
The method of how g_{int}, ps, and the voltages Vs are retrieved on the DFE need special attention. If a cell has N_{comps} and the compartments are sequentially connected then there are $N_{comps} - 1$ ratios between the cells, which are represented with single-floating-point v. The other variable g_{int} is also a floating-point variable, which gives a total of N_{comps} new variables (in comparison to kernels without multiple cell compartments). Therefore, for the implementation a variable gp is introduced, which is added to the *compConstants*. This variable will contain g_{int} when the first compartment of the cell is being processed (i is equal to 0). This variable needs to be changed only once per cell. Otherwise, when compartment i is processed, this variable holds $p_{i-1,i}$ as is shown in Equation (4.37) ($p_{i,i+1}$ can be retrieved using the `stream.offset` function). Note that, when either the first or last compartment is being processed, the value of either $p_{i-1,i}$ or $p_{i,i+1}$ is incorrect. However, this incorrect ratio has no influence on the output as is not needed for the calculation of I_{mc}. The voltage V_i will be present as it is needed in other equations. The voltages from the previous compartment (V_{i-1}) and the next compartment (V_{i+1}) are either read from memory or retrieved from the pipeline itself (in the case of the higher-order solvers not all voltages are stored).

$$gp = \begin{cases} g_{int} & \text{if } i = 0 \\ p_{i-1,i} & \text{else} \end{cases} \tag{4.37}$$

The extra loop has a variable limit N_{comps} which can vary per cell. This is the only variable which is needed per cell and therefore, is sent through its own stream. Consequently, to receive one value per cell, and one *compStructs* per compartment, two hardware counters with different limits are required to keep track of which compartment is being processed. Consequently, the variable $N_{channels}$ which consists of 32 bits is for the data

transfer split into two parts. The first 16 bits represent the channels per compartment ($N_{channels,comp} = N_{channels}$) and the other 16 bits are used to represent the channels per cell ($N_{channels,cell}$). This is done so that no extra data transfer is required for those values, while still being able to represent large enough maximums for the number of channels per either the compartment or cell. To support multiple compartments, the only extra variable which needs to be streamed into the kernel is gp, so only the *compConstants* is updated. Compared to the *HH+custom* implementation, the variable $vInit$ is replaced by gp and therefore $vInit$ is transferred by its own stream. Consequently, the total input size will change to $88 \cdot N_{gates,total} + (32 + 8) \cdot N_{comps,total} + 4 \cdot N_{cells}$ bytes for the *HH+custom+multi* implementation.
With the support of multiple cell compartments a new kernel segment *calcIComp* is added to a "'stage'. The kernel segment `calcIComp` contains the hardware needed to calculate $I_{compartment}$. For higher-order solvers a stream offset equal to N_{gates} is used (to retrieve V_{i-1}). However, this is expected to have a negligible effect on the hardware-usage as the offset will be relatively small (<20) and therefore, also the differences between configurations with different number of gates. Additionally, because the function `calcIComp` is needed to calculate $\frac{\mathrm{d}V}{\mathrm{d}t}$, this part will scale with the order of ODE. Because this is expected to be the only scaling factor, it could be placed under the `proc` segment. However, to let the segment `proc` be unambiguous `calcIComp` will be placed in its own kernel segment, which is called `calcIComp`).

4.4.4 Gap junctions

From Equation (4.23) it follows that two for-loops are needed to calculate the gap-junction currents. The order of the for-loops is of large significance to the performance. There are two loop-traversal possibilities for scheduling the loops which we will call row-wise and column-wise. The pseudocode of both situations can be seen in Algorithm 13 and Algorithm 14. Furthermore, both situations are visually presented in Figure 4.7. If the calculations are done per cell (row-wise) the execution order and the data dependencies of $I_{gap,i}$ are in the same direction (meaning that the same $I_{gap,i}$ is updated in consecutive iterations of the most inner loop). In this case, a latency of one tick is needed so as to make efficient use of the pipeline. Otherwise, the pipeline has to wait on the results before it can continue with the next iteration in order to give a correct output. However, the latency for calculating Equation (4.38) (which is the equation of one iteration) is larger than one and thus the pipeline will be used inefficiently. This will have a significant impact on performance. As a solution, the calculations are done column-wise. In that case, the data dependencies and the execution order are in a different directions (meaning different $I_{gap,i}$s are updated in consecutive iterations of the most inner loop). In that way the previous value of the accumulation is needed after N_{cells} ticks. Consequently, only if N_{cells} is bigger than the length of the pipeline, the pipeline can be filled completely and is efficiently used.

$$I_{gap,i,j} = w_{i,j}(0.8\exp(-0.01 \cdot V_{i,j}^2) + 0.2)V_{i,j} \tag{4.38}$$

Algorithm 13 Psuedocode of `calcIGap` (row-wise)

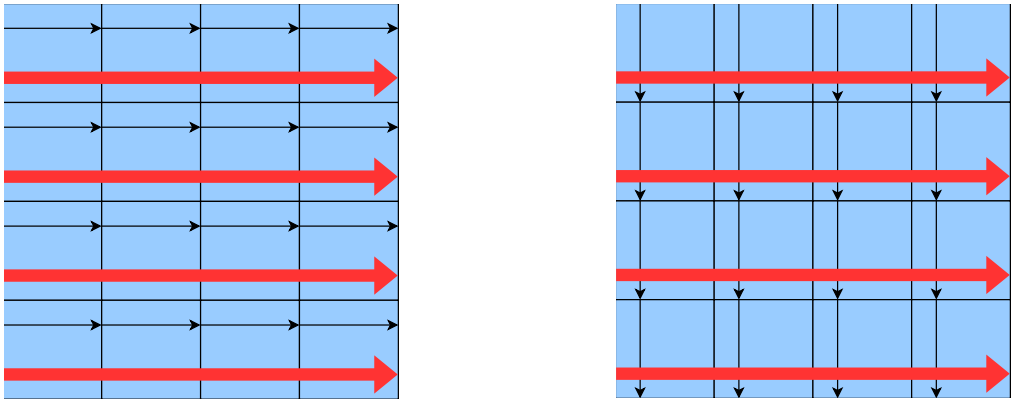
```
1: procedure CALCIGAP(N_cells, V, iGap, g_x s, W)
2:     for 0 ≤ i < N_cells do
3:         iGap[i] ← 0
4:         for 0 ≤ j < N_cells do
5:             vDiff ← V[i] − V[j]
6:             I_gap[i] ← I_gap[i] + W[i][j](g_x0 exp(g_x1 vDiff^2) + g_x2) vDiff)
7:         end for
8:     end for
9: end procedure
```

Algorithm 14 Psuedocode of `calcIGap` (column-wise)

```
1: procedure CALCIGAP(N_cells, V, iGap, g_x s, W)
2:     iGap ← 0
3:     for 0 ≤ i < N_cells do
4:         for 0 ≤ j < N_cells do
5:             vDiff ← V[j] − V[i]
6:             I_gap[j] ← I_gap[j] + W[i][j](g_x0 exp(g_x1 vDiff^2) + g_x2) vDiff)
7:         end for
8:     end for
9: end procedure
```

Because the column-wise traversal of the gap junctions is better for performance, the gap junctions are implemented this way on the DFE, as can be seen in Algorithm 15.

The implementation requires that the intermediate results of the gap junctions currents are stored, which is the reason of the use of `iGapMem` in Algorithm 15. This memory stores the intermediate results of the gap-junction currents of all cells. This data has its maximum size when when there is one compartment per neuron cell. Therefore, the size of this memory is set to $N_{comps,max}$ (in single-precision-floating-point numbers). So, the performance gain of the column-wise traversal comes at the price of more memory. The memory needs to be updated multiple times in a single simulation step. Therefore, FMem is used for `iGapMem`. Another adjustment which is required in the case multiple cell compartments are supported is that it must be indicated which of the compartments in the cell are connected through gap junctions. This is done by splitting N_{comps} into two variables. The first 16 bits of N_{comps} will contain the actual value of the N_{comps}, and the last 16 bits contain $gapAddress$. The latter variable is the address of voltage in `vMem` of the compartment which is connected by a gap-junction. The calculations for I_{gap} and the other currents are asynchronous and need gapAddress. Therefore, the variables of gapAddress are stored into FMem. The size of the memory is equal to $\lceil log2(N_{Comps,max}) \rceil \cdot N_{comps,max}$ bits as there are $\lceil log2(N_{comps,max}) \rceil$ bits needed to reach each element in `vMem`.
Besides the extra storage needed for `iGapMem` and $gapAddress$, more variables are required to be stored in FMem due to the gap-junction calculations. The sum of I_{mc}, I_{gates},



(a) row-wise (b) column-wise

Figure 4.7: Directions of the execution order (red arrows) and the data dependencies (black arrows) for both row-wise and column-wise calculations.

Algorithm 15 Psuedocode of `calcIGap`.

```
1:  procedure calcIGap(N_cells, iGapMem, vMem, g_xs)
2:      for i ∈ {0, uf, 2uf, ..., 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)
8:              end if
9:              vOwn ← vMem.read(j)
10:             for 0 ≤ k < uf pardo
11:                 vOther ← vMem.read(i + k)
12:                 vDiff ← vOwn − vOther
13:                 iGapTemp[k] ← w_{j,i+k}(g_x[0] exp(g_x[1] vDiff^2) + g_x[2]) vDiff)
14:             end for
15:             iGapNew ← iGapOld + sum(iGapTemp)
16:             iGapMem.write(j, iGapNew)
17:         end for
18:     end for
19: end procedure
```

I_{leak}, and I_{app} cannot be calculated fast enough to keep up with the gap junctions. This as due to the column wise computation at the final iteration of the gap junctions $I_{gap,i}$ is updated per tick, while the calculation of $I_{gates,i}$ takes $N_{gates}[i]$ ticks. Therefore, the sum of I_{mc}, I_{gates}, I_{leak}, and I_{app} is stored in another memory (`iRestMem`), for the compartments which are connected to gap junctions. Then, when the calculation of $iGap$ is

done, `iRestMem` is read and $\frac{dV}{dt}$ is calculated and the voltage is updated. This is shown in Algorithm 16, where the computations of `updateY` and `iRest` (where `calcICompartment` only is computed in case multiple compartments are supported) are done in parallel with the computations of `calcIGap`.

Algorithm 16 Psuedocode of a HH-type model with gap junctions

```
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:                  ys[i][j][k][h] ← updateY
6:              end for
7:              iRest ← calcIGates + calcILeak + calcICompartment + calcIApp
8:              iRestMem.write(iRest, k)
9:          end for
10:     end for
11:     calcIGap
12:     for 0 ≤ j < N_cells do
13:         vs[i][j][k] ← updateV
14:     end for
15: end for
```

The only adjustment in relation to the I/O is related to the weight variables $w_{i,j}$. The weight variables $w_{i,j}$ are stored in the so-called connectivity matrix of size N_{cells}^2 single-precision-floating-points numbers. Due to the size of this matrix, the matrix is stored into LMem and therefore, remains constant during the simulation.
For the use of higher-order solvers, another adjustment is required. In the kernels without gap junctions, it was possible to unroll all the stages of the higher-order solvers. However, when gap junctions are supported, the hardware resources are not enough. Therefore, another hardware counter is added for implementing the loop iterating over the solver stages. As a result, the number of ticks of the simulation now also scale with N_{ODE}. Additionally, the intermediate results of the state variables need to be stored too. Therefore, the size of both `vMem` and `yMem` is doubled, independent of the order of the solver.
Equations (2.17) to (2.22), the equations of the ODE solvers, can be represented by Equation (4.39). Here, x_{new} is the value of x (a state variable) of next stage (which can be the value of the next simulation step), x_{old} the value of the current simulation step, x_{ODE} the value of the intermediate stage, and $\frac{dx}{dt}$ the derivative. Furthermore, the cs (c_0, c_1, c_2) are constant and depend on the ODE method and the stage of the ODE method. The constants for the used ODE solvers can be seen in Table 4.10. This table shows that $c_1 = 1 - c_0$ and c_2 is equal to 1 in the first stage and equal to $1 - c_0$ in the other stages. Consequently, only the values of c_0 are stored in FMem to save memory space, although the saved space is negligible in comparison to the amount of used FMem. The asset of storing the constants of multiple ODE solvers is that the solver can be chosen on

the host CPU. Therefore, the same hardware implementation can be used for multiple ODE solvers. It must be noted that the forward-Euler method can also be used with this implementation, however, this is inefficient due to the extra storage needed for `vMem` and `yMem`.

$$x_{new} = c_0 \cdot x_{old} + c_1 \cdot x_{ODE} + c_2 \cdot dt \cdot \frac{\mathrm{d}x}{\mathrm{d}t} \tag{4.39}$$

Table 4.10: Constants for different ODE solvers, to be filled in Equation (4.39)

ODE	stage	c_1	c_2	c_3
fwd	1	1	0	1
rk2	1	1	0	1
	2	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
rk3	1	1	0	1
	2	$\frac{3}{4}$	$\frac{1}{4}$	$\frac{1}{4}$
	3	$\frac{1}{3}$	$\frac{2}{3}$	$\frac{2}{3}$

The gap-junction current $I_{gap,i}$ for cell i is calculated with Equation (4.22). As can be seen in the equation it is a summation of N_{cells} parts. Furthermore, each cell will calculate the gap-junction current. Consequently, the total ticks will scale with $\Theta(N_{cells}^2)$. The other calculations scale with $\Theta(N_{cells} \cdot N_{gates,avg,cell})$, where $N_{gates,avg,cell}$ are the average number of gates per cell. Therefore, the total simulation will scale with $\Theta(max(N_{cells} \cdot N_{gates,avg,cell}, N_{cells}^2))$. It is expected that N_{cells} will be larger than $N_{gates,avg,cell}$ in most cases and thus the gap junctions will be the most time-consuming part of the program. Therefore, it was decided to unroll the loop of the gap junctions instead of the loop of the gates, with an unroll factor uf_{gap}. Consequently, the overall number of simulation ticks used for the gap junctions is equal to Equation (4.40) and overall number of simulation ticks to complete the other ticks is equal to Equation (4.41). To synchronize the calculations, the control signals for the gap junctions and other calculations must be dependent on each other. Because it is expected that the gap junctions will take more ticks, the condition $N_{cells} \geq N_{gates,avg,cell} \cdot uf_{gap}$ must hold. This condition arises as there was not found a way to successfully build a single implementation which can simulate both $N_{cells} \geq N_{gates,avg,cell} \cdot uf$ and $N_{cells} < N_{gates,avg,cell} \cdot uf$. On the other hand, as uf is a maxConstant, this condition is checked to hold on the CPU so the user of the

flexHH-library cannot run an invalid simulation.

$$N_{Ticks,gap} = \frac{N_{cells}^2}{uf_{gap}} \cdot N_{ODE} \cdot N_{steps} \tag{4.40}$$

$$N_{Ticks,other*} = N_{cells} \cdot N_{gates,avg,cell} \cdot N_{steps} \tag{4.41}$$

Besides the double amount of memory when a higher-order solver is used, when a kernel supports the gap junctions the following kernel segments are added:

- `iGapMem`
  Memory used to store the intermediate results of the gap-junction calculations. The size of the memory is equal to $4 \cdot N_{comps,max}$ bytes.
- `gapAddressMem`
  Memory used to store the gapAddresses. The size of the memory is equal to $\lceil log2(N_{Comps,max}) \rceil \cdot N_{comps,max}$ bits.
- `iRestMem`
  Memory used to store the results of the sum of I_{mc}, I_{gates}, I_{leak}, and I_{app}. The size of the memory is equal to $4 \cdot N_{comps,max}$ bytes.
- `calcIGap`
  The hardware needed to do the calculations of the gap junctions, which can be seen in Algorithm 15. The hardware can be split between control (`gapControl`) and functional hardware `gapProc`. The control is expected to have a constant hardware-usage, while the functional hardware is expected to scale with uf_{gap}.

Furthermore, for the segments where in the kernels without gap junctions the gate equations were unrolled, this old unroll factor is set to one since, instead, the calculations of the gap junctions are unrolled.

5 Evaluation

In this chapter, the flexHH-library evaluation is presented. Firstly, the kernels on the Data-Flow Engine (DFE) are functionally validated in Section 5.1. Secondly, in Section 5.2 an exploration of time-step sizes using different Ordinary Differential Equation (ODE) solvers is done in the hopes of scoring better simulation speeds. Thirdly, the resource usage, depending on the hardware parameters and the flexHH kernel used, is analyzed in Section 5.3. The performance of the simulations is evaluated in Section 5.4 and finally, in Section 5.5 the energy usage is discussed.

5.1 Validation

This section discusses the validation of the output of the implementations on the DFE. To guarantee functional correctness of our implementations, we must validate our flexHH kernels in two levels. Initially, we need to make sure that the models in flexHH (one model per kernel instance) are validated against the established (by the community) versions of the same models. For this reason, we first check, in Section 5.1.1, the correctness of our kernels in C-code compared to reference NEURON and C implementations of the basic Hodgkin-Huxley (HH) model and the Inferior-Olive (IO) model . Secondly, we need to guarantee functional correctness in the porting from the C software implementations to the DFE versions, as precision errors can easily emerge when porting between different architectures. The functional correctness of the DFE versions is discussed in Section 5.1.2. A schematic overview of the validation steps is shown in Figure 5.1.

5.1.1 C-code validation

Reference code is only available for the original HH-model and for the IO-model, and will be used for the validation of the C-code. The reference code of the HH-model is used to ensure correctness for the basic HH equations and therefore used to validate the *HH* C kernel. The IO-model is used to validate all extra features that are utilized in this model and therefore, used to validate the *HH+custom+multi+gap* C kernel. The other C kernels (*HH+gap*, *HH+custom*, and *HH+custom+multi*) use the basic HH equations with one or two extra features. Every feature in those kernels is implemented in exactly the same way as in the *HH+custom+multi+gap* C kernel. Therefore, we hope that each kernel instance which utilizes an extra feature is functionally correct, if the *HH+custom+multi+gap* C kernel is functionally correct.
The reference code for the HH-model comes from NEURON, which is widely accepted as a standard by computational neuroscientists. However, the HH-model in NEURON does diverge a little compared to the originally defined HH-model. In NEURON, the resting potential is $-65\ mV$ instead of the $0\ mV$ defined in the formal definition in [7].

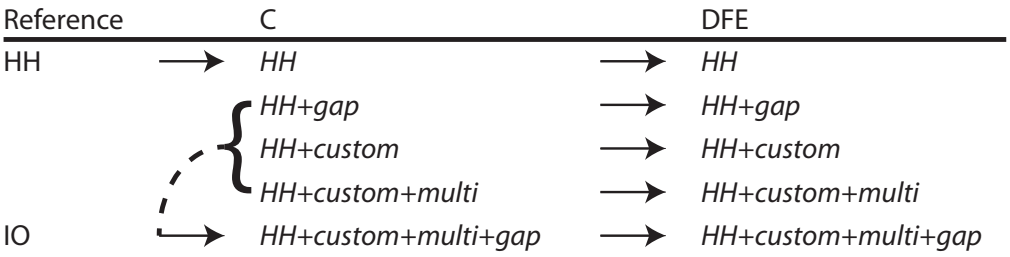


Figure 5.1: Schematic overview of the validation steps.

We follow the formal definition for both of our C and DFE implementations. To make the two implementations comparable, the neuron voltage output in NEURON is shifted accordingly after the simulation. Furthermore, each voltage is negated in comparison with the formal definition. This means that in all the equations where a voltage is involved, this variable is negated. Consequently, the voltage plots are mirrored in the y-axis. The voltage is also negated in the C and DFE implementations.
The simulator in NEURON uses a backward-Euler ODE solver while the forward-Euler method is not supported, because this solver can have numerical stability problems when the time-step becomes to large [8]. As a result, the output of the C-code and NEURON-code are not expected to be identical. However, a qualitative comparison–as is often common practice in this domain–is possible to asses correctness.
The simulation used for validation has a duration of 300 ms and a time-step-size (dt) equal to 0.01 ms. Moreover, the initial parameters and the values used for I_{app} can be seen in Appendix B.1.1.1.
The voltage trace of the output of the simulation in NEURON together with the errors for the output of the C-code for each of the three solvers is shown in Figure 5.2. The errors for each of the three solvers are substantial during the spiking period. Because there is only a significant difference between the voltage traces during the spiking period, those errors are likely to be the results of the use of different numerical methods. This assumption is reinforced by inspecting the voltage traces during a small time period, as shown in Figure 5.3. This figure shows the voltage traces are shifted, indicating a phase error, which can be expected when using different solvers.
To guarantee that the error is only caused by a phase shift and not caused by an accumulation error from other sources, extra tests are required. Firstly, the phase error is expected after a longer period of time to reduce to zero again, after full period. To verify this, a simulation is ran with the only difference being a longer simulation time. The output and the errors for this simulation are shown in Figure 5.4 revealing the error to be indeed a bounded phase error.
To see if the error is indeed the result of using different solvers, three additional simulations are ran. First, the same simulation is ran with a smaller dt. Consequently, each simulation (independently of the solver) is expected to be more accurate, resulting in a smaller difference/error between the solvers. The results in Figure 5.5, where a dt of 0.005 ms instead of 0.010 ms is used, indeed show improved accuracy in comparison to the results where a dt of 0.010 ms was used.

Secondly, the same simulation is run where the NEURON-code uses the Crank-Nicolson method instead of the backward-Euler method and thus uses a second-order instead of a first-order method. Therefore, it is expected that the NEURON-code becomes more accurate. The results, as can be viewed in Figure 5.6, show a smaller error than when using backward Euler. This reinforces the thought that the error is caused by using different solvers. Interestingly, the error peaks look mirrored when using Crank-Nicolson instead of the backward-Euler method (Figure 5.7 with Figure 5.3). Meaning that the forward-Euler solver has a positive phase shift in comparison to the backward-Euler solver method and a negative phase shift in comparison to the Crank-Nicolson solver. Furthermore, to make sure the error with Crank-Nicolson is also a phase error, and thus bounded, this simulation is also ran for an even longer time period (30000 ms). This result is shown in Figure 5.8 and shows indeed a bounded error.
Finally, the methods implemented in the C-code are compared to an exact solution. An exact solution of the HH-model is only possible under the condition that g_K and g_{Na} are equal to zero. Then, the equations of the HH-model simplify so that the ODE can be solved by using the method of separation of variables, The exact solution, with an initial condition of $V(0) = 0$, can be seen in Equation (5.1).

$$V(t) = \frac{I_{App} + g_L \cdot v_L - exp(-((g_L \cdot t)/C)(I_{App} + g_L \cdot v_L))}{g_L} \tag{5.1}$$

The other initial conditions can be seen in Appendix B.1.1.2.
The output of this function and the error which is calculated as the differences between the output of the exact solution and the output of the numerical solvers are depicted in Figure 5.9. Although the forward-Euler solver shows a larger error than the two Runge-Kutta solvers, the error is small enough to not affect functional correctness, as was confirmed by our in-house neuroscientists.
Since the error is periodical and bounded, a smaller dt gives a smaller error, a higher-order solver produces a smaller error, and the error in the exact case is relatively small, it can be concluded that the error is indeed a phase error. Looking at the output of the C-code it can be concluded to give qualitatively correct output and thus can be used as a reference for the accelerated implementation.
By showing that the HH-model is simulated correctly, the most basic simulation is verified. In order to verify all the extra features, the IO-model will be used. The IO-model was described in Section 2.2.2 and the equations can be found in Appendix A. For the IO-mode,l the reference code, which was verified by neuroscientist peers, is written in C and uses the forward-Euler method. However, to match a standard description of ODEs (splitting the equations used for the derivative and the solver) the code was rewritten. The new code separated the calculation of the derivatives (for the membrame voltages and gate-activation variables) from the rest of the code. Consequently, by using this modified version, the implementations using the Runge-Kutta methods were easier to implement. New simulations were run to validate the new code versions. For the simulation, a network of 480 cells was used. The initial values, the values of I_{app}, and the weights of the connectivity matrix are presented in Appendix B.1.1.3.
The axonal-voltage (from a single neuron in the grid) together with the output errors for each of the three solvers (rewritten in C) are shown in Figure 5.10. The plots indeed

show that the error between the two simulations using the forward-Euler method is small. However, the errors for both Runge-Kutta methods are larger which, again, is to be expected to be the result of using different solvers. This is because the large errors present themselves during the spiking phases and during the spiking phases a small phase shift leads to large error.

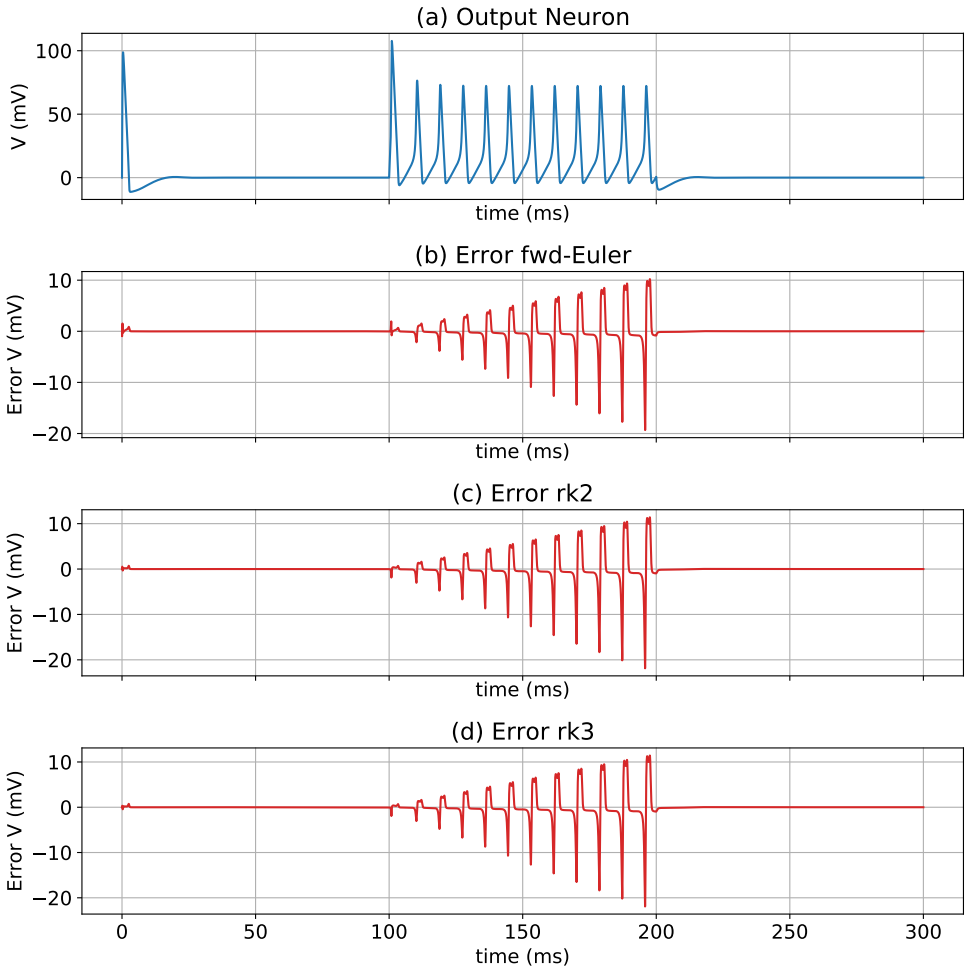


Figure 5.2: (a) Output of the voltage for a simulation in NEURON of a single HH cell. (b) Error between NEURON and fwd-Euler C. (c) Error between NEURON and rk2 C. (d) Error between NEURON and rk3 C. $dt = 0.01$ ms.

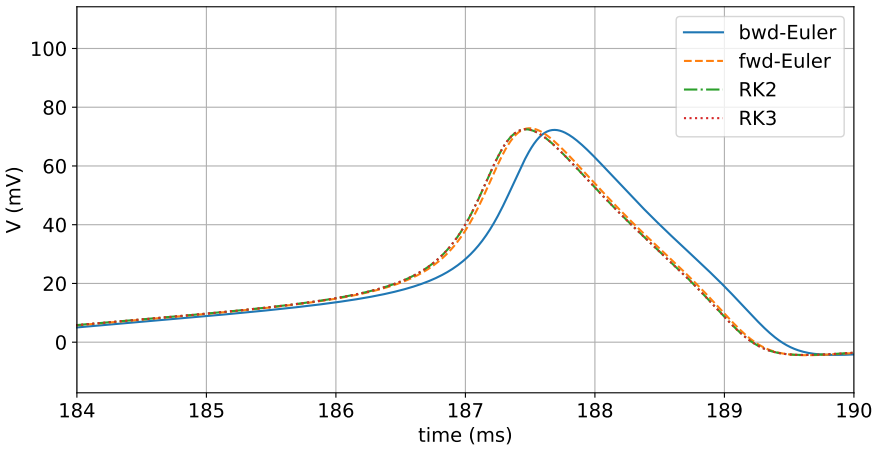


Figure 5.3: Output of the voltage for a simulation in NEURON of a single HH cell, $dt = 0.01$ ms.

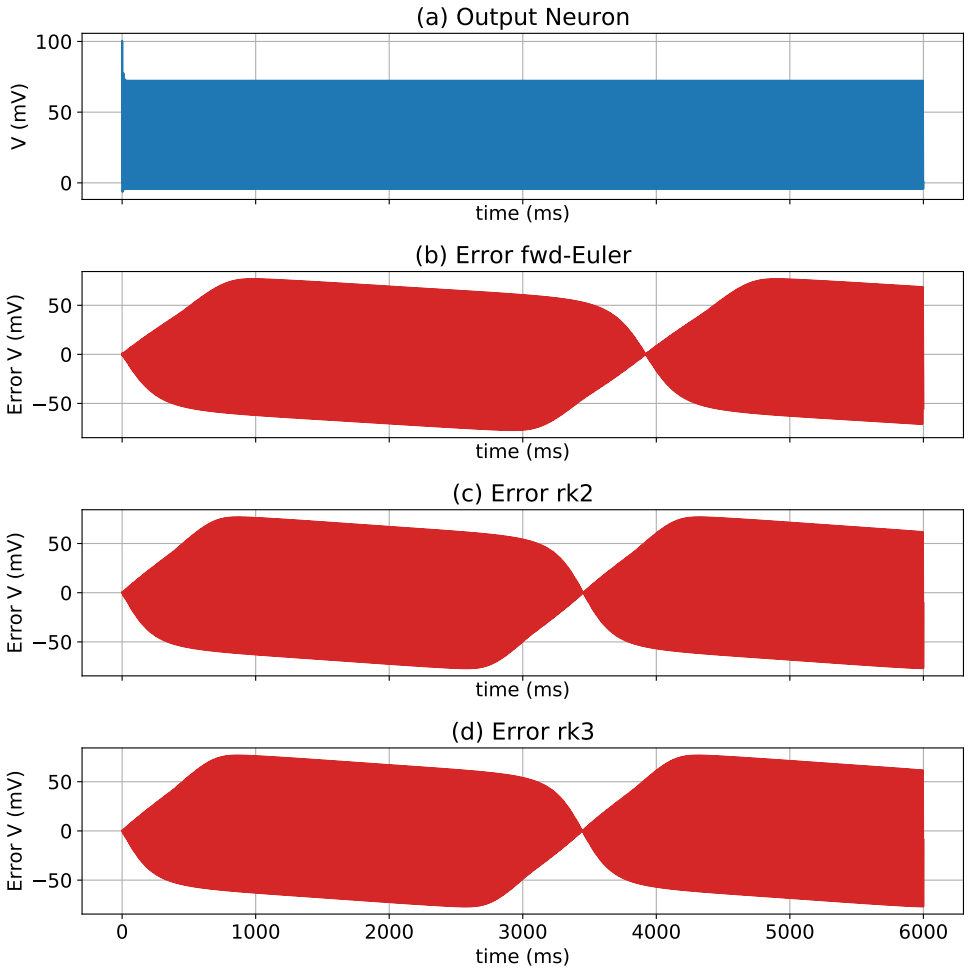


Figure 5.4: (a) Output of the voltage for a simulation in NEURON of a single HH cell. (b) Error between NEURON and fwd-Euler C. (c) Error between NEURON and rk2 C. (d) Error between NEURON and rk3 C. $dt = 0.01$ ms, $t_{sim} = 6000$ ms.

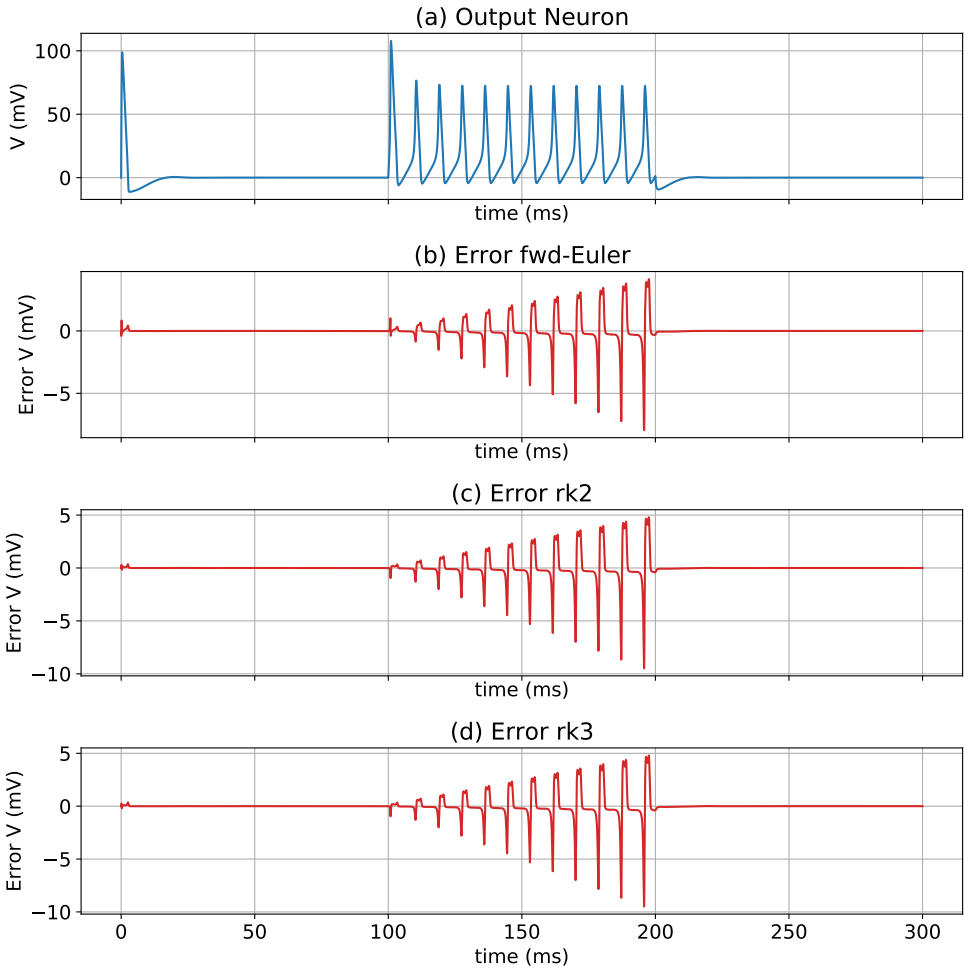


Figure 5.5: (a) Output of the voltage for a simulation in NEURON of a single HH cell. (b) Error between NEURON and fwd-Euler C. (c) Error between NEURON and rk2 C. (d) Error between NEURON and rk3 C. $dt = 0.005$ ms, $t_{sim} = 300$ ms.

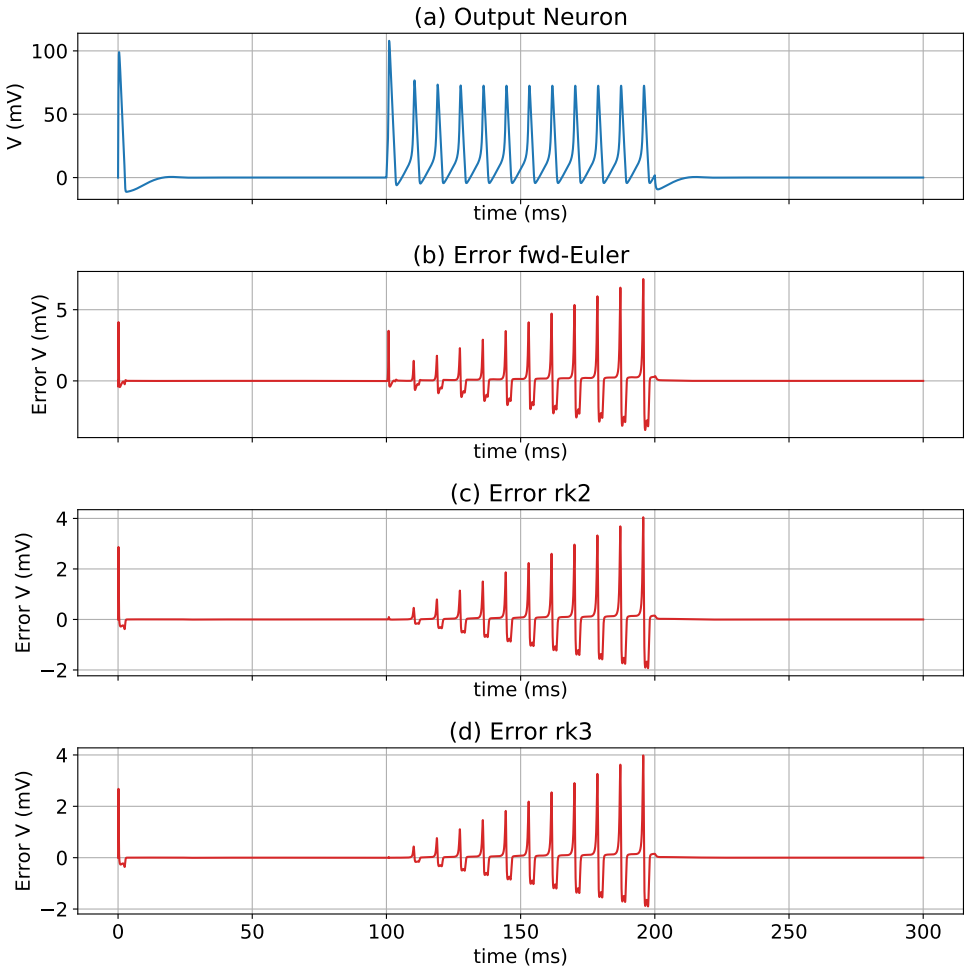


Figure 5.6: (a) Output of the voltage for a simulation in NEURON (using Crank-Nicolson) of a single HH cell. (b) Error between NEURON and fwd-Euler C. (c) Error between NEURON and rk2 C. (d) Error between NEURON and rk3 C. $dt = 0.01$ ms, $t_{sim} = 300$ ms.

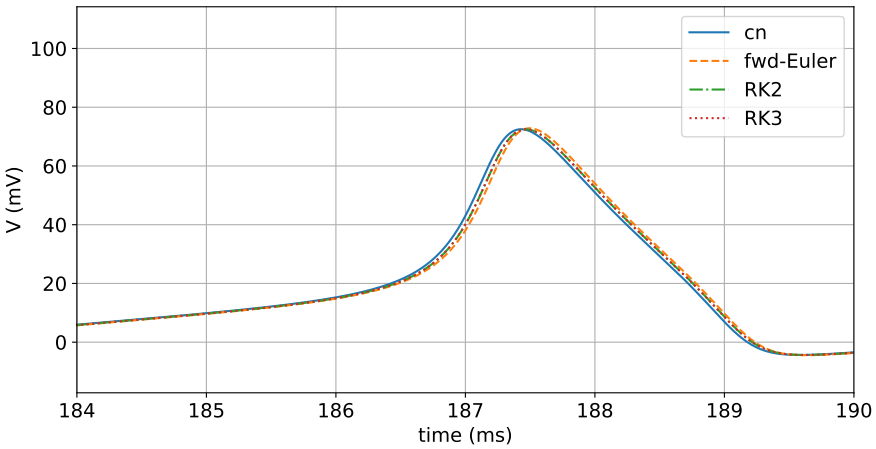


Figure 5.7: Output of the voltage for a simulation in NEURON (using Crank-Nicolson) of a single HH cell, $dt = 0.01$.

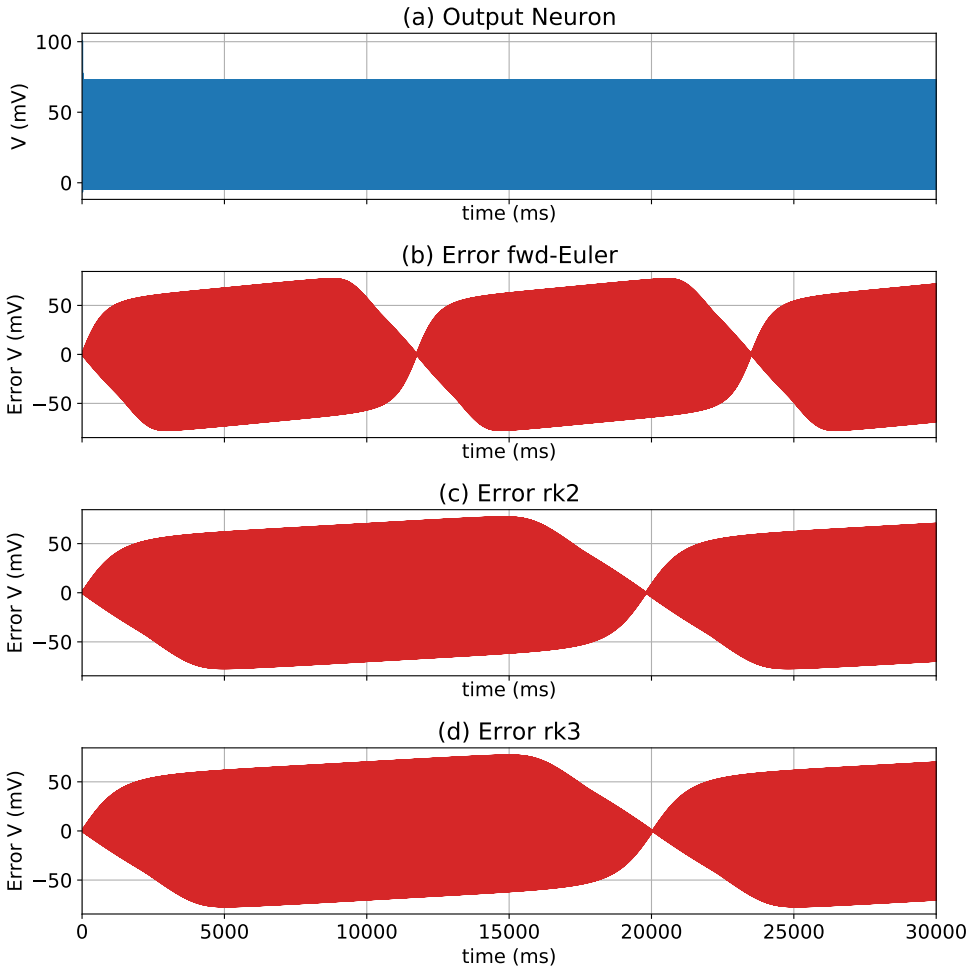


Figure 5.8: (a) Output of the voltage for a simulation in NEURON (using Crank-Nicolson) of a single HH cell. (b) Error between NEURON and fwd-Euler C. (c) Error between NEURON and rk2 C. (d) Error between NEURON and rk3 C. $dt = 0.01ms$, $t_{sim} = 30000ms$.

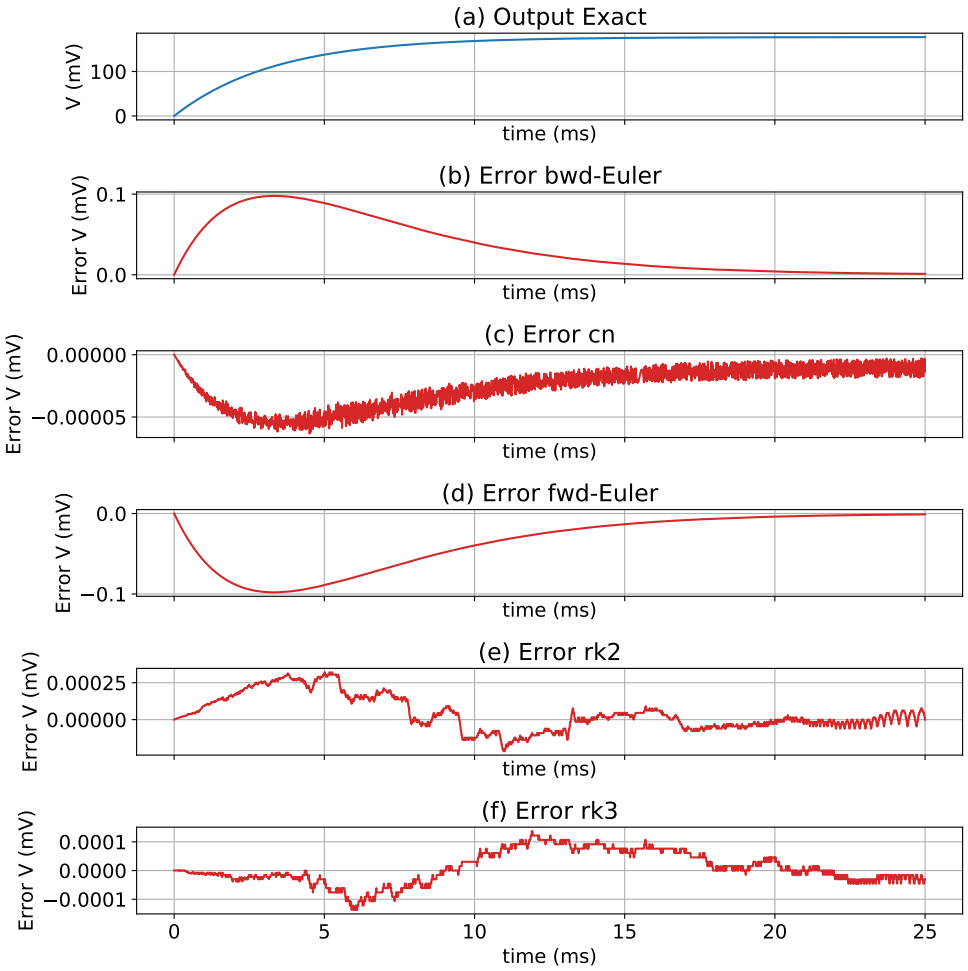


Figure 5.9: (a) Output of the exact solution using Equation (5.1). (b) Error between the exact solution and NEURON using the bwd-Euler method. (c) Error between the exact solution and NEURON using the Crank-Nicolson method. (d) Error between the exact solution and fwd-Euler C. (e) Error between the exact solution and rk2 C. (f) Error between the exact solution and rk3 C. $dt = 0.01$ ms, $t_{sim} = 25$ ms.

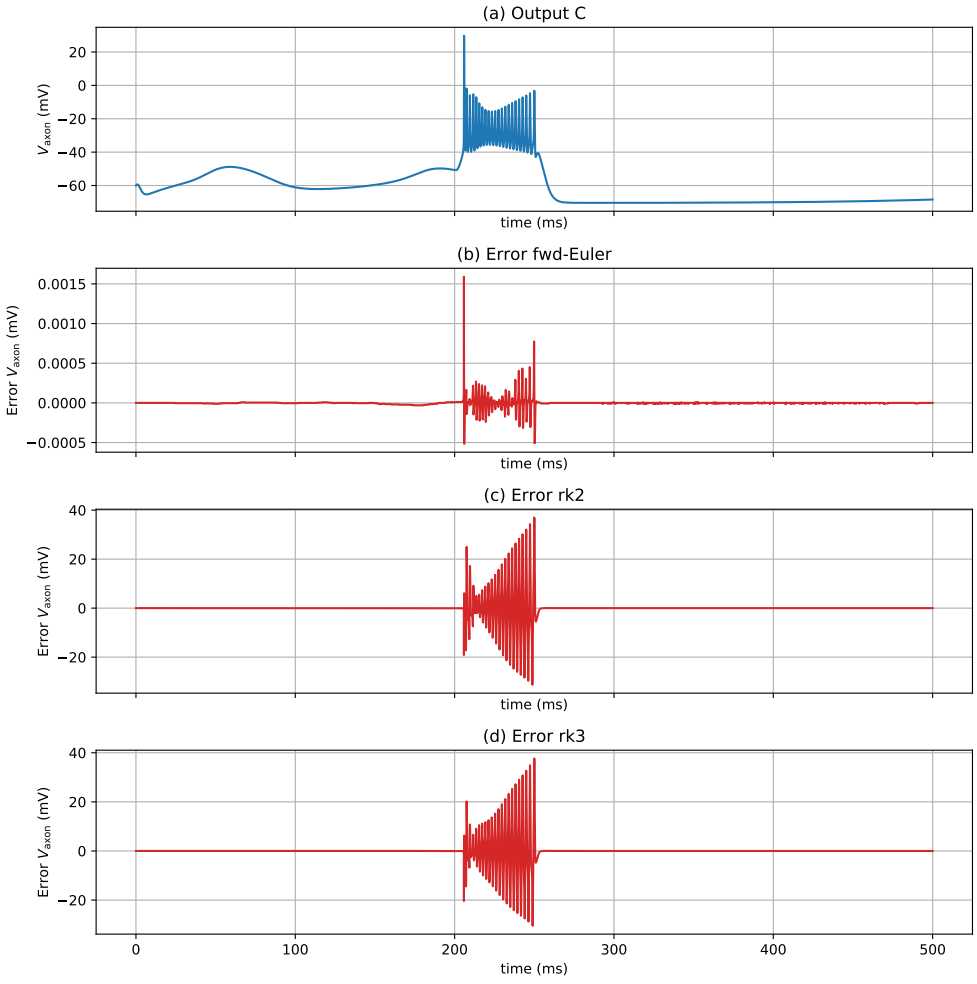


Figure 5.10: (a) Output of the axonal-voltage for a simulation with the reference code for a single cell (cell 0, in the code). (b) Error of fwd-Euler C. (c) Error of rk2 C. (d) Error of rk3 C. $dt = 0.01$ ms.

5.1.2 DFE-code validation

For the validation of the implementations on the DFE, the C-code is used as a reference, as previously mentioned. For each kernel, each ODE solver is compared against its own C counterpart. In the remainder of this section, the structure of presented results will be similar for all kernel instances. This structure is as follows: first, a reference to the initial values is given, after which a plot is shown. This plot shows the output of the forward-Euler C-code and the errors of each of three solvers (fwd-Euler, rk2, and rk3) on the DFE.

5.1.2.1 *HH*

The simulation, whose parameters can be found in Appendix B.1.1.1, to validate the *HH* DFE implementation is the same as the one used to validate the C-code against NEURON-code. The voltage trace and the errors are presented in Figure 5.11. The higher the order of the solver, the higher the error is observed. This can be explained by the fact that, for higher-order solvers, more calculation needs to be done and thus more rounding errors occur. However, the error in case of the third-order is concerning as it shows an increasing error over time, which is not the case when using the other two solvers.
To see if the error of the third-order solver arises from using different hardware, the same DFE-code was run in simulation mode, which means that the DFE-code was executed on the Central Processing Unit (CPU) host. The error of the simulation in comparison with the output of C is shown in Figure 5.12. This shows that the error when running the DFE-code on the CPU in simulation mode is just as large as when the code is run on the DFE.
As the use of different hardware is not the cause of the error between the C and DFE implementation, there are three possible causes for the error:

1. The MaxCompiler does some optimisations while generating the binary for the DFE implementation.
2. The use of generalized functions (as described in Section 4.3) instead of hard-coded functions introduces extra rounding errors.
3. There is a bug in the implementation.

To rule out either the first or second cause, a C implementation is made which also uses the general functions. If those results show the same error as before, then the optimisations of the MaxCompiler can be dismissed as the source of error since for generating the C binary, a different compiler is used. However, if the error is different, then the generalized functions can be dismissed as the source of the error as both C and the DFE use the generalized functions.
The error of the C implementation which uses the generalized functions is shown in Figure 5.13. This error is smaller than when simulating the DFE-code and therefore, the use of generalized functions can be excluded as the source of the error.
To see if either the different compilers or a bug is the source of the error, some variables, of C and DFE implementations which both use generalized functions are investigated.

Table 5.1: Variables during simulation. The index indicates to which stage the variable belongs to.

variable	CPU	DFE
$dvdt_1$	50.00	50.00
$dvdt_2$	49.66040802	49.66040802
$dvdt_3$	49.83286285	49.83285904
v_1	0.52567053	0.52567053
v_2	0.27482155	0.27482155
v_3	0.52399033	0.52388027

Table 5.2: Variables which are used to calculate $dvdt3$.

variable	CPU	DFE
I_{app}	50.00	50.00
I_{Na}	-1.22870898	-1.22870898
I_K	4.52340126	4.52340126
I_{leak}	-3.12755370	-3.12755370
C	1.00	1.00

To see which variable(s) are different, first a simulation of one step is conducted. The period of interest is where I_{app} is non-zero because this is where the error becomes relatively large. Therefore, the initial values for this simulation are set to values of time $t = 100.00$ ms which. Those initial values are shown in Appendix B.1.2.1.
First, we check in which stage the error is visible. Therefore, the values of the voltage derivative and the voltage after each stage are printed. The values show, see Table 5.1, that the error occurs in the third stage.
The derivative of the voltage is calculated with Equation (2.1), to check which variables differ between implementations; those values are printed in Table 5.2. It is interesting that all those variables are the same as those variables to calculate $dvdt3$. A possible explanation for this is that the MaxCompiler has done an optimisation which improves the precision. Therefore, introducing a difference between the C and DFE-code. This difference starts small however, becomes larger because old values are used to update the state variables.

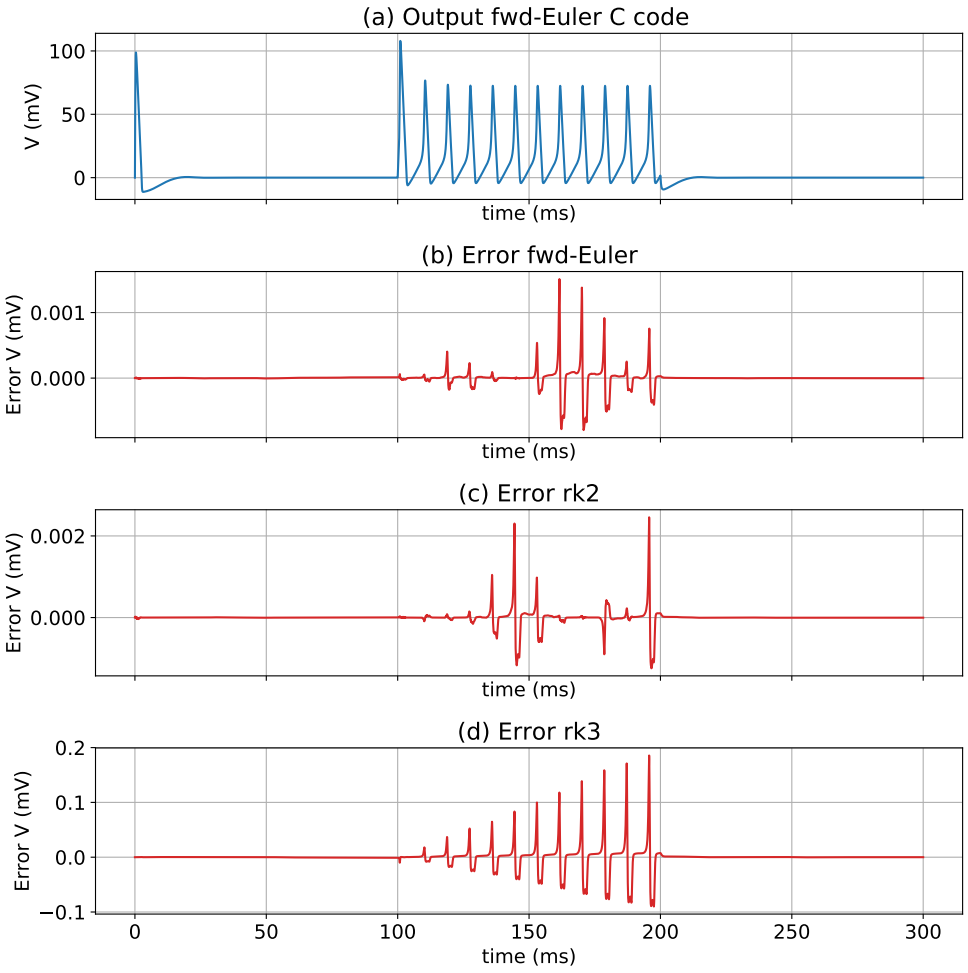


Figure 5.11: (a) Output of the voltage for a simulation in C of a single HH cell. (b) Error between C and the DFE-code using fwd-Euler. (c) Error between C and the DFE-code using rk2. (d) Error between C and DFE-code using rk3. $dt = 0.01$ ms.

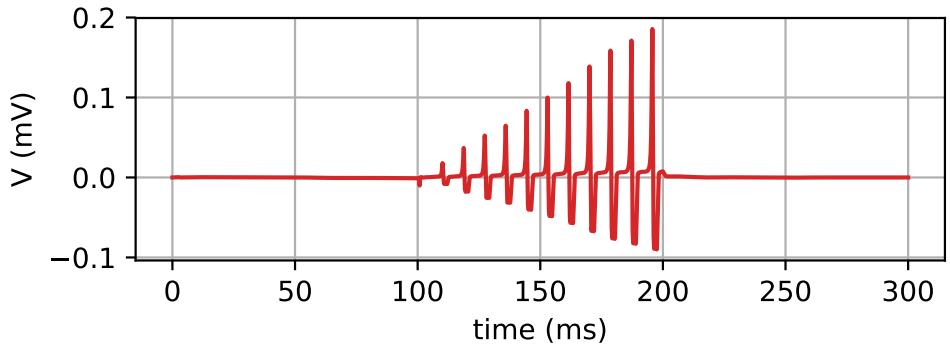


Figure 5.12: Error between C and the DFE-code in simulation mode using rk3. $dt = 0.01$ ms, $t_{sim} = 300$ ms.

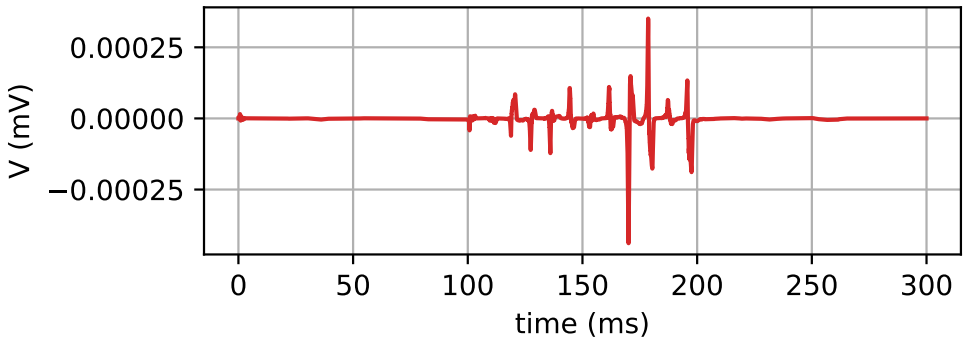


Figure 5.13: Error when using and not using generalized functions in C. $dt = 0.01$ ms, $t_{sim} = 300$ ms.

5.1.2.2 *HH+gap*

For the *HH+gap* implementation, a simulation of an network with 288 cells was simulated. The initial parameters for cell i, the values for the applied current, and the weight values are presented in Appendix B.1.2.2. The output and the errors are depicted in Figure 5.14. Results are similar to the *HH*-model; using higher-order solvers leads to bigger differences between the C and DFE implementation.
The validation of the *HH+gap* implementation was done with the output of a single cell. This is expected to be sufficient because each cell is connected through gap junctions and therefore, an error will manifest within each cell. However, to see the error on a network scale, a new simulation is done and the average error per simulation step is plotted for each cell in a colormap. The initial parameters for cell i can be seen, the parameters for the applied current, and the weight are shown in Appendix B.1.2.3. As shown in the appendix, the weights are different for three different regions in the network. This is done to study if there is a relation between the connectivity of the network and the produced error. Then. the average error per cell using the forward-Euler method is shown in Figure 5.15. The results when the second-order Runge-Kutta is used are shown in Figure 5.16. Figure 5.17 shows the error when the third-order Runge-Kutta method is used.
These figures show that there is no relation between the network connectivity and the error. A relatively higher error is observed for a specific neuron cell in the simulation using the third-order Runge-Kutta method (the error when using the other methods is expected to be small enough to assume functional correctness). Therefore, this cell is inspected further. The voltage trace on both the CPU and DFE of this cell can be seen in Figure 5.18. What stands out from the output is that the simulation on the CPU shows one more spike than the simulation on the DFE.
To find the reason for the discrepancies noticed between the output of the CPU and DFE, two additional tests on the CPU were performed. For these tests the following steps were taken:

1. Retrieve the values of the state variables of the CPU at time 195 ms of the previously discussed simulation.
2. Retrieve the values of the state variables of the DFE at time 195 ms of the previously discussed simulation.
3. Run a simulation, with a duration of 6 ms, on the CPU with the values of the state variables from step 1 as initial values (test 1).
4. Run a simulation, with a duration of 6 ms, on the CPU with the values of the state variables from step 2 as initial values (test 2).

By performing these two tests it could be inspected if the discrepancies were the result of a bug in the DFE code, rounding errors because different errors are used, or a small difference in the values of state variables. The output of both simulations on the CPU can be seen in Figure 5.19. As the both simulations were run with the same hardware and same kernel, it can be concluded ,that based on the different input values, there is a spike or not (it cannot be said if there should be a spike or not as we do not know if the

C or DFE code gives more accurate results). Consequently, the discrepancies are caused by a small difference in the values of the state variables. However, as the third-order Runge-Kutta solver was used to get these disparate results, the functional correctness of this solver on the DFE cannot be guaranteed. Furthermore, although it expected that the differences in state variables at time 195 ms is caused by rounding errors, since different hardware is used, the exact reason for the differences cannot be given, it cannot be concluded what the source of the error is. To find the source of the error this problem should be investigated further.

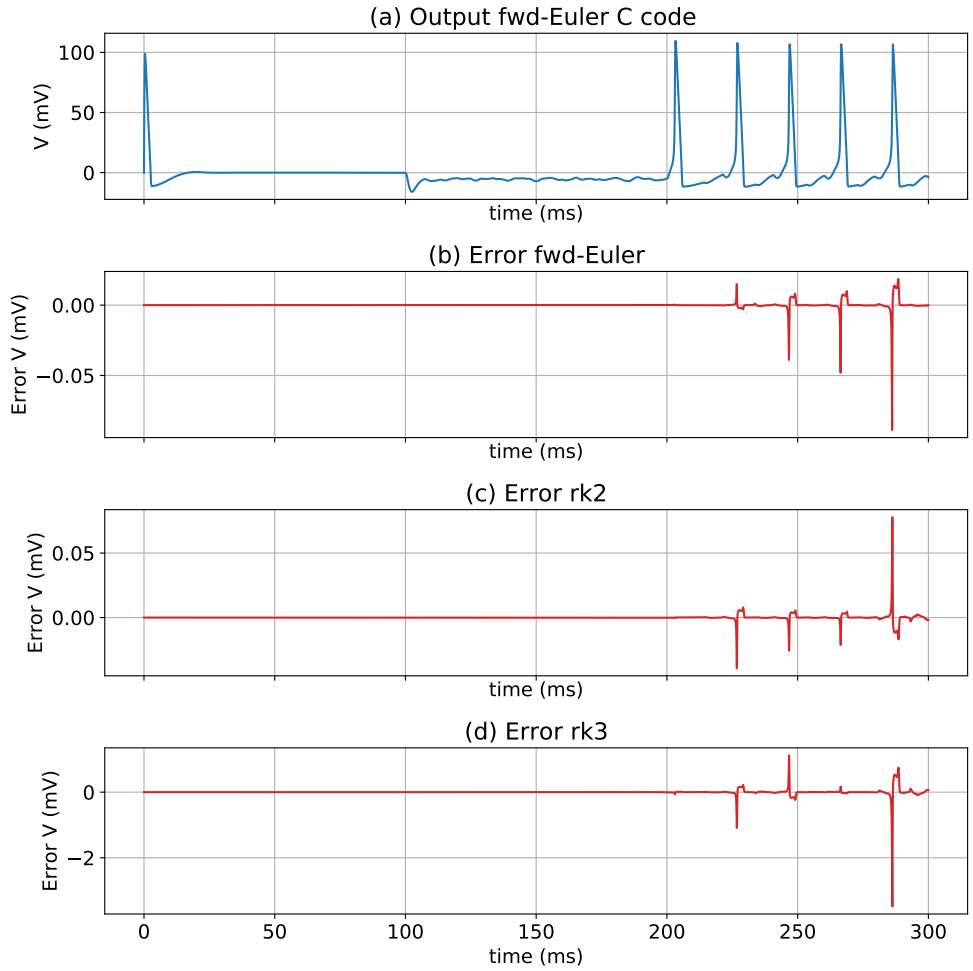


Figure 5.14: (a) Output of the voltage for a simulation in C of a single cell (cell 0, in the code) of a HH network with gap junctions. (b) Error between C and the DFE-code using fwd-Euler. (c) Error between C and the DFE-code using rk2. (d) Error between C and the DFE-code using rk3. $dt = 0.01$ ms.

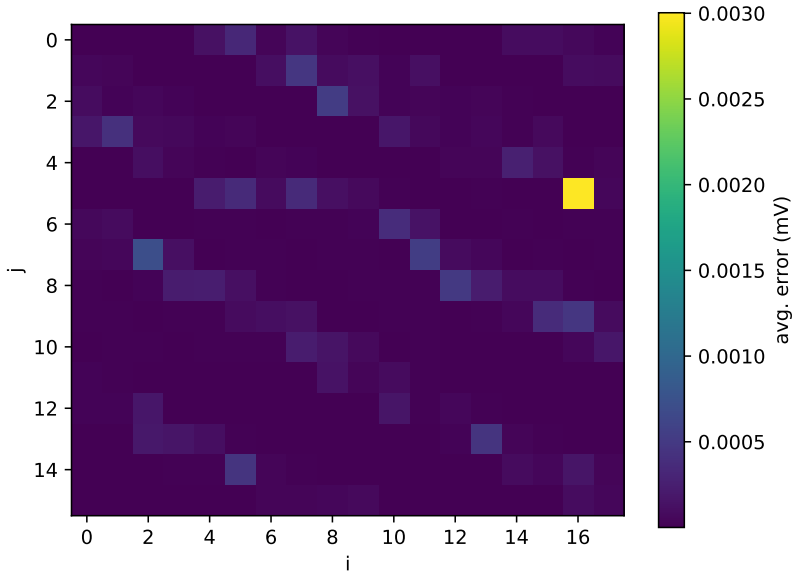


Figure 5.15: Average error per simulation step using the forward-Euler method for a simulation of the *HH+gap* implementation (i and j are indices of a cell).

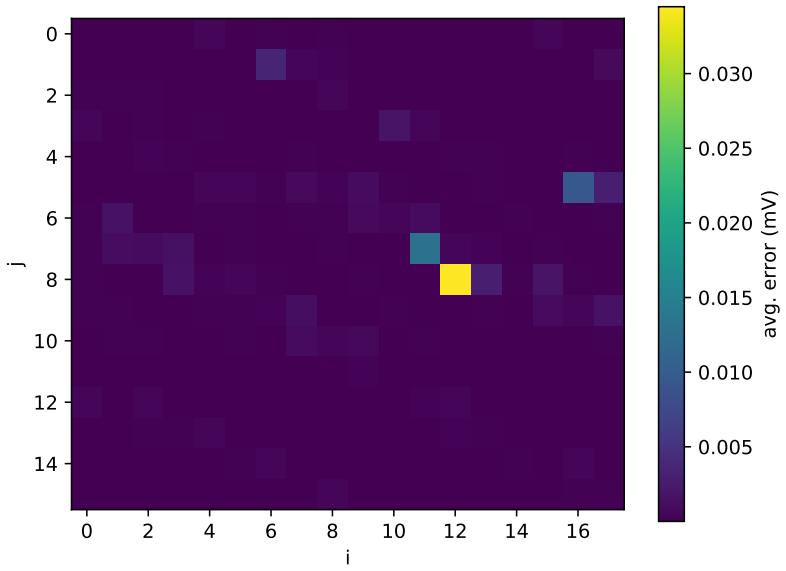


Figure 5.16: Average error per simulation step using the second-order Runge-Kutta method for a simulation of the *HH+gap* implementation (i and j are indices of a cell).

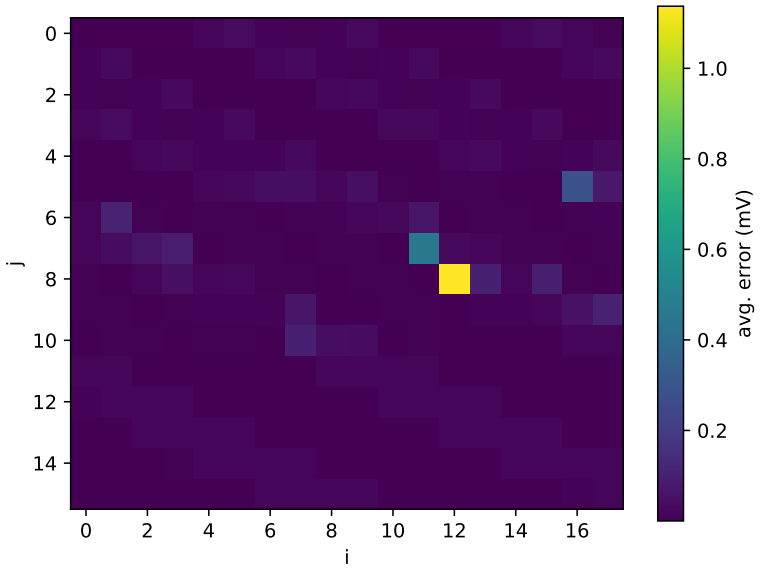


Figure 5.17: Average error per simulation step using the third-order Runge-Kutta method for a simulation of the *HH+gap* implementation (i and j are indices of a cell).

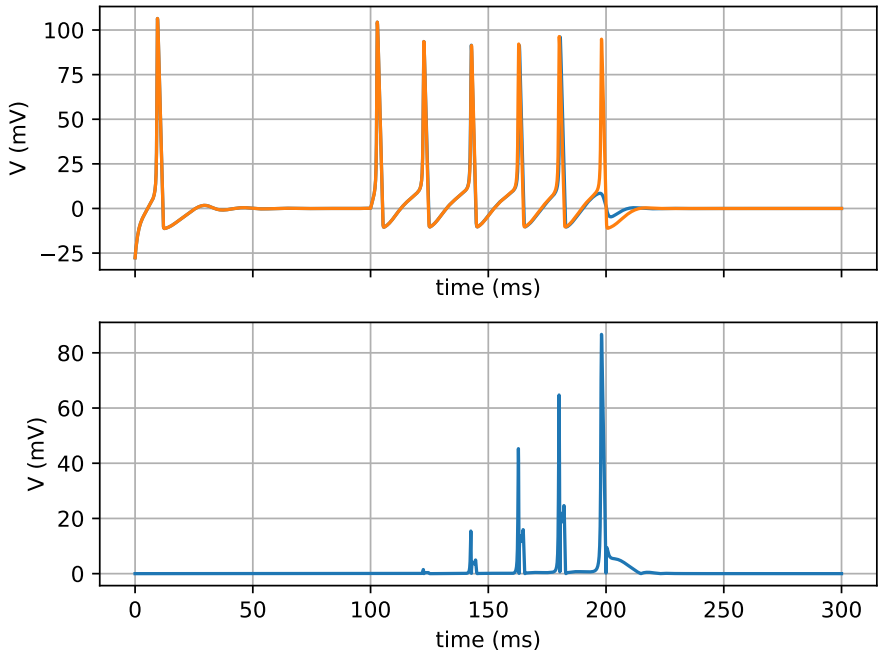


Figure 5.18: (a) Voltage trace, (b) error using the third-order Runge-Kutta method for a simulation of the *HH+gap* implementation.

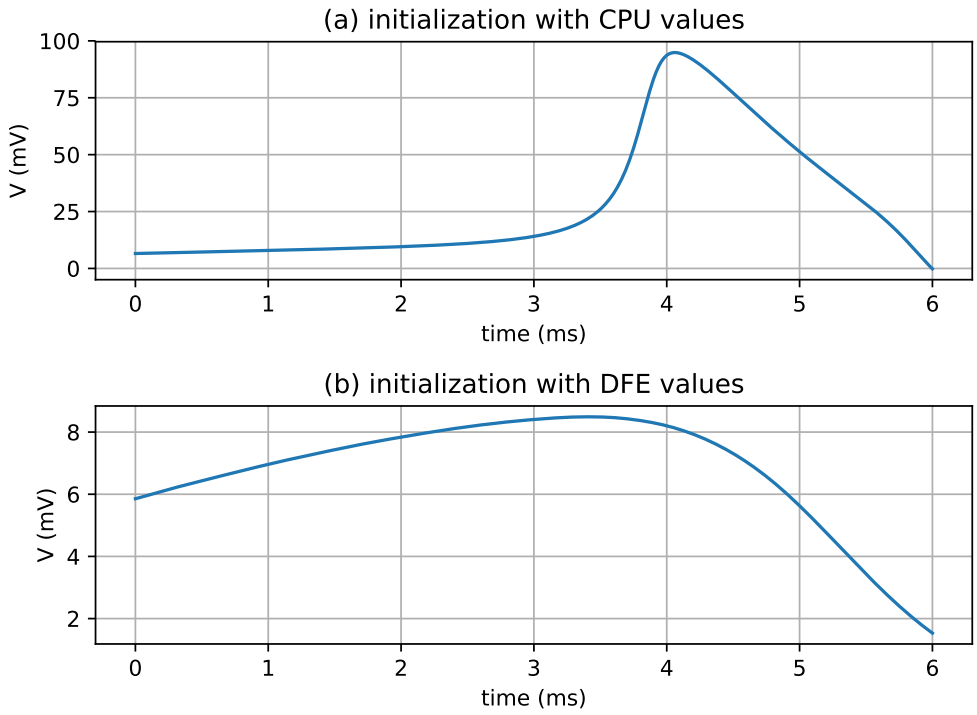


Figure 5.19: Voltage trace of the output of the *HH+gap* rk3 implementation in C (a) Voltage trace with initialization from CPU values, (b) Voltage trace with initialization from DFE values.

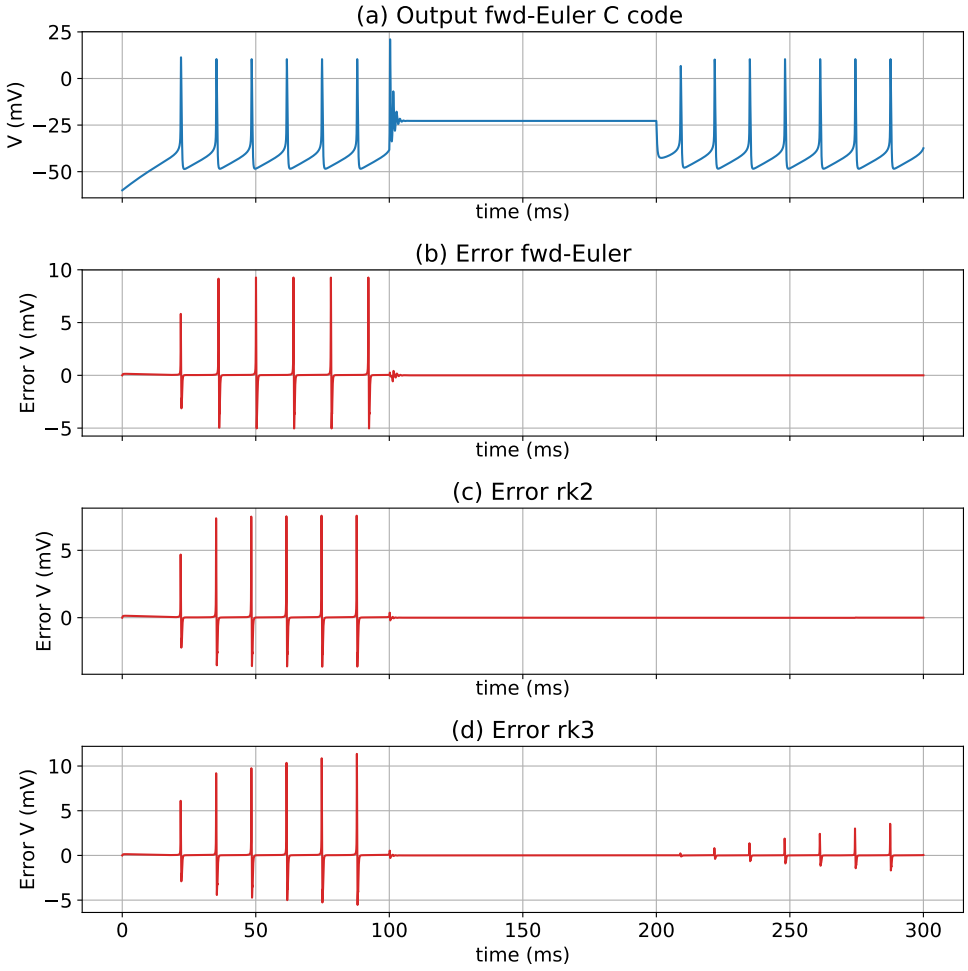


Figure 5.20: (a) Output of the voltage for a simulation in C of a single axon. (b) Error between C and the DFE-code using fwd-Euler. (c) Error between C and the DFE-code using rk2. (d) Error between C and the DFE-code using rk3. $dt = 0.01$ ms.

5.1.2.3 *HH+custom*

The *HH+custom* implementation simulates a single axon where the parameters for the simulation can be found in Appendix B.1.2.4. The output and the errors are depicted in Figure 5.20. The first thing which stands out is the fact that the error for each solver is relatively large in comparison with the voltage trace. The use of custom channels and thus more complex functions may have contributed to this increase of the errors. Secondly, it is interesting to note that the errors are relatively large during the first spiking period in comparison to the second spiking period. It is unknown why this is the case.

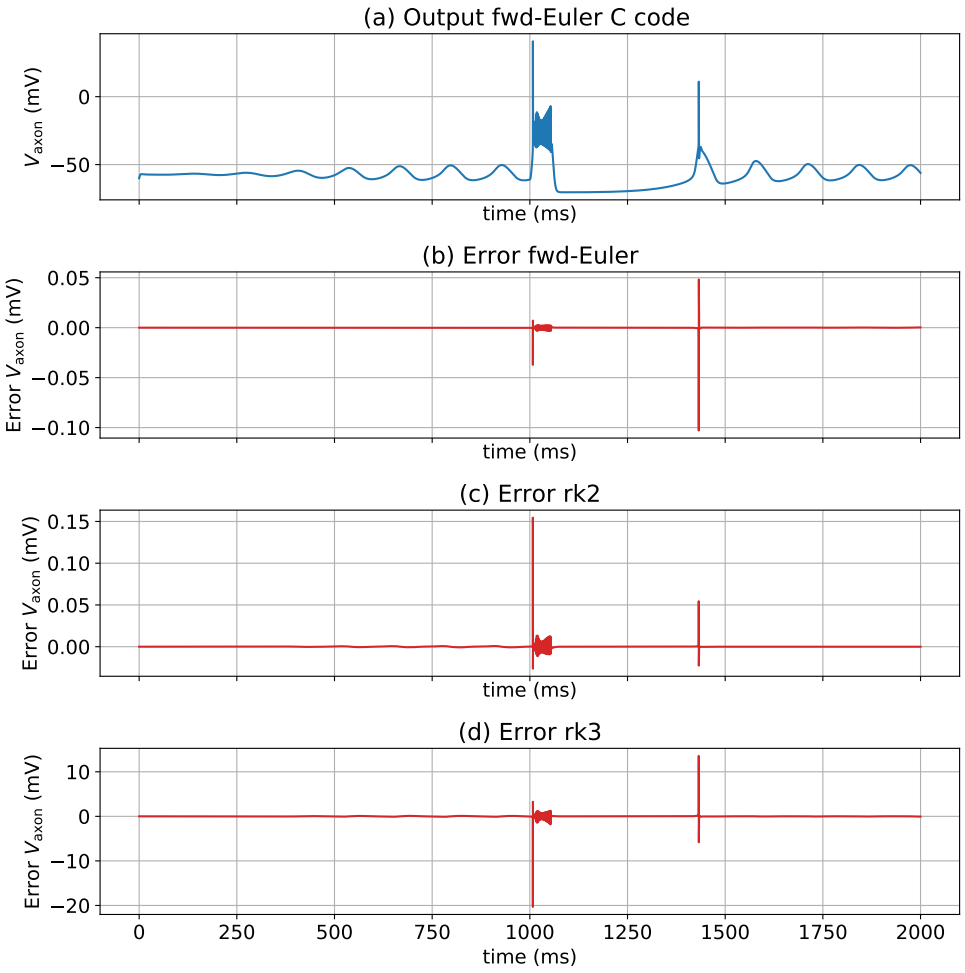


Figure 5.21: (a) Output of the axonal-voltage for a simulation in C of a single IO cell. (b) Error between C and the DFE-code using fwd-Euler. (c) Error between C and the DFE-code using rk2. (d) Error between C and the DFE-code using rk3. $dt = 0.01$ ms.

5.1.2.4 *HH+custom+multi*

The *HH+custom+multi* implementation simulates a single IO cell where the parameters for the simulation can be found in Appendix B.1.2.5. The output and the errors are shown in Figure 5.21. Those results show the same behaviour as discussed in the previous kernel instances.

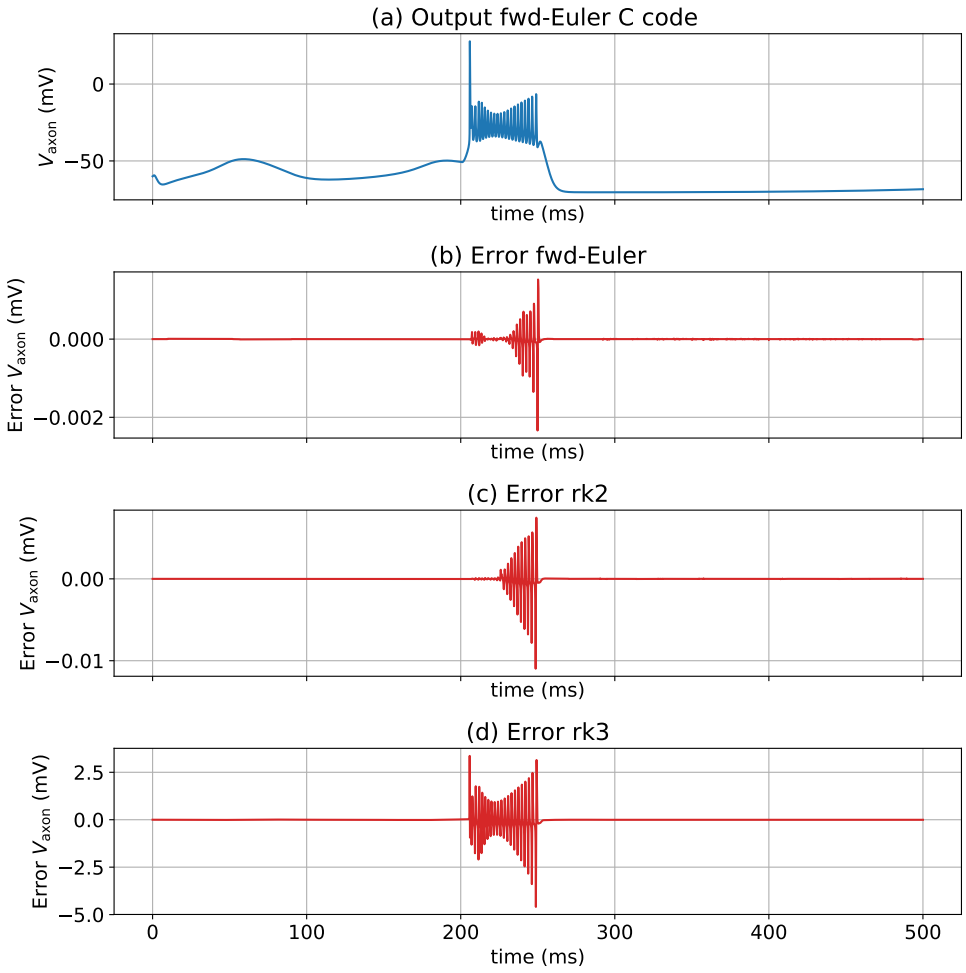


Figure 5.22: (a) Output of the voltage for a simulation in C of a single axon (the axon from cell 0, in the code) of an IO network. (b) Error between C and the DFE-code using fwd-Euler. (c) Error between C and the DFE-code using rk2. (d) Error between C and the DFE-code using rk3. $dt = 0.01$ ms.

5.1.2.5 *HH+custom+multi+gap*

The simulation of the *HH+custom+multi+gap* is the same as the one used to validate the C-code. Appendix B.1.1.3 shows the parameters of this simulation. The output and the errors are depicted in Figure 5.22. Those results show the same behaviour as discussed in the previous kernel instances.

5.1.2.6 Concluding remarks

The errors of most implementations show negligible effect on correctness. In these cases the errors are negligible. However, some of the errors when using the third-order Runge-Kutta method are relatively large in comparison with the output. Although no bug in the code could be found, it cannot be guaranteed that the output of those simulations will be functionally correct as the errors are relatively large in comparison to the output traces, for this specific solver using the default time-step-size. For the other two numerical methods, the errors are small enough to expect that the simulations on the DFE are functionally correct. To gauge the effect step size has on the above observed errors, in the next section we perform an exploration of different step sizes.

5.2 Exploration of time-step-size

After the validation of the different ODE solvers on the DFE, the influence of using a different ODE solver on the maximum time-step-size (dt) will be discussed in this section. To do this for each kernel, a simulation is executed with a dt equal to 0.01 ms using the forward-Euler method. The output of this simulation will be used as a reference, then with each of the solvers the time-step will be increased to see what the influence on the output is. By visually comparing the output between the solvers, an indication can be given of how much the time-step can be changed while still producing a correct trace. The visual comparison is justified as the goal of simulations of neuron models is to give a qualitative insight instead of a quantitative one as the precision of the model parameters is unknown [40].
Naturally it is expected that the higher-order solvers will allow for larger time-step sizes, in the general case, since the global error is presented by $\mathcal{O}(dt^p)$, where p is the order of the solver.

HH

The initial values and the variables used for I_{app} applied on each cell can be seen in Appendix B.2.1. To find the maximum value of dt while still producing a accurate output, the simulations start with a dt equal to 0.01 ms, after which the step-size is increased in increments of 0.01 ms until the simulation does not give any valid results any more. In the case of the forward-Euler solver, the maximum dt is equal to 0.06 ms, results of which can be seen in Figure 5.23. When the dt was further increased the output produced NaN values.
In the case of using the second-order Runge-Kutta solver the maximum dt which gives a correct trace is 0.05 ms. Increasing dt to 0.06 ms changes the shape of the first spike as is shown in Figure 5.24. It is interesting that the maximum time-step-size for the second-order Runge-Kutta solver is lower that the time-step-size of the forward-Euler solver. The third-order Runge-Kutta solver had a higher dt than the other solvers, as it could be increased to 0.07 ms, as is depicted in Figure 5.25. Increasing the time-step-size makes the trace even more inaccurate as NaN values are produced.

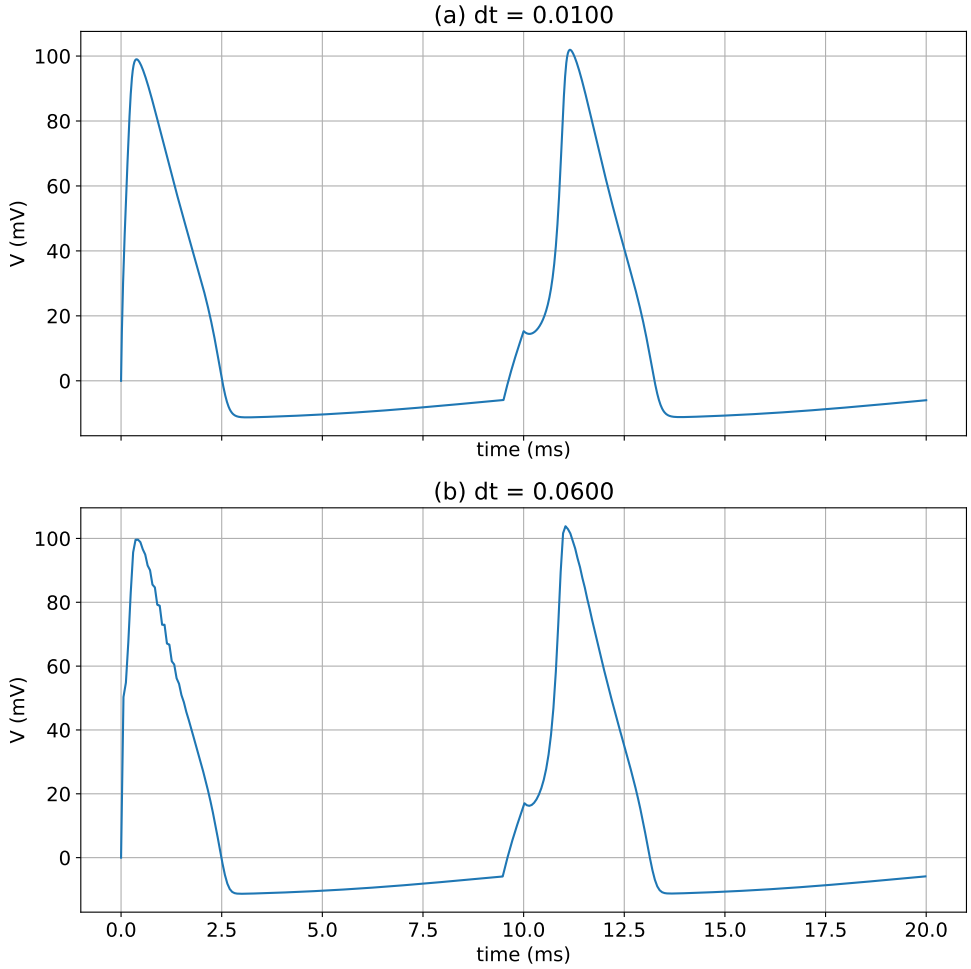


Figure 5.23: Output of the voltage for a simulation of a single HH cell using the forward-Euler solver.(a) $dt = 0.01$ ms, (b) $dt = 0.06$ ms

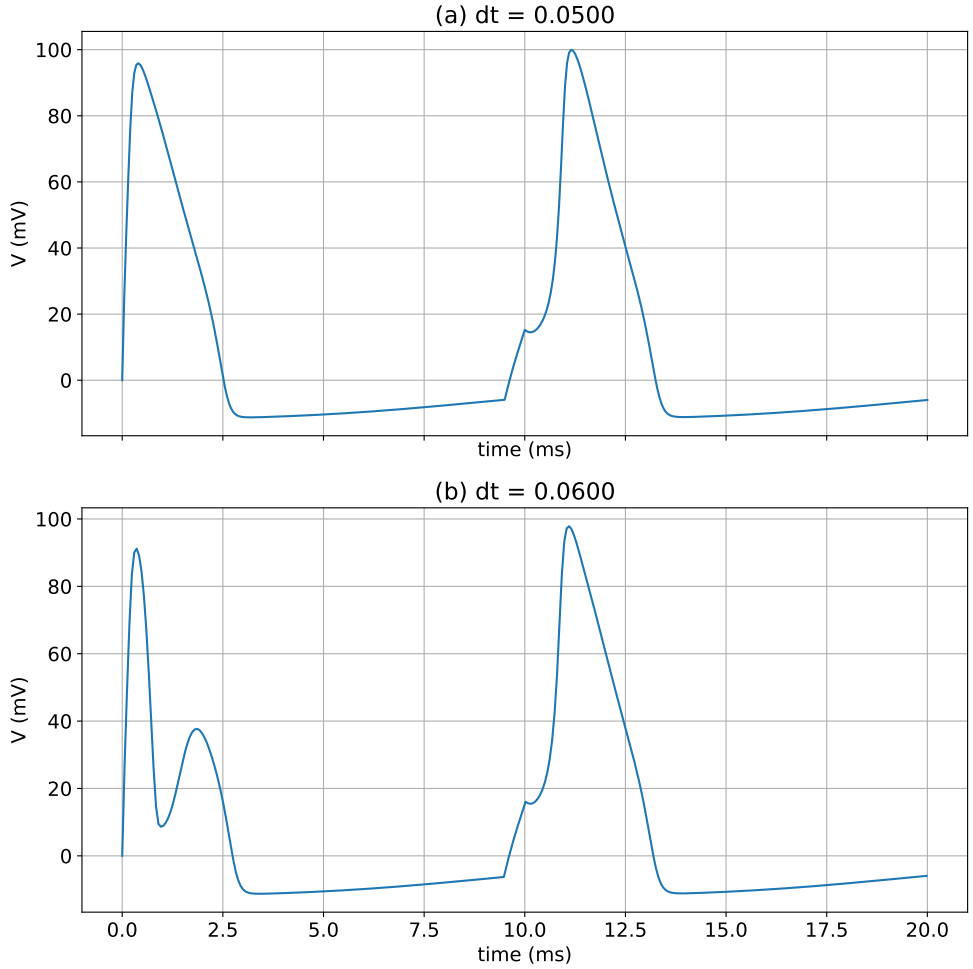


Figure 5.24: Output of the voltage for a simulation of a single HH cell using the second-order Runge-Kutta solver. (a) $dt = 0.05$ ms, (b) $dt = 0.06$ ms

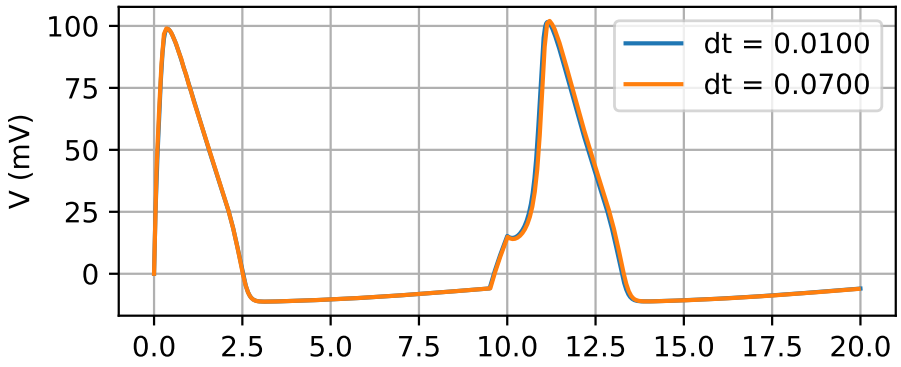


Figure 5.25: Output of the voltage for a simulation of a single HH cell using the third-order Runge-Kutta solver.

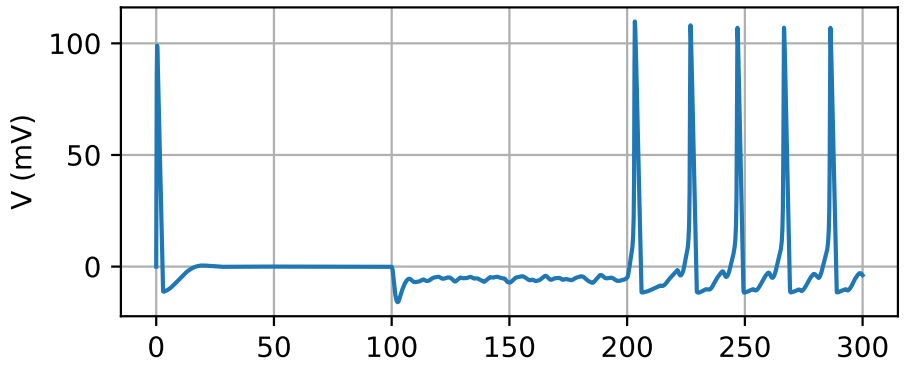


Figure 5.26: Output of the voltage of compartment 0 for a simulation of a *HH+gap* network using the forward-Euler method, $dt = 0.01$ ms.

HH+gap

In the case of the HH+gap, a network of 288 cells is simulated. The initial values, the variables used for I_{app}, and the values for the weights of the connectivity matrix are presented in Appendix B.2.2.
For the simulation of the *HH+gap* network, the maximum dt for each solver is equal to 0.01 ms. The voltage of compartment 0 for this simulation is shown in Figure 5.26. When dt was increased to 0.02 ms all solvers produced NaN values. This is a strong confirmation of the impact gap-junction modelling has on overall model stiffness, thus requiring small time-step sizes to maintain solution stability.

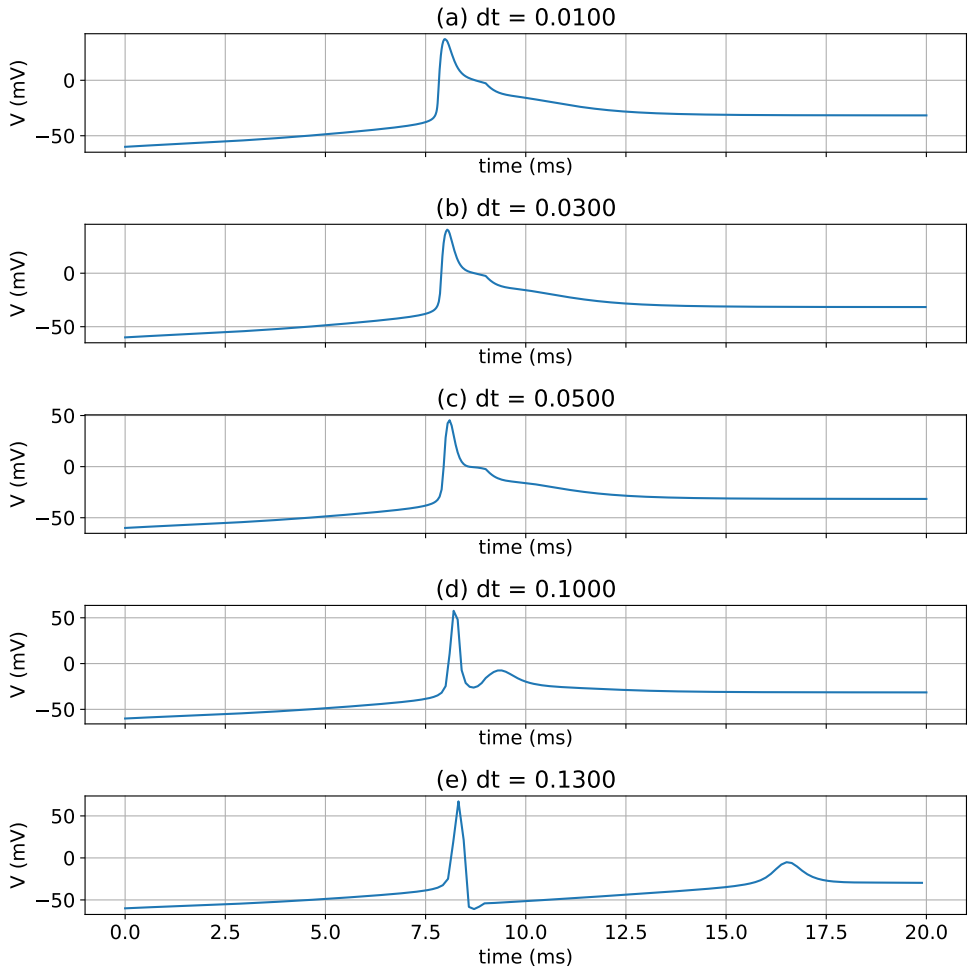


Figure 5.27: Output of the voltage of a single compartment for a simulation of a *HH+custom* (soma) compartment when using the forward-Euler solver for different time-step sizes.

HH+custom

The initial values of cell i used to simulate a soma compartment and additionally, the mathematical description can be seen in Appendix B.2.3.

The time-step-size (dt) is increased for the forward Euler solver from 0.01 ms to 0.13 ms in increments of 0.01 ms. Various simulation voltage traces can be seen in Figure 5.27. We see that for larger dt the behaviour around the spike diverges more from the most accurate solution where dt is 0.01 ms. For a dt larger than 0.05 ms, the shape of the spike is different. However, it must be noted that this is done with visual inspection of the plots and thus not with a specific accuracy specification.

The second-order Runge-Kutta solver allows for using larger time-step sizes in compar-

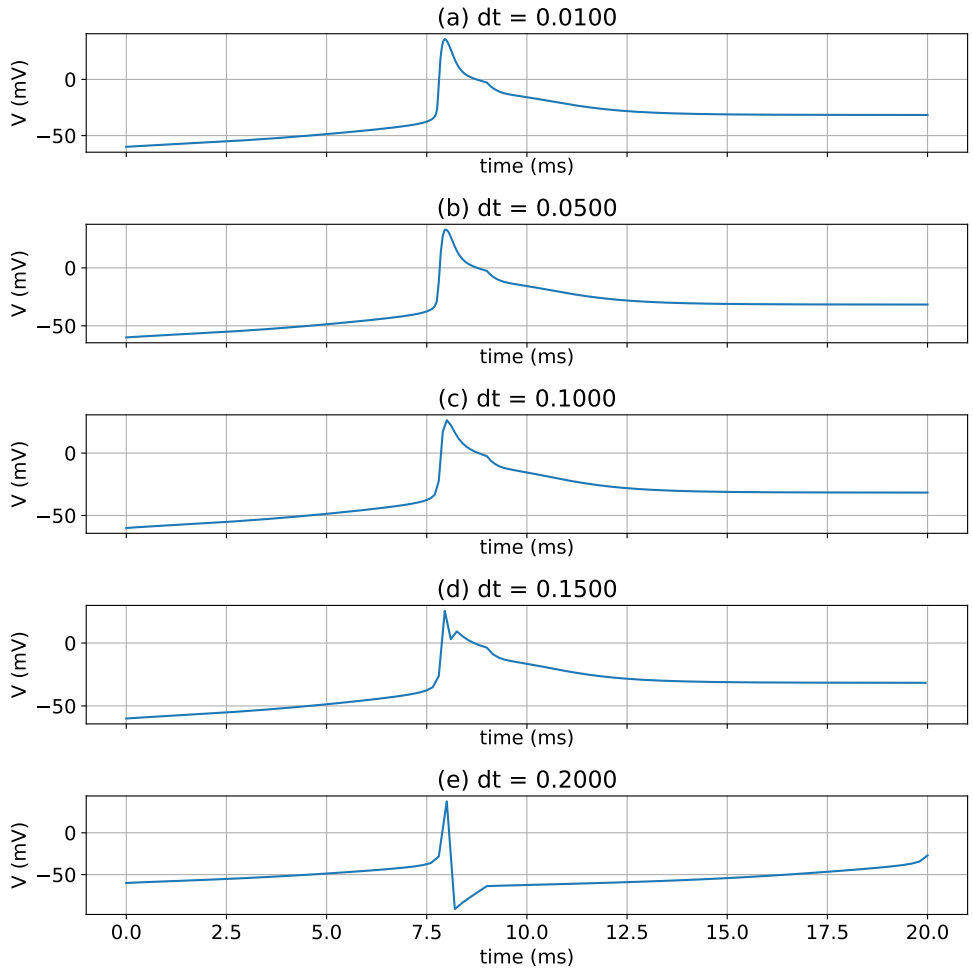


Figure 5.28: Output of the voltage of a single compartment for a simulation of a *HH+custom* (soma) compartment when using the second-order Runge-Kutta solver for different time-step sizes.

ison to the forward-Euler solver, as is shown in Figure 5.28. This figure shows that for a dt of 0.10 ms, only the maximum of the peak is lower. The shape of the peak only changes when dt increases to 0.15 ms or higher. The results in Figure 5.29 show that, for the same time-step sizes as the second-order Runge-Kutta solver, the third-order Runge-Kutta solver is more accurate. This can be seen when inspecting the shape and the maximum voltage of the spike. Furthermore, the shape of the peak starts to change at larger values of dt compared to using the second-order solver. In this case, the trace for a dt equal to 0.15 ms still matches the trace where dt is equal to 0.01 ms.

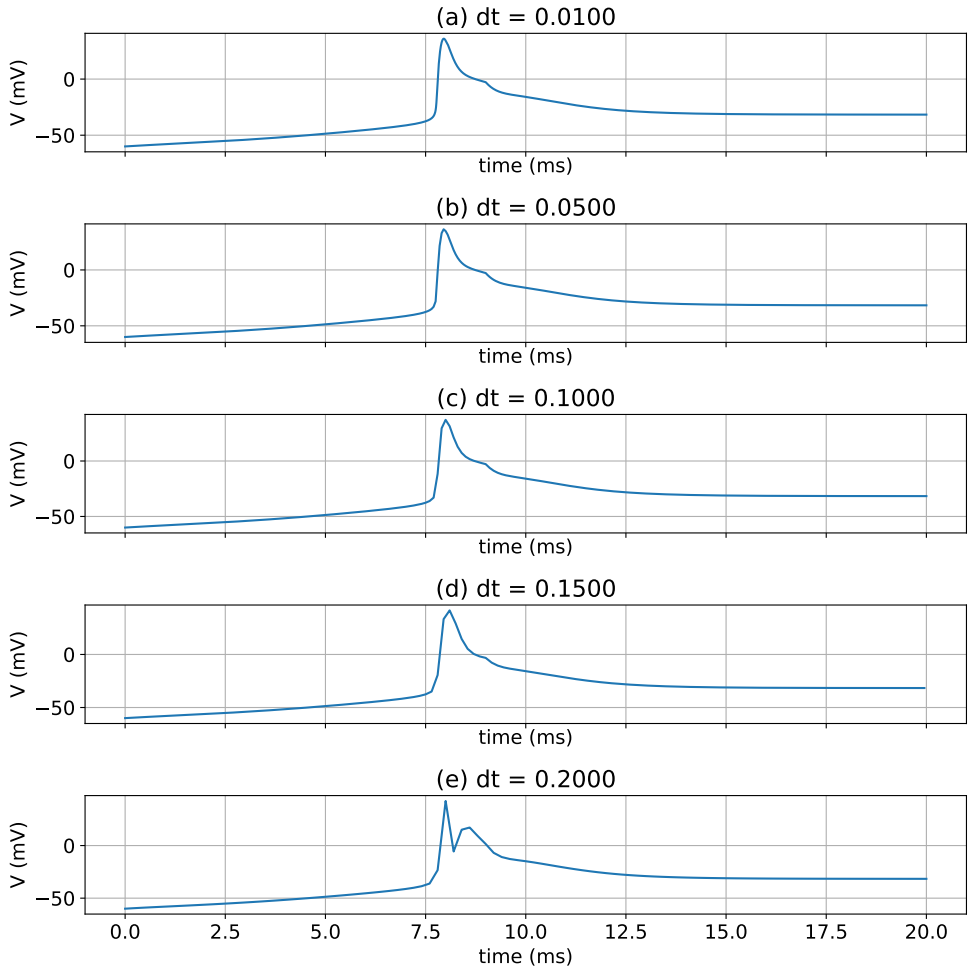


Figure 5.29: Output of the voltage of a single compartment for a simulation of a *HH+custom* (soma) compartment when using the third-order Runge-Kutta solver for different time-step sizes.

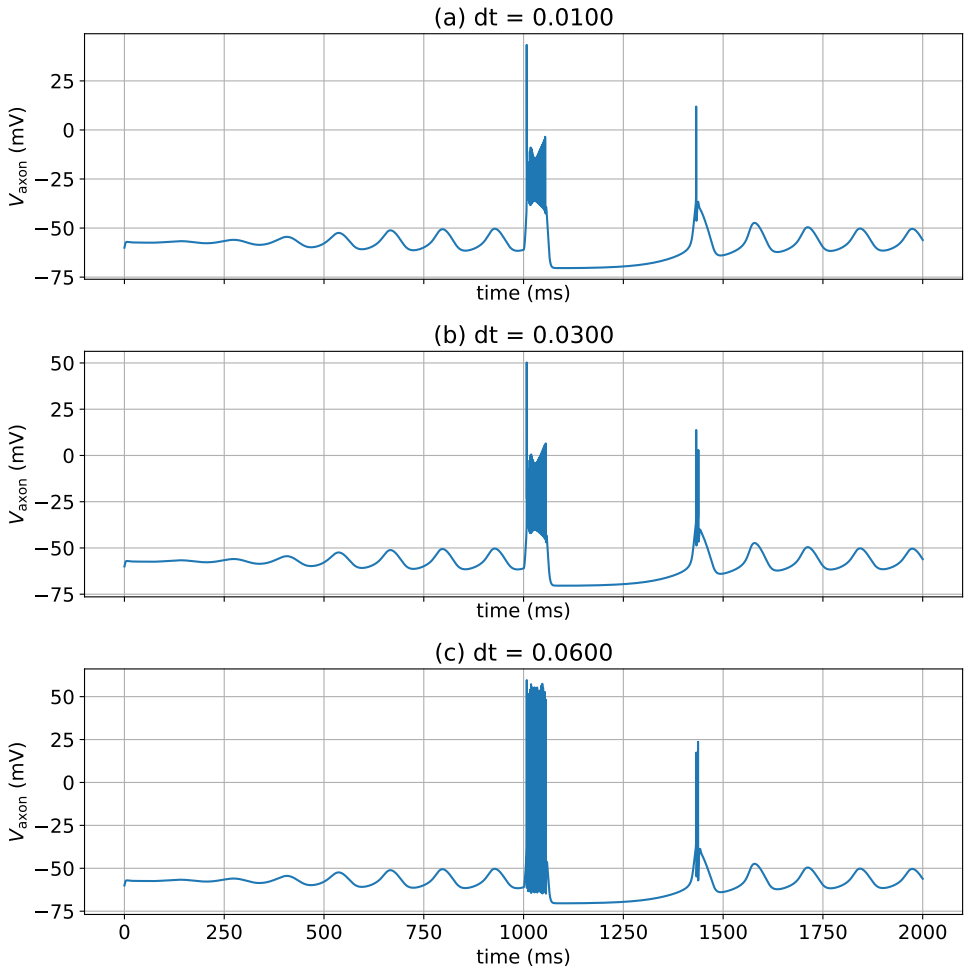


Figure 5.30: Output of the voltage of a single cell for a simulation of a *HH+custom+multi* (IO) cell when using the forward-Euler solver for different time-step sizes.

HH+custom+multi

The initial values of a single IO cell used for the accuracy tests and the mathematical description for the applied current I_{app} can be seen in Appendix B.2.4. The time-step-size (dt) is increased for the forward-Euler solver from 0.01 ms to 0.06 ms. The respective simulation voltage traces are shown in Figure 5.30. This reveals around 1000 ms that for larger dt tested, the amplitude of the post-spike oscillation becomes bigger.

The time-step-size can be increased more for the second-order and third-order solvers , while the amplitude of the oscillations does not vary as much in comparison with the first-order solver, as can be seen in Figure 5.31 and Figure 5.32. To define the maximum step-size, we need to pay attention to the amplitudes of the spikes and the amplitude of the oscillation. For the forward-Euler method, a maximum step-size of 0.03 ms is

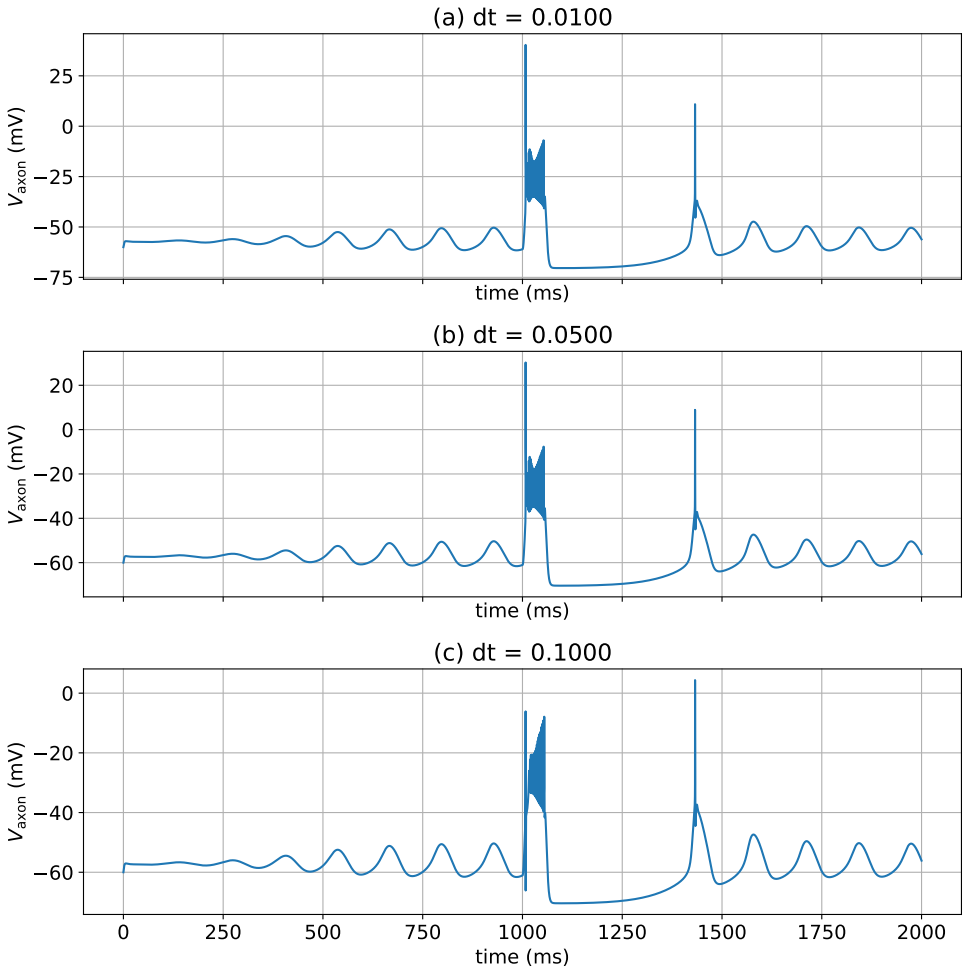


Figure 5.31: Output of the voltage of a single cell for a simulation of a *HH+custom+multi* (IO) cell when using the second-order Runge-Kutta solver for different time-step sizes.

chosen while for both of the Runge-Kutta methods a maximum of 0.05 ms is chosen.

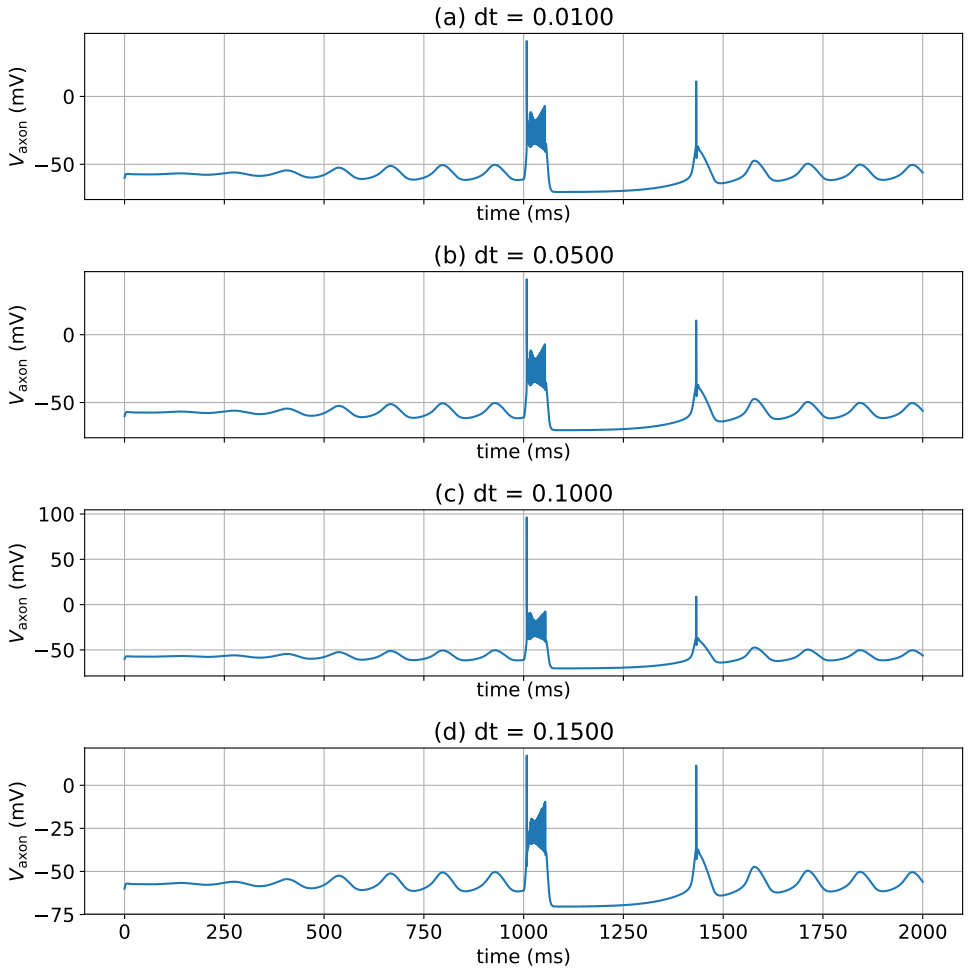


Figure 5.32: Output of the voltage of a single cell for a simulation of a *HH+custom+multi* (IO) cell when using the third-order Runge-Kutta solver for different time-step sizes.

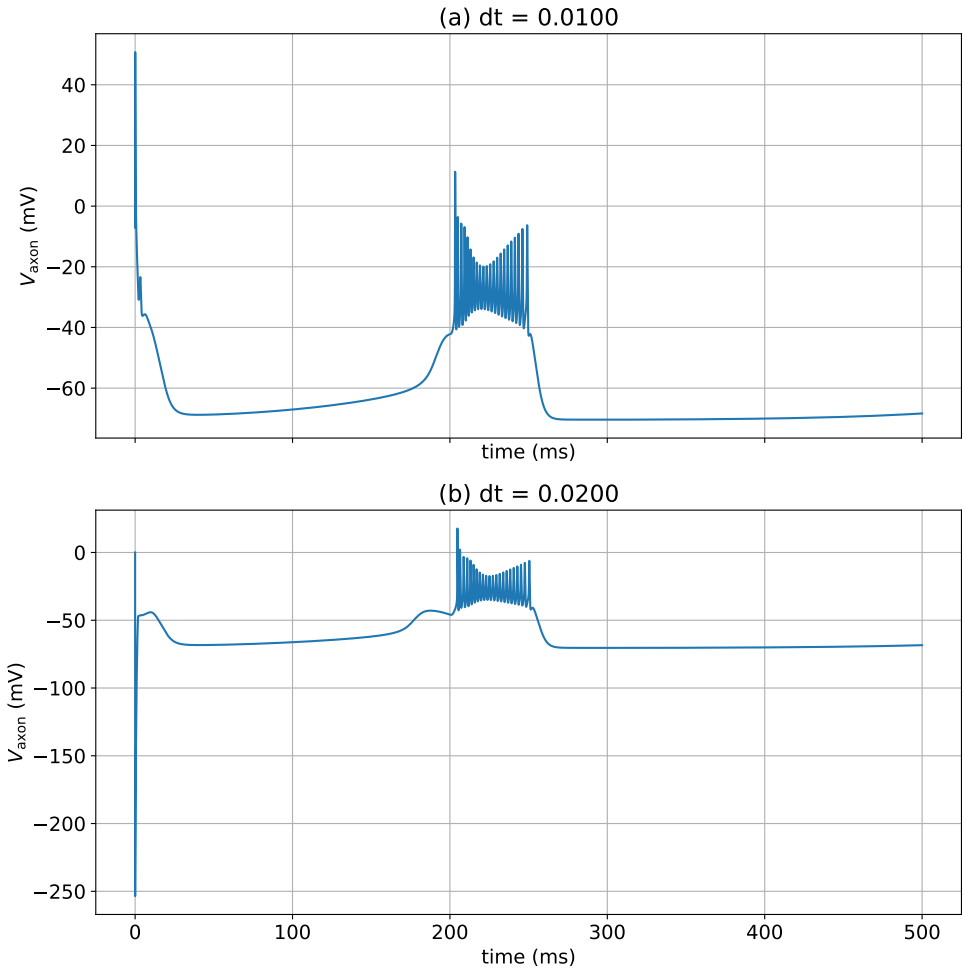


Figure 5.33: Output of the axonal-voltage of cell 0 for a simulation of the IO network.

HH+custom+multi+gap

The initial values of cell i used to simulate an IO network connected by gap junctions, the mathematical description for the applied current I_{app}, and the weights of the connectivity matrix can be seen Appendix B.2.5. Increasing $dt = 0.01$ ms, in increments of 0.01 ms, only led to valid results for the second-order Runge-Kutta solver, with output shown in Figure 5.33, as the other solvers do produce NaN values when dt is increased. Although the simulation with the use of the second-order Runge-Kutta solver does not produce any NaN values, the change of dt still produces a trace which varies from the reference trace as can be seen by the large negative spike at the start of the simulation. Furthermore, it is interesting that the third-order Runge-Kutta solver does not produce a valid output while theoretically having a better precision than the second-order solver.

Table 5.3: Maximum time-step-size in ms for a simulation of each kernel.

Kernel	fwd	rk2	rk3
HH	0.06	0.05	0.07
HH+gap	0.01	0.01	0.01
HH+custom	0.05	0.10	0.15
HH+custom+multi	0.03	0.05	0.05
HH+custom+multi+gap	0.01	0.01	0.01

5.2.1 Discussion

The maximum steps sizes, which can be seen in Table 5.3, do not show an exponential increase of the time-step-size when using higher-order solvers. What is more, the only simulation where the time-step-size can be significantly increased is the simulation of the *HH+custom* kernel. But even in this case, the time-step-size increase is only linear. The computational cost scale linearly. Because the DFEs work with the data-flow principle, this will either influence performance or the resource usage also linearly. Therefore, the extra resources needed when the higher-order solvers are used, could be committed, instead, to achieve a higher unroll factor (in case of the first-order solver). Thus, there is no gain in using higher-order solvers for the use cases in flexHH, given the low observed gains in time-step sizes. The performance, and resource usage of the different numerical methods is discussed in more detail in Section 5.4.1.
These important findings are in line with a recent publication by Börgers et al. [40], the use of different ODE solvers for HH-like models is discussed. It is stated that using explicit numerical methods requires a dt in the order of 0.01 ms. Using larger dts, even for higher-order methods, will have catastrophic results for the simulations. However, it is also stated that the use of such small dts gives often more accuracy than is needed. This is supported by the following statements from [40]:

- The goal of simulations of neuron models is more likely to be for qualitative insight than quantitative precision as there is uncertainty about the precision of the model parameters.
- Constraints on dt are caused by voltage spikes. Therefore, dt must be much shorter than the duration of a voltage spike for the output to be produced accurately.
- Between spikes there is no need to use a small dt, as $dt = 1$ ms can produce adequate accuracy.
- Adaptive time-step-size is not useful when spiking is asynchronous in networks with connected neurons. In such cases, the frequency of spikes in the network is high, rendering time-step adaptivity a useless feature.
- When using implicit methods the dt size must be constrained by the same amount as when using explicit methods, due to convergence conditions of solving a system of equations needed for the implicit solution.

These conclusions indicate that the use of other ODE solvers would not be beneficial in comparison to the forward-Euler solver for HH-like models and, thus, also not for the flexHH-library. Additionally, it shows that the results obtained for the Runge-Kutta solvers are not unique to the simulations done in this thesis. However, to further explore whether the use of higher-order solvers can be beneficial, two other tests in which no spikes are generated at all are done. Therefore, a larger time-step-size is expected for each solver, as the spikes are the constraint for the small time-step-size.

The first test is of a single HH cell for which the initial values can be seen in Appendix B.3.1. The output of the simulation when using a dt equal to 0.01 ms can be seen in Figure 5.34.

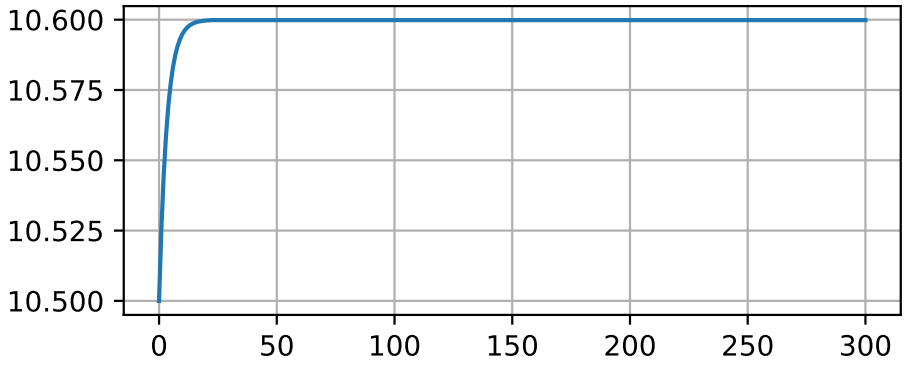


Figure 5.34: Output of the voltage for a relaxed simulation of a single HH cell, $dt = 0.01$ ms.

For the second test, a single IO cell is simulated. The initial values used for this simulation can be seen in Appendix B.3.2 and the output of the axon can be seen in Figure 5.35.

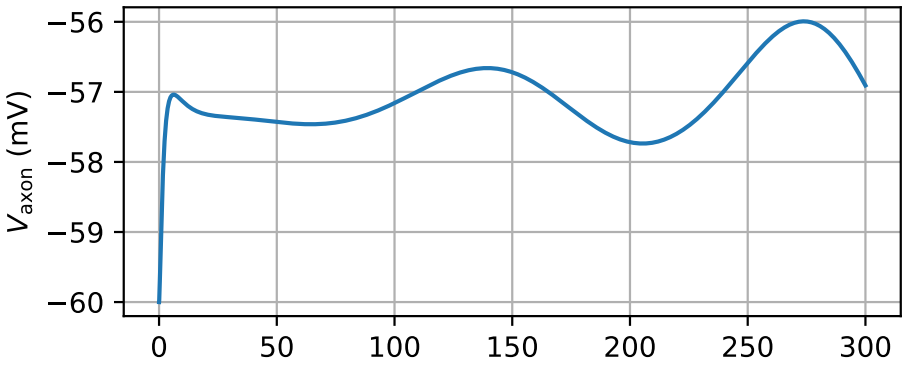


Figure 5.35: Output voltage of the axon for a relaxed simulation of a single IO cell, $dt = 0.01$ ms.

For both tests the time-step was increased as much as possible before the voltage trace became incorrect. The maximum step sizes which could be achieved are presented in

Table 5.4: Maximum time-step-size in ms for relaxed simulations.

Model	fwd	rk2	rk3
HH	0.70	0.70	0.90
IO	0.80	0.80	0.80

Table 5.4. If time steps higher than those reported in the table are used (in increments of 0.1 ms), then the simulation produces NaNs. These results show that, indeed also for relaxed simulations, the use of higher-order solvers is expected not to be beneficial on a DFE. This is because, the computational cost scale linearly and the DFEs work with the data-flow principle, a higher-order solver will either influence performance or the resource usage also linearly. Therefore, the extra resources needed when the higher-order solvers are used, could be committed, instead, to achieve a higher unroll factor (in case of the first-order solver).

5.3 Hardware-resource usage

As previously discussed in Chapter 4, the hardware-usage of the kernel instances of flexHH depends on which features are supported, $N_{comps,max}$[1], $N_{gates,max}$[2], uf[3], and N_{ODE}[4]. The features $N_{comps,max}$, and $N_{gates,max}$ dictate what neural networks can be simulated and how large those networks can be. On the other hand, uf and N_{ODE} influence the performance of the simulations. An increase in any of those variables leads to an increase in hardware-usage. Consequently, the variables influence each other in which values can be chosen, before the hardware-usage reaches the capacity of the DFE. Therefore, it is interesting to inspect how each of those variables influence the hardware-usage. Moreover, as neuroscientists should only choose what neural network (including size of the network) they want to simulate, the performance of the kernel instances should be maximised automatically. An approach to optimizing the performance, is to resynthesize hardware different configurations for each set of the above features in order to get resource usage statistics however, this will be prohibitively time-consuming as a single synthesis cycle requires multiple hours to complete. As a solution, we try to *model and predict* the hardware-usage to get insight how the hardware parameters ($N_{comps,max}$, $N_{gates,max}$, uf, and N_{ODE}) and the features influence the hardware-usage. In the remainder of this section, first the methodology of hardware-usage prediction is discussed. Then, the validation of the prediction method is done and finally, the prediction method is used to inspect the influence of using different features on the hardware-usage.

[1] $N_{comps,max}$ are the maximum number of compartments in a network.
[2] $N_{gates,max}$ are the maximum number of gates per compartment.
[3] uf is the unroll factor.
[4] N_{ODE} is the order of the solver.

Table 5.5: Configurations of the *HH* implementation with different frequencies.

uf	$N_{comps,max}$	$N_{gates,max}$	N_{ODE}	f	LUTs	FFs	BRAMs	DSPs
3	52	10	1	100	143318	195656	2168	106
3	52	10	1	140	143355	196001	2168	106
3	52	10	1	180	143373	196157	2168	106
4	44	10	1	100	158758	216561	2139	127
4	44	10	1	140	158833	216604	2139	127
4	44	10	1	180	158767	216651	2139	127
6	28	10	1	100	194181	261642	2158	169
6	28	10	1	140	194197	261942	2158	169
6	28	10	1	180	194180	261840	2158	169
8	12	10	1	100	229475	310643	2224	219
8	12	10	1	140	229442	311099	2224	219
8	12	10	1	180	229444	311724	2224	219

5.3.1 Methodology for the prediction of hardware-usage

For the prediction, it is assumed that the clock frequency does not influence the hardware-usage. To confirm this assumption, four configurations using the same hardware parameters (uf, $N_{comp,max}$, $N_{gates,max}$, N_{ODE}) of the *HH* kernel instance are synthesized with different frequencies. The results, which can be seen in Table 5.5, show that the influence of the frequency on the hardware-usage is minimal. Therefore, we can conclude that the assumption that the operating frequency does not influence the hardware is valid. A possible explanation for this might be that because of the data-flow principle a deep pipeline is made to increase performance. As a side effect the stages could function with different frequencies without changing the hardware much, as the path of a single stage is relatively small.
To predict the resource usage, the kernels are divided into the same kernel segments as those discussed in Chapter 4 and some more, which will be introduced below. For each of the segments discussed in Chapter 4, the scaling factor is known as is the same as discussed in Chapter 4. The scaling factor per segment for each kernel instance can be seen in Tables 5.6 to 5.10. To predict the hardware-usage using the scaling factors, a reference for the hardware-usage is required. By using the reference and calculating the ratio between the scaling factors of the configuration which needs to be predicted and the reference configuration, using Equation (5.2), the hardware-usage can be predicted, using Equation (5.3). The hardware-usage per segment of the reference can be obtained by parsing the annotated kernel file (in the annotated kernel file per line of code the resource usage is given).

$$ratio(hw_{params}, hw_{params,ref}) = \frac{factor_{scaling}(hw_{params})}{factor_{scaling}(hw_{params,ref})} \tag{5.2}$$

where

Table 5.6: Scaling factors per kernel segment for *HH* kernel instances.

Segment	Scaling factor
vMem	$N_{comps,max}$
yMem	$\left\lceil \frac{N_{gates,max} \cdot N_{comps,max}}{uf} \right\rceil \cdot uf$
calcIGates	$N_{gates,max} \cdot N_{ODE}$
iGate	$(uf-1)N_{ODE}$
pipe	$uf \cdot N_{ODE}$
proc	N_{ODE}
control	constant

Table 5.7: Scaling factors per kernel segment for *HH+custom* kernel instances.

Segment	Scaling factor
vMem	$N_{comps,max}$
yMem	$\left\lceil \frac{N_{gates,max} \cdot N_{comps,max}}{uf} \right\rceil \cdot uf$
calcIGates	$N_{gates,max} \cdot N_{ODE}$
iGate	$(uf-1) \cdot N_{ODE}$
pipe	$uf \cdot N_{ODE}$
proc	N_{ODE}
control	constant

$$hw_{params} = (uf, N_{comps,max}, N_{gates,max}, N_{ODE})$$
$$hw_{params,ref} = (uf_{ref}, N_{comps,max,ref}, N_{gates,max,ref}, N_{ODE,ref})$$

$$res_{segment}(hw_{params}) = res_{segment,ref}(hw_{params,ref}) \cdot ratio(hw_{params}, hw_{params,ref}) \quad (5.3)$$
where
$$hw_{params} = (uf, N_{comps,max}, N_{gates,max}, N_{ODE})$$
$$hw_{params,ref} = (uf_{ref}, N_{comps,max,ref}, N_{gates,max,ref}, N_{ODE,ref})$$

There are more segments than the previously discussed ones, which contribute to the total hardware-usage of the kernel instances in the DFE. The file, which gives a complete overview of the hardware-usage on the DFE (`report.txt`) and is generated by the MaxTools, shows the segments ”kernel extra” and ”manager” Finally, there is also hardware used for things such as I/O streams, First In First Out (FIFO) buffers and memory controllers. Those things are placed under the label ”rest”. The "kernel extra", "manager", and "rest" segments are all automatically generated. Therefore, it is not

Table 5.8: Scaling factors per kernel segment for *HH+custom+multi* kernel instances.

Segment	Scaling factor
vMem	$N_{comps,max}$
yMem	$\left\lceil \frac{N_{gates,max} \cdot N_{comps,max}}{uf} \right\rceil \cdot uf$
calcIGates	$N_{gates,max} \cdot N_{ODE}$
iGate	$(uf - 1) \cdot N_{ODE}$
pipe	$uf \cdot N_{ODE}$
proc	N_{ODE}
control	constant
calcICompartment	N_{ODE}

Table 5.9: Scaling factors per kernel segment for *HH+gap* kernel instances.

Segment	Scaling factor
vMem	$N_{comps,max} \cdot N_{ODE} \cdot uf$
yMem	$\left\lceil \frac{N_{gates,max} \cdot N_{comps,max}}{uf} \right\rceil \cdot uf*$
calcIGates	$N_{gates,max} * N_{ODE}$
proc	constant
control	constant
iCellMem	$N_{comps,max}$
iGapMem	$N_{comps,max}$
gapProc	uf
gapControl	constant

known what resources are in those segments and thus no scaling factor could be given of how the particular segment will scale. However, by inspecting the hardware-usage of different configurations, it was found out which parameter influences which kernel segment. The results of the inspection can be seen in Table 5.11. It is likely that the "manager" is affected by how many pipelines it drives and therefore, is influenced by uf). Additionally, "kernel extra" could be anything and therefore, it is not surprising that more hardware parameters influence this segment.

Although an exact scaling cannot be given it is still handy to predict the resource usage of the automatically generated segments. To be able to predict, for now a first-order approach, based on the parameters in Table 5.11, is used. The first-order approach is made with the use of Python scripts. In the python scripts, to predict "manager" `polyfit` from the Python `numpy` library was tried. The `polyfit` from `numpy` was able to predict "manager" as its hardware-usage is only dependent on one parameter. If more a prediction with more parametersneeds to be predicted (as is the case with

Table 5.10: Scaling factors per kernel segment for *HH+custom+multi+gap* kernel instances. *N_{ODE} is added after inspection of hardware-usage.

Segment	Scaling factor
vMem	$N_{comps,max} \cdot nODE \cdot uf$
yMem	$\left\lceil \frac{N_{gates,max} \cdot N_{comps,max}}{uf} \right\rceil \cdot uf \cdot N_{ODE}$*
calcIGates	$N_{gates,max} \cdot N_{ODE}$
proc	constant
control	constant
calcICompartment	N_{ODE}
iCellMem	$N_{comps,max}$
addressMem	$N_{comps,max} \cdot uf$
iGapMem	$N_{comps,max}$
gapProc	uf
gapControl	constant

Table 5.11: Overview of which hardware parameters influence the hardware-usage of which kernel segment.

Segment	Hardware parameters
kernel extra	uf, $N_{gates,max}$, N_{ODE}
manager	uf
rest	constant

"kernel extra"), `polyfit` will not be working as it only supports linear models with one variable. Consequently, another library which supports multiple parameters is used, called `sklearn`, for the prediction of "kernel extra". Finally, to use the prediction methods reference configurations (with their hardware-usage) are required. This as their hardware-usage is required as a reference. The kernel segments from Tables 5.6 to 5.10 require one reference configuration, to use the scaling factor to predict the hardware-usage of these segments. The `polyfit` library requires two reference configurations and `sklearn` requires four reference configurations. Consequently, in total four reference configurations are used as the hardware-usage of those configurations can be reused.

5.3.2 Hardware-usage prediction validation

The previously discussed method to predict the hardware-resource usage is simple and its purpose is then only to get a good first indication of the hardware-usage on the DFE. To validate how accurate the predictions are for each kernel instance the predicted hardware-usage is compared against the actual hardware-usage of multiple configurations for all the kernel instances. The results of the hardware-usage are all

Table 5.12: Reference configurations used for the hardware-usage prediction of the *HH* kernel instances.

Segment	uf	$N_{comps,max}$	$N_{gates,max}$	N_{ODE}
kernel & rest	4	12288	10	1
kernel extra	2	12288	10	1
	4	12288	6	1
	4	12288	10	1
	4	12288	10	2
manager	2	12288	10	1
	4	12288	10	1

Table 5.13: Reference configurations used for the hardware-usage prediction of the *HH+custom* kernel instances.

Segment	uf	$N_{comps,max}$	$N_{gates,max}$	N_{ODE}
kernel & rest	2	16384	10	1
kernel extra	2	16384	6	1
	2	16384	10	1
	4	16384	10	1
	2	16384	10	2
manager	2	16384	10	1
	4	16384	10	1

Table 5.14: Reference configurations used for the hardware-usage prediction of the *HH+custom+multi* kernel instances.

Segment	uf	$N_{comps,max}$	$N_{gates,max}$	N_{ODE}
kernel and rest	4	24	10	1
kernel extra	4	12	4	1
	4	24	10	1
	2	16	10	2
	1	4	10	3
manager	4	24	10	1
	2	16	10	2

Table 5.15: Reference configurations used for the hardware-usage prediction of the *HH+gap* kernel instances.

Segment	uf	$N_{comps,max}$	$N_{gates,max}$	N_{ODE}
kernel & rest	12	12288	10	1
kernel extra	12	12288	10	1
	12	12288	6	1
	4	12288	10	1
	16	12288	10	2
manager	12	12288	10	1
	4	12288	10	1

Table 5.16: Reference configurations used for the hardware-usage prediction of the *HH+custom+multi+gap* kernel instances.

Segment	uf	$N_{comps,max}$	$N_{gates,max}$	N_{ODE}
kernel & rest	4	12288	10	1
kernel extra	4	12288	10	1
	4	12288	6	1
	6	12288	10	1
	4	12288	10	2
manager	4	12288	10	1
	6	12288	10	1

taken from the Maia DFE; the available resources on this DFE can be seen in Table 5.17. Additionally, because during the inspection of the resource usage it was found that the Block Random-Access Memories (BRAMs) always were the limiting factor while synthesizing, only the accuracy of the prediction of the BRAMs is inspected. The used configurations for the validation of the prediction method can be found in Appendix C. The results per kernel segment for each kernel instance are shown in Tables 5.18 to 5.22.

For the discussion on how accurate the hardware predictions are, the accuracy of the total resource prediction can be seen in Table 5.23. One finding is that if a kernel instance has more segments, the error becomes larger. This is as expected as there are more sources for error. The results show that the maximum mean error is 6.60 %. Therefore, it is expected that the prediction method can give a good first indication of the hardware-usage on a Maia DFE.

Table 5.17: Available resources on the Maia DFE.

	available resources
LUTs	524800
FFs	1049600
BRAMs	2567
DSPs	1963

Table 5.18: Error of the resource prediction of *HH* configurations as a percentage of the available resources on the Maia DFE.

Segment	mean (%)	std (%)
total	3.38	3.56
vMem	0.02	0.03
yMem	-0.12	0.69
calcIchannels	-0.09	0.65
iChannels	0.01	0.06
pipe	0.70	1.21
proc	0.12	0.20
control	0.21	0.28
extra kernel	-0.40	2.70
manager	0.46	1.45
rest	0.14	0.30

Table 5.19: Error of the resource prediction of *HH+custom* configurations as a percentage of the available resources on the Maia DFE.

Segment	mean (%)	std (%)
total	2.77	5.46
vMem	0.03	0.07
yMem	0.33	0.58
calcIChannels	0.21	0.69
iChannels	0.04	0.15
pipe	0.87	2.49
processing	0.02	0.11
control	0.24	0.47
extra kernel	0.09	1.47
manager	0.31	1.07
rest	0.03	0.08

Table 5.20: Error of the resource prediction of *HH+custom+multi* configurations as a percentage of the available resources on the Maia DFE.

Segment	mean (%)	std (%)
total	3.92	5.37
vMem	0.27	0.73
yMem	0.83	2.26
calcIChannels	-0.24	0.69
iChannels	0.06	0.19
iComp	-0.03	0.06
pipe	1.32	3.06
processing	0.05	0.13
control	0.13	0.36
extra kernel	-0.11	0.83
manager	0.40	1.24
rest	0.14	0.23

Table 5.21: Error of the resource prediction of *HH+gap* configurations as a percentage of the available resources on the Maia DFE.

Segment	mean (%)	std (%)
total	5.51	6.62
vMem	0.30	1.73
yMem	-1.28	1.44
iMem	0.01	0.02
calcIChannels	-0.76	1.55
processing	-0.33	0.98
control	0.00	0.00
extra kernel	0.54	1.32
iGapMem	0.01	0.02
gapFunct	-1.60	2.89
gapControl	0.00	0.00
manager	-0.09	0.12
rest	-0.26	0.37

5.3.3 Influence of model features on hardware-usage

Besides the hardware parameters $N_{comps,max}$, $N_{gates,max}$, uf, and N_{ODE}, the support of extra features also influences the hardware-usage. To see the influence on the hardware-usage of different features different kernel instances will be compared. For the comparison between the kernel instances, the hardware-usage is obtained by using the prediction model. This has a drawback that the hardware-usage is not perfectly accurate, as the

Table 5.22: Error of the resource prediction of *HH+custom+multi+gap* configurations as a percentage of the available resources on the Maia DFE.

section	mean (%)	std (%)
total	6.60	6.60
vMem	-3.04	5.90
yMem	-0.42	1.02
iMem	0.01	0.03
gapAddressMem	-0.96	1.91
calcIChannels	-0.35	0.71
calcIComp	-1.18	1.44
processing	-0.01	1.05
control	0.51	0.72
extra kernel	-0.19	0.80
iGapMem	0.01	0.03
gapFunct	-0.54	2.05
gapControl	0.00	0.00
manager	-0.02	0.12
rest	-0.08	0.26

Table 5.23: Error of the total resource prediction for all kernel instances as a percentage of the available resources on the Maia DFE.

Kernel	mean (%)	std (%)
HH	3.38	3.56
HH+gap	5.51	6.62
HH+custom	2.77	5.46
HH+custom+multi	3.92	5.37
HH+custom+multi+gap	6.60	6.60

predictions give a small error. However, the predictions are good enough to get a good impression of how the kernel instances scale.

Firstly, the influence of using custom channels is analysed. This is done by choosing the same hardware parameters and then comparing the hardware-usage between the *HH* and *HH+custom* kernel instances. This is interesting because both implementations have the same kernel segments with the same scaling factors. The hardware parameters are set to: $uf = 1$, $N_{comps,max} = 1024$, $N_{gates,max} = 1$, and $N_{ODE} = 1$. The used BRAMs for each kernel segment can be seen in Table 5.24.

The difference of BRAMs used for `yMem`, `iChannels`, and `proc` is interesting. Although the same code is used in both implementations, there is still a difference. This can be expected as heuristics are used for the synthesis of the kernel instances. This difference

Table 5.24: BRAMs used for *HH*, *HH+custom*, and *HH+custom+multi* kernel instances with the hardware parameters, $uf = 2$, $N_{comps,max} = 10240$, $N_{gates,max} = 4$, and $N_{ODE} = 1$.

Segment	*HH*	*HH+custom*	*HH+custom+multi*
total	1089.50	1175.00	1461.00
vMem	17.00	17.00	52.00
yMem	70.00	65.00	65.00
calcIchannels	42.00	48.00	51.00
iChannels	4.00	8.00	8.00
iComp	-	-	40.00
pipe	142.00	266.00	266.00
proc	22.00	23.00	27.00
control	0.00	0.00	0.00
extra kernel	204.50	178.00	289.00
manager	500.00	482.00	575.00
rest	88.00	88.00	88.00

shows an inaccuracy of the predictions, however, the differences between those segments are relatively small. On the other hand, the difference between the implementations for the pipe segment can be as high as 87.3 %, which is significantly larger. This segment is expected to use significantly more hardware resources in case of the *HH+custom* kernel, because the equations of the gate-activation variables when custom gates are used are more complex and therefore, require more resources. It must be noted that the pipe scales with the unroll factor, which was equal to 2. Consequently, an increase of 43.7 % in BRAMs per uf unit for the kernel segment pipe is expected.

To measure the influence of using multiple cell compartments, the predicted resource usage of a *HH+custom* kernel instance and *HH+custom+multi* kernel instance are compared. Those results can be seen in Table 5.24. The first interesting observation is that when multiple compartments are supported, the resource usage of `vMem` is increased while $N_{comps,max}$ is equal between the implementations. This can be explained by the increment in number of reads of the `vMem` as besides the voltage of the cell being processed, also the voltages of the 2 neighbouring compartments are read. This explanation is reinforced by the fact that the increment in usage of BRAMs is 3.06, with the number of reads being 3 times as high. The second interesting thing is that the BRAMs required to calculate *iComp* are relatively small in comparison with the total amount of BRAMs needed. On the other hand, the increase in BRAM usage in both the extra kernel and manager is relatively large. A possible explanation for this increment is the addition of two extra streams (`vCompIn` and N_{comps}), however, we offer this hypothesis cautiously, as it is unknown how those two segments are implemented.

To measure the influence of enabling gap junctions, two comparisons are done. The first comparison is done between a *HH* and a *HH+gap* kernel instance. The second comparison is between a *HH+custom+multi* and a *HH+custom+multi+gap* kernel instance. Table 5.25 shows the predicted resource usages for the *HH* and *HH+gap* kernel instances

Table 5.25: BRAMs used for the kernel segments of the *HH* and *HH+custom* kernel instances with the hardware parameters, $uf = 1$, $N_{comps,max} = 30720$, $N_{gates,max} = 8$, and $N_{ODE} = 1$.

Segment	*HH*	*HH+gap*
total	1302.00	1493.00
vMem	52.00	65.00
yMem	424.00	420.00
iMem	-	52.00
calcIChannels	84.00	105.00
iChannels	0.00	-
pipe	71.00	-
proc	22.00	85.00
control	0.00	0.00
extra kernel	108.00	106.25
iGapMem	-	52.00
gapFunct	-	23.00
gapControl	-	0.00
manager	454.00	496.75
rest	88.00	88.00

with the hardware parameters, $uf = 1$, $N_{comps,max} = 30720$, $N_{gates,max} = 8$, and $N_{ODE} = 1$. The unroll factor is set to 1 because in this case the only difference in hardware between the two kernels is the hardware used for the gap junctions. The results show a significant increase in the amount of resources used for `calcIChannels`. This is surprising as, in both implementations, the same code is used for this segment. Secondly, the *HH+gap* kernel instance contains more segments and therefore, more resources. Another interesting observation o is the fact that the BRAMs used for the `pipe` are larger than for the `gapFunct`. Therefore, it is expected that the unroll factor could be higher when unrolling gap junctions instead of pipes when the rest of the hardware parameters are equal.
When comparing the results of the *HH+custom+multi* and *HH+custom+multi+gap* kernels, BRAM predictions are shown in Table 5.26; the same conclusions can be drawn as when the *HH* and *HH+gap* implementations were compared. Additionally, there is a interesting observation which are the difference between resource usage of the `calcIChannels` and `yMem` segments as they are expected to be roughly the same. This, because `calcIChannels` uses the same equations between both kernels and `yMem` has the same size and the same number of reads in both kernels. Due to automatic synthesis, an explanation for this difference could not be found.

5.4 Performance evaluation

In this section, the performance of the implementations on the DFE will be discussed. Before discussing the performance on the DFE, first the influence of numerical meth-

Table 5.26: BRAMs used for the kernel segments of the *HH+custom+multi* and *HH+custom+multi+gap* kernel instances with the hardware parameters, $uf = 1$, $N_{comps,max} = 30720$, $N_{gates,max} = 8$, and $N_{ODE} = 1$.

Segment	*HH+custom+multi*	*HH+custom+multi+gap*
total	1461.00	1842.50
vMem	52.00	118.00
yMem	65.00	420.00
iMem	-	52.00
gapAddressMem	-	45.00
calcIChannels	51.00	124.00
iChannels	8.00	-
calcIComp	40.00	40.00
pipe	266.00	-
processing	27.00	154.00
control	0.00	0.00
extra kernel	289.00	177.00
iGapMem	-	52.00
gapFunct	-	22.00
gapControl	-	0.00
manager	575.00	550.50
rest	88.00	88.00

Table 5.27: Specifications of the hardware used for the performance measurement.

Specification	Maia	Intel Core i7-4870HQ
On-board DRAM (GB)	48	16
RAM bandwidth (GB/s)	76.8	25.6
On-chip memory	6 MB (FPGA BRAMs)	256 KB (L2 Cache)
Chip frequency (GHz)	Implementation specific	2.5
Chip Architecture	Stratix V (5SGSD8)	Crystal Well
Manufacturing Technology (nm)	28	22

ods on the number of ticks is discussed. Secondly, an indication is given of how the performance of the kernel instances is expected to scale and what the expected bottlenecks are. Then, the results of the performance measurements will be shown. The performance measurements are done on a Maia DFE, whose specifications can be seen in Table 5.27. To set a reference point, the performance of the DFE implementations are compared against C implementations run on an Intel Core i7-4870HQ processor. Finally, a comparison against the High Performance Computing (HPC) framework BrainFrame is made.

Table 5.28: Optimized kernel configurations, based on the resource prediction method, which support 10 channels and minimally 16384 compartments.

Kernel	N_{ODE}	uf	$N_{comps,max}$	$N_{gates,max}$
HH	1	8	36864	10
	2	4	39936	10
	3	3	26624	10
HH+gap	1	16	21504	10
	2/3	4	18432	10
HH+custom	1	6	19456	10
	2	3	20480	10
	3	1	39936	10
HH+custom+multi	1	4	33792	10
	2	2	26624	10
	3	1	22528	10
HH+custom+multi+gap	1	8	18432	10
	2/3	1	16384	10

5.4.1 Performance influence of numerical methods

In Section 5.2, the influence of numerical methods on the time-step-size was discussed. Additionally, the numerical methods have an influence on what unroll factor can be used, which directly affects performance through influencing the number of ticks. Therefore, for each kernel the best performing (the one with the highest unroll factor within the bounds of the hardware resources available) configuration of each numerical method is defined, using the prediction method discussed in Section 5.3.1, which supports 10 channels and minimally 16384 compartments. The result can be seen in Table 5.28.

The total number of ticks of a simulation is given either by Equation (5.4) or Equation (5.6) depending whether the simulation uses gap junctions. To simplify Equation (5.4), the number of gates per compartment is assumed to be constant, changing the equation to Equation (5.5), where N_{gates} is the number of gates per compartment. N_{steps} can be calculated by dividing the simulation time ($SimTime$) by the time-step-size (dt), see Equation (5.7). Therefore, to compare the numerical methods for the different kernel instances N_{comps}/N_{cells}, N_{gates}, and $SimTime$ needs to be chosen. The values used for the simulation can be seen in Table 5.29. With the use of those values it is possible to compare the performance between the three different numerical methods.

$$N_{ticks,noGap} = \sum_{k=0}^{N_{comps}-1} \left\lceil \frac{N_{gates}[k]}{uf} \right\rceil \cdot N_{steps} \tag{5.4}$$

Table 5.29: Parameters used to see influence numerical methods on performance.

Variable	Value
N_{comps}, N_{cells}	16384
N_{gates}	10
$SimTime$	1000.0

Table 5.30: Parameters used to the see influence of different numerical methods on performance.

	dt			speedup		
	fwd-Euler	rk2	rk3	fwd-Euler	rk2	rk3
HH	0.06	0.05	0.07	1.00	0.56	0.58
HH+gap	0.01	0.01	0.01	1.00	0.13	0.08
HH+custom	0.05	0.10	0.15	1.00	1.00	0.60
HH+custom+multi	0.03	0.05	0.05	1.00	1.00	0.50
HH+custom+multi+gap	0.01	0.01	0.01	1.00	0.06	0.04

$$N_{ticks,noGap} = N_{comps} \cdot \left\lceil \frac{N_{gates}}{uf} \right\rceil \cdot N_{steps} \tag{5.5}$$

$$N_{ticks,gap} = \frac{N_{cells}^2}{uf} \cdot N_{steps} \cdot N_{ODE} \tag{5.6}$$

$$N_{steps} = \frac{SimTime}{dt} \tag{5.7}$$

The results, from the non-relaxed simulations to test how high the time-step-size could be increased (the dts from the relaxed simulations do not improve the performance of higher-order solvers, as shown in Section 5.2.1), can be seen in Table 5.30. These show that using a higher-order solver for HH models is at best as good as the forward-Euler solver (i.e. a speedup of 1.00 in the table). However, in most cases the performance is significantly worse. Consequently, the use of higher-order solvers seems not beneficial and therefore, will no be taken into account for the evaluation of the actual performance and energy. This observation is done based on 5 different tests (one for each kernel instance) and additionally, the relaxed simulations did not show better performance. Therefore, it is expected that in general higher-order solvers will not be beneficial in terms of performance.

5.4.2 Theoretical performance scaling

The first indication of how the execution time scales is given by the number of ticks required. As discussed in Chapter 4, the number of ticks needed to complete the

simulation depends on whether gap junctions are supported. If gap junctions are not supported, the number of ticks needed to complete the simulation is given by Equation (5.4). To simplify this equation, the compartments are assumed to be homogeneous so that the number of gates is constant per compartment. If this assumption holds, then the number gates per compartment is constant, changing the number of ticks for kernel instances without gap junctions to Equation (5.5). This shows that the number of ticks scales linearly with the N_{comps}, $N_{gates,comp}$, and N_{steps}.
If gap junctions are supported, the number of ticks is given by Equation (5.6), under the assumption that the calculation of the gap junctions requires more ticks than the calculation for the gate-activation variables. See Section 4.4.4 for the reason this assumption should hold.
Apart from the number of ticks, the execution time is expected to scale linearly to the frequency as this is the rate at which the calculations are done. For the performance measurements, the frequency is kept constant at 180 MHz.
Under optimal circumstances, the performance scales with $N_{ticks} \cdot f$. Therefore, the minimum number of ticks will be achieved with the highest uf possible. However, an increase of uf will lead to higher throughput needs between the chip and the Large Memory (LMem). Therefore, before showing the actual performance measurements, how different parameters will influence the throughput is analysed.

5.4.3 LMem bandwidth evaluation

To analyse whether the LMem bandwidth of the DFE is a bottleneck for the performance, the total required throughput is needed. We can derive total throughput using the throughput per stream. The required throughput per stream for the five kernel instances is shown in Tables 5.31 to 5.35, respectively.
The total required throughput is calculated as the sum of the throughput of all streams. This sum includes the stream of $vCompIn$, although it is only needed in the first simulation step. Knowing that the frequency is equal to 180 MHz, the required throughput of each kernel instance can be seen in Table 5.36.
For the *HH* kernel, the total required throughput can be seen in Figure 5.36a. This shows that the unroll factor has the biggest influence on the required throughput. Additionally, it can be seen that the bandwidth is exceeded when the uf is bigger than six. In the case of the *HH+custom* it is interesting that, depending on N_{gates}, the throughput will or will not exceed the bandwidth when the unroll factor is equal to four. However, as it is the theoretical bandwidth, it is expected that increasing uf to higher values than four will not lead to a performance benefit as it will saturate the LMem bandwidth.
For the *HH+custom+multi* kernel instance, for which the results can be seen in Figure 5.37a, the throughput depends on three parameters, N_{comps}, N_{gates}, and uf. However, as can be seen the influence of N_{comps} is small in comparison to either N_{gates} or uf. Therefore, to have a more clear indication, a 2D plot is made, which kept N_{comps} constant at 960. This plot is shown in Figure 5.37b. For the *HH+custom+multi* kernel instance it, just as the *HH+custom* kernel, depends on the number of gates if the throughput exceeds the bandwidth when uf is 4.
For the *HH+gap* kernel instance, it can be seen that next to uf, when N_{comps} is small the

Table 5.31: I/O communication with LMem of the *HH* kernel instance. *vCompIn is only needed in the first simulation step.

variable	size (B)	throughput (B/s)
gateConstants	$12 \cdot 4$	$size \cdot f \cdot uf$
compConstants	$6 \cdot 4$	$size \cdot f \cdot \frac{1}{\left\lceil \frac{N_{gates}}{uf} \right\rceil}$
vCompIn	4	$size \cdot f \cdot \frac{1}{\left\lceil \frac{N_{gates}}{uf} \right\rceil}$*
N_{gates}	4	$size \cdot f \cdot \frac{1}{\left\lceil \frac{N_{gates}}{uf} \right\rceil}$
yOut	4	$size \cdot f \cdot uf$
vOut	4	$size \cdot f \cdot \frac{1}{\left\lceil \frac{N_{gates}}{uf} \right\rceil}$

Table 5.32: I/O communication with LMem of the *HH+custom* kernel instance.

variable	size (B)	throughput (B/s)
gateConstants	$24 \cdot 4$	$size \cdot f \cdot uf$
compConstants	$8 \cdot 4$	$size \cdot f \cdot \frac{1}{\left\lceil \frac{N_{gates}}{uf} \right\rceil}$
N_{gates}	4	$size \cdot f \cdot \frac{1}{\left\lceil \frac{N_{gates}}{uf} \right\rceil}$
yOut	4	$size \cdot f \cdot uf$
vOut	4	$size \cdot f \cdot \frac{1}{\left\lceil \frac{N_{gates}}{uf} \right\rceil}$

required throughput significantly increases. However, for the parameter space shown in Figure 5.38 the bandwidth is not exceeded.

In the case of the *HH+custom+multi+gap* kernel instance, the throughput depends on four parameters namely uf, N_{cells}, $N_{comps,cell}$, and N_{gates}. Therefore, no clear plot of a

Table 5.33: I/O communication with LMem of the *HH+custom+multi* kernel instance. *vCompIn is only needed in the first simulation step

variable	size (B)	throughput (B/s)
$gateConstants$	$24 \cdot 4$	$size \cdot f \cdot uf$
$compConstants$	$8 \cdot 4$	$size \cdot f \cdot \frac{1}{\left\lceil \frac{N_{gates}}{uf} \right\rceil}$
$vCompIn$	4	$size \cdot f \cdot \frac{1}{\left\lceil \frac{N_{gates}}{uf} \right\rceil}$*
N_{comps}	4	$size \cdot f \cdot \frac{1}{\left\lceil \frac{N_{gates}}{uf} \right\rceil} \cdot \frac{1}{N_{comps}}$
N_{gates}	4	$size \cdot f \cdot \frac{1}{\left\lceil \frac{N_{gates}}{uf} \right\rceil}$
$yOut$	4	$size \cdot f \cdot uf$
$vOut$	4	$size \cdot f \cdot \frac{1}{\left\lceil \frac{N_{gates}}{uf} \right\rceil}$

Table 5.34: I/O communication with LMem of the *HH+gap* kernel instance. *vCompIn is only needed in the first simulation step.

variable	size (B)	throughput (B/s)
$gateConstants$	$12 \cdot 4$	$size \cdot f \cdot \frac{uf \cdot N_{gates}}{N_{comps}}$
$compConstants$	$6 \cdot 4$	$size \cdot f \cdot \frac{uf}{N_{comps}}$
$vCompIn$	4	$size \cdot f \cdot \frac{uf}{N_{comps}}$*
N_{gates}	4	$size \cdot f \cdot \frac{uf}{N_{comps}}$
w	4	$size \cdot f \cdot uf$
$yOut$	4	$size \cdot f \cdot \frac{uf \cdot N_{gates}}{N_{comps}}$
$vOut$	4	$size \cdot f \cdot \frac{uf}{N_{comps}}$

Table 5.35: I/O communication with LMem of the *HH+custom+multi+gap* kernel instance. *vCompIn is only needed in the first simulation step.

variable	size (B)	throughput (B/s)
$gateConstants$	$24 \cdot 4$	$size \cdot f \cdot \frac{uf \cdot N_{gates,cell}}{N_{cells}}$
$compConstants$	$8 \cdot 4$	$size \cdot f \cdot \frac{uf \cdot N_{comps,cell}}{N_{cells}}$
$vCompIn$	4	$size \cdot f \cdot \frac{uf \cdot N_{comps,cell}}{N_{cells}}$*
N_{gates}	4	$size \cdot f \cdot \frac{uf \cdot N_{comps,cell}}{N_{cells}}$
N_{comps}	4	$size \cdot f \cdot \frac{uf}{N_{cells}}$
w	4	$size \cdot f \cdot uf$
$yOut$	4	$size \cdot f \cdot \frac{uf \cdot N_{gates,cell}}{N_{cells}}$
$vOut$	4	$size \cdot f \cdot \frac{uf \cdot N_{comps,cell}}{N_{cells}}$

Table 5.36: Required throughput for each kernel instance.

Kernel	Throughput
HH	$(52 \cdot uf + 32 \cdot \left\lceil \frac{N_{gates}}{uf} \right\rceil) 180 \cdot 10^6$
HH+custom	$(100 \cdot uf + 40 \cdot \left\lceil \frac{N_{gates}}{uf} \right\rceil) 180 \cdot 10^6$
HH+custom+multi	$(100 \cdot uf + (\frac{4}{N_{comps}} + 44) \cdot \frac{1}{\left\lceil \frac{N_{gates}}{uf} \right\rceil}) 180 \cdot 10^6$
HH+gap	$uf(4 + \frac{1}{N_{comps}}(36 + 52 \cdot N_{gates})) 180 \cdot 10^6$
HH+custom+multi+gap	$uf(4 + \frac{1}{N_{cells}}(44 \cdot N_{comps,cell} + 100 \cdot N_{gates,cell})) 180 \cdot 10^6$

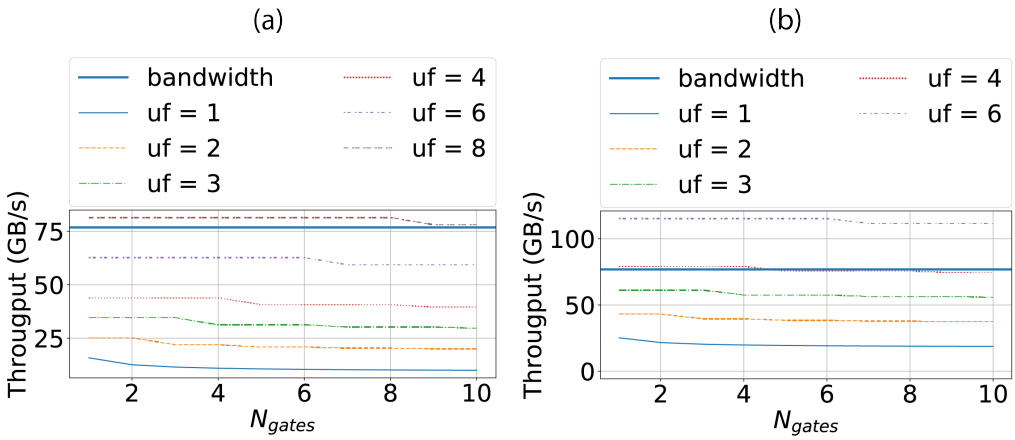


Figure 5.36: LMem bandwidth (thick line) and total required throughput for multiple unroll factors (other lines) $f = 180MHz$. (a) For the *HH* kernel instance. (b) For the *HH+custom* kernel instance.

complete parameters space can be made. To still be able to get an indication of how the throughput will scale, two different plots where $N_{comps,cell}$ is constant are made. The values chosen for $N_{comps,cell}$ are 1 and 3 as this are the number of compartments per cell for a HH or IO cell. Furthermore, the number of gates per compartment is assumed to be identical between compartments. The predictions of the throughput can be seen in Figure 5.39. It is interesting that, for a small number of cells, the throughput exceeds the bandwidth depending on the number of gates and unroll factor. Intuitively, this result can be understood as the number of ticks scales quadratically (see Equation (5.6)) while the amount of data scales linearly (see Table 5.35). Therefore, the ratio of data/ number of ticks will increase if the number of cells decreases. Additionally, it must be noted that besides the extra data sent per cell, if $N_{comps,cell}$ is increased, the $N_{gates,cell}$ increases faster as N_{gates} represents the amount of gates per compartment.
To see if the calculation of the throughput holds in practice, each kernel is simulated. To simplify the plots and thus make the plots less cluttered, $N_{gates,max}$ is set to 10. The used configurations can be seen in Tables 5.37 to 5.41.
The performance measurements of the *HH* kernel are shown in Figure 5.40a. It can be seen that until a uf 4 is reached, the performance increases. For larger uf, no performance gain is observed. This is the consequence of reaching the bandwidth limit between the DFE and the LMem. It must be noted that the bandwidth limit is reached with a lower unroll factor than was calculated. This can be explained by the fact that a throughput equal to the bandwidth can be achieved under optimal circumstances, meaning that the data alignment and clock speed of the memory can cause deviations from the predicted behaviour.
The performance measurements of the *HH+custom* kernel instance can be seen in Figure 5.40b. This shows that the maximum performance is achieved when uf is equal to 2. This is lower than the expected maximum uf of four. This could again be expected

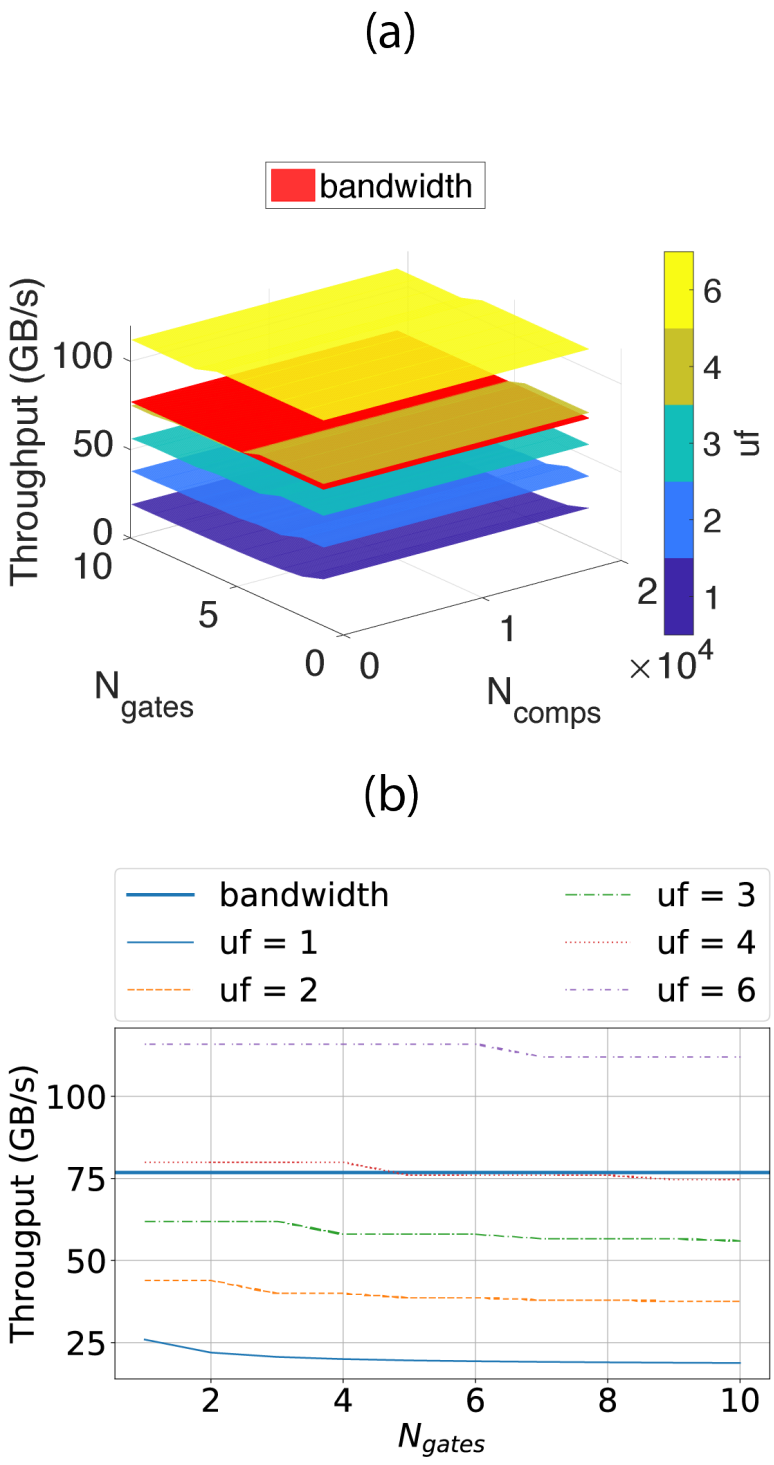


Figure 5.37: LMem bandwidth (red plane or thick line) and total required throughput for multiple unroll factors (other planes or other lines) for *HH+custom+multi* kernel instance, $f = 180MHz$. (a) $960 \leq N_{comps} \leq 19600$. (b) $N_{comps} = 960$.

Table 5.37: Configurations used for performance measurements of the *HH* kernel instance.

uf	$N_{comps,max}$	$N_{gates,max}$
1	61440	10
2	57344	10
3	40960	10
4	53248	10
6	28672	10

Table 5.38: Configurations used for performance measurements of the *HH+gap* kernel instance.

uf	$N_{comps,max}$	$N_{gates,max}$
1	65536	10
2	65536	10
3	61440	10
4	57344	10
6	53248	10
8	49152	10
12	40960	10
16	32768	10
24	24576	10

Table 5.39: Configurations used for performance measurements of the *HH+custom* kernel instance.

uf	$N_{comps,max}$	$N_{gates,max}$
1	57344	10
2	53248	10
3	53248	10

Table 5.40: Configurations used for performance measurements of the *HH+custom+multi* kernel instance.

uf	$N_{comps,max}$	$N_{gates,max}$
1	40960	10
4	28672	10

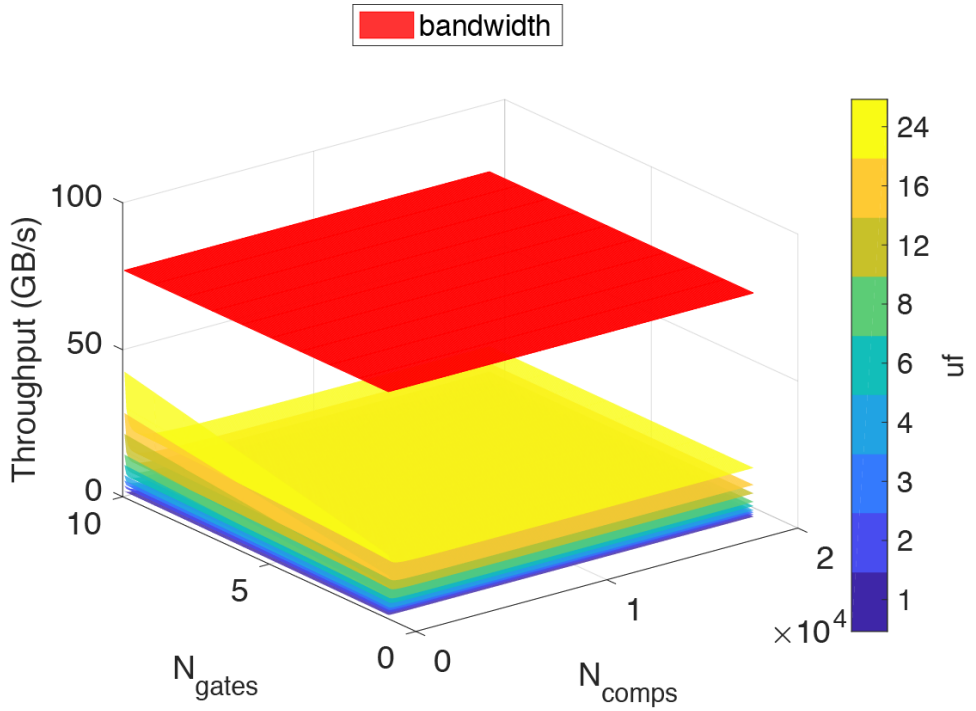


Figure 5.38: LMem bandwidth (red plane) and total required throughput for multiple unroll factors (other planes) for *HH+gap* kernel instance, $f = 180MHz$.

Table 5.41: Configurations used for performance measurements of the *HH+custom+multi+gap* kernel instance.

uf	$N_{comps,max}$	$N_{gates,max}$
8	23552	6
12	23552	6
16	24576	10

as the LMem bandwidth could be achieved only under optimal circumstances.

The performance measurements of the *HH+custom+multi* kernel instance are depicted in Figure 5.40c. Although increasing the unroll factor from 1 to 4 achieves a performance gain, the expected performance ($10/\lceil 10/4 \rceil = 3.33$) gain is not achieved. This is as expected because the maximum achievable throughput is lower than the bandwidth as results from the *HH* and *HH+custom* kernels have shown.

The performance results of the *HH+gap* kernel can be seen in Figure 5.41a. This shows that, as expected (see Figure 5.38) the *HH+gap* kernel instance is not bounded by the LMem bandwidth.

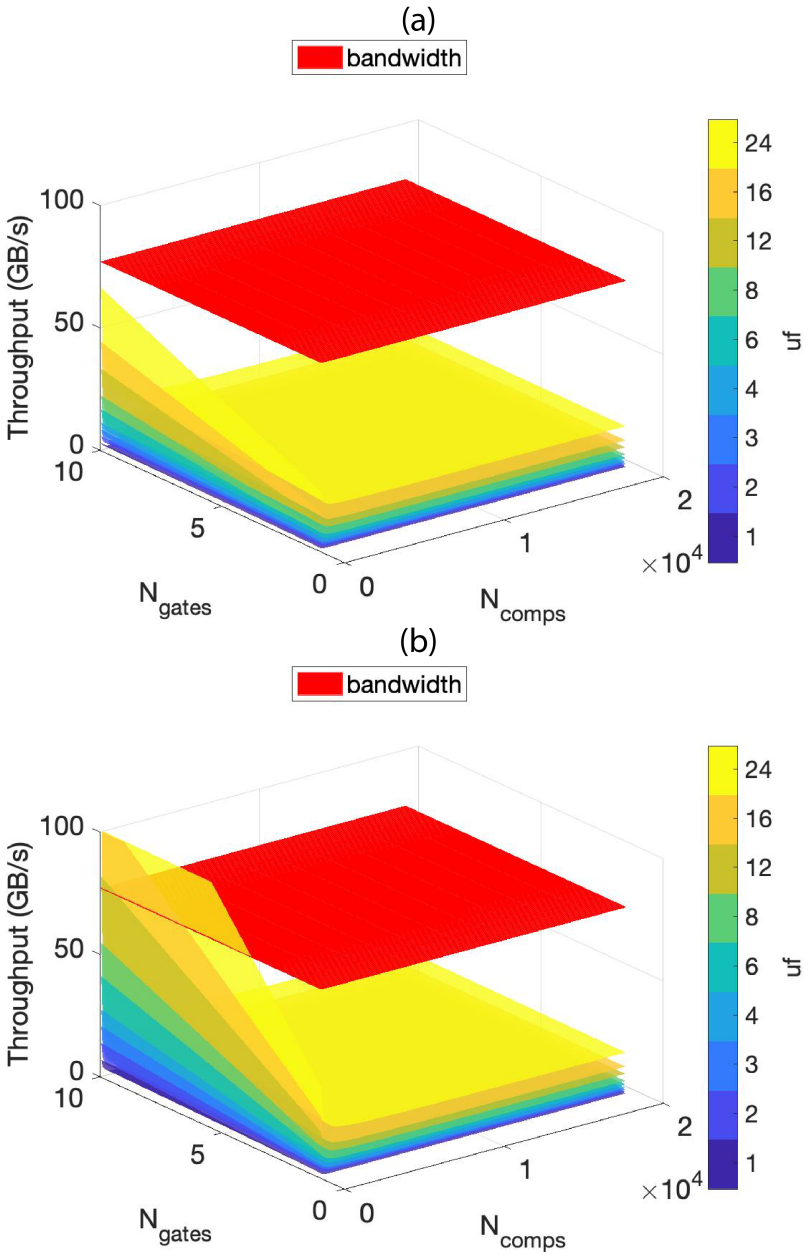


Figure 5.39: LMem bandwidth (red plane) and total required throughput for multiple unroll factors (other planes) *HH+custom+multi+gap* kernel instance, $f = 180MHz$. (a) $N_{comps,cell} = 1$. (b) $N_{comps,cell} = 3$

Finally, the performance measurements of the *HH+custom+multi+gap* kernel instance can be seen in Figure 5.41b. Those show that, as expected (see Figure 5.39), the throughput is not a bottleneck for the performance. Thus, in both kernels featuring gap junctions, the value of uf is only dictated by the available hardware resources, rather than the

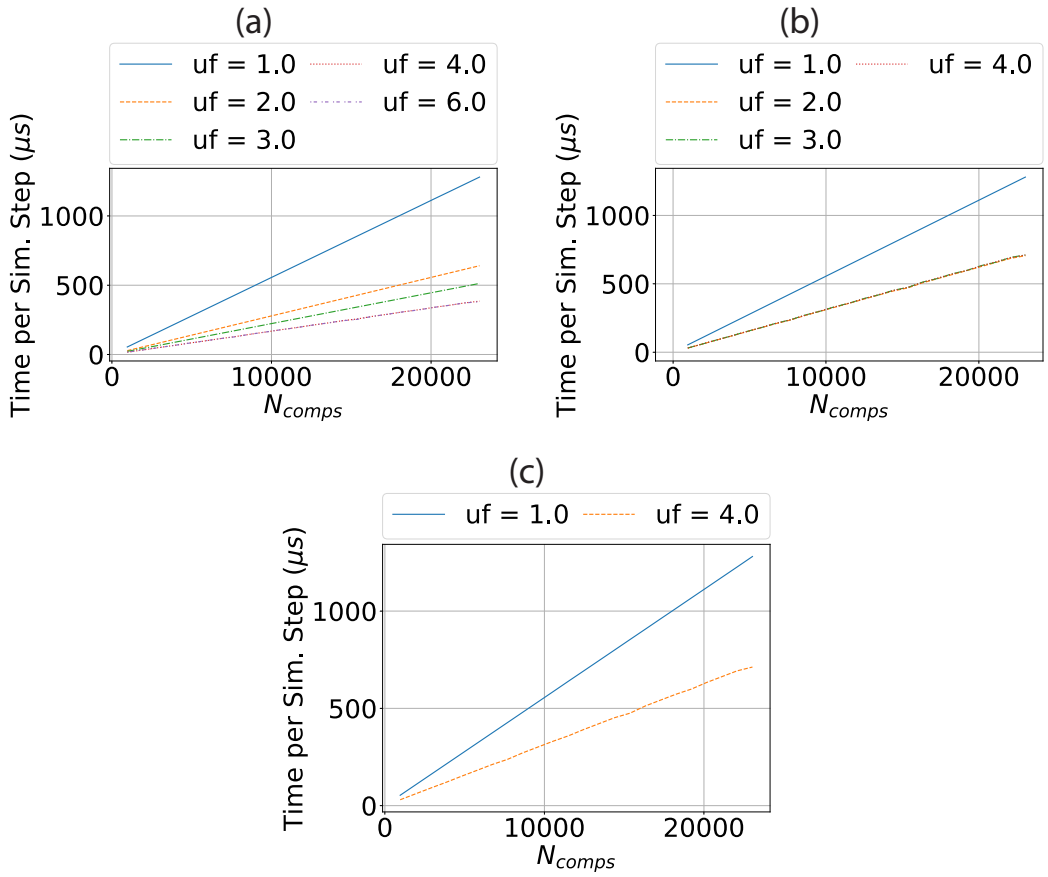


Figure 5.40: Time per simulation step for the kernel instances without gap junctions, $N_{gates} = 10$, $f = 180MHz$. (a) *HH*. (b) *HH+custom*. (c) *HH+custom+multi*.

LMem bandwidth. In other words, this means that the kernels become compute-bound instead of data-bound.

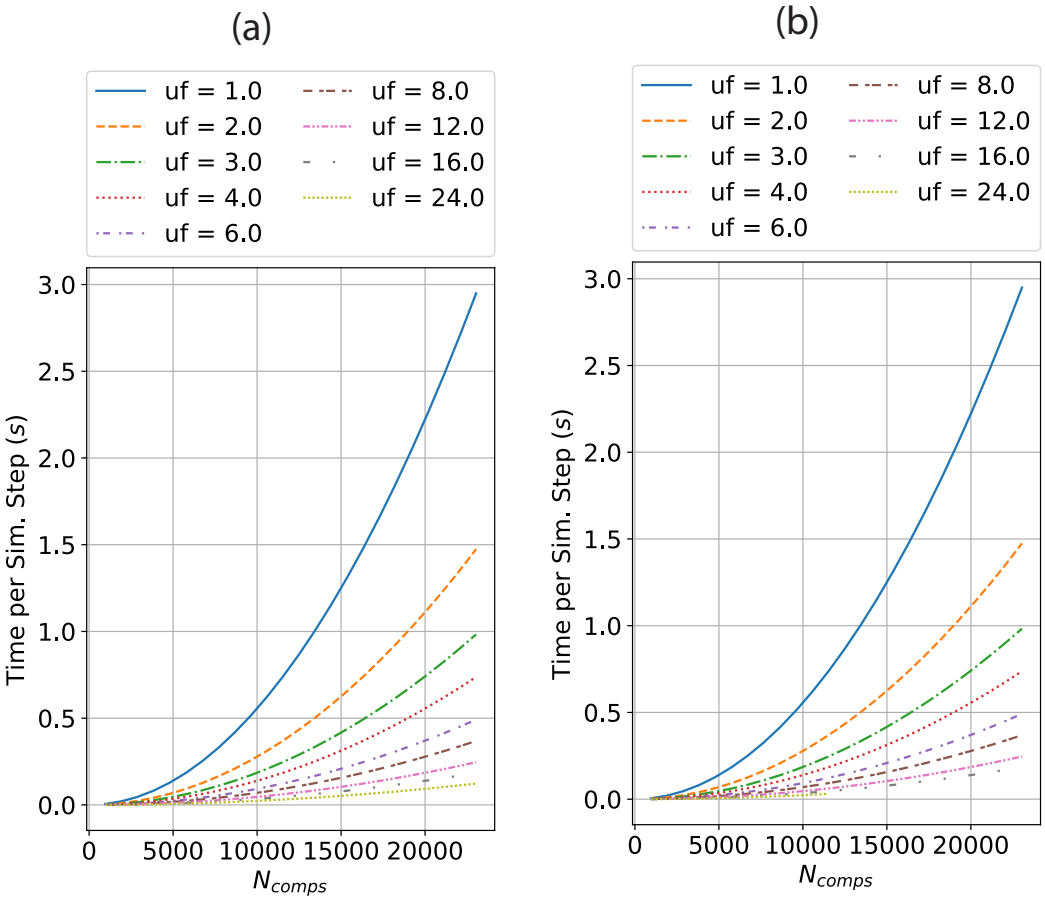


Figure 5.41: Time per simulation step for the kernel instances with gap junctions, $N_{gates} = 10$, $f = 180MHz$. (a) *HH+gap*. (b) *HH+custom+multi+gap*.

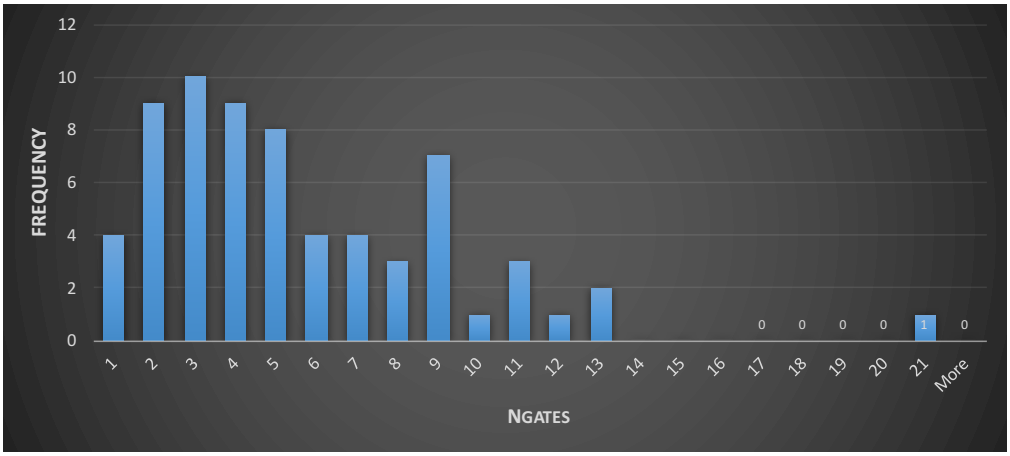


Figure 5.42: Estimation of the frequency of compartments per compartment.

5.4.4 Experimental performance results

Based on the scaling analysis above, it can be seen that the performance of the kernel instances without gap junctions depends on N_{comps}, N_{gates} and uf. On the other hand, the kernel instances with gap junctions scale with N_{cells} and uf (under the assumption that $N_{cells} \geq N_{gates,avg,cell} \cdot uf_{gap}$). As the maximum values $N_{comps,max}$, $N_{gates,max}$, and uf are dependent on each other, the performance depends on which network sizes are supported. To set a starting point, $N_{gates,max}$ is set to 10. The value of 10 is chosen as most of the models are covered. To justify this decision, 10% of the 660 realistic single-neuron models in modelDB[5] were investigated. To get an estimation of the gates per compartment, the number of different channels on the front page of the model in modelDB were counted. This estimation has as a downside that the channels are counted per cell and thus not the gates per compartment. The results can be seen in Figure 5.42 and show 10 gates per compartment covering 89% of the cases.
After this simplification of having a constant value for $N_{gates,max}$, for each uf that is viable the maximum of $N_{comps,max}$ is found with an accuracy of 4096 compartments, while not letting the throughput exceed the theoretical LMem bandwidth. The results can be seen in Tables 5.42 to 5.46.
As the best performance is achieved with an as high as possible unroll factor, smaller networks will be able to be simulated faster if the configuration is not bounded by the bandwidth. For smaller networks, a higher unroll factor can be used than when larger networks are simulated. Therefore, the *HH* kernel models up to and including 53248 compartments can best be simulated with the configuration of $uf = 4$, while if the number of compartments is between 53248-57344 the configuration with $uf = 2$ gives the best performance. The best performance for each range of the number of compartments per kernel instance is shown in Figures 5.43 to 5.44. Interesting is that the performance of the kernel instances without gap junctions show a linear relation

[5] https://senselab.med.yale.edu/ModelDB/ModelList.cshtml?id=3537

Table 5.42: Best uf possible for configurations with maximum $N_{comps,max}$, where $N_{gates,max} = 10$ and $f = 180MHz$ for the *HH* kernel instance.

uf	$N_{comps,max}$
1	61440
2	57344
3	40960
4	53248
6	28672

Table 5.43: Best uf possible for configurations with maximum $N_{comps,max}$, where $N_{gates,max} = 10$ and $f = 180MHz$ for the *HH+gap* kernel instance.

uf	$N_{comps,max}$
1	65536
2	65536
3	61440
4	57344
6	53248
8	49152
12	40960
16	32768
24	24576

Table 5.44: Best uf possible for configurations with maximum $N_{comps,max}$, where $N_{gates,max} = 10$ and $f = 180MHz$ for the *HH+custom* kernel instance.

uf	$N_{comps,max}$
1	57344
2	53248
3	53248
4	45056

Table 5.45: Best uf possible for configurations with maximum $N_{comps,max}$, where $N_{gates,max} = 10$ and $f = 180MHz$ for the *HH+custom+multi* kernel instance.

uf	$N_{comps,max}$
1	40960
4	28672

Table 5.46: Best uf possible for configurations with maximum $N_{comps,max}$, where $N_{gates,max} = 10$ and $f = 180MHz$ for the *HH+custom+multi+gap* kernel instance.

uf	$N_{comps,max}$
1	49152
2	45056
3	45056
4	40960
6	36964
8	32768
12	28672
16	24576
24	12288

to N_{gates}, while uf is not equal to one so by equation Equation (5.8) (the number of ticks required per compartment), the number of ticks does not scale linear. This can be explained by the fact that the kernel instances are bounded by the bandwidth of the memory. Secondly, the number of compartments per cell in the *HH+custom+multi* kernel instance does not influence the performance. This is as expected because the total number of compartments stays the same. Note that N_{comps} represents the total number of compartments.

$$N_{ticks,comp} = \left\lceil \frac{N_{gates,comp}}{uf} \right\rceil \tag{5.8}$$

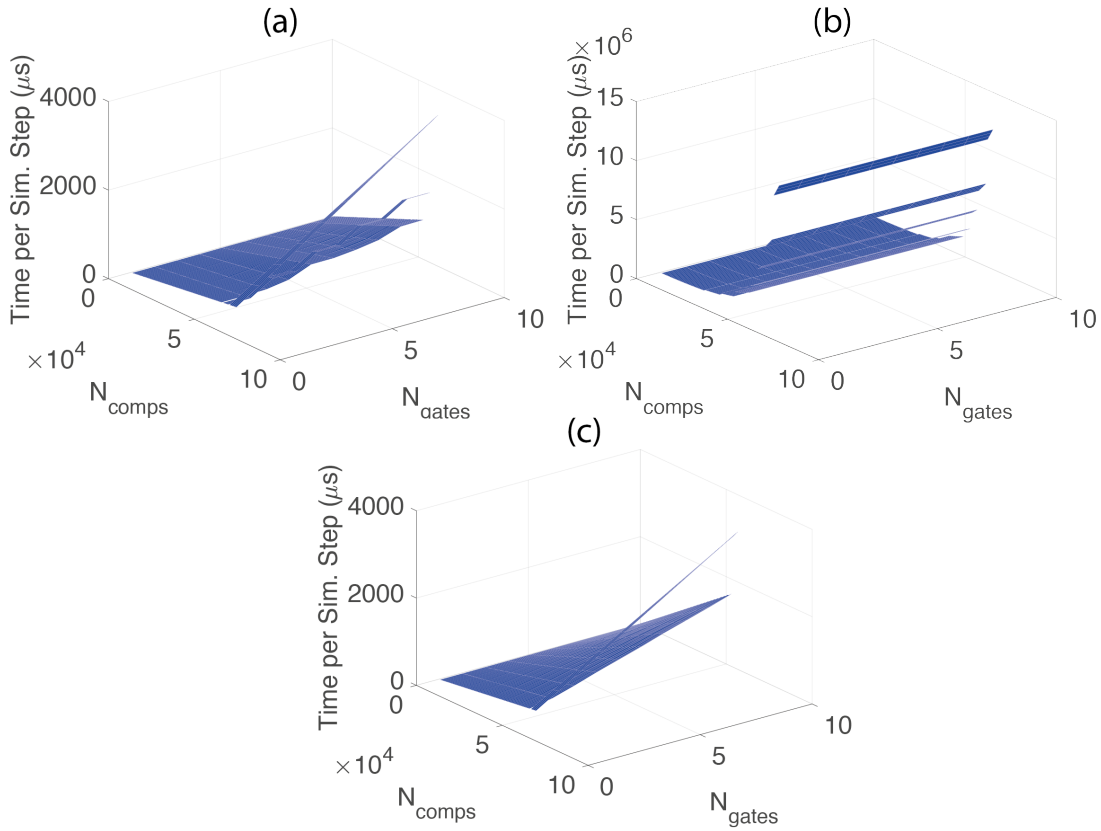


Figure 5.43: Time per simulation step for the kernel instances without multiple cell compartments, $f = 180MHz$. (a) *HH* (b) *HH+gap* (c) *HH+custom*

.

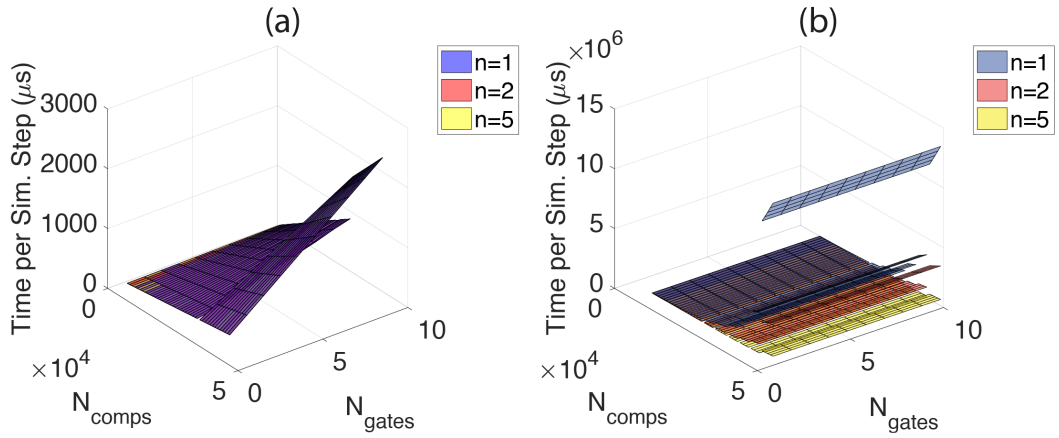


Figure 5.44: Time per simulation step for the kernel instances with multiple cell compartments, $f = 180MHz$. (a) *HH+custom+multi* (b) *HH+custom+multi+gap*

.

Table 5.47: Configurations used to get performance results for the comparison against the CPU and the speedup against the CPU with 23040 compartments.

kernel	uf	$N_{comps,max}$	$N_{gates,max}$	Speedup
HH	4	53248	10	35.49
HH+gap	24	24576	10	36.36
HH+custom	3	53248	10	15.88
HH+custom+multi	4	28672	10	14.33
HH+custom+multi+gap	16	24576	10	23.98

5.4.5 Performance comparison

Previously, the scaling of the execution time has been shown. In the remainder of this section, the performance of each kernel instance on the DFE will be compared against its counterpart on the CPU. Additionally, the *HH+custom+multi+gap* kernel instance is compared against the implementations of BrainFrame.

5.4.5.1 Performance comparison against sequential CPU

For the simulation of the *HH* and *HH+gap* kernels, HH cells are simulated, without gap junctions in case of the *HH* kernel and with gap junctions in case of the *HH+gap* kernel instance. The *HH+custom* simulation simulates soma compartments from the IO-model. Finally, for both the *HH+custom+multi* and *HH+custom+multi+gap* kernel instances IO cells, again with or without gap junctions depending if the kernel instance supports the gap junctions. For each of the simulations 23040 compartments are simulated, which means that in case of the simulations using the *HH+custom+multi* and *HH+custom+multi+gap* kernels 7680 cells are simulated as a single IO cell consists out of 3 compartments.
The optimal configurations of the DFE used for the simulations and the speedups against the CPU can be seen in Table 5.47. The first interesting finding of the speedups is that although the unroll factor of the *HH+gap* junction kernel instance is much higher than the unroll factor of the *HH* kernel instance, the speedup is not. An explanation for this is that the computations which are unrolled in the HH-model are more computationally intensive than the computations which are unrolled in the *HH+gap* kernel. As the part which is more computationally intensive takes more time, execution of the more computationally intensive parts in parallel will achieve a higher speedup for the same parallelization factor in comparison with a less computationally intensive part. The second interesting observation is that the speedup of both the *HH+custom* and *HH+custom+multi* kernel instance is relatively low in comparison with the *HH* kernel. It is expected that this is caused by bottleneck of the LMem bandwidth. Finally, the speedup of the *HH+custom+multi+gap* kernel instance is in line with our expectations if compared against the speedup of the *HH+gap* kernel instance based on the ratio of unroll factors, as follows: $(16/24) \cdot 36.36 = 24,24$. Additionally, simulations of the HH-model are compared to the implementation in NEURON (Python-based) . The C-code was

Table 5.48: Fabric specialisations of the Xeon Phi and NVidia Titan X used in [3].

Specification	Xeon Phi 5110P	NVidia Titan X
On-Board DRAM	8 Gb	12 Gb
RAM bandwidth	320 Gb/s	336.5 Gb/s
Memory streams/channels	16	-
On-chip Memory	30 Mb (L2 cache)	3 Mb (L2 cache)
Number of chip cores	61	3072 CUDA Cores
Chip Frequency	1.053 GHz	1 GHz
Manufacturing Technology (nm)	22	28

30.17x faster than NEURON which accounts for an overall speedup of 1065x of the DFE kernel against NEURON.

5.4.5.2 Performance comparison against BrainFrame

Additionally to the comparision with the CPU, the performance of the flexHH *HH+custom+multi+gap* kernel, with an unroll factor of 16, is compared to three BrainFrame [3] implementations. The three implementations of BrainFrame are on a Maxeler Maia DFE (this is a hard-coded implementation), an Intel Xeon Phi, and a NVidia Titan X GPU. The specifications of the Xeon PHI and the GPU can be seen in Table 5.48.
The performance of the simulation of the IO-model for the implementations of BrainFrame and the flexHH kernel is shown in Figure 5.45. This shows that the flexHH kernel is performing better than all three implementations of BrainFrame. It must be noted that, the performance of the Xeon Phi and Graphics Processing Unit (GPU) are expected to have better perfromance for larger scale networks as a lower unroll factor will be required for the DFE kernels.
The speedup of flexHH against the hard-coded DFE implementation of BrainFrame, which kernels both have the same unroll factor, shows a constant speedup of 1.36 $\times$. The speedup of the flexHH implementation is partly caused by a higher frequency (180 MHz vs 150 MHz). As the increase in frequency accounts only for 20 % of the speedup (assuming linear correlation of frequency to performance) there must be another factor for this gain. It is expected that it is caused by the fact that the flexHH implementation uses a column-wise calculation for the gap junctions (as explained in Section 4.4.4). Because of the column-wise calculations, there is no need to explicitly flush any pipeline during the execution, which is needed in the hard-coded implementation.
A disadvantage of the kernel in flexHH is the data transfer. This is due to the fact that, besides the initial values of the state variables, also the parameters of the equations are sent to the DFE in the flexHH implementation, however, this is the price to pay for deploying general kernel instances in hardware. Additionally, during the execution of the kernel, all those parameters are repeatedly transferred from the LMem in flexHH as opposed to the hard-coded version. Another disadvantage is the case that all the parameters used in the flexHH implementation need to be stored in the LMem of the

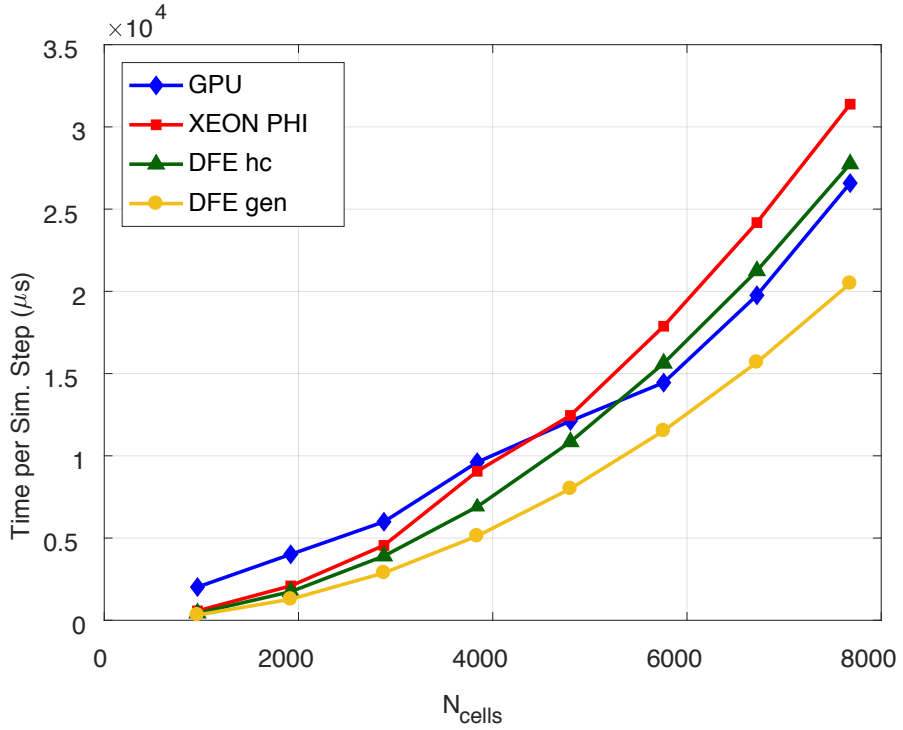


Figure 5.45: Time per simulation step for the implementations of BrainFrame and the flexHH *HH+custom+multi+gap* kernel.(DFE hc is the hard-coded DFE variant from BrainFrame, DFE gen is the flexHH kernel)

DFE as opposed to the BrainFrame implementation where the parameters are hard-coded.
On the other hand, an advantage in favour of the kernel in flexHH, and possibly the reason why a higher frequency is achieved, is that in case of the hard-coded kernel, all the gate equations are calculated simultaneously which requires all the hardware to be available. In contrast, in the flexHH implementation the gate equations are implemented with the use of general equations, which require more hardware per gate equations in comparison to the hard-coded solution. In this case, as those general equations can be used for each of the gate equations, the hardware can be reused and these equations do not have to be unrolled to improve the performance as the calculation of the gap junctions is the critical part of the simulation. This hardware re-usability comes yet, at the cost of higher BRAM use: The hard-coded version is more efficient as the flexHH implementation uses more `YMem` than is needed as the three compartments have a different number of gates and the maximum of the `YMem` is a multiplication of $N_{comps,max}$ and $N_{gates,max}$ (see Chapter 4). However, this overhead is not big enough to undo the more efficient use of hardware needed for the gate equations. It must be noted that the flexHH implementation only functions correctly when the gap junctions require more time to calculate than the gate equations, which is the case if $N_{cells} > N_{gates,avg,cell} \cdot uf$,

as discussed in Section 4.4.4.

5.5 Energy results

In this section, the energy usage of the implementations on the DFE will be discussed. To get the power usage of the implementations, the `maxtop` command was used. The configurations used for the power measurements are the same as the configurations used for the performance measurements. To have a variety of measurement points per configuration, the power is measured with the minimum and maximum number of compartments and gates it supports. Consequently, four points per configuration are measured. All the results of the power measurements can be seen in Appendix D. Those results show that when a higher unroll factor is used, the power usage increases. The reason is, of course, that more computations are done simultaneously. Consequently, the data rate of the LMem is increased. Additionally, the number of compartments seems to have a negligible influence on power usage. This is to be expected as the number of compartments only changes the duration of the simulation and therefore, the power usage while the kernel is active will not change.
On the other hand, the number of gates is of influence on the power usage for the kernel instances without gap junctions. This can be explained, as the unroll factor for those kernel instances creates multiple pipelines which are active or inactive based on the number of gates. Based on this, it is expected that when more pipelines are active, the power usage increases.
Inspecting the power results in Appendix D, this seems to be the case. The only kernel instance where this relation is not clear is the *HH+custom* kernel between $uf = 3$ and $uf = 4$. However, because the unroll factor only differs by one and the results are not significantly different, this is assumed to not contradict the general relation between the power usage and unroll factor. Although a general trend is observed between the uf and the power usage, a precise relation between uf, N_{comps}, N_{gates} and the power usage has not been found.
Consequently, for the calculation of the energy, the maximum value per kernel instance is taken. The maximum values of the power measurements per kernel instance are presented in Table 5.49. Interesting from those results is a difference in power consumption depending on whether the kernel instance supports gap junctions. Possibly this due to the fact that kernel instances without gap junctions do not have to transfer the connectivity matrix. Another possible explanation might be that the power of the kernel instances with gap junctions is measured while only the calculations of the gap junctions were active and not the rest of calculations. The power results of each of the implementations are shown in Figures 5.46 to 5.47. These figures are showing a power consumption between 40.1 and 46.8 Watts. The energy results are calculated by multiplying the power from Table 5.49 with the performance results of Figures 5.43 to 5.44. Being able to show those energy results is an important practical consideration for the using flexHH in HPC-systems like BrainFrame.

Table 5.49: Maximum power consumption per kernel instance.

Kernel	Power (W)
HH	46.8
HH+gap	40.1
HH+custom	45.8
HH+custom+multi	44.3
HH+custom+multi+gap	40.5

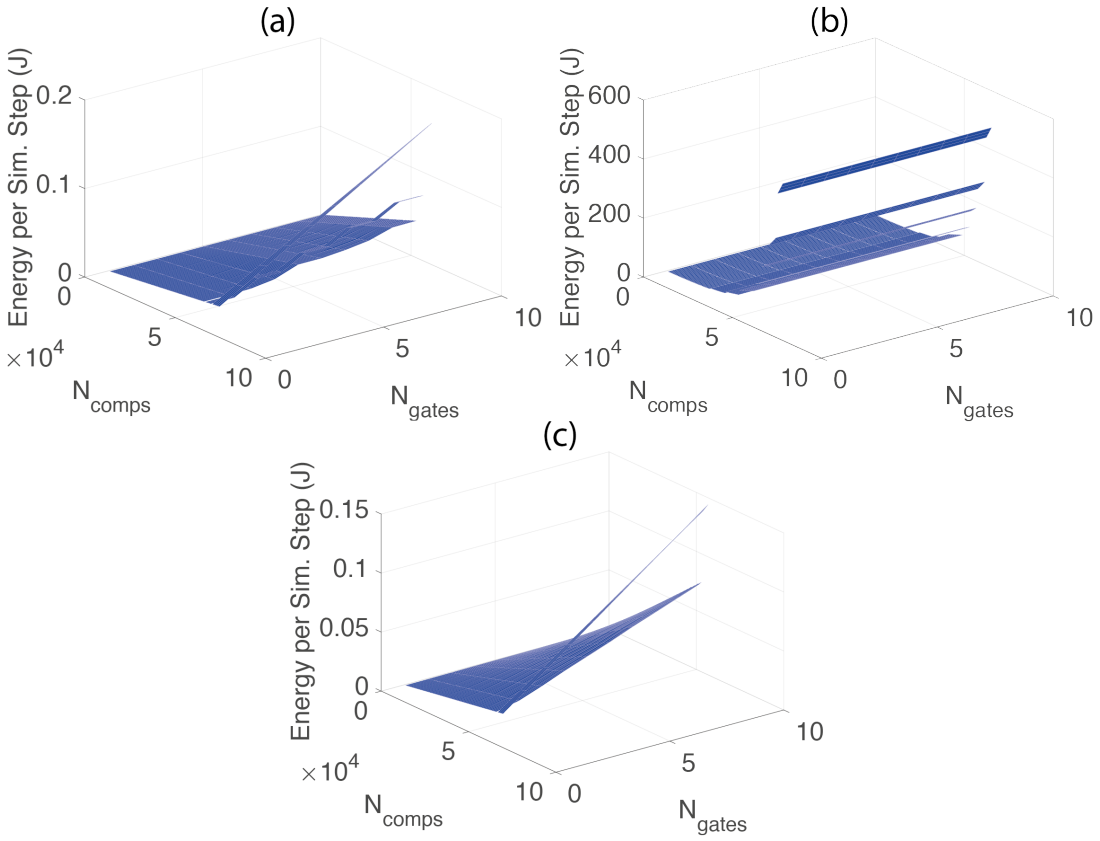


Figure 5.46: Energy usage for the implementations without multiple cell compartments, $f = 180MHz$. (a) *HH*. (b) *HH+gap* (c) *HH+custom*

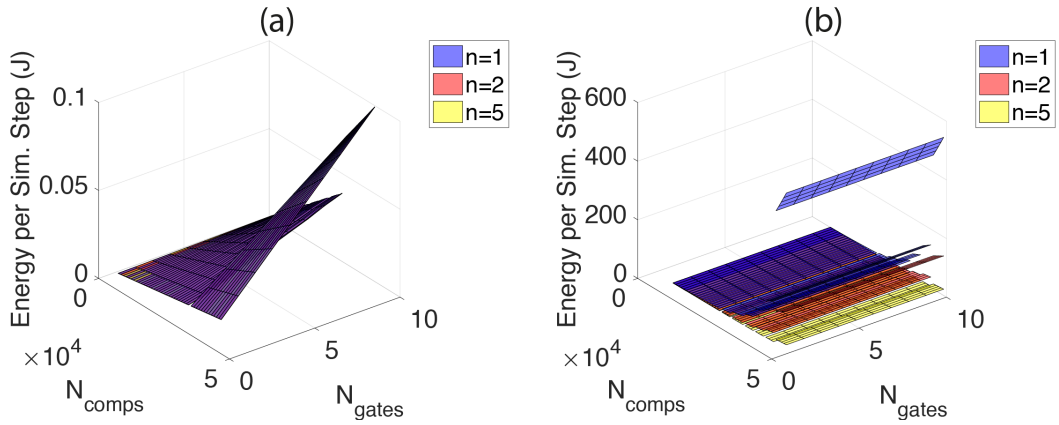


Figure 5.47: Energy usage for the implementations with multiple cell compartments, $f = 180MHz$. (a) *HH+custom+multi*. (b) *HH+custom+multi+gap*.

6 Conclusions

6.1 Discussion

In this thesis, we presented a flexible, scalable and high performing HH-model library called flexHH implemented on a Data-Flow Engine (DFE) (Field Programmable Gate Array (FPGA)-based) platform. flexHH provides clear performance benefits compared to C-based, single-threaded execution and traditional NEURON-based simulation environments. flexHH also performs uniformly better than a prior hard-coded hardware implementation, while not sacrificing simulation flexibility. The flexibility and compliance with NeuroML makes flexHH immediately useful for computational neuroscientists, giving this library a major advantage towards community adoption. Besides the performance benefit that hardware-based libraries like flexHH provide, they also give performance guarantees. Since all design features are defined at design time and resources are allocated statically, both power and execution times can be accurately predicted based on problem size, resulting in accurate energy usage predictions, which is an important practical consideration for the use of High Performance Computing (HPC)-systems on the field. This makes this DFE-based, HH-model library a significant addition for HPC-based acceleration platforms like BrainFrame that wish to provide a heterogeneous-computing system incorporating multiple acceleration platforms, including FPGA-based substrates.
To make flexHH flexible, the equations of the models were parametrized so that they can be stored in the Large Memory (LMem) on board of the DFE. Consequently, for new simulations there is no need for a new time-consuming synthesis cycle. However, all those variables require to be stored in memory, creating a limitation in the network size and simulation time as at some point there won't be any more memory left. Furthermore, the amount of hardware resources used is dependent on the variables $N_{comps,max}$, $N_{gates,max}$, and uf, creating a trade-off between the network size and performance. Other limitations which this version of the library has are:

- Not every gate and gap-junction equation is supported as this would have required an infinite amount of resources. Consequently, the focus was laid to support the equations used in the classical Hodgkin-Huxley (HH)-model and the Inferior-Olive (IO)-model. However, there are other HH-type models which require other equations, than are able to be generated with the predefined set of generalized equations, to describe their gates and/ or gap junctions.
- The multiple cell compartments only support one structure (a sequential structure).
- The gap-junction calculations always take $\frac{N_{cells}^2}{uf}$ ticks even if the network is not all

to all connected.

- It must hold that $N_{cells} > N_{gates,avg,cell} \cdot uf$ as this is how the control signals are implemented. Otherwise, an incorrect simulation will be run.

6.2 Contributions

The contributions of this thesis are as follows:

- A scalable, hardware library of accelerated, parameterizable and NeuroML-compliant [10] HH-model implementations which offer high performance gains.
- A set of crucial model extensions: custom ion gates, gap junctions connectivity and multi-compartmental neurons resulting in five different kernels (independent of the numerical solver used).
- A analysis of the difference between using three different orders of solvers for HH-type models. This showed only in some cases a linear (instead of exponential) benefit in increasing the time-step-size. As the benefits were not uniform and the extra hardware-usage required for the higher-order solvers, the higher-order solvers were not beneficial performance-wise. Consequently, the decision was made to only further evaluate the forward-Euler solver.
- A simple prediction method for the resource usage of the kernels on the DFE. During the analysis of this method, it became clear that the Block Random-Access Memories (BRAMs) of the underlying FPGA were the limiting resource factor. This method could predict the BRAM usage with a mean error between 3.28 and 7.30 % of the Maia DFE resources, where the error increased when more features were supported. Furthermore, it showed that the influence of the frequency on the hardware-usage was minimal.
- A comprehensive performance analysis showing that the bandwidth of the LMem was a bottleneck for the kernel instances without gap junctions. The kernel instances with gap junctions are compute bound. However, speedups of 1065x against (sequential Python-based) NEURON, between 14-36x against sequential C implementations. Furthermore, the flexHH library performed uniformly better than all BrainFrame implementations, with a speedup of 1.36x against a hard-coded DFE implementation were achieved.
- A power analysis showing a power consumption between 40.1 and 46.8 Watts, which is only a fraction of the power consumed on the other acceleration platforms of BrainFrame (Intel Xeon-Phi Central Processing Unit (CPU) and NVidia Graphics Processing Unit (GPU)).

6.3 Future work

In this section recommendations are given for further improvements of the flexHH-library.

- *Develop a parser to translate NeuroML to C-code.*
  The flexHH-library is NeuroML-compliant, however, it is not possible to use NeuroML-code with this version of flexHH. as there is no automatic method which translates the variables of the NeuroML-code to the variables in the C-code of the host CPU. Therefore, the first recommendation is to develop a parser which can translate the NeuroML-code to C-code which the CPU host can use. Developing this parser will increase the usability of the library for neuroscientists.

- *Add extra functionality to the kernel instances.*
  Adding more features (for example other equations for the gates and gap junctions) will increase the range of neural models which can be simulated.

- *Remove limitations of current version.*
  Per limitation (see Section 6.1) a possible migration strategy is given:

  - *Not every gate and gap-junction equation is supported as this would have required an infinite amount of resources.*
    To add extra equations the current functions need to be extended with more function branches.
  - *The multiple cell compartments only support one structure (a line).*
    To add support for a general three structure a matrix multiplication between the compartments in a cell is required. The matrix multiplication is expected to be sparse, as it is not expected that in the most compartments within a single cell are connected. When supporting multiple structures, first challenge is how to map the multiple cell compartments. Specifically challenging is how this mapping will efficiently support a variable number of compartments per cell.
  - *The gap-junction calculations always take* $\frac{N_{cells}^2}{uf}$ *ticks even if the network is not all to all connected.*
    The current gap-junction calculations do a naive matrix multiplication. To reduce the number of ticks when the network is not all to all connected, a sparse matrix multiplication could be implemented.
  - *It must hold that* $N_{cells} > N_{gates,avg,cell} \cdot uf$ *as this is how the control signals are implemented. Otherwise an incorrect simulation will be run.*
    To address this limitation a way needs to be found how the control signals of both the gap-junction and gate equations depend on each other. Otherwise, another kernel instance could be made which functions when $N_{cells} \leq N_{gates,avg,cell} \cdot uf$

- *Efficiently implement instantaneous variables.*
  The instantaneous variables are treated as gate-activation variables in the first version of the flexHH library. Therefore, the instantaneous variables unnecessary use space of the `yMem` (BRAMs) and the LMem. If the instantaneous variables are processed detached from the gate-activation variables, then the space which

is used by the instantaneous variables of both memories will be available. The challenge to efficiently implement this, is how to manage the addresses of the memories, as it is flexible which variable is an instantaneous variable.

- *Use fixed-point instead of floating-point variables.*
  In the current version of the library single-precision-floating-point variables are used. The use of floating-point variables leads to a high resource usage on the DFE in comparison to the use of fixed-point variables. For using fixed-point variables for the simulations of a HH-model, an analysis should be done on the precision of the variables as a small rounding error can lead to faulty simulation. As HH models are biophysically meaningful the state variables are within specific bounds and therefore, such an analysis can be done. Furthermore, the Maxeler tools enable the use of look-up tables with fixed-point computations. Those look-up tables can replace the hardware needed for some complex functions, such as divisions or multiple exponentials. Consequently, the hardware-usage for such functions can be lower and thus those hardware resources can be used for extra features, bigger networks, and/or better performance.

- *Study of ODE solvers to simulate HH-type models.*
  The use of different numerical solvers was compared and showed that there was no exponential increase of the time-step-size when using higher-orders solvers. Moreover, in most cases there was increase in time-step-size when using higher-order solvers. Further investigation of this point can potentially lead to added benefits. Understanding the full details of using different Ordinary Differential Equation (ODE) solvers for HH-type models, can give a explanation why which solver performs best. Consequently, a future version of flexHH that provides automatic selection of the most optimal solver per experiment can be constructed.

- *Supporting variable-time step-size ODE solvers.*
  Additionally to the above study, solvers with a variable time-step-size could be implemented, as a higher time-step-size could be used for non-spiking periods. This will reduce the overall execution time of the simulation, as the simulation can be done in less steps, when there are periods without spiking activity, which can be a significant portion of the execution in simulations without gap junctions. However, it must be noted that for the use of the variable step-size, the accuracy of the simulation needs to be checked at runtime, which will require extra development effort and hardware resources.

Bibliography

[1] E. M. Izhikevich, "Which model to use for cortical spiking neurons?" *IEEE transactions on neural networks*, vol. 15, no. 5, pp. 1063–1070, 2004.

[2] R. A. Tikidji-Hamburyan, V. Narayana, Z. Bozkus, and T. A. El-Ghazawi, "software for brain network simulations: a comparative study," *Frontiers in neuroinformatics*, vol. 11, p. 46, 2017.

[3] G. Smaragdos, G. Chatzikonstantis, R. Kukreja, H. Sidiropoulos, D. Rodopoulos, I. Sourdis, Z. Al-Ars, C. Kachris, D. Soudris, C. I. De Zeeuw *et al.*, "Brainframe: a node-level heterogeneous accelerator platform for neuron simulations," *Journal of neural engineering*, vol. 14, no. 6, p. 066008, 2017.

[4] "Reverse-engineer the brain," http://www.engineeringchallenges.org/challenges/9109.aspx, accessed: 2018-29-01.

[5] W. Maass, "Noisy Spiking Neurons with Temporal Coding have more Computational Power than Sigmoidal Neurons," in *Neural Information Processing Systems*, 1996, pp. 211–217.

[6] ——, "Networks of Spiking Neurons: The Third Generation of Neural Network Models," *Neural Networks*, vol. 10, pp. 1659–1671, 1997.

[7] A. L. Hodgkin and A. F. Huxley, "quantitative description of membrane current and application to conduction and excitation in nerve," *Journal Physiology*, vol. 117, pp. 500–544, 1954.

[8] N. T. Carnevale and M. L. Hines, *The NEURON book*. Cambridge University Press, 2006.

[9] J. M. Bower, "Constructing new models," in *The Book of GENESIS*. Springer, 1998, pp. 195–201.

[10] 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 in Neuroinformatics*, vol. 8, p. 79, 2014. [Online]. Available: http://journal.frontiersin.org/article/10.3389/fninf.2014.00079

[11] M. A. Bhuiyan, A. Nallamuthu, M. C. Smith, and V. K. Pallipuram, "Optimization and performance study of large-scale biological networks for reconfigurable computing," in *High-Performance Reconfigurable Computing Technology and Applications (HPRCTA), 2010 Fourth International Workshop on*. IEEE, 2010, pp. 1–9.

[12] G. Smaragdos, C. Davies, C. Strydis, I. Sourdis, C. Ciobanu, O. Mencer, and C. De Zeeuw, "Real-Time Olivary Neuron Simulations on Dataflow Computing Machines," in *Supercomputing*, ser. Lecture Notes in Computer Science, J. Kunkel,

T. Ludwig, and H. Meuer, Eds. Springer International Publishing, vol. 8488, pp. 487–497.

[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 computational biology*, vol. 8, no. 12, p. e1002814, 2012.

[14] W. Gerstner and W. M. Kistler, *Spiking neuron models: Single neurons, populations, plasticity*. Cambridge university press, 2002.

[15] G. Wulfram and W. Werner, *Spiking Neuron Models*. Cambridge University Press, 2002.

[16] E. M. Izhikevich, "Simple model of spiking neurons," *IEEE Transactions on neural networks*, vol. 14, no. 6, pp. 1569–1572, 2003.

[17] A. L. Hodgkin and A. F. Huxley, "A quantitative description of membrane current and its application to conduction and excitation in nerve," *The Journal of Physiology*, vol. 117, no. 4, pp. 500–544, 1952. [Online]. Available: http://dx.doi.org/10.1113/jphysiol.1952.sp004764

[18] E. M. Izhikevich, *Dynamical systems in neuroscience*. MIT press, 2007.

[19] N. Schweighofer, E. J. Lang, and M. Kawato, "Role of the olivo-cerebellar complex in motor learning and control," *Frontiers in neural circuits*, vol. 7, p. 94, 2013.

[20] D. Xu, T. Liu, J. Ashe, and K. O. Bushara, "Role of the olivo-cerebellar system in timing," *Journal of Neuroscience*, vol. 26, no. 22, pp. 5990–5995, 2006.

[21] N. Schweighofer, K. Doya, and M. Kawato, "Electrophysiological properties of inferior olive neurons: a compartmental model," *Journal of neurophysiology*, vol. 82, no. 2, pp. 804–817, 1999.

[22] M. V. Mascagni, A. S. Sherman *et al.*, "Numerical methods for neuronal modeling," *Methods in neuronal modeling*, vol. 2, 1989.

[23] C. T. Kelley, *Solving nonlinear equations with Newton's method*. Siam, 2003, vol. 1.

[24] S. Gottlieb, C.-W. Shu, and E. Tadmor, "Strong stability-preserving high-order time discretization methods," *SIAM review*, vol. 43, no. 1, pp. 89–112, 2001.

[25] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, 2008.

[26] A. Davison, D. Brüderle, J. Eppler, J. Kremkow, E. Muller, D. Pecevski, L. Perrinet, and P. Yger, "PyNN: a common interface for neuronal network simulators," *Front. Neuroinform*, vol. 2, no. 11, 2008.

[27] M. L. Hines, T. Morse, M. Migliore, N. T. Carnevale, and G. M. Shepherd, "Modeldb: a database to support computational neuroscience," *Journal of computational neuroscience*, vol. 17, no. 1, pp. 7–11, 2004.

[28] D. F. Goodman and R. Brette, "The brian simulator," *Frontiers in neuroscience*, vol. 3, p. 26, 2009.

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

[30] N. Brunel and X.-J. Wang, "What determines the frequency of fast network oscillations with irregular neural discharges? i. synaptic dynamics and excitation-inhibition balance," *Journal of neurophysiology*, vol. 90, no. 1, pp. 415–430, 2003.

[31] B. V. Atallah and M. Scanziani, "Instantaneous modulation of gamma oscillation frequency by balancing excitation with inhibition," *Neuron*, vol. 62, no. 4, pp. 566–577, 2009.

[32] R. A. Tikidji-Hamburyan, J. J. Martínez, J. A. White, and C. C. Canavier, "Resonant interneurons can increase robustness of gamma oscillations," *Journal of Neuroscience*, vol. 35, no. 47, pp. 15 682–15 695, 2015.

[33] Y. Zhang, J. Nuñez-Yañez, J. McGeehan, E. Regan, and S. Kelly, "A biophysically accurate floating point somatic neuroprocessor," in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*. IEEE, 2009, pp. 26–31.

[34] H. T. Blair, J. Cong, and D. Wu, "Fpga simulation engine for customized construction of neural microcircuits," in *Proceedings of the International Conference on Computer-Aided Design*. IEEE Press, 2013, pp. 607–614.

[35] E. Graas, E. Brown, and R. H. Lee, "An fpga-based approach to high-speed simulation of conductance-based neuron models," *Neuroinformatics*, vol. 2, no. 4, pp. 417–435, 2004.

[36] V. Booth and J. Rinzel, "A minimal, compartmental model for a dendritic origin of bistability of motoneuron firing patterns," *Journal of computational neuroscience*, vol. 2, no. 4, pp. 299–312, 1995.

[37] R. K. Weinstein and R. H. Lee, "Architectures for high-performance fpga implementations of neural models," *Journal of Neural Engineering*, vol. 3, no. 1, p. 21, 2005.

[38] K. Cheung, S. R. Schultz, and W. Luk, "Neuroflow: a general purpose spiking neural network simulation platform using customizable processors," *Frontiers in neuroscience*, vol. 9, p. 516, 2016.

[39] 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 *International Symposium on Applied Reconfigurable Computing*. Springer, 2017, pp. 299–310.

[40] C. Brgers and A. R. Nectow, "Exponential time differencing for hodgkin–huxley-like odes," *SIAM Journal on Scientific Computing*, vol. 35, no. 3, pp. B623–B643, 2013.

Mathematical description Inferior-Olive model

A

Constants

$$
\begin{aligned}
C_{m,d} = C_{m,s} = C_{m,a} &= 1 \\
g_{cah,d} &= 4.5 \\
V_{cah,d} &= 120 \\
g_{kca,d} &= 35 \\
V_{kca,d} &= -75 \\
g_{h,d} &= 0.125 \\
V_{h,d} &= -43 \\
g_{cal,s} &= 0.68 \\
V_{cal,s} &= 120 \\
g_{na,s} &= 150 \\
V_{na,s} &= 55 \\
g_{kdr,s} &= 9 \\
V_{kdr,s} &= -75 \\
g_{k,s} &= 5 \\
V_{k,s} &= -75 \\
g_{na,a} &= 240 \\
V_{na,a} &= 55 \\
g_{k,a} &= 20 \\
V_{k,a} &= -75 \\
g_{leak,d} &= 0.016 \\
V_{leak,d} &= 10 \\
g_{leak,s} &= 0.016 \\
V_{leak,s} &= 10 \\
g_{leak,a} &= 0.016 \\
V_{leak,a} &= 10 \\
p_1 &= 0.25 \\
p_2 &= 0.15 \\
g_{int} &= 0.13 \\
w_{i,j} &= 0.4
\end{aligned}
$$

$$simStep = \text{index of current simulation step, starts at 0}$$
$$iApp = \begin{cases} 6, & \text{if (simStep} \geq \text{20000) \& (simStep<20500-1)} \\ 0 & \text{otherwise} \end{cases}$$

Derivatives

$$\frac{dV_{dend}}{dt} = \frac{-I_{gap} + I_{app} - I_{sd} + I_{cah,d} + I_{kca,d} + I_{h,d} + I_{leak,d}}{C_{m,d}} \tag{A.1}$$

where:

$$\begin{aligned} i &= \text{number current cell} \\ j &= \text{number other cell} \\ V_{i,j} &= V_{dend,i} - V_{dend,j} \\ I_{gap_i} &= \sum_{j=0}^{N-1} (w_{i,j} \cdot (0.8 \cdot e^{-0.01 \cdot V_{i,j}^2} + 0.2) \cdot V_{i,j}) \\ I_{app} &= \text{input current, can vary per simulation step} \\ I_{sd} &= \frac{g_{int}}{1 - p_1} \cdot (V_{dend} - V_{soma}) \\ I_{cah,d} &= g_{cah,d} \cdot r_d^2 \cdot (V_{cah,d} - V_{dend}) \\ I_{kca,d} &= g_{kca,d} \cdot s_d \cdot (V_{kca,d} - V_{dend}) \\ I_{h,d} &= g_{h,d} \cdot q_d \cdot (V_{h,d} - V_{dend}) \\ I_{leak,d} &= g_{leak,d} \cdot (V_{leak,d} - V_{dend}) \end{aligned}$$

$$\frac{dr_d}{dt} = 0.2(\cdot \frac{1.7}{1 + e^{-\frac{V_{dend} - 5}{13.9}}} \cdot (1 - r_d) - \frac{0.1 \cdot \frac{V_{dend} + 8.5}{-5}}{1 - e^{\frac{V_{dend} + 8.5}{5}}} \cdot r_d) \tag{A.2}$$

$$\frac{ds_d}{dt} = min(0.00002 \cdot Ca2Plus_d, 0.01) \cdot (1 - s_d) - 0.015 \cdot s_d \tag{A.3}$$

$$\frac{dq_d}{dt} = \frac{\frac{1}{1 + e^{\frac{V_{dend} + 80}{4}}} - q_d}{\frac{1}{e^{-0.086 \cdot V_{dend} - 14.6}} + e^{0.070 \cdot V_{dend} - 1.87}} \tag{A.4}$$

$$\frac{dCa2Plus_d}{dt} = -3 \cdot I_{cah,d} - 0.075 \cdot Ca2Plus_d \tag{A.5}$$

$$\frac{dV_{soma}}{dt} = -\frac{I_{ds} - I_{as} + I_{cal,s} + I_{na,s} + I_{kdr,s} + I_{k,s} + I_{leak,s}}{C_{m,s}} \tag{A.6}$$

where:

$$
\begin{aligned}
I_{ds} &= \frac{g_{int}}{p_1} \cdot (V_{soma} - V_{dend}) \\
I_{as} &= \frac{g_{int}}{1 - p_2} \cdot (V_{soma} - V_{axon}) \\
I_{cal,s} &= g_{cal,s} \cdot k_s^3 \cdot l_s \cdot (V_{cal,s} - V_{soma}) \\
m_s &= \frac{1}{1 + e^{-\frac{V_{soma} + 30}{5.5}}} \\
I_{na,s} &= g_{na,s} \cdot m_s^3 \cdot h_s \cdot (V_{na,s} - V_{soma}) \\
I_{kdr,s} &= g_{kdr,s} \cdot n_s^4 \cdot (V_{kdr,s} - V_{soma}) \\
I_{k,s} &= g_{k,s} \cdot x_s^4 \cdot (V_{k,s} - V_{soma}) \\
I_{leak,s} &= g_{leak,s} \cdot (V_{leak,s} - V_{soma})
\end{aligned}
$$

$$
\frac{dk_s}{dt} = \frac{1}{1 + e^{-\frac{(V_{soma} + 61)}{4.2}}} - k_s \tag{A.7}
$$

$$
\frac{dl_s}{dt} = \frac{\frac{1}{1 + e^{-\frac{V_{soma} + 85.5}{-8.5}}} - l_s}{\frac{20 \cdot e^{\frac{V_{soma} + 160}{30}}}{1 + e^{\frac{V_{soma} + 84}{7.3}}} + 35} \tag{A.8}
$$

$$
\frac{dh_s}{dt} = \frac{\frac{1}{1 + e^{\frac{V_{soma} + 70}{5.8}}} - h_s}{3 \cdot e^{\frac{V_{soma} + 40}{-33}}} \tag{A.9}
$$

$$
\frac{dn_s}{dt} = \frac{\frac{1}{1 + e^{-\frac{V_{soma} + 3}{10}}} - n_s}{5 + 47 \cdot e^{\frac{-(-50 - V_{soma})}{900}}} \tag{A.10}
$$

$$
\frac{dx_s}{dt} = \frac{1.3 \cdot \frac{V_{soma} + 25}{10}}{1 - e^{-\frac{V_{soma} + 25}{10}}} \cdot (1 - x_s) - 1.69 \cdot e^{\frac{V_{soma} + 35}{-80}} \cdot x_s \tag{A.11}
$$

$$
\frac{dV_{axon}}{dt} = -\frac{I_{sa} + I_{na,a} + I_{k,a} + I_{leak,a}}{C_{m,a}} \tag{A.12}
$$

where:

$$
\begin{aligned}
I_{sa} &= \frac{g_{int}}{p_2} \cdot (V_{axon} - V_{soma}) \\
m_a &= \frac{1}{1 + e^{-\frac{V_{axon}+30}{5.5}}} \\
I_{na,a} &= g_{na,a} \cdot m_a^3 \cdot h_a \cdot (V_{na,a} - V_{axon}) \\
I_{k,a} &= g_{k,a} \cdot x_a^4 \cdot (V_{k,a} - V_{axon}) \\
I_{leak,a} &= g_{leak,a} \cdot (V_{leak,a} - V_{axon})
\end{aligned}
$$

$$
\frac{dh_a}{dt} = \frac{\frac{1}{1 + e^{-\frac{V_{axon}+60}{-5.8}}} - h_a}{1.5 \cdot e^{\frac{V_{axon}+40}{-33}}} \tag{A.13}
$$

$$
\frac{dx_a}{dt} = \frac{1.3 \cdot \frac{V_{axon}+25}{10}}{1 - e^{-\frac{V_{axon}+25}{10}}} \cdot (1 - x_a) - 1.69 \cdot e^{\frac{V_{axon}+35}{-80}} \cdot x_a \tag{A.14}
$$

Initial values

$$
\begin{aligned}
V_{dend}^0 &= -60 \\
r_d^0 &= 0.0112788 \\
s_d^0 &= 0.0049291 \\
q_d^0 &= 0.337836 \\
Ca2Plus_d^0 &= 3.7152 \\
V_{soma}^0 &= -60 \\
k_s^0 &= 0.7423159 \\
l_s^0 &= 0.0321349 \\
h_s^0 &= 0.3596066 \\
n_s^0 &= 0.2369847 \\
x_s^0 &= 0.1 \\
V_{axon}^0 &= -60 \\
h_a^0 &= 0.9 \\
x_a^0 &= 0.2369847
\end{aligned}
$$

B Simulation parameters

In this appendix the simulation parameters, consisting of the initial values , the values for the applied current and the weight values for the connectivity matrix, of the simulations used in Section 5.1 are shown.

B.1 Validation

B.1.1 C-code validation

B.1.1.1 HH

$$V = 0 \tag{B.1}$$
$$m = 0.5 \tag{B.2}$$
$$h = 0.5 \tag{B.3}$$
$$n = 0.5 \tag{B.4}$$
$$I_{app} = \begin{cases} 50, & \text{if (t} \geq 100\text{) \& (t < 200)} \\ 0 & \text{otherwise} \end{cases} \tag{B.5}$$

where

t is the time in ms

B.1.1.2 Exact solution HH

$$V = 0 \tag{B.6}$$
$$m = 0.5 \tag{B.7}$$
$$h = 0.5 \tag{B.8}$$
$$n = 0.5 \tag{B.9}$$
$$I_{app} = \begin{cases} 50, & \text{if (t} \geq 0\text{) \& (t < 20)} \\ 0 & \text{otherwise} \end{cases} \tag{B.10}$$

where

t is the time in ms

B.1.1.3 IO

$$V_{dend} = -60 \cdot -5(i\%10) \tag{B.11}$$

$$r_d = 0.0112788 \tag{B.12}$$
$$s_d = 0.0049291 \tag{B.13}$$
$$q_d = 0.0337836 \tag{B.14}$$
$$Ca2Plus = 3.7152 \tag{B.15}$$
$$V_{soma} = -60 \tag{B.16}$$
$$k_s = 0.7423159 \tag{B.17}$$
$$l_s = 0.0321349 \tag{B.18}$$
$$h_s = 0.3596066 \tag{B.19}$$
$$n_s = 0.2369847 \tag{B.20}$$
$$x_s = 0.1 \tag{B.21}$$
$$V_{axon} = -60 \tag{B.22}$$
$$h_a = 0.9 \tag{B.23}$$
$$x_a = 0.2369847 \tag{B.24}$$
$$I_{app} = \begin{cases} (i\%20), & \text{if (t} \geq 200\text{) \& (t < 250)} \\ 0 & \text{otherwise} \end{cases} \tag{B.25}$$
$$w_{i,j} = 0.005 \tag{B.26}$$

where

t is the time in ms

i, j are indexes of the cells

B.1.2 DFE-code validation

B.1.2.1 *HH*

$$V = 0.02567053 \tag{B.27}$$
$$m = 0.05309296 \tag{B.28}$$
$$h = 0.59523809 \tag{B.29}$$
$$n = 0.31807682 \tag{B.30}$$
$$I_{app} = \begin{cases} 50, & \text{if (t} \geq 0\text{) \& (t < 0.01)} \\ 0 & \text{otherwise} \end{cases} \tag{B.31}$$

where

t is the time in ms

B.1.2.2 *HH+gap*

$$V = 0 \tag{B.32}$$
$$m = 0.5 \tag{B.33}$$
$$h = 0.5 \tag{B.34}$$

$$n = 0.5 \tag{B.35}$$

$$I_{app,i} = \begin{cases} 20(i\%10), & \text{if (t} \geq 100\text{) \& (t < 200)} \\ 0 & \text{otherwise} \end{cases} \tag{B.36}$$

$$w_{i,j} = 0.003 \tag{B.37}$$

where

t is the time in ms

i, j are indexes of the compartments

B.1.2.3 *HH+gap* colormap

$$V = -10 + 20 * sin(i) \tag{B.38}$$

$$m = 0.05293139070272 \tag{B.39}$$

$$h = 0.59613484144211 \tag{B.40}$$

$$n = 0.31768223643303 \tag{B.41}$$

$$I_{app,i} = \begin{cases} (i\%10), & \text{if (t} \geq 100\text{) \& (t < 200)} \\ 0 & \text{otherwise} \end{cases} \tag{B.42}$$

$$w_{i,j} = \begin{cases} 0.003, & \text{if (i < 96 \& j < 96)} \\ 0.006, \cdot sin(i), & \text{if (96} \leq \text{i < 192 \& 96} \leq \text{j < 192)} \\ 0.0 & \text{otherwise} \end{cases} \tag{B.43}$$

where

t is the time in ms

i, j are indexes of the compartments

B.1.2.4 *HH+custom*

$$V = -60 \tag{B.44}$$

$$h = 0.90 \tag{B.45}$$

$$x = 0.2369847 \tag{B.46}$$

$$I_{app} = \begin{cases} 50, & \text{if (t} \geq 100\text{) \& (t < 200)} \\ 0 & \text{otherwise} \end{cases} \tag{B.47}$$

where

t is the time in ms

B.1.2.5 *HH+custom+multi*

$$V_{dend} = -60 \tag{B.48}$$

$$r_d = 0.0112788 \tag{B.49}$$

$$s_d = 0.0049291 \tag{B.50}$$
$$q_d = 0.0337836 \tag{B.51}$$
$$Ca2Plus = 3.7152 \tag{B.52}$$
$$V_{soma} = -60 \tag{B.53}$$
$$k_s = 0.7423159 \tag{B.54}$$
$$l_s = 0.0321349 \tag{B.55}$$
$$h_s = 0.3596066 \tag{B.56}$$
$$n_s = 0.2369847 \tag{B.57}$$
$$x_s = 0.1 \tag{B.58}$$
$$V_{axon} = -60 \tag{B.59}$$
$$h_a = 0.9 \tag{B.60}$$
$$x_a = 0.2369847 \tag{B.61}$$
$$I_{app} = \begin{cases} 6, & \text{if (t} \geq 1000\text{) \& (t} < 1050\text{)} \\ 0 & \text{otherwise} \end{cases} \tag{B.62}$$

where

t is the time in ms

i, j are indexes of the cells

B.2 Exploration time step-size

B.2.1 *HH*

$$V = 0 \tag{B.63}$$
$$m = 0.5 \tag{B.64}$$
$$h = 0.5 \tag{B.65}$$
$$n = 0.5 \tag{B.66}$$
$$I_{app} = \begin{cases} 50, & \text{if (t} \geq 9.5\text{) \& (t} < 10.0\text{)} \\ 0 & \text{otherwise} \end{cases} \tag{B.67}$$

where

t is the time in ms

B.2.2 *HH+gap*

$$V = 0 \tag{B.68}$$
$$m = 0.5 \tag{B.69}$$
$$h = 0.5 \tag{B.70}$$
$$n = 0.5 \tag{B.71}$$

$$I_{app} = \begin{cases} 50 \cdot i, & \text{if (t} \geq 5) \text{ \& (t} < 6) \\ 0 & \text{otherwise} \end{cases} \tag{B.72}$$

$$w_{i,j} = 0.01 \tag{B.73}$$

where

t is the time in ms

i, j are indexes of the compartments

B.2.3 *HH+custom*

$$V = -60 \tag{B.74}$$

$$k = 0.7423 \tag{B.75}$$

$$l = 0.0321349 \tag{B.76}$$

$$h = 0.3596066 \tag{B.77}$$

$$n = 0.2369847 \tag{B.78}$$

$$x = 0.1 \tag{B.79}$$

$$I_{app} = \begin{cases} 30, & \text{if (t} \geq 8) \text{ \& (t} < 9) \\ 0 & \text{otherwise} \end{cases} \tag{B.80}$$

where

t is the time in ms

B.2.4 *HH+custom+multi*

$$V_{dend} = -60 \tag{B.81}$$

$$r_d = 0.0112788 \tag{B.82}$$

$$s_d = 0.0049291 \tag{B.83}$$

$$q_d = 0.0337836 \tag{B.84}$$

$$Ca2Plus = 3.7152 \tag{B.85}$$

$$V_{soma} = -60 \tag{B.86}$$

$$k_s = 0.7423159 \tag{B.87}$$

$$l_s = 0.0321349 \tag{B.88}$$

$$h_s = 0.3596066 \tag{B.89}$$

$$n_s = 0.2369847 \tag{B.90}$$

$$x_s = 0.1 \tag{B.91}$$

$$V_{axon} = -60 \tag{B.92}$$

$$h_a = 0.9 \tag{B.93}$$

$$x_a = 0.2369847 \tag{B.94}$$

$$I_{app} = \begin{cases} 6, & \text{if (t} \geq 1000) \text{ \& (t} < 1050) \\ 0 & \text{otherwise} \end{cases} \tag{B.95}$$

where

t is the time in ms

(B.96)

B.2.5 *HH+custom+multi+gap*

$$V_{dend} = -4 \cdot (i\%20) \tag{B.97}$$
$$r_d = 0.0112788 \tag{B.98}$$
$$s_d = 0.0049291 \tag{B.99}$$
$$q_d = 0.0337836 \tag{B.100}$$
$$Ca2Plus = 3.7152 \tag{B.101}$$
$$V_{soma} = -2 \cdot (i\%30) \tag{B.102}$$
$$k_s = 0.7423159 \tag{B.103}$$
$$l_s = 0.0321349 \tag{B.104}$$
$$h_s = 0.3596066 \tag{B.105}$$
$$n_s = 0.2369847 \tag{B.106}$$
$$x_s = 0.1 \tag{B.107}$$
$$V_{axon} = -6 \cdot (i\%10) \tag{B.108}$$
$$h_a = 0.9 \tag{B.109}$$
$$x_a = 0.2369847 \tag{B.110}$$
$$I_{app} = \begin{cases} i\%20, & \text{if (t} \geq 200\text{) \& (t < 250)} \\ 0 & \text{otherwise} \end{cases} \tag{B.111}$$
$$w_{i,j} = 0.01 \tag{B.112}$$

where

t is the time in ms

i, j are indexes of the cells

B.3 Discussion

B.3.1 HH

$$V = 10.50 \tag{B.113}$$
$$m = 0.17 \tag{B.114}$$
$$h = 0.25 \tag{B.115}$$
$$n = 0.48 \tag{B.116}$$
$$I_{app} = 0 \tag{B.117}$$
(B.118)

B.3.2 IO

$$V_{dend} = -60 \tag{B.119}$$
$$r_d = 0.0112788 \tag{B.120}$$
$$s_d = 0.0049291 \tag{B.121}$$
$$q_d = 0.0337836 \tag{B.122}$$
$$Ca2Plus = 3.7152 \tag{B.123}$$
$$V_{soma} = -60 \tag{B.124}$$
$$k_s = 0.7423159 \tag{B.125}$$
$$l_s = 0.0321349 \tag{B.126}$$
$$h_s = 0.3596066 \tag{B.127}$$
$$n_s = 0.2369847 \tag{B.128}$$
$$x_s = 0.1 \tag{B.129}$$
$$V_{axon} = -60 \cdot (i\%10) \tag{B.130}$$
$$h_a = 0.9 \tag{B.131}$$
$$x_a = 0.2369847 \tag{B.132}$$
$$I_{app} = 0 \tag{B.133}$$
$$\tag{B.134}$$

Hardware configurations used for the evaluation of the resource usage

C

Table C.1: Hardware configurations used for prediction of the hardware-usage of the *HH* kernel instances.

uf	$N_{comps,max}$	$N_{gates,max}$	N_{ODE}
2	12	4	1
2	12	10	1
2	12	16	1
2	24	10	1
3	24	10	1
4	4	10	1
4	12	4	1
4	12	6	1
4	12	10	1
4	12	16	1
4	24	10	1
5	24	10	1
6	12	6	1
6	12	10	1
6	12	16	1
6	24	10	1
6	55	10	1
8	24	10	1
2	12	10	2
4	4	10	2
4	12	5	2
4	12	10	2
4	12	14	2
4	16	10	2
2	6	6	3
2	6	10	3
2	6	14	3
2	18	10	3
3	6	6	3
3	6	10	3

Table C.2: Hardware configurations used for prediction of the hardware-usage of the *HH+custom* kernel instances.

index	uf	$N_{comp,max}$	$N_{gates,max}$	N_{ODE}
0	2	12	10	1
1	2	16	6	1
2	2	16	10	1
3	2	16	14	1
4	3	23	10	1
5	3	36	6	1
6	3	48	6	1
7	4	12	10	1
8	4	16	10	1
9	4	16	16	1
10	4	23	6	1
11	4	24	10	1
12	2	12	6	2
13	2	12	10	2
14	2	16	10	2
15	1	4	10	3

Table C.3: Hardware configurations used for prediction of the hardware-usage of the *HH+custom+multi* kernel instances.

index	uf	$N_{comp,max}$	$N_{gates,max}$	N_{ODE}
0	2	12	6	1
1	3	32	9	1
2	4	8	10	1
3	4	12	4	1
4	4	12	10	1
5	4	12	12	1
6	4	23	12	1
7	4	24	10	1
8	2	16	10	2
9	1	4	10	3

Table C.4: Hardware configurations used for prediction of the hardware-usage of the *HH+gap* kernel instances.

index	uf	$N_{comp,max}$	$N_{gates,max}$	N_{ODE}
0	2	12	6	1
1	2	12	10	1
2	2	16	10	1
3	4	12	10	1
4	12	12	6	1
5	12	12	8	1
6	12	12	10	1
7	12	16	10	1
8	12	20	10	1
9	16	23	6	1
10	24	16	10	1
11	24	23	6	1
12	4	12	10	2
13	8	16	6	2
14	8	16	10	2
15	8	16	12	2
16	12	12	6	2
17	12	12	8	2
18	12	12	10	2
19	16	12	6	2
20	16	12	10	2

Table C.5: Hardware configurations used for prediction of the hardware-usage of the *HH+custom+multi+gap* kernel instances.

index	uf	$N_{comp,max}$	$N_{gates,max}$	N_{ODE}
0	1	12	10	1
1	2	12	10	1
2	4	4	10	1
3	4	4	20	1
4	4	12	6	1
5	4	12	10	1
6	4	12	14	1
7	4	16	10	1
8	4	23	6	1
9	4	32	10	1
10	6	12	10	1
11	6	12	12	1
12	6	16	10	1
13	6	23	6	1
14	8	12	10	1
15	8	23	6	1
16	12	12	10	1
17	16	23	6	1
18	24	12	10	1
19	1	4	10	2
20	2	4	10	2
21	4	4	6	2
22	4	4	10	2
23	4	4	14	2
24	4	12	10	2
25	8	4	10	2
26	8	8	10	2
27	12	4	10	2
28	12	8	10	2

Power results

D

HH

Table D.1: Power results for the *HH* kernel instance on a Maia DFE. The parameters uf, $N_{comps,max}$, and $N_{gates,max}$ are configuration parameters while N_{comps} and N_{Gates} are parameters of the simulation itself.

uf	$N_{comps,max}$	$N_{gates,max}$	N_{comps}	N_{gates}	Power (W)
1	61440	10	57600	1	34,7
		10	57600	10	33,7
		10	61440	1	35,2
		10	61440	10	33,6
2	57344	10	53760	1	35,3
		10	53760	10	39,0
		10	56640	1	35,7
		10	56640	10	39,0
4	53248	10	960	1	36,5
		10	960	10	45,8
		10	52800	1	37,0
		10	52800	10	46,8

HH+gap

Table D.2: Power results for the *HH+gap* kernel instance on a Maia DFE. The parameters uf, $N_{comps,max}$, and $N_{gates,max}$ are configuration parameters while N_{comps} and N_{gates} are parameters of the simulation itself.

uf	$N_{comps,max}$	$N_{gates,max}$	N_{comps}	N_{gates}	Power (W)
2	65536	10	62400	1	29,9
		10	62400	10	30,0
		10	65280	1	30,0
		10	65280	10	30,0
3	61440	10	57600	1	30,6
		10	57600	10	30,7
		10	61440	1	30,7
		10	61440	10	30,7
4	57344	10	53760	1	31,0
		10	53760	10	31,0
		10	56640	1	31,0
		10	56640	10	31,0
6	53248	10	49920	1	31,8
		10	49920	10	31,8
		10	52800	1	31,9
		10	52800	10	31,9
8	49152	10	41280	1	32,4
		10	41280	10	32,6
		10	48960	1	32,5
		10	48960	10	32,5
12	40960	10	33600	1	34,3
		10	33600	10	34,1
		10	40320	1	34,3
		10	40320	10	34,1
16	30720	10	24960	1	35,9
		10	24960	10	35,9
		10	32640	1	36,0
		10	32640	10	35,9
24	24576	10	960	1	39,3
		10	960	10	40,1
		10	24000	1	39,1
		10	24000	10	39,2

HH+custom

Table D.3: Power results for the *HH+custom* kernel instance on a Maia DFE. The parameters uf, $N_{comps,max}$, and $N_{gates,max}$ are configuration parameters while N_{comps} and N_{gates} are parameters of the simulation itself.

uf	$N_{comps,max}$	$N_{gates,max}$	N_{comps}	N_{gates}	Power (W)
1	57344	10	53760	1	39,6
		10	53760	10	38,2
		10	56640	1	40,2
		10	56640	10	38,1
3	53248	10	45120	1	41,7
		10	45120	10	45,8
		10	52800	1	42,3
		10	52800	10	45,7
4	45056	10	960	1	43,2
		10	960	10	44,8
		10	44160	1	43,2
		10	44160	10	45,0

HH+custom+multi

Table D.4: Power results for the *HH+custom+multi* kernel instance on a Maia DFE. The parameters uf, $N_{comps,max}$, and $N_{gates,max}$ are configuration parameters while N_{comps} and N_{gates} are parameters of the simulation itself.

uf	$N_{comps,max}$	$N_{gates,max}$	N_{comps}	N_{gates}	Power (W)
1	40960	10	28800	1	40,4
		10	28800	10	37,3
		10	40320	1	40,4
		10	40320	10	37,4
4	28672	10	960	1	44,3
		10	960	10	43,8
		10	27840	1	43,9
		10	27840	10	43,8

HH+custom+multi+gap

Table D.5: Power results for the *HH+custom+multi+gap* kernel instance on a Maia DFE. The parameters uf, $N_{comps,max}$, and $N_{gates,max}$ are configuration parameters while N_{comps} and N_{gates} are parameters of the simulation itself.

uf	$N_{comps,max}$	$N_{gates,max}$	N_{comps}	N_{gates}	Power (W)
1	49152	10	45120	1	30,3
		10	45120	10	30,3
		10	48960	1	30,3
		10	48960	10	30,3
3	45056	10	41280	1	31,4
		10	41280	10	31,5
		10	44160	1	31,5
		10	44160	10	31,5
4	40960	10	37440	1	31,6
		10	37440	10	31,6
		10	40320	1	32,0
		10	40320	10	32,0
6	36864	10	33600	1	32,3
		10	33600	10	32,2
		10	36480	1	32,8
		10	36480	10	32,8
8	32768	10	28800	1	33,1
		10	28800	10	33,1
		10	32640	1	33,2
		10	32640	10	33,2
12	28672	10	24960	1	34,6
		10	24960	10	34,7
		10	27840	1	34,7
		10	27840	10	34,6
16	24576	10	12480	1	36,2
		10	12480	10	36,3
		10	24000	1	36,3
		10	24000	10	36,3
24	12288	10	960	1	39,5
		10	960	10	40,5
		10	11520	1	39,6
		10	11520	10	39,9