BrainFrame:
From FPGA to heterogeneous acceleration of brain simulations

Georgios Smaragdos

Layout: Georgios Smaragdos

Cover: Sofia Nevezhina

Printed by: Optima

© Georgios Smaragdos, 2024. All rights reserved. No part of this publication may be reproduced, stored in retrieval system, or transmitted in any form by any means, electronic, mechanical, photocopying, recording or otherwise without permission of the author or, when appropriate, of the scientific journal in which parts of this dissertation have been published.

ISBN: 978-94-6361-970-7

BrainFrame:
Van FPGA tot heterogene versnelling van hersensimulaties
BrainFrame:
From FPGA to heterogeneous acceleration of brain simulations

Proefschrift

ter verkrijging van de graad van doctor aan
de Erasmus Universiteit Rotterdam
op gezag van de rector magnificus

Prof.dr. A.L. Bredenoord

en volgens besluit van het college voor promoties.

De openbare verdediging zal plaatsvinden op
donderdag 28 maart 2024 om 15.30 uur

door

Georgios Smaragdos

geboren te Thessaloniki, Griekenland.

Erasmus University Rotterdam

Erasmus

Promotiecommissie

Promotor:

Prof.dr. C.I. de Zeeuw

Overige leden:

Dr. J. Pel
Dr. C. Frenkel
Prof.dr.rer.nat. A. Morrison

Copromotoren:

Dr.ir. C. Strydis
Prof.dr. I. Sourdis

Contents

List of Tables V

List of Figures VII

List of Acronyms XIII

1 Introduction 1
- 1.1 Motivation for in-silico brain simulations . . . 3
- 1.2 The era of accelerated brain simulations . . . 5
- 1.3 Thesis scope . . . 6
- 1.4 Thesis contributions . . . 7
- 1.5 Thesis organization . . . 8

2 Background 11
- 2.1 The workloads of computational neuroscience . . . 12
  - 2.1.1 Abstract view of neuronal models . . . 13
  - 2.1.2 Neuron model types . . . 14
  - 2.1.3 Network connectivity modeling . . . 17
- 2.2 The inferior olive . . . 19

3 Accelerated SNNs on Reconfigurable Hardware 21
- 3.1 Explanatory power and computational complexity of neuron models . . . 22
- 3.2 Numerical analysis . . . 23
  - 3.2.1 Numerical methods . . . 24
  - 3.2.2 Behaviour of numerical methods on SNNs . . . 25
  - 3.2.3 Model fitting . . . 27
- 3.3 Complexity analysis . . . 27
- 3.4 Summary of design considerations and trade-offs . . . 33
- 3.5 FPGA spiking neural network implementation categorization 35
  - 3.5.1 SNN applications for brain simulations . . . 35

3.5.2 SNNs for specific AI applications 38
3.5.3 Model choice, network size and real-time performance 40
3.5.4 Architectural choice and their effect on FPGA designs 42
3.6 Tool-flows 44
3.7 Summary 48

4 The Inferior Olive on FPGA-based Hardware 51
4.1 Application description 52
4.1.1 The IO-cell model 52
4.1.2 The IO-network model 54
4.1.3 C-code profiling 54
4.2 HLS FPGA-based inferior olive implementation 56
4.2.1 Overview of the hardware design 56
4.2.2 Time-multiplexing execution 58
4.2.3 HLS C-Code optimizations 59
4.3 Evaluation of the Vivado HLS implementation 61
4.3.1 Experimental methodology 61
4.3.2 Experimental results 62
4.3.3 Error estimation 64
4.3.4 Comparison to related work 66
4.4 DFE-based inferior olive implementation 69
4.4.1 The IO-kernel DFE architecture 70
4.4.2 Additional design optimizations 71
4.5 Evaluation of the DFE implementation 71
4.6 Satellite efforts on FPGA implementations 74
4.6.1 Embedded-HPC inferior olive - ZedBrain 74
4.6.2 Custom (non-HLS-based) acceleration of the inferior olive 76
4.7 The Problem of traditional FPGA workflow 78
4.8 Summary 80

5 Comparison with other HPC solutions 83
5.1 The inferior olive on volunteer computing 84
5.1.1 The mCluster framework and programming model . 86
5.1.2 mCluster implementation 86
5.1.3 Evaluation of the IO on the mCluster 90
5.2 The inferior olive on GPGPUs 92
5.2.1 The GPU implementation 93
5.2.2 GPU evaluation 94
5.3 The inferior olive on a many-core Processor 97
5.3.1 InfOli use cases 97

5.3.2 Quantifying neuron interconnectivity 98
5.3.3 Target platforms 99
5.3.4 Performance evaluation 102
5.4 The problem of complex workload diversity 109
5.5 Summary 111

6 The BrainFrame Platform 113
6.1 Methods 116
6.1.1 Application use cases detailed profiling 116
6.1.2 HPC fabrics and Implementation 117
6.1.3 BrainFrame & the PyNN front-end 123
6.2 Results 125
6.2.1 Performance evaluation 125
6.2.2 Accelerator-selection algorithm 133
6.3 Multi-node potential of BrainFrame back-ends 136
6.3.1 Mutli-DFE implementation and evaluation 137
6.3.2 Mutli-node PHI implementation and evaluation 142
6.3.3 Mutli-node GPU implementation and evaluation . . . 147
6.4 flexHH: BrainFrame library for HH-based neural simulations 151
6.4.1 The DFE implementation 152
6.4.2 Experimental setup and evaluation 155
6.4.3 flexHH on Intel Xeon CPUs (on Amazon Cloud) . . . 158
6.5 Summary 159

7 Conclusions 163
7.1 Scientific contributions 165
7.2 Future work 166

Bibliography 169

Acknowledgments 188

Curriculum Vitae 190

List of Publications 193

Summary 196

Samenvatting 198

List of Tables

3.1 Design factor trade-offs for neuron model workloads with regard to performance and supported network sizes. 34

3.2 A.I. applications of FPGA-based SNNs. 38

3.3 Performance efficiency of works that achieve real-time performance on conductance models. 41

3.4 Average Performance efficiency of FPGA SNN implementations utilizing (only) on-chip and utilizing on-board memory. 43

4.1 Neuron compute requirements per simulation step. 55

4.2 Synthesis Estimation for each optimization case with Vivado HLS 2013.2 for a Virtex 707 evaluation board. Opt1: Gap-junction calculations' optimizations. Opt2: Division-by-constant replacement in dendritic compartment. Opt3: Division-by-constant replacement in all 3 compartments. LUT: look up tables. 61

4.3 Speed-up of C implementations and the FPGA-based accelerator compared to original Matlab code for simulating a real-time network of 96 neurons. 62

4.4 Area utilization for the Virtex 707 evaluation board. 64

4.5 Overview of FPGA SNN Implementations on achievable real-time network sizes. For [1] compared to a Pentium 4 3GHz/3GB RAM and for the IO design to a Xeon 2.66GHz/20GB RAM. 67

4.6 Overview of current and related work SNN Implementations on achievable real-time network sizes. CPU Speed-up for the IO designs is compared to a Xeon E5430 2.66GHz/20GB RAM. 73

4.7 Performance in FLOPS and development time for the various IO FPGA-based works presented in this chapter. Performance is normalized to the design of [2] which was a preliminary exploratory design before the implementation of section 4.2.1 was made. It achieved to simulate a single IO neuron and its immediate neighbouring GJ connections in real-time. 80

5.1 Platform comparison table 93
5.2 Maximum achievable network size (cells) on different platforms of unoptimized and optimized implementation. 97
5.3 Specifications of Evaluation Platforms 100
5.4 Logic utilization for the Virtex 6 FPGA chip on the Vectis. 101
5.5 Real-time achievable network for each use case on each platform. 106
5.6 Comparison between tested platforms using the IO RGJ as benchmark (full GJ with all-to-all communication). Speed-up compared to single thread C implementations used in [3]. 109

6.1 Specifications of the accelerator fabrics used. 117
6.2 RT-achievable network size (#cells) for each use case. . . . 130
6.3 Time savings (in minutes) with BrainFrame for the tested experimental RGJ batch scenario compared to three homogeneous-accelerator systems. The % savings are shown in parenthesis. 135
6.4 Energy savings with BrainFrame for the assumed experimental scenario compared to three homogeneous-accelerator systems. We assume nominal (TDP) power figures (see Table 6.1). 135
6.5 Overview of supported model features per flexHH kernel. . 152
6.6 Specifications of the h/w used for performance measurements 155
6.7 Optimized flexHH-kernel configurations, used for evaluation and the speedup against the CPU. 156

List of Figures

1.1 von Neumann and neural network computing approaches [4]. 3
1.2 Applicability of brain simulation research. 4

2.1 Basic illustration of a biological neuron [5]. 12
2.2 Generalized simulation flowchart of a neuronal model. . . . 13
2.3 Examples of Basic I/O responses of neuronal behavior [6]. . 15
2.4 General STDP modification function [7]. 17
2.5 Depiction of the Olivocerebellar System (left) [8] and abstract depiction of main Olivocerebellar loop (right) 19

3.1 Computational complexity (#State Variables vs model dimentionality and brain size modeled. n-d/mcomp assume an example of a 3 compartmental neuron) - Higher complexity denotes higher explainability. 23
3.2 FLOPS vs Network size per simulation step for the Resonate Integrate and Fire model. 28
3.3 FLOPS vs network size per simulation step for the AdEx model. 29
3.4 FLOPS vs network size per simulation step for the Hodgkin-Huxley model. 29
3.5 Arithmetic Intensity vs Network size for the Resonate Integrate and Fire model. 30
3.6 Arithmetic Intensity vs Network size for the AdEx model. . 31
3.7 Arithmetic Intensity vs Network size for the Hodgkin-Huxley model. 31
3.8 Basic organization of hybrid network adapted from [9]. . . 37
3.9 Neural Network implementing navigation control in the Khepera robot (adapted from [10]). 39
3.10 Whiskerbot block diagram adapted from [11]. 39
3.11 Neuron-model types implemented in each topic. 42

3.12 Percentage of FPGA SNN implementations utilizing (only) on-chip and utilizing on-board memory. 43
3.13 Percetage of FPGA SNN implementations utilizing control-flow and data-flow architecture. 44
3.14 Initial code description environment for HDL generation for SNN FPGA designs. 46

4.1 Graphical representation of the inferior-olivary network model. a) 6-neuron network b) single-neuron model in detail c) sample axon response. 53
4.2 Software profiling of the arithmetic operations in the model for a 96-cell, fully interconnected network. 55
4.3 Block diagram of the Olivocerebellar neuromodeling hardware design. 57
4.4 Time multiplexing of hardware neurons. 59
4.5 Accelerator step execution time for different network sizes. . 63
4.6 Accelerator performance comparison to double-FP C implementation. 63
4.7 Initialization delay for different network sizes. 64
4.8 Graphical comparison of numerical-precision error. Externally evoked input current (Iapp) in green, axonal voltage in blue and error signal in red (Va). (a) Reference trace in double-FP precision. (b) The same trace generated with single-FP precision and all three code optimizations. (c) The error signal (i.e. difference) between the two traces. Observe the amplitude units of the error. 65
4.9 Illustration of (a) A single instance of the FPGA IO Kernel (b) A single instance of the DFE IO Kernel. 69
4.10 Simulation step execution time for the DFE kernels and the FPGA kernel of section 4.2.1. 72
4.11 Speed-up vs network size of best DFE and FPGA implementations compared to single-FP CPU-based execution on an Intel Xeon 2.66GHz with 20GB RAM. 72
4.12 Overall system architecture of Zedbrain. The GJ is managed by one thread of the ARM A9, via AXI4, while the other thread resolves software calculations and I/O outside of the ZedBoard. 76
4.13 Dataflow of a PhC. 77
4.14 A diagram showing how the controllers are housed within the cluster controller 78

4.15 Typical workflow for accelerating neuron applications using HPC devices. 79

5.1 Application-task annotation with the mCluster pragma keywords. Developers need only to define the input/output starting addresses and sizes. 87
5.2 The mCluster BOINC implementation: the original pragma-annotated code is source-to-source translated as described in the previous section and generates the BOINC-oriented back-end files, where each task is associated with a BOINC application and a task graph describes all task dependencies. At runtime, the mCluster starts all applications that have their inputs ready. When an application reports that assimilated results are ready, mCluster traverses deeper in the graph to start new applications that now have their inputs ready. Processing is finished when all applications have reported their assimilated results. 88
5.3 Illustration of the Inferior-Olive application execution procedure; in each simulation time step (simStep) each task calculates a cell's state within the 2-d cell grid with dimensions DIM_X $\times$ DIM_Y. 89
5.4 Our experimental system consisted of four Asus Nexus 7 (2013) tablets and two Samsung Galaxy S4 smartphones. The BOINC client app is installed in each device. 90
5.5 Average simStep execution time of the Inferior-Olive application on all client devices. Overall time is divided into data download/upload, execution and the mCluster scheduling overhead, such as BOINC application creation/deletion and graph traversing. 91
5.6 Projected processing time of a single simStep when simulating different brain types using up to 10,000 IoT devices; execution time is lower bounded by the server work. 92
5.7 a) C implementation; (b) Un-optimized CUDA implementation; (c) Optimized CUDA implementation. 94
5.8 Performance comparison between unoptimized vs optimized implementation (Fermi platforms - Block size 64). 95
5.9 Performance comparison to the CPU execution among all 4 GPU platforms. 96
5.10 Example of connection probability, using the exponential distribution with mean value μ and the distance between neurons (r). Both quantities are measured per unit (p.u.). . . . 99

5.11 Xeon-Phi architecture of the IO kernel. 101
5.12 Execution time per time step for small networks for Vectis DFE. Colored areas correspond to the range of possible execution-time values due to different connectivity densities (0%-100%). (Note: Graph areas are unstacked.) 104
5.13 Execution time per time step for small networks for XEON PHI. Colored areas correspond to the range of possible execution-time values due to different connectivity densities (0%-100%). (Note: Graph areas are unstacked.) 104
5.14 L1 cache hit rates ratio for each use case for 100% connectivity networks of the Xeon Phi implementation. 105
5.15 L1 compute-to-data ratio for each use case for 100% connectivity networks of the Xeon Phi implementation. 105
5.16 DRAM data-transfer overheads for small network on the Vectis. It is clear that the platform has enough provisions and is not DRAM-bound for any of the explored use cases. Colored areas correspond to the range of possible timing-overhead values due to different connectivity densities (0%-100%). (Note: Graph areas are unstacked.) 107
5.17 DRAM data-transfer overheads for small network on the XEON PHI. It is clear that the platform has enough provisions and is not DRAM-bound for any of the explored use cases. Colored areas correspond to the range of possible timing-overhead values due to different connectivity densities (0%-100%). (Note:Graph areas are unstacked.) 108

6.1 Workflow framework proposed to support the scientific process through a heterogeneous HPC platform. 115
6.2 Floating-point operations required per simulation step of the IO model for each use case and for different connectivity density percentages (%). 118
6.3 Compute-to-Memory-Access Ratio per simulation step of the IO model for each use case and for connectivity density percentages (%). 118
6.4 DFE implementation of the IO application. 120
6.5 Xeon-Phi implementation of the IO application. 121
6.6 GPU implementation of the IO application. Pre-compute and compute operations are issued by the host. 122
6.7 PyNN architecture and the proposed BrainFrame framework. 123
6.8 RGJ execution time (TYPE I, 100% connectivity). 127
6.9 SGJ execution time (TYPE I, 100% connectivity). 127

6.10 RGJ execution time (TYPE I, <100% connectivity). 128
6.11 SGJ execution time (TYPE I, <100% connectivity). 128
6.12 NGJ execution time (TYPE I, no connectivity). 129
6.13 RGJ execution time (TYPE II, 100% connectivity). 131
6.14 SGJ execution time (TYPE II, 100% connectivity). 131
6.15 RGJ execution time (TYPE II, <100% connectivity). . . . 132
6.16 SGJ execution time (TYPE II, <100% connectivity). 132
6.17 NGJ execution time (TYPE II, no connectivity). 133
6.18 BrainFrame accelerator-selection map for TYPE-II experiments. Selection is heavily (dependent on the experiment, involving all three accelerator fabrics. For TYPE-I experiments, the DFE is always the optimal choice (not shown). . 134
6.19 Architecture of 8-DFE design. 138
6.20 Sim-step execution time for RGJ case vs network size (single vs. 2-DFE run). 139
6.21 Sim-step execution time for NGJ vs network size (single vs. 2-DFE run). 140
6.22 Sim-step execution time for RGJ case vs network size (2-DFE vs. 8-DFE run). 141
6.23 Sim-step execution time for the NGJ case vs network size (2-DFE vs. 8-DFE run). 141
6.24 Multi-KNL implementation. 143
6.25 Sim-step execution time for RGJ case in PHI KNLs (single vs. 2-KNL run). 144
6.26 Sim-step execution time for NGJ case in KNL system vs network size (single vs. 2-KNL run). 145
6.27 Sim-step execution time for RGJ case vs network size (2-KNL vs. 8-KNL run). 145
6.28 Sim-step execution time for NGJ in KNL multi-node systems (2-KNL vs. 8-KNL run). 146
6.29 Gaussian distribution in formed synapses throughout the neuronal network, for different neuron counts and synapses per neuron. The evaluation shows scalability for 1, 2, 4 and 8 Xeon-Phi KNL configurations. High scalability is shown for the heaviest network workload of 2 million neurons/1,000 synapses per neuron. 148

6.30 Uniform distribution in formed synapses throughout the neuronal network, for different neuron counts and synapses per neuron. The evaluation shows scalability for 1, 2, 4 and 8 Xeon-Phi KNL configurations. Scalability is observably lower, particularly for 8-KNL configurations. 148
6.31 Obtained total execution times when scaling GPU-world size for 1M neurons for densities of 10 and 1k synapses/neuron (s/n). For (a) uniform and (b) Gaussian connectivity networks. (c) Obtained total execution times when scaling neuron network size with a per neuron connectivity of 1k synapses/neuron (s/n) for 1 and 32 GPU(s) for uniform and (d) Gaussian connectivity networks. The numbers in the graphs are the speedups of the results in relation to the single GPU-world execution. * 4M cell measurements are extrapolated. 150
6.32 Schematic overview of implementation on the DFE. 154
6.33 Execution Time per step for the *HH+custom+multi*. N=Number of compartments. 156
6.34 Execution Time per step for the *HH+custom+multi+gap*. N=Number of compartments. 157
6.35 Execution Time per step vs network size for the *HH+custom+multi+gap*.158

List of Acronyms

A.I. Artificial Intelligence
ANN Artificial Neural Networks
ASIC Application Specific Integrated Circuits
BRK Booth-Rinzel-Keihn
DAC Digital-to-Analog Converter
DFE Dataflow Machine
DMA Direct Memory Access
DSP Digital Signal Processing Units
FLOPS Floating-Point Operations per Second
FP Floating Point
FPGA Field-Programmable Gate Array
FSM Finite State Machine
GA Genetic Algorithm
GJs Gap Junctions
GPU Graphics Processing Unit
GUI Graphics User Interface
HH Hodgkin-Huxley
HLS High Level Synthesis
HPC High Performance Computing
HW Hardware
I&F Integrate and Fire
IO Inferior Olive
I/O Input/Output
LUTs Look-up Tables
NN Neural Network
NoC Network-on-Chip
PBC pre-Bötzinger Complex
RDMA Remote direct memory access
SNN Spiking Neural Network
SoC System-on-Chip

STDP Spike-Time Dependant Plasticity
SW Software
VHDL VHSIC Hardware Description Language
VLSI Very Large Scale Integration
XML Extensible Markup Language

CHAPTER 1

Introduction

The first steps in brain-behaviour research took place in the end of the 19th century and, since then, the subject has fascinated scientists and engineers. In the early 1940s, the first mathematical model of a Neural Network (NN) was proposed, inspired by brain behavior [12]. Contrary to the typical computer system based on the Von Neumann model, also introduced around the same time, a neural network does not execute explicit sequential instructions to solve its computational problems [4]. The network is a group of processing elements interconnected to each other. Each function on each node (or neuron) is computed in parallel and the relation between input and output of the NN is determined by the network topology and method of interconnectivity. This topology can also be adaptive, in terms of its computational dynamics, mimicking further the biological behaviour (Figure 1.1). This achievement paved the way to further explore the idea of Artificial Neural Networks (ANNs) for creating more advanced systems that abstractly mimic biological behaviour. Such research as the perceptron [13] and other works on connectionist networks are more suitable for certain computational problems. This made neural networks a significant computational tool over the decades even though far removed from replicating the actual brain behaviour.

Advances in neuroscience and engineering eventually led to the creation of mathematical models of networks that do not simply mimic biological behaviour in an abstract fashion but emulate it in significant detail. Such an example is the ***Spiking Neural Network*** (SNN) [14]. An SNN can model a variety of additional behavioural characteristics, like encoding data and adapting according to a spike train‘s amplitude, frequency and general precise pattern of arrival of spiking events on a neuron. As a result, an SNN can have greater computational potential than typical ANNs [15, 16].

This, alongside with the advances of technology in computer science and engineering, has opened up the possibility of implementing larger-scale NNs that have the ability to more accurately simulate brain behaviour. This made neural networks a viable research tool for neuroscience, in addition to a powerful computational tool [4]. A multitude of possibilities then arise in using in-silico simulations both as an experimentation tool to understand brain function] [17] but also as the basis of novel computing systems beyond the current Von Neumann-based technology [18]. These large scale, in-silico neuron simulations with added features come with high computational cost, often beyond the capabilities of typical computing platforms [19, 20]. The primary focus of this thesis, then, is to utilize advanced technologies specifically designed to address the high computational demands incurred by SNNs.

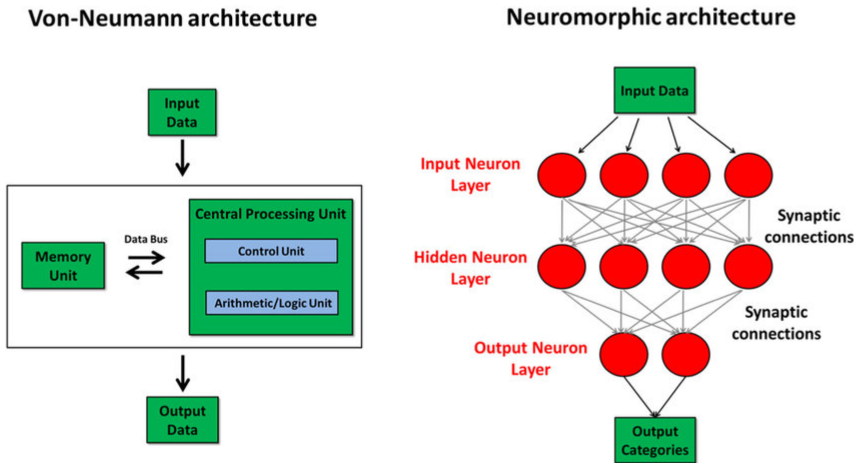


Figure 1.1: *von Neumann and neural network computing approaches [4].*

1.1 Motivation for in-silico brain simulations

The United States National Academy of Engineers has already classified brain simulation as one of the Grand Engineering Challenges [21]. Brain simulation in-silico presents a number of potential benefits (Figure 1.2) for:

- **Neuroscientific Research:** Neuroscientists plan to gain greater understanding of brain behaviour by simulations based on biologically plausible models. As powerful experimentation as tools ***in-vivo*** and ***in-vitro*** experiments are, they have limited scale and are time-consuming to conduct reliably. ***In-silico*** models (based on biological data) can provide insight from single-cell behaviour [22] to network dynamics of whole brain sections [17] in a much more controlled environment. Then findings can again be validated with more targeted biological experiments greatly accelerating brain experimentation.

- **AI research:** The natural potential of biologically inspired neural networks for artificial intelligence can also be an important benefit. ANNs have already been successfully used in this field even though they cannot even begin to reach the computational capacity of biological systems. It is believed that greater understanding of biological systems and their richer computational dynamics can lead to more advanced, bio-inspired artificial intelligence models for autonomous and robotic applications [11].

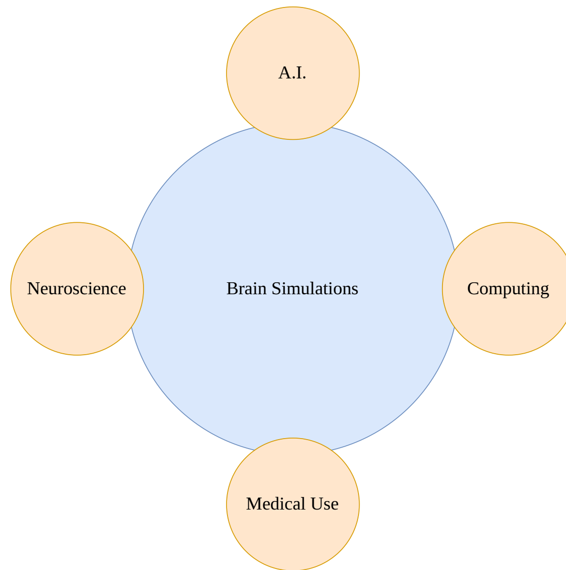


Figure 1.2: *Applicability of brain simulation research.*

- **Computing research:** this type of research could lead to new computer architecture paradigms, alternatives to the typical von Neumann architectures [18]. Such architectures can be very useful for massively parallel applications (such as Digital Signal Processing or inference) and could potentially provide defect-tolerant systems emulating the brain's adaptability.

- **Medical use:** One important eventual goal of the field is brain recovery. If brain function can be emulated in silicon accurately enough and in real time, it can lead to brain prosthetics and implants that can recover lost brain functionality due to health conditions and accidents [23].

The main challenge of complex, biologically accurate SNNs is the computational load they often entail. Furthermore, the biological NNs simulated execute these computations alongside high communication traffic between neurons, something that often conventional CPU execution cannot cope with very well. As a result, the speed of simulations and population of executed models is quite low when running on traditional workstations (usually implemented using MATLAB or standard neuron modeling languages such as NEURON [24] and GENESIS [25]). For example, a 96-

neuron network of the inferior olive model in [22], simulating just 6 seconds of brain time takes close to 1.2 hours to complete in MATLAB in a typical workstation [3]. This greatly impedes the efficiency of brain research. ***High Performance Computing (HPC)*** solutions need to be employed, if such simulations are to be accelerated to speed-up the research process.

1.2 The era of accelerated brain simulations

The process of accelerating a neuroscientific application intuitively follows the same process of any typical acceleration efforts in other fields. Initially the neuroscientific question is formulated. Based on that, a relevant neuroscientific computational model is defined. Then, the computational neuroscientist develops the model using a simulation language. Typical examples include MATLAB, NEURON [26], PyNN [27] etc. To achieve optimized acceleration of simulation, an acceleration engineer needs to subsequently take over, in a sense to mediate between the scientist and the HPC hardware (as the hardware is often too specialized), porting the initial description to the acceleration platform.

An increasingly popular HPC platform for brain simulations is the Graphics Processing Unit (GPU). Since GPUs are ideal for repetitive and highly parallel operations they are an good fit for executing neuron network models and provide good performance and scalability([28, 29]). Yet, in the cases of complex models or very large-scale networks, they may not be able to provide real-time performance, due to the high rates of data exchange between the computational elements of models. Additionally, applications are required to fully utilize the parallel elements of the GPU before the device can show its full efficiency, but the network sizes in which the technology can realistically provide real-time performance for complex models often cannot do that due to the computational intensity of models.

An alternative would be the use of multithreading using supercomputer setups. These systems can emulate the behaviour and parallelism of biological networks with sufficient speed [30]. Supercomputer systems require immense physical space, have high implementation overheads, high maintenance and energy consumption costs [31] while lacking any kind of mobility. A solution using similar multithreading programming environments at a smaller package would be many-core-CPUs such as the Xeon Phi. It has been proven to be efficient for SNNs before [32] but can also suffer from drawbacks similar as GPUs when it comes to real-time experimentation while also cache communication overheads can also affect performance.

Mixed-signal Very Large Scale Integration (VLSI) circuits is another widely used option for SNNs due to combining advantages both from the

digital (flexibility and performance) and analogue domain (low power and large scale) [33]. Mixed-signal VLSI designs are, on the other hand, much more difficult to implement, while often encumbered by simulation accuracy issues, such as transistor matching. Additionally they lack flexibility since each system must be tailor-made for a certain neuron model. Thus, would be problematic to use during model ***fitting*** where there is constant need to edit the developing model [34], especially with more biophysically complex modeling.

Implementing the neural network in parallel custom digital hardware can naturally exploit the parallelism of biological models fully and provide real-time or hyper real-time performance useful for simulations, prosthetics and robotics applications. Digital Application Specific Integrated Circuit (ASIC) design, as otherwise called, does suffer from other drawbacks. They are expensive and time consuming to implement while not flexible. A digital ASIC chip, when implemented, cannot be altered. Minor model changes, again often required in fitting, would require a new development cycle, just like Mixed-signal VLSI.

Many of the aforementioned drawbacks of digital ASIC and mixed-VLSI hardware options can be avoided with the use of Field Programmable Gate Arrays (FPGAs). FPGAs are similar to ASICs, in the sense that they are application specific digital circuits but have the added benefit of being "field programmable". The nature of the implemented hardware on an FPGA can be modified (reconfigured) on the fly without the need of redeveloping the chip from scratch, like with digital ASICs or mixed-signal VLSI. FPGAs, though slower than ASIC designs, still provide enough performance & parallelism for real-time and hyper real-time neuron simulations. Besides being able to be used as an embedded platform, the reconfiguration property of FPGAs gives the ability for the same device to be used for different SNN models, providing good flexibility in simulations, model fitting and prototyping. Reconfigurability can also potentially provide a way to emulate the plasticity of biological neural networks in ways other solutions cannot.

1.3 Thesis scope

Because of the aforementioned reasons and the fact that FPGA design were less explored for the specific application domain compared to other HPC platforms like GPUs [35], the work presented in this thesis, initially focused on FPGA-based acceleration of complex biophysically meaningful SNNs, assuming that this would be the best choice for serving both large-scale and real-time SNN simulations. The FPGA solutions were later compared with other typical HPC solutions to validate its benefits. It was clear,

though, during the evaluation of the FPGA proof-of-concept designs that the problem of accelerating brain simulations is far more complex than just the engineering challenge of the acceleration:

- The large variety of experiment types, even when focusing on a single model type, cannot be always supported efficiently by a single HPC platform.
- The acceleration devices are too complex to be programmed and set-up reliably (assuring reproducibility and portability) by non-engineer experts (something not unique on brain model acceleration). Thus mediation by the accelerator engineer is required. This, holds true not only for the more specialized FPGA platforms but also for more widely familiar technologies. But the dynamic nature of computational modeling creates significant delays and even forces the complete repetition of engineering development cycles adding significant delays to experimental process that could balance out the performance benefits of the acceleration.

The above limitations made apparent that the ideal HPC platform for large-scale and real-time-brain simulations requires programming constructs familiar to the neuroscientist that are portable and accessible and even potentially provide multi-node support (for larger-scale experiments). Thus, the final part of the work proposes and argues in favor of a unified, flexible, cloud-based, heterogeneous HPC simulation platform, called BrainFrame. The BrainFrame front-end provides a familiar (to neuroscientists) interface using on a Python-based simulation language, PyNN [27]. The back-end integrates FPGA-based, GPU-based and CPU-based HPC resources, operating transparently to the user, removing the engineering work from the critical path of the experiment.

1.4 Thesis contributions

The contributions of this thesis can be summarized as follows:

- An extensive review of the modelling aspects of Spiking Neural Networks as HPC workloads that reveal the main characteristics that can affect performance and scalability when using different HPC solutions. This analysis can be used as a guide to assess suitability of an HPC technology in relation to the neuron network application that is the focus of the in-silico experiment.

- A comprehensive analysis of prior art on the field using on FPGA technology and the identification of FPGA potential and shortcomings based on this prior art.
- The acceleration of a biologically realistic, Hodgkin-Huxley (HH) [36] based model of the Inferior Olive [22], an important subsystem of the Olivocerebellar brain system using various FPGA-based HPC technologies. Besides the scientific usefulness of the model itself, the Inferior Olive model is complex enough to provide generalized insights on the benefits and drawbacks of executing HH models on FPGAs.
- The evaluation of the FPGA-based acceleration effort and a comprehensive comparison with other HPC technologies implementing the same application highlighting the need for heterogeneity.
- The proposal and development of the BrainFrame HPC platform, supporting the Inferior Olive and standard Hodgkin-Huxley models. BrainFrame provides both a powerful heterogeneous platform for acceleration and also a familiar to the neuroscientist front-end that hides the HPC-platform porting intricacies from the neuron modellers.

1.5 Thesis organization

The thesis is organized as follows: Chapter 2 presents some basic background about computational neuron models (such as the Hodgkin-huxley model) and biological systems (such as the inferior olive) which collectively form the focus of the thesis. Chapter 3 presents a review of the modelling aspects of Spiking Neural Networks as HPC workloads. It provides a comprehensive overview and description of what kind of characteristic network neuron models have; characteristics that directly affect how well they are executed in HPC hardware. The chapter concludes with an analysis of prior art of FPGA-based SNN implementations. Chapter 4 presents all the FPGA-based acceleration efforts conducted on the Inferior Olive model as part of this thesis. The goal of the work in this chapter is to employ the use case of the inferior olive to assess the limits of the FPGA platform when accelerating such complex neuron models. The next chapter presents a comparison and analysis of the aforementioned FPGA designs with other potentially beneficial HPC technologies and reveals the merits of various HPC solutions for these workloads. Chapter 6, informed by the previous chapters' conclusions, proposes and evaluates a heterogeneous HPC system for neuron networks, BrainFrame. Finally, the thesis work is summarized in chapter 7 with conclusions and future work.

CHAPTER 2

Background

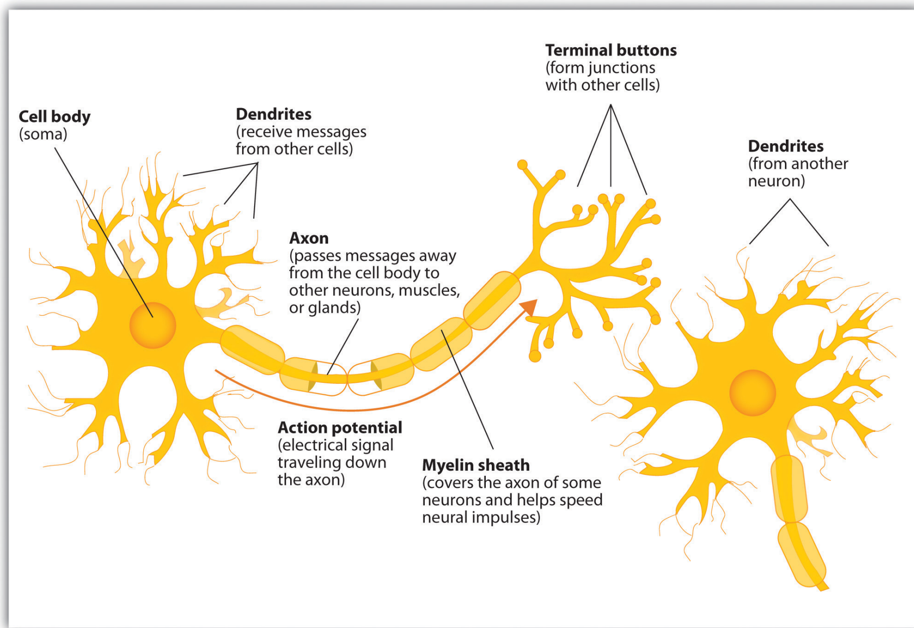


Figure 2.1: *Basic illustration of a biological neuron [5].*

2.1 The workloads of computational neuroscience

The biological neuron is comprised in general (in truth it is a much more complicated system) by three parts (called ***compartments*** by neuroscientists and modelers). The ***dendrites***, ***soma*** and ***axon*** (Figure 2.1) [26]. The dendritic compartment represents the cell input. The dendrites receive input via electrical and chemical synapses from other cells and transfer them to the soma. The soma processes the stimuli and generates them into a ***Membrane Potential***, which in turn generates a response (***Action Potential***). This response is transferred through the axon, that basically represents the cell output, to other cells. The soma accumulates inputs, influencing the membrane potential which subsequently (based on its electrochemical characteristics) may or may not generate (or fire) a pattern of spikes as output. A connection between axon and dendrite of two neurons is referred to as a synapse, which also greatly influences network behaviour. Neuron models try to represent this behaviour.

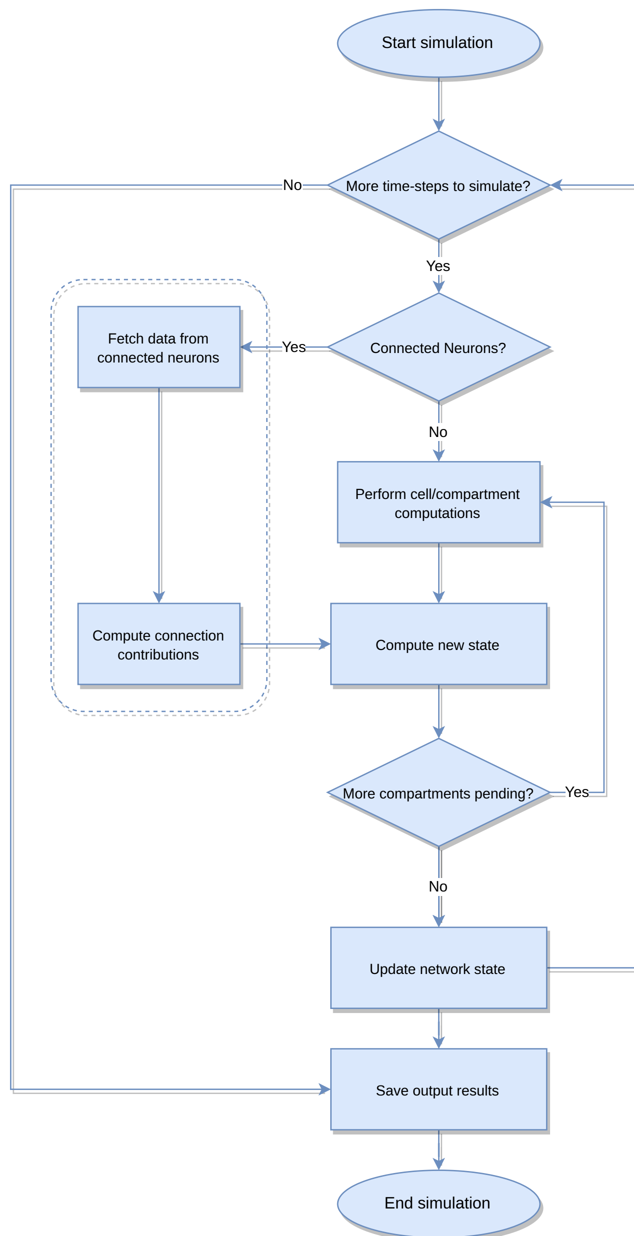


Figure 2.2: *Generalized simulation flowchart of a neuronal model.*

2.1.1 Abstract view of neuronal models

In order to execute such models on HPC resources, complex, task-intensive applications that can run in parallel need to be generated, usually denoted

as ***workloads***. Model characteristics, naturally, can greatly affect the HPC workload behaviour. A typical representation of a neuronal network simulation workload can be seen in Figure 2.2. Of course, the various workloads in the field can be much more complex, but this flow can be generally found in the vast majority of workloads. A simulation is separated into simulations time steps. The computations of the previous step are required to be completed to move to execution of the next one. Within a simulation step the various compartmental and network connectivity influences on the state of the neuron compartments (represented by ***state variables***) are calculated. If network connectivity is present, data from other neurons need to be retrieved first and their contribution added to the neuron and compartment states.

In the simplest abstract view of a neuronal network model the cells of the network are connected with each other and exchange information (potentially) in each time-step. These connections can be simple data transfers but could also incur additional computations on top of the computation within a cell, modeling specific electrochemical activities of the connections between biological neurons. In each step they can also receive external evoked inputs (network input). The cell system works in lock-step computing discrete state and output values for each cell that, when aggregated in time, contribute to form the electrical waveform response of the network.

2.1.2 Neuron model types

There are a number of mathematical models describing the behaviour of compartments of spiking neurons. The more complex the model,it can emulate more biological dynamics and is a more biologically meaningful representation of the real cell (Figure 2.3). This does not mean that the more complex the model the more useful it is. Models of every complexity have merit. The level of complexity required is highly dependent on which scientific question is being explored. Without exhausting the field, there are two main subfamilies of spiking neural models that are widely used; phenomenological and biophysically plausible models.

Phenomenological models use simple solvers that emulate the Input/Output behavior of a neuron but are not emulating the biophysical properties of a cell (chemical channels, compartments etc.) of the neuron as shown in Figure 2.1. Some typical examples of widely used phenomenological models are the Integrate-and-fire, Izhikevich [6] and AdEx [37] (AdEx is a middle ground between phenomenological and more biophysically plausible models but for simplicity grouped here) models.

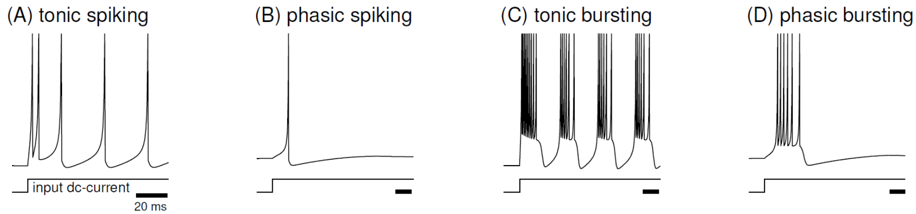


Figure 2.3: *Examples of Basic I/O responses of neuronal behavior [6].*

Integrate-and-Fire (IaF) models, emulate the most basic property of a spiking neuron. It is essentially modelling an integrator function. The model accumulates spikes coming from other cells as inputs, and for every spike it receives, increases the membrane potential. When the potential surpasses a set threshold, the neuron fires an output spike. Then, the membrane potential is reset to a resting value in which it cannot fire any output spikes, from which it gradually returns to a state whence it can begin again to accumulate inputs. There are a number of extensions upgrading the basic model to include more behavior found in biology, which on its own is very simple and models a very small portion of biological behaviour.

One immediate and simple extension is the Leaky-Integrate-and-Fire model [6]. Here, an extra property of membrane potential decay is added. The potential reduces in value over time as no new inputs are coming to the cell. Another extension is an integrate-and-fire model with adaptation. The model's response adapts according to the frequency of spike inputs and not only to their number. Smith et al. [38] proposed an Integrate-and-Fire-or-Burst model. It was developed as a model of thalamo-cortical neurons. The extension provides the model with extra neuron behaviours, such as bursting instead of just tonal spiking of the original models. Another extension of the IaF is Resonate-and-Fire [39]. Here, the model incorporates the characteristics of a resonator. A resonator neuron only responds (fires) to a certain frequency of spiking inputs, instead of all. Under certain parameters the model becomes again an integrator. Finally, an interesting alternative is the Quadratic Integrate-and-Fire model, referred also as the Theta-neuron [40,41]. The model is powerful enough to incorporate spiking latencies, input-dependent threshold value and bi-stability characteristics.

Izhikevich neurons [42] are a special type of model which emulate an impressive fraction of the biological-neuron behaviour. It is presented as a interesting compromise between IaF models and biophysically plausible models attempting to bridge the gap between the biophysically-meaningful

model, which provide high biological plausibility but also high computational demands, and IaF models that are computationally efficient but often too simple. It boasts the ability to emulate most input/output spiking activity found in the biological neuron such as spiking, bursting, and mixed mode firing patterns, post-inhibitory (rebound) spikes and bursts etc [6]. At the same time, it has similar computational demands as the more complex IaF models. Its flexibility permits computational neuroscientists to create very accurate high-level representations of large-scale, biological-neural-network behaviour.

The Adaptive Exponential integrate-and-fire model, also called AdEx [37], model that combines properties from the Izhikevich and the exponential versions of IaF models. It has similar complexity to and can reproduce all features the Izhikevich model can, with the main difference being that the rate of change of cell voltage is not quadratic (as in Izhikevich) but exponential. The Izhikevich model suffers from the upswing of its action potential being too slow compared to the biological neuron, a problem that the AdEx neuron does not have, due to the exponential voltage dependence. As a result, the AdEx provides a much more realistic representation of the biological neuron, even though technically supporting the exact same features as the Izhikevich model.

If the topic of research is focused on the internal dynamics of a neuron and how that affects its input and output behaviour, a phenomenological neuron representation cannot be employed. **Biophysically plausible models** need to be used then. A typical subfamily of biophysically plausible models are ***conductance-based*** models. These models capture the biophysical properties (multiple compartments, electrochemical channels and gates etc) of the biological neuron. The first and most pivotal conductance model is the one presented by Hodgkin and Huxley (HH) in 1952 [36]. It is considered one of the most accurate mathematical representations of neuron behaviour. The main challenge with HH models is their extreme computational costs required to model all neuronal characteristics (about 2 orders of magnitude more computational demanding than Izhikevich and AeEx neurons [6]). This has huge impact on simulation times and can be prohibiting for achieving real-time simulations speeds or modeling large scale networks.

For this reason, simplifications of the HH model have been created over the years, with reduced capability in biological emulation, but lighter computational needs. Such simplifications are the FitzHugh-Nagumo [43] and Wilson [44] models. FitzHugh-Nagumo has significantly less computational demands but cannot represent chaotic spiking dynamics and bursting. The

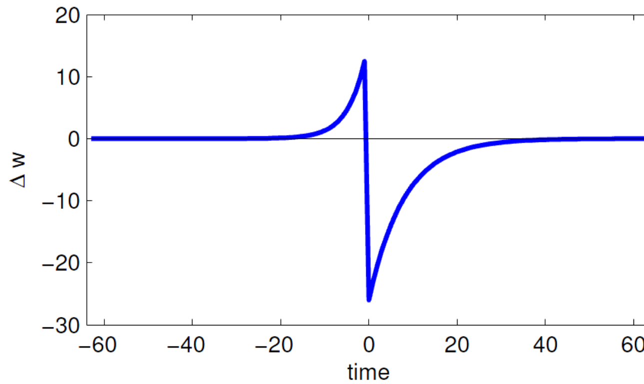


Figure 2.4: *General STDP modification function [7].*

Wilson model is much simpler than the HH model but significantly more complex than the FitzHugh-Nagumo. It can model almost all possible behavior with correct parameter tuning (also known as ***fitting***). Correctly fitting Wilson models, on the other hand, is often characterized as a cumbersome process [6]. Another example is the BoothRinzel model [45], which is a two-compartmental model created to study bi-stability properties of dendrites in motor neurons.

2.1.3 Network connectivity modeling

An important role in the neural network dynamics of biological systems is network connectivity. There are various types of connections between neurons within a network. One is the synapse which, as mentioned previously, is the connection between the axons and dendrites of different cells. Synapse modeling can be as simple as a data transfer, an accumulator or even modeling more complex electrochemical behavior, that can add to the overall network computation requirements significantly. The synapses have strength that is represented by weights influencing the inputs coming from other cells and affect the resulting membrane potential. These weights can also be adaptive based on the state of the network (***synapse plasticity***). In 1945, Donald Hebb formulated the theory that synaptic plasticity is dependent on synapse usage [46]. If a synapse stimulates a cell repeatedly and causes the cell to fire, then the synapse efficiency (weight) increases. Synapse weight decreases when a connection is rarely used.

An approximation of the hebbian rule useful for synapse modeling is the ***Spiked Time Dependent Plasticity*** (STDP) algorithm (Figure 2.4) [47,

48]. This learning rule is based on the time difference between post-synaptic spikes and pre-synaptic spikes. According to this rule, a reference time span is defined, usually in the order of tens of milliseconds. If a pre-synaptic spike arrives in that timespan before a post-synaptic spike, it is considered to contribute to the cell spiking activity. According to the rule, this synapse should be strengthened. If it arrives in the timespan after the occurrence of a post-synaptic spike, the synapse weight is reduced, assuming that cell activity is independent of the activity of this synapse. A hybrid STDP and ***Genetic Algorithm (GA)*** learning rule was even published in 2006 [49], in which the GA was controlling network topology adaptation.

Another type of connection between neuron cells is the ***Gap Junction***. Gap junctions are electrical connections that can develop between any part of neighboring neurons due to physical proximity. Thus gap junction connections do not only manifest between axon and dendrite but between any neuron compartment (like between somata or dendrites of neighboring neurons). Since gap junctions are virtually channels through which charged ions move, they are represented by complex electrical equations that increase the per time-step computations significantly.

When modeling a neuron network, a neuroscientist can usually combine most cell models types with most connectivity models into the same network, depending on the biological network they want to represent. This adds even more to how dynamic and diverse the field of computational neuroscience is in terms of workloads.

In many cases, neuronal network models are effectively behaving like an assortment of integrators. This is often the case with networks using ***Phenomenological*** cell models or simple connectivity modelling. In such cases event driven simulation flows can be employed. Each time step calculations are only done in case of a spiking event that affects the output of a cell. Otherwise that specific cell is ignored, assuming that its state is unchanged, thus computations are skipped. For non-complex networks, event driven flows can be used to reduce computation requirements greatly. On the other hand, complex connectivity modeling or the use of ***Biophysically Plausible*** cell models often create coupled oscillators that need to be co-simulated in strict lockstep among them even with the absence of a spiking event in the input. This enforces the use of cycle-accurate, transient simulators where simulation steps are hardly compressible and all neuron states need to be completely updated at each simulation step. Here, event-driven execution cannot be employed, thus computation requirements increase significantly.

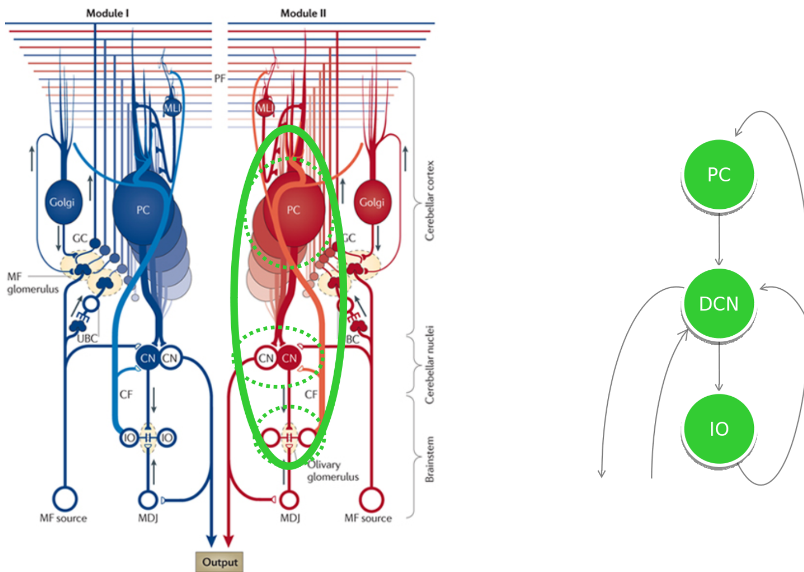


Figure 2.5: *Depiction of the Olivocerebellar System (left) [8] and abstract depiction of main Olivocerebellar loop (right). PC=Purkinje cell, DCN=Deep Cerebellar Nuclei, IO=Inferior Olive.*

2.2 The inferior olive

The inferior-olivary nucleus forms an intricate part of the olivocerebellar system (Fig 2.5), which is one of the most dense brain regions and plays an important role in sensorimotor control. Activity in the inferior olive only directly triggers movements, when it is synchronized among multiple neurons [50, 51]. In addition, the olivary neurons can provide rhythm and coordination signals for motor functions [8]. It is considered to be imperative for the instinctive learning and smooth completion of motor actions [52]. The olive provides one of the two main inputs to the cerebellum through the climbing fibers.

What makes the inferior-olive neurons special is their dense interconnection through gap junctions (GJs). The gap junctions facilitate the synchronization behavior between the olivary neurons and, subsequently, influence the synchronization and learning properties of the entire olivocerebellar system [8]. A model of the inferior olive will be a main benchmark for the work in this thesis. The multiple compartments, GJ connectivity and synchronization characteristics result in a highly demanding workload for use with HPC technology, as shown in later chapters.

CHAPTER 3

Accelerated SNNs on Reconfigurable Hardware

In this chapter, we shall review the modelling aspects of Spiking Neural Networks as HPC workloads for reconfigurable hardware. This includes an analysis of the explanatory strength and the computational complexity of a model, the effects of the numerical implementation. This analysis allows us to form some key observations that can be key for a beneficial acceleration of such workloads in HPC technologies and FPGAs specifically. This chapter will also present a summary of the state-of-the-art of FPGA-based neuron model acceleration and present the platforms place in the field, both for the use in A.I. applications and simulations for brain experimentation. Finally, based on this survey, observe the strong and weak aspects of such FPGA implementations to identify possible topics for improvement.

3.1 Explanatory power and computational complexity of neuron models

The choice of which neuron model to use for an experiment as mentioned above is not an easy one. Just because a model's representation is more accurate and more detailed it does not mean it is the most suitable for use in any possible simulation scenario, compared to a model with a more coarse representation. Suitability depends heavily of the topic of research. For example if the research question revolves around large scale experiments that do not require small time-steps, I&F neurons can be suitable [53] even if limited in the features they support. There is also a trade-off between computational simplicity and explanatory power (behavioral feature support) of a model. The explanatory power of a model is directly tied to the input/output or electrochemical behavior it tries to represent. The more behaviors or biophysical details are modeled, the greater the number of the state variables will be needed to describe the neuron. Thus, we can make the reasonable assumption that the number of neuron state variables a neuron contains is an indication of its explanatory power. A more powerful model also tends to be more computationally complex, thus poses a more challenging workload for a computing system and vice versa. It would be interesting to quantify the biological explainability that a given neural network offers – i.e., how faithfully to biology it can explain certain phenomena – with respect to the compute complexity it requires. If this relationship is not linear, it means that there will be interesting model niches offering sufficient explainability to modelers at a proportionally lower compute cost.

We can plot model explainability as a function of model type employed for different sizes of brain networks all the way to the size of a whole human brain (ca. 1 liter in volume), assuming average anatomical characteristics

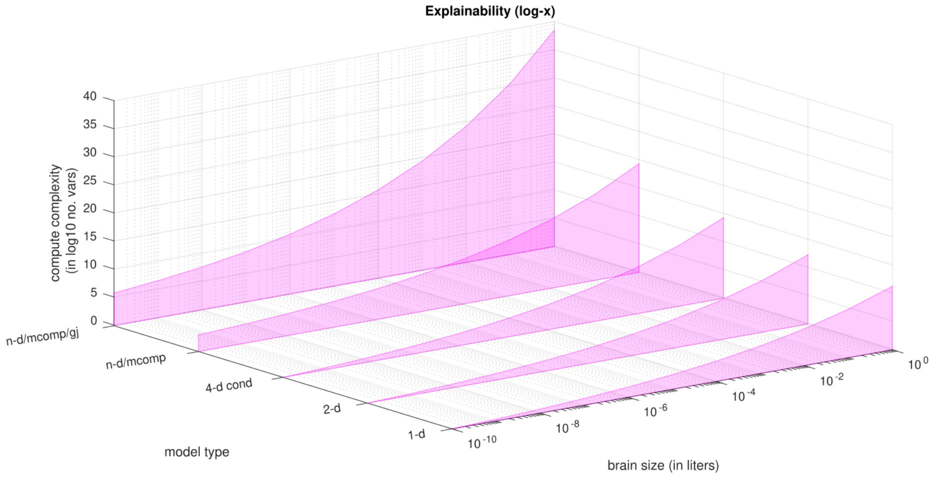


Figure 3.1: *Computational complexity (#State Variables vs model dimentionality and brain size modeled. n-d/mcomp assume an example of a 3 compartmental neuron) - Higher complexity denotes higher explainability.*

of the human brain [54], within a single graph (Figure 3.1). Typical 1-d models would be the integrate and fire variants that usually implement 5-10 floating point operations per neuron (FLOPs). Typical 2-d models are the Quadratic IaF and AdEx model that vary between 7-30 FLOPs while single compartmental models are often characterized as 4-d models. Single compartmental model can vary from tens of FLOPs (like the FitzHugh–Nagumo with 72) to a few thousands like the Hodgkin-Huxley (1200 FLOPs) [6]. Adding multicompartmental models (denoted in the graph as n-d) the compute complexity increases significantly. It must be noted though, that the greater increase in the state variable number comes when adding biophysically meaningful network connectivity (like for example Gap Junction modeling - n-d/mcomp/gj on the graph) and multicompartmental models making their computation, potentially, the most challenging type of workload when such high explanatory power is required. The IO model that is the focus of the work later in this thesis is such a model.

3.2 Numerical analysis

Biophysically plausible SNN models are in practice dynamical systems. The behaviour of the SNNs are described by state variables that change over

time, as mentioned in section 2.1.1. The models consists of ordinary differential equations) [55], describing the derivatives of the state variables. These equations, in most cases, cannot be solved analytically and therefore, require the use of numerical methods. In this case the model assumes a uniform voltage potential on each cell membrane.

The type of numerical methods used to solve these ODEs directly affects the size of the time-step Δt , and subsequently has an huge influence on the computation time of a simulation. A smaller Δt gives more accuracy, however, requires more computation steps than a larger Δt for the same simulation. Furthermore, how accurate the solution is for a certain time-step size is dependent on which numerical method is used and, thus, the numerical methods have a significant influence on the both the performance and accuracy of the simulations. To give some insight on the behaviour of different numerical methods, different varieties and method characteristics will be discussed further.

3.2.1 Numerical methods

Numerical methods approximate the solutions of the mathematical equations numerically. This is done by using finite computational processes. Consequently, the continuous problems are solved using discretization, introducing truncation errors. Furthermore, as a computer can only represent numbers with a finite precision, rounding errors are also introduced. To analyse the sources of the errors introduced by the use of numerical methods three concepts are used: **stability**, **consistency** and **convergence**. A numerical method is called stable if the error remains bounded after the use of multiple iterations. The consistency of a numerical method signifies that the discretization error of the method goes to zero if the step size goes to zero. Finally, convergence describes that the error of a numerical method is always lower than a certain arbitrary value. Furthermore, **stiffness** is used to measure the difficulty of solving an ODE system. A system is stiff when the different equations in the system are numerically unstable except when the step size is small.

Another categorization of numerical methods is whether they are using **explicit** or **implicit** solvers. An example of an explicit method, meaning that the calculation of future steps in time only requires the values of the current step, is the forward-euler method [56]. In an implicit method, on the other hand, the calculation of the future time steps, implicitly involves the values of the current and future steps in its calculations. An example of an implicit method is the backward-euler method [56].

Both the forward-Euler and backward-Euler method are methods of the first order. Meaning that only one evaluation per time-step is computed and resulting in a truncation error bounded by $\mathcal{O}(\Delta t)$. Generally, the error bound is p where p is the solver order. Thus, the truncation error is defined as $\mathcal{O}(\Delta t^p)$. Higher-order methods do multiple evaluations per time-step to get better accuracy. As the number of evaluations per time-step scale linearly and the error bound exponentially, extra computation is required per time-step, which in turn will result in a larger time-step for the same accuracy in comparison to lower-order solvers.

All aforementioned methods used a fixed time-step size. The maximum value of the time-step is determined by the most rapid changes in a simulated spiking neuron, during active phases like during an output spike. During the less active (silent) phases, e.g., before or after an output spike, an accurate solution might be possible with a larger time-step size than when the output is changing rapidly. To exploit this behavior, methods with adaptive step size can be used. In models using adaptive methods the time-step will be large during silent phases and small during active phases to give an accurate solution. This reduces the total number of time-steps of a given simulation providing greater performance. For this method to be applied, an indication of the local truncation error (truncation error per time-step) is required to be calculated. This error can, for example, be calculated by the difference between solutions of different orders [57].

Another way of solving the ODE system is with the use of exponential time differencing (ETD). The idea behind ETD is to freeze some variables, so that the equations become linear and then solve them analytically. The simplest ETD method is called the exponential-Euler method. Using an ETD method will, on the one hand, take longer per computation, especially when the linear system that needs to be solved is large. Furthermore, due to the linearization, the solution will be less accurate. On the other hand, if the problem is tolerant to the lower accuracy then larger time-step values can be used increasing potential simulation speed [58].

3.2.2 Behaviour of numerical methods on SNNs

As discussed before, there is a wide variety of numerical methods to solve ODE systems. There is not a single method which performs best when applied on simulations of SNNs, as the performance of the numerical methods depends on the model used, the network size, the connectivity of the network, the desired accuracy and the spiking activity.

Henker et al. in [59] evaluate the accuracy of numerical methods specifically on simulations of integrate-and-fire neuron models, which were linearly solvable. The results reveal that:

- The error of explicit fixed time-step methods is independent of the order. Consequently, a higher order method only adds extra computation without increasing the accuracy.
- Decreasing the size of the time-step led to an increase in accuracy for adaptive time-step methods compared to fixed time-step methods.
- The error thresholds, used within the adaptive methods, have a significant influence on the accuracy and have to be set manually before using the solver.
- The ETD method only fails for large time-step sizes. However, the accuracy does not increase with smaller time-step sizes, which is the result of the behaviour of solving linear equations.

The previous results presented the behaviour of linearly solvable methods of integrate-and-fire models. However, many ODE systems describing SNNs are non-linear and analytically unsolvable (like HH networks). Börgers et al. [58] discuss the behaviour of numerical methods on non-linear HH models. The paper analyzes the rising phase of a voltage spike which defines the accuracy. Since this is an active phase of the simulation, the time-step size must be small to achieve good accuracy independent of the order of the method. This small time-step is only required during spiking behaviour and, between the spikes, much larger time-steps achieve good accuracy. This would indicate that the use of adaptive time scaling methods is suitable, however, this might be not beneficial when the cells are connected and produce asynchronous spikes. In such cases, even if only one spike is present in the network, the time-step has to be sufficiently small for an accurate solution.

Furthermore, the use of implicit methods is not useful for the case of complex, non-linear SNNs. In order to solve the non-linear system constraints, the time-step size for an implicit method has to be of the same order as the respective explicit method to provide accurate results, thus providing no further benefit. The use of ETD methods shows accurate results with larger time steps. However such ETD methods are only useful when an accurate and precise shape of the spikes is not required. Therefore, when features like gap junctions are present, this method cannot be used as the shape of the spikes has an effect on overall network synchronization.

3.2.3 Model fitting

When building new models, neuroscientists need to perform so-called model fitting; a process by which they subject their models to different input or state conditions and measure their output. This is an essential step for matching the in-silico to the biological behavior of a modeled neuron, and it typically calls for large parameter-space sweeps. This way the modeler can decide, based on these trial runs, the correct parameters for the models and numerical solvers chosen and ensuring stability and functional correctness before the actual simulation runs. This process is of course very time-consuming and includes significant amount of trial and error, often more time consuming than the actual experiments [34]. Model fitting makes acceleration relevant not only for the experimentation but also for the model development process.

3.3 Complexity analysis

As is clear from the previous sections, the computational-neuroscience field is diverse and it is difficult to provide a complete picture of the various SNN models, seen as computational workloads. Still, it is useful to attempt to perform a complexity analysis on a few, representative models among those discussed. For this, we choose to analyse three of the most feature-rich models in their category: The Resonate IaF, the AdEx and the HH models. We measure computational complexity of SNNs in terms of performed single-floating point operations (FLOPs) and as memory complexity in terms of required memory space. We also assume that memory data need to transfer at least once from memory to the processing unit per simulation time-step (as per time-step parameters and connectivity data should be fetched at least once), thus, the memory footprint is also an indication of the required memory traffic in an execution of the model. Estimations of the cell-model operation number are taken from [6] assuming a 1 ms time step. To accurately calculate memory and computational requirements, we use three use cases of connectivity modeling, which are representative of typical neuroscientific neuroscience simulations:

1. Realistic Gap Junctions (GJ) – Cells are modeled with (biophysically) realistic GJ interconnectivity, based on the GJ modeling of [22], profiled as in section 4.1.3. The highest amount of detail is included in this connectivity modeling from our list of use cases.

2. Simplified connection – Cells modeled with the Gjs replaced by simplified, passive connections. This instance essentially models a sim-

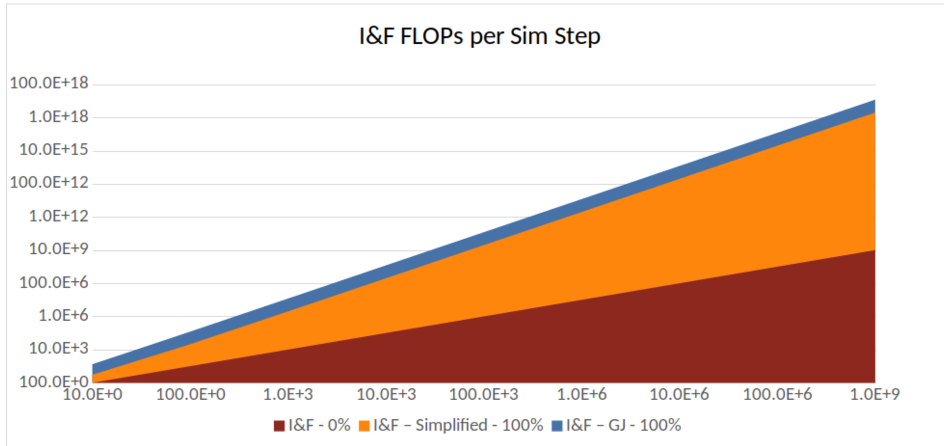


Figure 3.2: *FLOPS vs Network size per simulation step for the Resonate Integrate and Fire model.*

ple input accumulator. The accumulation is parameterized using the weights that are assigned to each connection between two neurons. The implementation does not have a specific biological meaning. It is dummy paradigm that is a good tool to see how workload behaviour when connections have significantly lower processing requirements while keeping the connectivity the same.

3. Cells modeled without any interconnectivity (0% connectivity density). This is the simplest use case, whereby the neurons are modeled as separate computational islands; no updates among them are needed. In practice such simulations can be useful when connectivity data are processed on the host or when single neuron models are explored. This drastically simplifies simulations and, from a graph standpoint, makes them embarrassingly parallel.

In Figures 3.2, 3.3 and 3.4 we can see the FLOPs per simulation step for each cell model type vs network size. The top lines of each area represent the FLOPs for the maximum connectivity density which is 100%. The area below these lines, up until the line for the 0% connectivity use case, represent all the possible FLOPs for each use case for all connectivity densities between 0% and 100%. The 0% connectivity area (red) also represents the part of the computation that comes purely from the neuron computations without the influence of connectivity computations. If we focus on the cell computations only (ignoring any connectivity) we can see clearly the

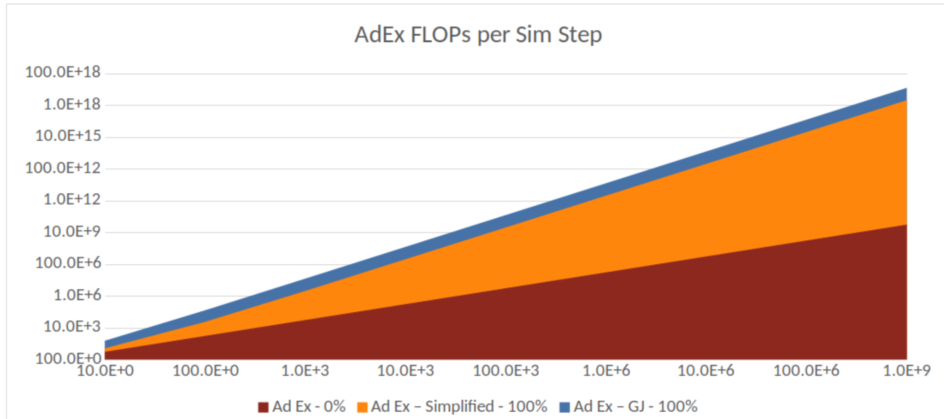


Figure 3.3: *FLOPS vs network size per simulation step for the AdEx model.*

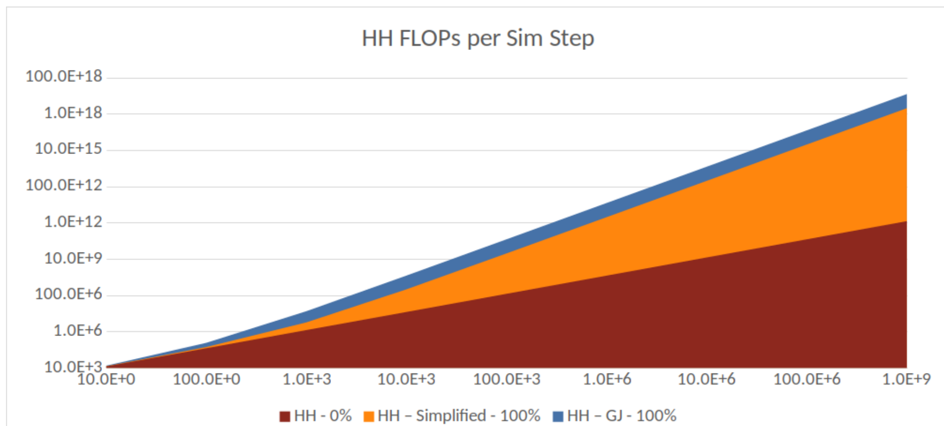


Figure 3.4: *FLOPS vs network size per simulation step for the Hodgkin-Huxley model.*

significant difference in cell computations between the phenomenological and the bio-physically meaningful HH model. An interesting observation is also the significant influence on computations when connectivity is present, especially for higher connectivity densities. The presence of connectivity computations causes a quadratic increase in operations for higher network sizes, even for the simplified connection that is just a simple accumula-

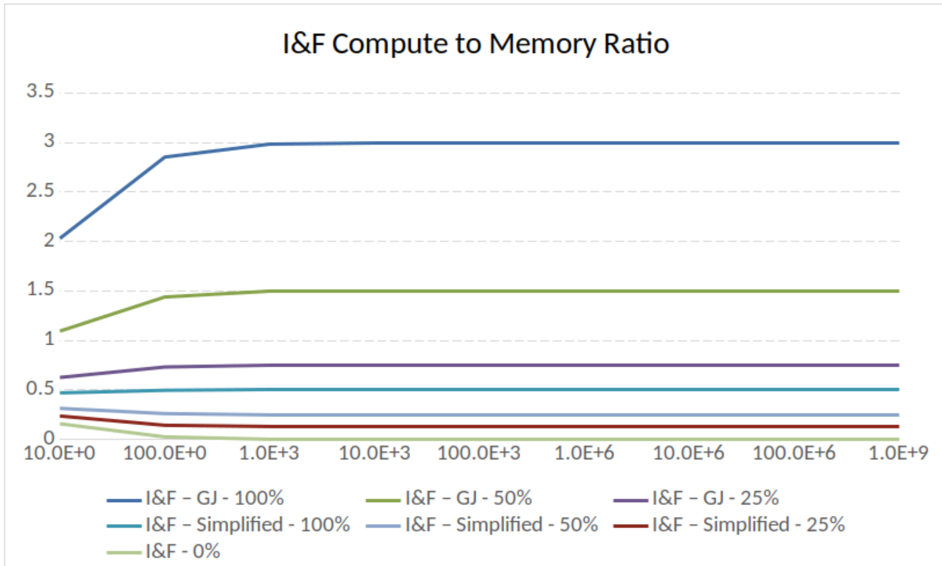


Figure 3.5: *Arithmetic Intensity vs Network size for the Resonate Integrate and Fire model.*

tion operation, compared to the more complex GJ connections, that have around 6 times more operations per connection.

Another useful aspect to analyze is the arithmetic intensity of the workloads (Figures 3.5, 3.6 and 3.7). We define arithmetic intensity as the compute to memory ratio for each use case (FLOPs/Byte). In essence, this metric attempts to quantify the amount of operations performed on every Byte of data fetched from memory, since data movement is nowadays the largest challenge in high-performance computing [60, 61]. Thus, a larger arithmetic intensity signifies a more efficient use of memory. The main source of memory footprint in the SNN applications are the neuron state variables and the connectivity information, which takes the form of a connectivity matrix. The matrix stores the weights of the connections in floating point format. If a value is equal to 0 then the specific connection is not present. With the assumption that each byte of data will need to be transferred at least once within a time-step, an arithmetic intensity higher than 1 can signify that the workload is **compute bound** (more than 1 floating point operation for each byte transferred), while lower than 1 signifies a **memory-bound** workload (more than 1 byte transferred for each floating point operation).

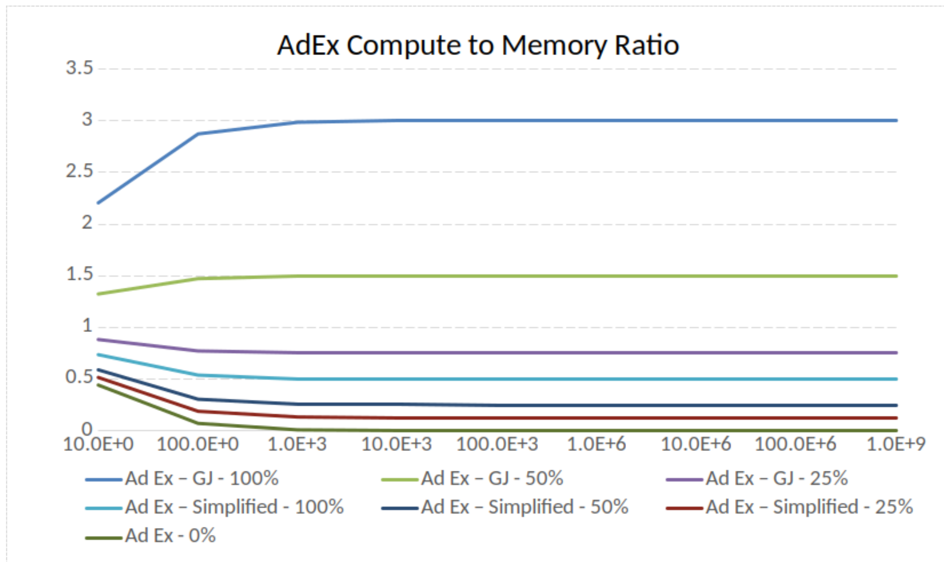


Figure 3.6: *Arithmetic Intensity vs Network size for the AdEx model.*

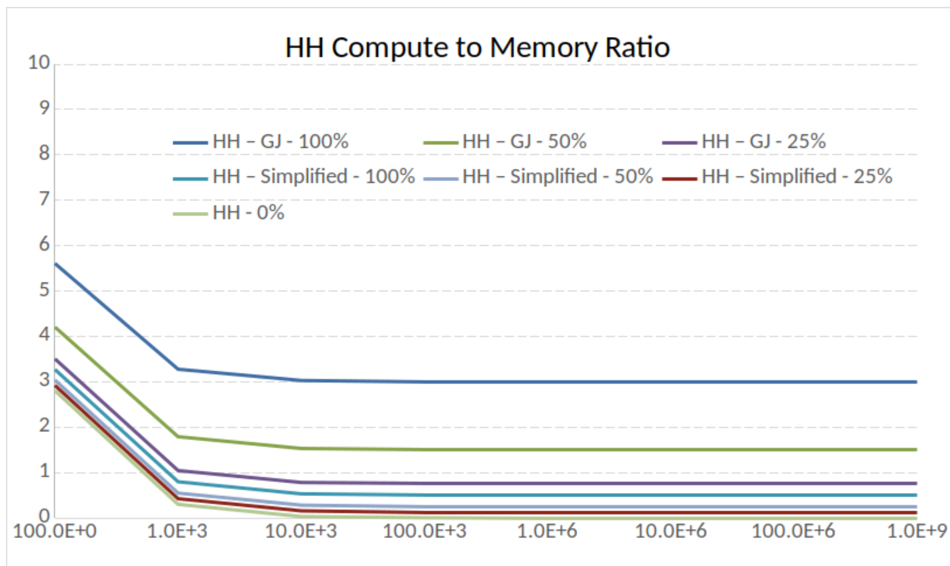


Figure 3.7: *Arithmetic Intensity vs Network size for the Hodgkin-Huxley model.*

We can see right away that the connectivity density, for the cases of the simpler IaF and AdEx models (Figures 3.5 and 3.6), is a major factor in deciding if a workload is compute or memory bound for the two phenomenological models. When connections between neurons are complex (GJ) and dense (50% and over) the workloads for both the IaF and AdEx models are compute bound. But the GJ-enabled workloads with lower density and all other use cases with simpler or no connectivity actually are memory bound. It must be noted here that the connectivity density in experiments is user defined and it would change often during/between simulation runs. This means that the same workload with different connectivity densities can under certain circumstances be either memory or compute bound. The situation becomes even more interesting for the HH models (Figure 3.7) where the cell computations are a lot more than the computations within the connections compared to the the I&F and AdEx cases. The HH model use cases in smaller network sizes begin largely as compute bound as the cell computations dominate compared to the memory needed for the connectivity matrix. As the network sizes increase the connectivity matrix increases at a higher rate to the total computations and the workloads become less compute bound and around the 1000 neuron mark, they revert to a similar situation as the the I&F and AdEx models; the use cases with the higher density and complex connections remain compute bound but all other cases become memory bound.

Summarizing the key observations from the aforementioned analysis:

- Neuron model computational requirements increase with network (problem) size and with model complexity/type.

- Memory (and I/O) requirements stay more or less the same and are somewhat independent from model/synapse type, for a given network size . The size of the connectivity matrix keeping the connection data is far larger compared to any other neuron model data, except for very small network sizes (assuming no storage/data optimizations) or the complete lack of connectivity.

- Compute-to-memory ratio also highlights the impact of the network/synapse modeling when simple phenomenological models are used. Very high density and complex synapse modeling make a workload compute bound. With sparser connections or a simpler synapse, the application can become memory bound.

- For bio-physically meaningful modeling (HH): Depending on the density and complexity of the connectivity an HH application is heavily

compute bound for small problem sizes and becomes memory bound with increasing problem size, depending on density and synapse model complexity.

  – On small problem sizes, the neuron computation dominates while the size of the connectivity matrix storing the connectivity information in memory and the synapse computations remain small. The HH neuron model operations are dominant and the workload is compute bound.
  – As problem size increases (over 1000 neurons), the connectivity matrix increases quadratically but also the computations for the synapse modeling. The applications with complex synapses of high density still remain compute bound. Computational requirements increase fast enough to keep up with the increased memory requirements of the larger matrix.
  – For less dense networks (50% connectivity ratio and under) or simpler synapse modeling, applications turn from compute bound to memory bound. The increase of the memory requirements overcomes the increase in operations.

- The complex behavior of network neuron models (especially for the biophysically meaningful cases), poses a significant challenge for accelerating in an HPC environment, as a "one size fits all" implementation strategy that caters to each model's needs – even to the same model's needs under different starting conditions – does not exist.

3.4 Summary of design considerations and trade-offs

Taking into account all the aspects of SNN modeling and acceleration discussed in the previous sections, we can derive a comprehensive list of trade-offs when designing neuron models while having in mind their performance and acceleration potential. Some of the design considerations should be kept in mind from the start of the process, as basic modeling decisions can affect the potential performance of the eventual workload substantially. In practice a neuroscientist developing a model may not often be aware of how their modeling decisions are affecting the end performance, and subsequently the speed and network capacity of their in-silico experiments (Table 3.1).

Model complexity and **connectivity density** have a clear effect on performance and supported network size, as discussed in the previous

Table 3.1: *Design factor trade-offs for neuron model workloads with regard to performance and supported network sizes.*

Design Factor	**Performance**	**Supported network size**
Increased # of compute units	↑	↓
Larger time step	↑	-
Event Driven Execution	↑	↑
Off-chip memory	↓	↑
On-chip memory	↑	↓
Higher Conn. Density	↓	↓
Transient simulation	↓	↓
Higher model complexity	↓	↓

sections. A less obvious factor, caused by the chosen numerical implementation, is **time step** duration of the model. Besides the time step, the numerical implementation can also affect the requirement to run the model using a **transient** simulator, that is much more challenging computationally, or if it is possible for the model to be executed in a **event driven** fashion, which is better performing and less computationally demanding for HPC acceleration.

A separate kind of trade-off is also related to the architectural decisions taken in the accelerated implementation. The memory architecture of the accelerator can have an important effect. For example, the use of **on-chip memory** for storing neuron states and connectivity matrices can result in significant performance benefits. On the other hand, on-chip memory tends to be significantly small in capacity resulting in the maximum supported network size to be limited compared to using the larger **off-chip** (on-board or system) memory [62].

An interesting aspect is also the **size of the compute units** provided or implemented on the HPC platform. Larger but fewer compute units within the same accelerator can enhance performance per neuron by providing more computation resources. But larger compute units also consume more logic resources that can limit the number of neurons that can be evaluated per time step and visa-versa. Whether a workload is more suited to be accelerated by small but numerous compute units or by fewer but more powerful ones is strictly application dependent and can be significantly affected by modeling and numerical decisions.

3.5 FPGA spiking neural network implementation categorization

In this section, we collect and present the most significant works in literature on the implementation of SNNs on FPGAs. We divide them in 2 main categories, each bearing its own significance. First, we shall present the largest group, that of implementations targeting either simulation acceleration or general neural modelling or, even, explicit experimental applications. The second group covers a number of bio-inspired SNN implementations for the purposes of specific applications, such as pattern matching and robotic control. Later in this section we present and discuss other notable aspects of prior art such as architectural characteristics and performance efficiency.

3.5.1 SNN applications for brain simulations

A significant amount of work has been conducted for the purpose of brain simulations/exploration. The choice of model and type of neural network varies based on the required simulation accuracy and resource constraints, and can range from the simplest models to full HH representations.

Implementations of IaF models vary for networking sizes from a few dozens or hundreds [63–70] in the early implementations, to hundred of thousands [71–74], in more mature ones. FPGA implementations have also successfully incorporated learning, most notably STDP algorithms [67, 75]. A notable example of such an implementation is presented by Glackin et al. [53] that managed to emulate the function of the Medial Superior Olive in terms of sound localization. The brain uses the time difference of the same sound coming to different ears to establish the general direction of the sound source. The network implemented simulated 1029 neurons (49 physical model instances time-multiplexed) and 25221 STDP synapses. It performed in real time and boasted very accurate sound localization results. The implementation was done on a Xilinx Virtex-4 FPGA. Based on the FPGA resources used, the authors estimated that the model could simulate at most 7.5k neurons with 1.35M synapses. A population that is quite close to the human superior olive subsystem, estimated around 10-11 thousand neurons.

The need for more complex or high scale networks for the emulation of more realistic brain systems has lead researchers to employ multi-FPGA designs for simulating IaF models [76, 77]. Kousanakis et al. [78] have developed a multi-FPGA system that emulates the cerebral cortex neurons using IaF models and highly complex dendrites modelling alongside NMDA

synapses. The design was demonstrated on a system simulating 240 neurons with about 12000 dendrites and more than 800000 synapses. Even though it was not able to reach real-time performance it boasted a speed-up of $\times 923$ compared to a conventional single-threaded CPU solution. As the complexity of synapses and dendrites were configurable the design could theoretically support network sizes up to 15360 neuron with 521 synapses per neuron. Another multi-FPGA implementation was proposed by [62] realizing a system incorporating 35 Zynq-7000 devices for the simulation of a maximum network of 89600 IaF neurons, that was demonstrated by the simulation of a cortical microsection.

One of the first attempts to make an FPGA implementation of an Izhikevich neuron was done in 2005 by La Rosa et al. [79]. The goal was to test the applicability of the model on an FPGA. For this reason, just 2 interconnected neurons were implemented. The model was converted to discrete time by Euler integration and the results showed accurate behavior at faster than real-time performance. Soon after, more advanced, larger scale implementations appeared [80–82], some even often including advanced connectivity between neurons like gap junctions [83]. Very large scale Izhikevich networks have also been achieved using very simple fixed-point arithmetic [84]. Here, the focus was large scale, real-time performance alongside high energy efficiency. The demonstrated network was of 28.9k neurons with the system supporting up to 250K neurons. FPGA implementations of this type often use event-driven implementations [19,85], allowing them to alleviate the computational requirements and achieve large scale network simulations. Impressive implementations designed for multi-FPGA setups have been also developed such as the hybrid neuromorphic compute (HNC) node of [86] and the Bluehive system [20].

As discussed previously, if the goal is to simulate and study the neuronal behavior in greater detail, one has to use more complex models, like **conductance-based models**. Moreover, for future possible implantable devices, a more biophysically meaningful representation than the previous models might be required [23]. However, the added complexity creates greater limitations in terms of network size and simulation speed but hardware implementations are still capable of providing real-time behavior. Consequently, there have been some attempts to create FPGA implementations of typical conductance models [87,88]. Additionally, there are quite a few conductance model implementations using fixed-point precision [89–91]. Although fixed-point arithmetic reduces resource cost, compared to floating point, whether it can provide small enough precision errors to accurately implement biophysically-meaningful neuron models without alteration in

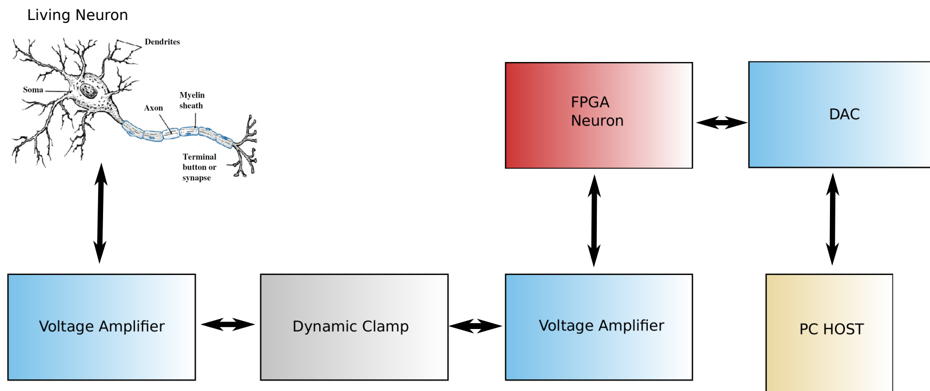


Figure 3.8: *Basic organization of hybrid network adapted from [9].*

behavior is not self-evident. The fixed-point analysis of a neuron model (initially designed with floating-point arithmetic in mind), is a rigorous process and not guaranteed to give functionally correct results. For this reason, alongside the gradual maturity of floating-point arithmetic on FPGAs, a more recent trend is to employ floating-point arithmetic instead of fixed-point for such implementations [1, 92–99].

A notable use of FPGAs in conductance-based modeling was presented by Sorensen in [9]. The research did not just tackle the real-time demand for hardware implementations but also the actual interconnection of a possible artificial neuron with its biological counterpart. Here, a number of neuron models were implemented (from an HH model to simpler IaF models) to model the behavior of a heart neuron of a leech. These models were deployed on a Virtex II FPGA which was then included on a hybrid network including a live leech neuron. Two interconnected heart-neurons have oscillatory behavior. That was also the behavior of the network that included the live and the artificial neuron on the FPGA. The Input/Output from both neurons were captured by a dynamic clamp and DAC devices were used for communication with the host PC. The FPGA model runs 36 times faster than real-time and so downsampling was incorporated to provide real-time Input/Output (Fig.3.8). These experiments proved that such a network can operate as the biological system and that also the behavior of the biological neuron can be influenced and controlled by tweaking the parameters of the artificial neuron. Finally, the influence and differences in behavior of using simpler models for the silicon neuron were explored.

Table 3.2: *A.I. applications of FPGA-based SNNs.*

Application	FPGA implementations
Visual	[107], [108], [109], [109], [110]
Olfactory	[70]
Taste	[111]
Sound	[64], [76], [53], [112]
General Pattern Matching	[113], [114], [115]
Robotic Control	[105], [106], [116], [10], [117], [118], [119], [120]

Besides standardized models, attempts have been made to accelerate completely custom-made (not adhering to typical conductance-based model implementation) conductance models. The design choices and type for these models vary depending on the simulation demands, complexity and the specific scientific question explored [100–106].

3.5.2 SNNs for specific AI applications

Besides the attempt to accelerate simulation for brain exploration, there has been work on applying biologically plausible NNs for specific applications like AI control and visual recognition. In these cases the demand for accuracy, modeling details and network size are less strict since the system has a specific task usually less demanding than a complete biological system. Especially for simple models, FPGA SNN implementations have been used for a number of AI tasks including visual, olfactory, auditory and control applications providing advanced computational power with satisfactory performance (Table 3.2).

One of the more notable AI applications for SNNs was proposed by Johnston et al. in 2010 [10] which created a control SNN for the Khepera robot. The researches developed an FPGA-based SNN using IaF neurons with synapses including both STDP learning and learning with a Genetic Algorithm (GA). The STDP controlled synaptic weights, while the GA adapted network characteristics, mainly axonal delays and synapse polarity. The network had 10 neurons, 8 of them receiving sensory input and 2 providing output for motor function according to the sensory neuron outputs (Figure 3.9). It provided control so that the robot navigates itself through a small maze, while also avoiding obstacles. The I/O interface between motor, sensors and the NN was done by UART and the design included a microprocessor downloaded on the Virtex II FPGA to execute the GA. The final implementation was 105 times faster than PC simulation.

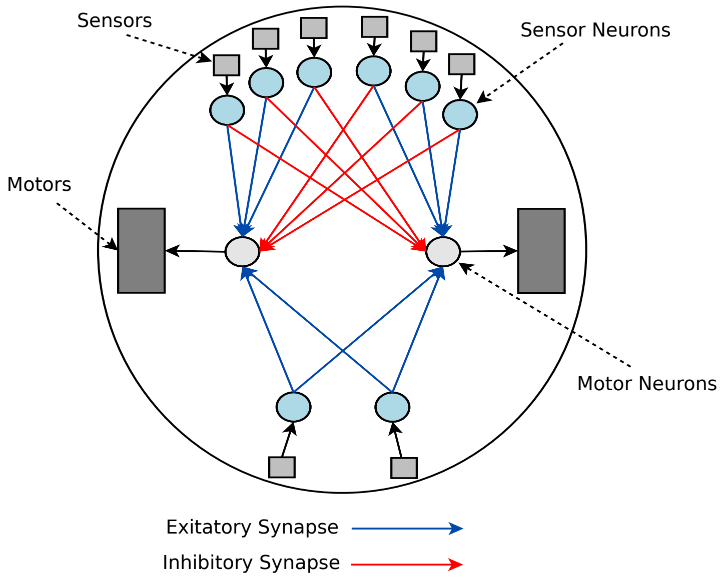


Figure 3.9: *Neural Network implementing navigation control in the Khepera robot (adapted from [10]).*

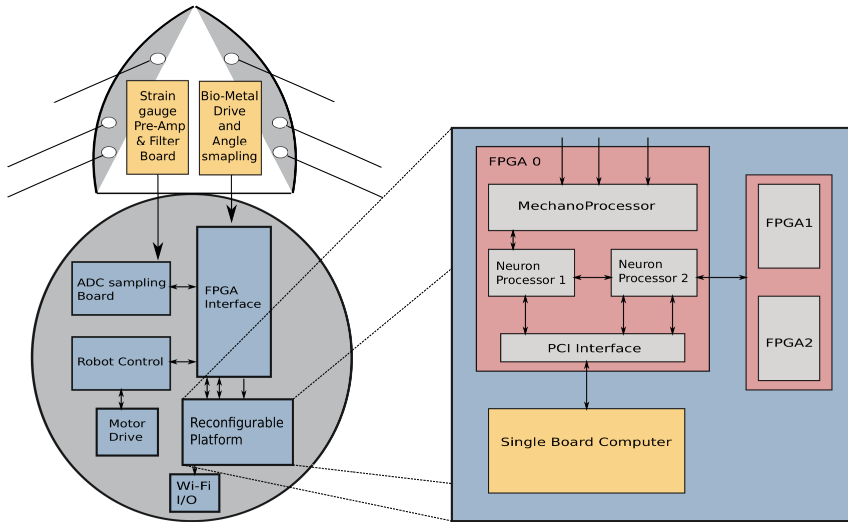


Figure 3.10: *Whiskerbot block diagram adapted from [11].*

Another notable use of SNNs demonstrating its use for robotic control was presented by Pearson et al. [117–119]. The architecture implemented

IaF neurons that included axonal delays and noise. Two types of architectures were implemented: The a-architecture had neural cores featuring 112 neurons with 9 synapses per neuron (912 synapses, in total). The system had 10 such processing elements bringing the total network size to 1,100 neurons. Since the amount of synapses is as important for biological plausibility as the network size, a second version was implemented with 64 neurons per processing element (640 neurons for the whole network using 10 elements), but with 16 synapses per neuron (b-architecture). The architectures included input and output modules using buses and data were stored on FPGA block RAMs. The device used was a Virtex II FPGA and the network was run at real-time performance.

The b-architecture core was later used to provide control for the Whiskerbot [11]. The Whiskerbot is a robot using a sensory system emulating the whisker sensory system of a mouse (Figure 3.10). A tactile sensory system (such as the mouse whiskers) has the advantage of sensing the environment without the need of any kind of illumination using either visual, sonic or Infrared (IR) signals. It is ideal if a robotic system needs to operate in confined, completely dark or noisy environments. The researchers created 6 artificial whiskers used to acquire sensory input (similarly to the actual mouse) and used a population of 6 networks with 24 neurons each, implemented with the b-architecture for control. The robot was tested on a test arena and the SNN provided navigation control based on the whisker sensory input with similar procedures as in the biological system.

3.5.3 Model choice, network size and real-time performance

The amount of work conducted in the recent decades shows a growing interest for the use of more biologically plausible NNs because of their superior computational capabilities [15, 16]. It is also clear, from the previous sections, that the computational complexity of such networks poses difficult challenges for their implementation into practical applications. Hardware, and FPGA implementations specifically, seem quite promising because of their inherent use of parallelism.

Although any model complexity can be of scientific use, depending on the experiment conducted, the more detailed the models, the closer they are to the biological counterparts and the more behaviors can be studied and replicated. SNN implementations are a natural option for brain simulations. Still, the constraints and issues of incorporating complex models in FPGAs are apparent in many cases. These could include either model choice and network size, system design automation, tool availability and resource constraints.

Table 3.3: *Performance efficiency of works that achieve real-time performance on conductance models.*

Design	**RT Net. Size #**	**Per. Efficiency (MFLOPS)**	**Arithmetic Precision**
Zhang et al. [1]	4	0.0044	Floating Point
Beuler et al. [97]	400	0.44	Floating Point
Sorensen [9]	1	0.001	Fixed Point
Weinstein [121]	40	4.4	Fixed Point
Luo et al. [89]	500	11	Fixed Point
Graas et al. [88]	17	18	Fixed Point
Christiaanse et al. [99]	1,188	19	Floating Point
Yang et al. [91]	160,000	183	Fixed Point
Pourhaj et al. [96]	1,024	189	Fixed Point

Brain simulation demands are more versatile, depending on the designers' simulation and study goals. IaF neurons can create very large networks but the behavioral detail of the neurons is too basic for most studies and probably unusable for brain recovery. These models are useful as a case study and easier to implement in large networks. Yet, such models seem to fall sort for other purposes. As such, there is a significant tendency to begin using more complex models in FPGA implementations, such as Izhikevich and also conductance models. A significant amount of implementations incorporate conductance models, either HH and their simplifications or custom tailored ones to specific characteristics.

In order to make meaningful observations on the real-time performance of conductance SNNs on FPGAs, we can compare the ***performance efficiency*** of designs that achieve real-time performance in terms of MFLOPS per FPGA device. We only take into consideration the main neuron model and not the computation overhead of the connectivity. Although connectivity is an important aspect of the neuron-application computational part, it is not possible to make reasonable estimations of their computational requirements from the data that most works present. On the contrary, we can make a reasonable estimation for the main neuron model based on the FLOPS estimation presented in [6], by using the model time step size. We also assume real-time performance even for designs that provide hyper-real time performance. Even though, theoretically, they can achieve higher than the reported network sizes on real-time performance, there are other aspects of the design that may prevent higher than the reported sizes (memory storage and I/O for example) which is not possible to reasonably

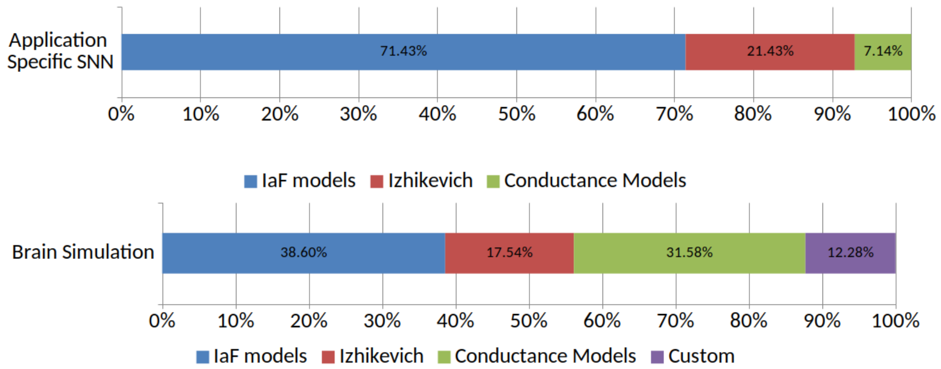


Figure 3.11: *Neuron-model types implemented in each topic.*

estimate in most cases. Looking at the comparison (Table 3.3) we can see FPGAs give promising results in supporting real-time conductance modelling in meaningful sizes.

In Figure 3.11, we can see an overview of SNN-based models chosen for AI applications and brain simulations, within the works presented in this chapter. In contrast to brain simulations, AI applications so far tend to opt for simpler SNNs mostly owing to the complexity of the task entailed. Tuning more complex models seems unnecessary when the task can be tackled with simpler models. Thus we can see a trend to use as simple as possible models in FPGA AI implementations. Most designers are satisfied with using IaF models and a few tried using Izhikevich models that basically have similar resource and computational costs. There is much less interest in using more complex models for specific tasks. In general FPGA implementations seem to be one of the promising platforms for AI use as they offer the necessary compute power and low latency for demanding AI applications.

3.5.4 Architectural choice and their effect on FPGA designs

FPGA-based SNN simulation speeds are drastically affected – perhaps even more than moderate differences in model detail – by (a) the memory architecture, and (b) the overall compute architecture; namely, whether the SNN simulation is implemented following a control-flow or a data-flow paradigm. FPGAs employing BRAMs (on-chip memory) provide a very high speed choice for data storage, thus potentially a significant performance boost. Indeed the vast majority of surveyed FPGA implementations do employ On-chip memory for the performance benefits (Figure 3.12) and, on aver-

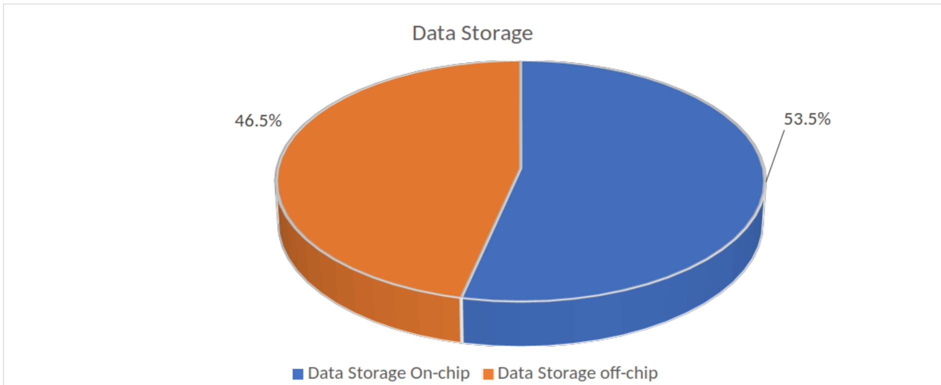


Figure 3.12: *Percentage of FPGA SNN implementations utilizing (only) on-chip and utilizing on-board memory.*

Table 3.4: *Average Performance efficiency of FPGA SNN implementations utilizing (only) on-chip and utilizing on-board memory.*

	on-chip	**on-board**
Per. Efficiency (GFLOPS)	8.38	0.93

age, have higher performance efficiency than designs using on-board memory only (Table 3.4). Even though the comparison is not completely fair (since the FPGAs used in the various works are very varied), it is still an indication of what has been accomplished using either design choices. This is related to the classic memory wall problem in computing [122] in which processing of the computational elements outpace the ability of the memory to transfer the data fast enough to be processed, thus becoming the performance bottleneck. That is why metrics such as arithmetic intensity (section 3.3) are relevant in understanding the HPC potential of such workloads.

BRAM is limited, though, and larger scale network size would require significant data resources. So, using the on-chip memory would provide significant performance benefits but could potentially limit the size of the simulated network. So often there is a network size/performance trade-off when using the BRAM.

The other important choice is that of control-flow vs data-flow architecture. In a control-flow schemes the execution is organized into tasks whose execution is centrally controlled. It leads to elegant designs that can be easily maintainable and extended. The centralized control does, though,

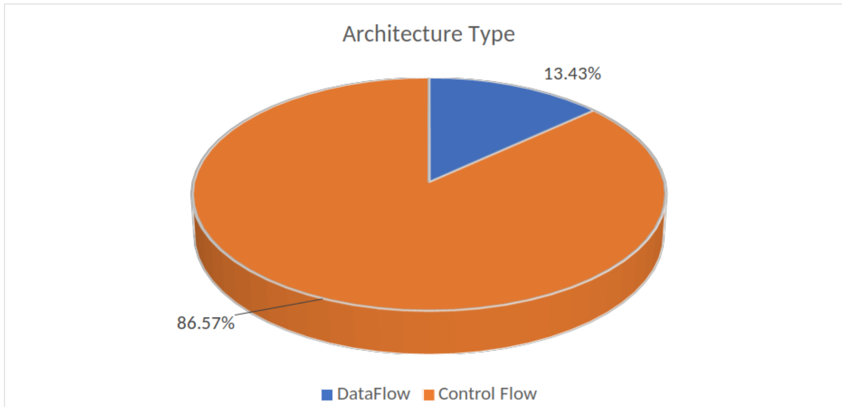


Figure 3.13: *Percetage of FPGA SNN implementations utilizing control-flow and data-flow architecture.*

add an overhead that can affect performance in certain cases. Most prior art tended to use control-flow architectures, as this is the more direct and traditional choice (Figure 3.13). Data-flow schemes, on the other hand, abstract and remove the central control of execution. The computational units just execute their operations the moment data are available in their input without waiting for any centralized control signals. Such a scheme can be beneficial for specific kind of applications, like embarrassingly parallel workloads, which by nature fit into a data-flow scheme. Big parts of the computations in an SNN are actually embarrassingly parallel. Specifically the computation within each neuron within a single simulation step. Thus, a data-flow scheme can be suitable. On the other hand, connectivity, especially computationally complex connections between neurons like gap junctions, can break the data-flow nature of the computations, removing some of the benefits of the data-flow scheme. So the suitability of a data-flow depends a lot on the model of the SNN being implemented and the density and complexity of neuron connections. Nevertheless, the designs that attempted to utilize data-flow architectures managed to present impressive results. Many of them even use data-flow optimized reconfigurable hardware platforms, like the Maxeler Dataflow Engine (DFE) [85, 123].

3.6 Tool-flows

In terms of the tool-flow used for the hardware implementation of the SNN, some trends are also forming. Most engineers seem to initially implement

their models directly on a Hardware Description Language (HDL). Especially for simpler models, this does give great control over the design and can produce efficient results. The most common choice is VHDL (Figure 3.14). The main problem is, of course, that only hardware experts have the knowledge to code directly on an HDL language such as VHDL or Verilog. Additionally conductance models, like HH, are so complex that manually describing them in HDL can be cumbersome even for a hardware expert. As a result, there is a trend to begin using High Level Synthesis (HLS) tools for the implementation of SNNs designs.

Such an alternative that is often used in SNNs is Handel-C. This language describes HW in a sequential fashion, much more natural for non-hardware designers. A Handel-C description is afterwards easily converted to an HDL synthesizable code. Many researchers (especially in the non-computer engineering realm) are more familiar with MATLAB code and so there is also the trend of using MATLAB as the initial code and using automated tools (such as the Xilinx System Generator) for HDL descriptions. Another reason is that tools like Simulink can be easily used for validation and data-process purposes. A more recent option used by Cheung et al. [85] is the Maxeler Dataflow Machines that include their own java-based HLS language (MaxJ). Although HLS greatly automates the procedure, generic automated tools, not specifically developed for such designs, may not always have the best possible results. Additionally even with HLS tools, a good knowledge of hardware design is required, if the capabilities of FPGAs are to be fully exploited.

The lack of an automated tool for hardware neuron modeling, which does not required hardware expertise to efficiently use, is apparent. Several works have attempted to not just create a one-off SNN implementation but make a more general framework for SNN FPGA design to remove the difficulty of designing with HDL/HLS languages from scratch [124], [125], [126], [127], [128], [129].

All these works have tried to create a complete design framework only for very simple neuron models. Weistein and Lee studied the possibility of creating an architectural framework for more complex conductance models to emulate biological behavior accurately. They begin with a manual FPGA implementation of a 10-compartment conductance motorneuron model [130]. Each neuron channel was implemented in HW and a 10-stage pipeline provided a throughput of 1 compartment state per cycle. The model was deployed on a Virtex II and run almost 4 times faster than real-time and 12 times faster than the software simulation.

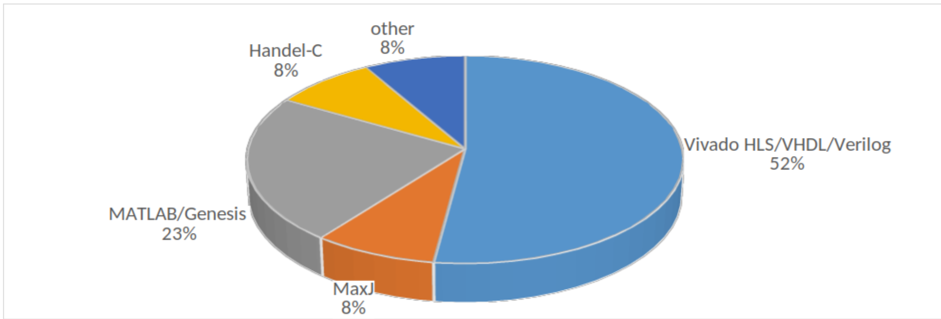


Figure 3.14: *Initial code description environment for HDL generation for SNN FPGA designs.*

Using the experience of their first design, researchers later developed an architectural framework for SNN FPGA modeling of a Fitz-Nagumo neuron [131]. In this research 3 architectures were developed. One of a single neuron model (including both single and multi-cycled versions), one simulating 10 unconnected neurons exploiting pipelining/time-multiplexing and one with 10 interconnected (coupled) neuron models. The architectures were based on standardized IP hardware modules provided by the Xilinx tools. Using this as a basis, a semi-automatic strategy was later presented for implementing neural networks and a pre-Bötzinger complex (PBC) simulation was designed as an example [132]. The model included 3 compartments having HH type conductances. The semi-automatic tool flow begins with MATLAB code that is translated into VHDL, and using the same HW libraries as before, produces the design to be deployed on reconfigurable hardware. The model parameters/inputs of the system were stored on shared RAMs and the design communicated with the Host PC for on-line configuration purposes. The network output was captured and displayed using DACs. Four fully interconnected NNs of various sizes were implemented as a proof of concept, the largest one comprised of 40 HH neurons running 8.7 times faster than real-time.

The results of this previous research led to the creation of the DYNAMO modeling language and compiler that implemented a fully automated design framework for SNN modeling on FPGAs [121] . The Dynamo Modeling Language (DML) is a general mathematical modeling language, although developed for use in neural models. The execution flow is organized in a loop and can be translated to C (and from C to JAVA), MATLAB and VHDL code, either for simulation or HW implementation purposes. The

compiler included an application scheduler and floating-to-fixed-point conversion for the FPGA modeling. Model parameters in the produced systems were stored in shared RAMs and constantly feed in the models by registers. Input/Output was given in the FPGA through FIFO buffers. A number of SNNs using conductance models were presented as proof of concept, all of them running at higher than real-time performance. The largest HH network presented was a fully interconnected, 10-neuron neural network, deployed on a Virtex 4 device.

The DYNAMO compiler creates a fully automated process beginning with a custom generic modeling language that could potentially model every complexity. A mathematical modeling language, like the one developed under the DYNAMO compiler, is much more comprehensible to a variety of other fields, including neuroscience, than HDL languages are. Although DYNAMO did bridge the gap between modelling definition and HDL implementation, one drawback it had was precision problems stemming from the fixed-point conversion, although the compiler included numerous optimization techniques to reduce them. A final drawback of the design was also that DML was relatively complex to use and still unfamiliar to the neuroscientific field.

Experience from efforts such as DYNAMO (which had very limited adoption even though functionally sound) makes it quite obvious that a framework that attempts to create a more general tool for neural networks used in Computational Neuroscience requires to use pre-existing frameworks that scientists are already familiar with and validated by their community. Additionally using a front-end that neuroscientists already use seem essential for community adoption. Most modelers use Matlab, NEURON [26] and GENESIS [25] and there is an extensive library of models in these descriptions that any complete tool-flow targeting FPGAs should be able to use, in order to be immediately useful for the scientific community. That is the gap that NeuroFlow [123] is trying to fill by using a PyNN front end, translating the source into MaxJ and conducting HDL generation for the Maxeler DFE hardware.

Proposed by Cheung et al. NeuroFlow [123] integrates PyNN (an established python-based neuroscientific development framework) to their Maxeler Dataflow Machine hardware library. Neuroflow also provides a very complete library of IPs in the back-end, covering a great portion of possible applications, thus providing FPGA-based high performance with user friendliness. It supports HH, IaF and Izhikevich neurons with a number of different synapse implementations. Performance results are only presented for the Izhikevich use case supporting almost 100K neurons per

device. It supports multi-DFE designs providing significant scalability to Neuroflow. The performance and efficiency analysis is only presented for a single use case of a generally simpler model (Izhikevich) and with connectivity modelling of medium complexity (STDP) and relatively lower neuron inter-connectivity density (about 10%). The behaviour and performance of the system for the rest of the supported features is not self-evident and is expected to be significantly different, especially for accurate modelling such as the HH with high connectivity densities, for example. Furthermore, many of the performance benefits are accomplished using event-driven simulations (neurons are evaluated only when their inputs are triggered), that cannot always be employed with complex conductance models.

3.7 Summary

In this chapter an extensive review of the modeling aspects of Spiking Neural Networks was presented. The review reveals the high diversity of model and subsequently produced workloads this field includes. Workload behavior is also decisively affected by early model decisions that complicated their execution in HPC resources even more. Modeling aspects like the model numerical methods and time-step sizes can affected HPC execution as much as hardware implementation design decisions, like memory organization.

Advances in the development of FPGAs make them a natural design option for practical SNNs, especially for smaller and more computationally complex neuron simulations. Looking at prior FPGA-based implementations, we can see that FPGAs can in practice provide good performance for many A.I. applications of SNNs. Additionally such designs can exploit their modularity, speeding up prototyping and making on-field system updates possible. In terms of SNNs for brain simulations and recovery, FPGAs also show promise as a design paradigm. The potential of real-time performance of complex models on FPGAs, in network sizes not possible before, could have an important impact in brain research. The area and memory constraints that are present in FPGAs should be relaxing as newer and better devices emerge. The real-time performance potential also makes brain implantable devices prototyped on FPGAs a possibility, although the power consumption/budget issue would need to be also solved if actual FPGAs are to be used on implants. FPGAs can also provide reliability; the regular and re-configurable nature of the device can also be exploited to provide fault-tolerance.

What is severely missing in the field, though, is a process to streamline and automate the design procedure in a way that the engineering work required for the FPGA acceleration is transparent to the neuroscientist. Any

work that attempted to provide user friendliness with more typical ways (like GUIs) and custom-made modelling languages had limited scope and success. Additionally, developing these systems manually directly using HDL descriptions requires great design effort; an automated system, optimized for the specific domain, would be an extremely useful tool. Finally, bridging the gap between HDL descriptions and the vast verified work on traditional neuroscietific environments (GENESIS, NEURON, PyNN) is a relevant goal to be pursued as very few prior published works have tried to tackle the problem holistically.

CHAPTER 4

The Inferior Olive on FPGA-based Hardware

- **"FPGA-based Biophysically-Meaningful Modeling of Olivocerebellar Neurons"**, **Georgios Smaragdos**, Sebastian Isaza, Martijn Van Eijk, Ioannis Sourdis and Christos Strydis, 22nd ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), Monterey, California, February, 2014.
- **"Real-Time Olivary Neuron Simulations on Dataflow Computing Machines"**, **Georgios Smaragdos**, Craig Davies, Christos Strydis, Ioannis Sourdis, Catalin Ciobanu, Oskar Mencer and Chris I. De Zeeuw, Int'l Supercomputing Conference (ISC) - Also Appearing in LNCS 8488 proceedings, Leipzig, Germany, June 2014.
- **"A Real-Time Reconfigurable Multichip Architecture for Large-Scale Biophysically Accurate Neuron Simulation"**, Amir Zjajo, Jaco Hofmann, Gerrit Jan Christiaanse, Martijn F. van Eijk, **Georgios Smaragdos**, Christos Strydis, Alexander de Graaf, Carlo Galuzzi, Rene van Leuken in IEEE Trans. Biomed. Circuits and Systems, Volume: 12, Issue:2 , April 2018.
- **"Prototyping a Biologically Plausible Neuron Model on a Heterogeneous CPU-FPGA Board"**, Kaleb Alfaro-Badilla, Alfonso Chacón-Rodríguez, **Georgios Smaragdos**, Christos Strydis, Andrés Arroyo-Romero, Javier Espinoza-González, Carlos Salazar-García in IEEE 10th Latin American Symposium on Circuits and Systems (LASCAS), February 2019.
- **"DeSyRe: On-Demand Adaptive and Reconfigurable Fault-Tolerant SoCs"**, I. Sourdis, C. Strydis, A. Armato, C.S. Bouganis, B. Falsafi, G.N. Gaydadjiev, S. Isaza, A. Malek, R. Mariani, D.K. Pradhan, G. Rauwerda, R.M. Seepers, R.A. Shafik, **G. Smaragdos**, D. Theodoropoulos, S. Tzilis, M. Vavouras, S. Pagliarini, and D. Pnevmatikatos, 10th Int'l Symp. on Applied Reconfigurable Computing (ARC), Vilamoura, Algarve, Portugal, LNCS 8405, pp. 312–317, 2014.
- **"ESL Design of Customizable Real-Time Neuron Networks"**, Martijn van Eijk , Carlo Galuzzi , Amir Zjajo , **Georgios Smaragdos**, Christos Strydis and Rene van Leuken, IEEE Biomedical Circuits and Systems Conference (BIOCAS 2014).

In this chapter the initial effort of accelerating the Inferior Olive (IO) model that existed in-house within Erasmus MC, a state-of-the-art model accelerated as a first step towards building a high-performance olivocerebellar simulation platform is presented. The choice to focus the effort on FPGA-accelerated technology was made using the conclusions from looking at the prior art of the previous chapter. One extra constraint was the use of single-precision floating-point (FP) arithmetic computations. Biophysically meaningful models such as the IO model are quite stiff, meaning that the use of fixed-point arithmetic could severely affect functional correctness. The complexity and stiffness of the IO model would allow for the use fixed-point computation only after a substantial arithmetic analysis that moved way beyond the scope of the acceleration effort, making it a non-viable option to pursue at the time. In short, the IO FPGA acceleration had these immediate goals:

- The analysis of the Inferior Olive model to recognize characteristics and possible performance bottlenecks.
- The optimization of the original model algorithm achieving performance improvements in hardware.
- The design, implementation and validation of the FPGA-based accelerator, achieving significant speed-up and real-time performance.
- The performance evaluation of the designed accelerator and evaluation of precision error, issued by design optimizations, to guarantee preservation of the biological behavior reflected in the original model.

4.1 Application description

The IO model not only divides the cells in multiple compartments but also creates a network in which neurons are interconnected. We start by describing the single-cell model, while the rest of the section deals with the cell-network model.

4.1.1 The IO-cell model

The IO cell model we have implemented has been originally developed by de Gruijl et al. [22]. The model contains active conductances based on the original HH-based model [36] and divides the neuron into three computational compartments – closely resembling their biological counterparts – as shown in Figure 4.1(b). For every compartment, a few ion channels are present in the model so as to contribute to the total compartment

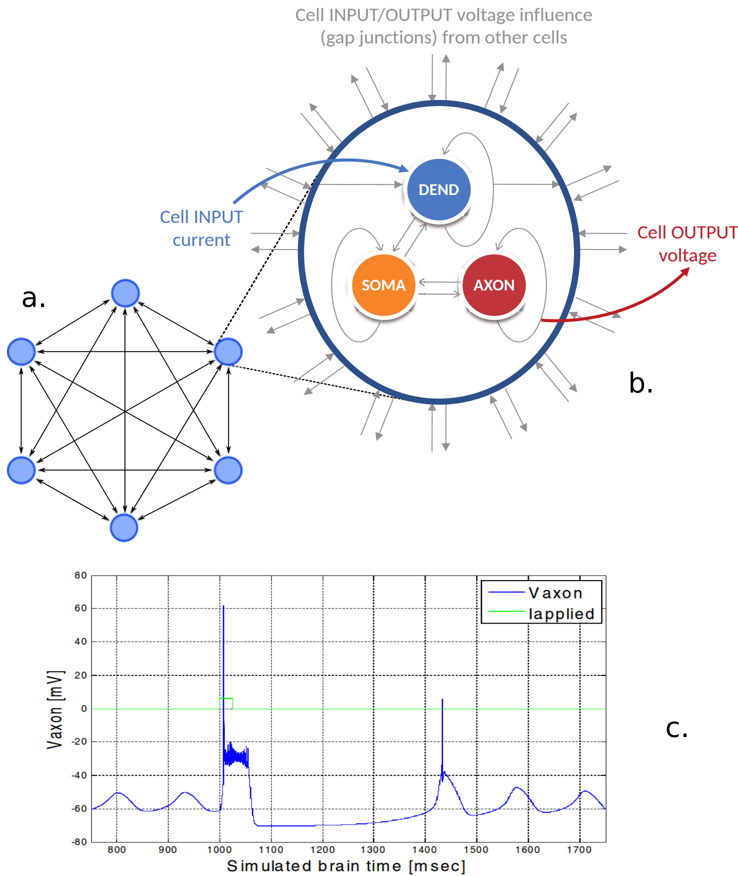


Figure 4.1: *Graphical representation of the inferior-olivary network model. a) 6-neuron network b) single-neuron model in detail c) sample axon response.*

membrane potential. Every compartment has a state that holds the electrochemical variables and, on every simulation step, the state is updated based on: i) the previous state, ii) the other compartments' previous state, iii) the other neuron's previous state and iv) the externally evoked input.

The computational model operates in a fashion that allows ***concurrent*** execution of the three compartments. The model is calibrated to produce every per-neuron output value with a 50 μsec time step. This means that, in order to support real-time simulations, all neurons are required to compute one iteration of compartmental calculations within 50 μsec.

4.1.2 The IO-network model

The IO network interconnectivity is implemented with Gap Junction (GJ) modelling based on the implementation of [133]. The GJs are associated with important aspects of cell behavior as they are not just simple connections; rather, they involve significant and intricate electrical processes, which is reflected in their modeling details [50].

Figure 4.1a depicts a representation of the IO network model. The GJs are part of the dendritic compartment, thus the compartment receives the extra input coming from the inter-neuron connection. This network model defines effectively a transient simulator through computing discrete output axon values in 50 μsec time steps which, when integrated in time, recreate the output response of the axon (Figure 4.1c). The IO network must be synchronized in order to guarantee the correct exchange of previous dendritic data within a step. Thus, the execution can only be parallelized in space (simultaneous evaluation of neurons within a simulation step), but not in time (parallelization of multiple simulation steps). The cells – even when not actively spiking – present an oscillatory behavior, thus affecting network synchronization. As a result, event-driven execution of the network model is not an option.

4.1.3 C-code profiling

The IO-network model was initially available to us in Matlab but was re-written in C for profiling purposes and so as to be used in High-Level Synthesis (HLS) tools (Vivado 2013.2 targeting an Virtex 7 XC7VX485T device).

By profiling the application using an operation and memory-access profiler [134], it is revealed that the GJs have great impact on the total model complexity. As seen in Table 4.1, the total number of floating-point (FP) operations needed for simulating a single step of a single cell including a single GJ are 871. In an N-cell network, assuming that each neuron maintains a constant number of connections C to neighboring cells, the overall GJ computation cost exhibits linear complexity: $\mathcal{O}_{gj}(N)$. For many complex experiments, it is not the number of connections C but, rather, the ***connectivity density*** that is indicative of neuron interconnectivity. That is, the ***average percentage of the total neuron population to which neuron cells are connected*** (measured in %), whereby the complexity becomes quadratic: $\mathcal{O}_{gj}(N^2)$. This makes GJ computations the prevalent contributor, as they break the dataflow nature of the application and dominate computational demands. This is true even for small-scale networks.

Table 4.1: *Neuron compute requirements per simulation step.*

Computation	**FP Operations per neuron**
Gap Junction	12 per connection
Cell Compartment	859
I/O and storage	**FP values per neuron**
Neuron States	19
Evoked Input	1
Connectivity Vector	1 per connection
Neuron Conductances	20
Axon Output	1 (Axon Voltage)
Neuron Computation Task	**% of FP ops for 96 cells**
Compartmental Computations	43
Gap Junctions	57

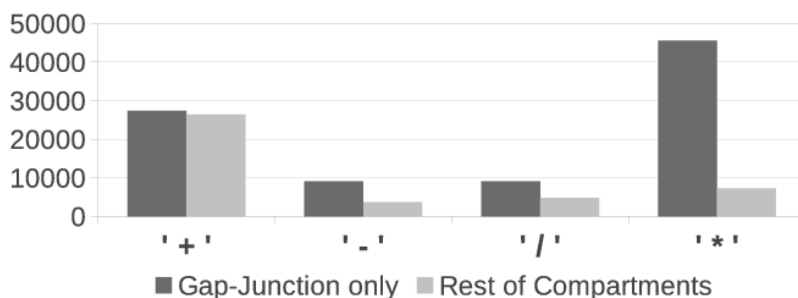


Figure 4.2: *Software profiling of the arithmetic operations in the model for a 96-cell, fully interconnected network.*

As an example, for a 96-cell, all-to-all connected network (Table 4.1) the GJs comprise almost 60% of the overall computations. Figure 4.2 shows the profiling of the arithmetic operations performed in the model, for one iteration of the whole IO-network. We differentiate the operations belonging to the gap-junction compartment from the rest again, to show its importance. Results show that (FP) multiplications performed in the gap-junction compartment dominate the distribution. Finally, we can see that the gap junctions contain the largest fraction of operations for all operation types and will, therefore, consume more FPGA resources.

4.2 HLS FPGA-based inferior olive implementation

After profiling the C code based on the Matlab model, we used it as the basis for generating the proposed hardware solution using the Vivado HLS tool. The resulting hardware accelerator simulates the behavior of multiple IO-cells ***step by step*** based on the aforementioned model. The hardware accelerator is designed to work alongside a softcore or host CPU that controls the total number of simulation steps and handles the I/O of the accelerator. The CPU feeds the accelerator with initialization data (initial state) and with evoked-current inputs (external stimuli of neurons) and outputs the result of the computations at every simulation step. Output data can be stored in on-board memory (e.g., SD cards) or sent to an off-board PC host.

Both, neuron states and evoked inputs – required at every simulation step – are stored in on-chip BRAMs, so as to avoid incurring off-chip latency. The performance benefit of using on-chip storage is substantial compared to going off-chip, especially for complex models such as ours, which require handling large amounts of data to represent the network state. On the other hand, this creates a constraint on the maximum network size that can be simulated, which depends on the storage capacity of the FPGA BRAMs.

The remainder of this section offers the details of our FPGA-based approach and the optimizations performed to improve the performance and area efficiency of our design.

4.2.1 Overview of the hardware design

The general block diagram of the proposed system can be seen in Figure 4.3. The actual execution is performed at the "IO Network" component, which consists of multiple identical parallel neuron-processing modules, each modeling the dendrite, soma and axon parts of a single IO cell. Our design further includes a set of BRAMs for storing the evoked inputs to the neurons as well as their state, which is updated after each simulation step. The execution flow of the IO network is controlled by a – local to the accelerator – kernel control unit. Our actual implementation of the IO network consists of eight hardware neuron-processing modules, which are able to simulate eight IO-cells in parallel.

The accelerator was designed to give run-time control over a number of simulation parameters, providing flexibility and the ability for more com-

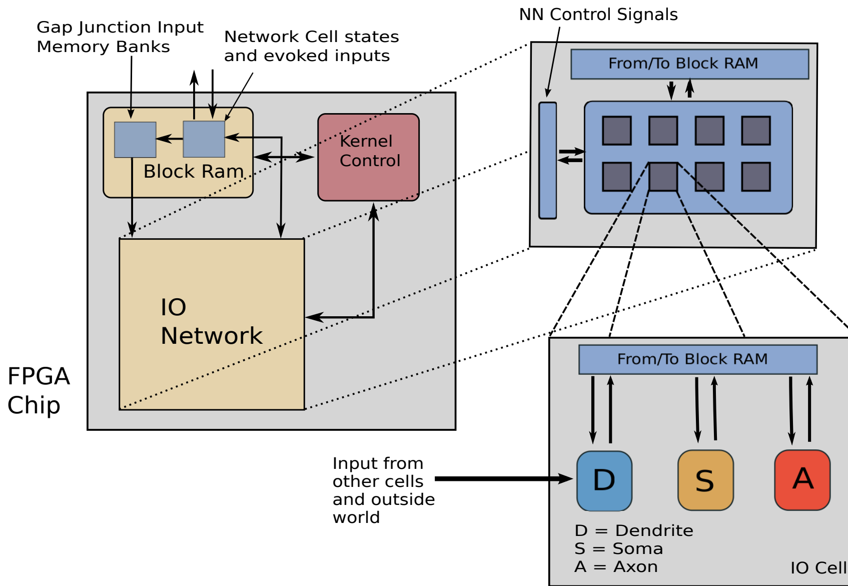


Figure 4.3: *Block diagram of the Olivocerebellar neuromodeling hardware design.*

plex experiments. During execution, each neuron state parameter can be modified. Interconnectivity density is also adjustable during simulation.

Next, we describe the functionality of our FPGA-based accelerator. First, the neurons in the network are initialized with data streamed from the CPU to the FPGA. The initialization data are either produced by the CPU itself or read by on or off-board resources. This introduces a delay which is however paid only once at the IO-network simulation onset.

After initialization, the actual execution of the network simulation is performed. Each simulation step begins with storing new evoked inputs of the neurons in BRAM, representing the network external input vector. Following the storing of this vector, the kernel-control copies to dedicated BRAM banks part of the other cells' state (the dendritic voltages) needed for computing the gap-junction effect. Each hardware neuron-processing module has a separate dual-port, BRAM bank to store its respective gap-junction data. By making this design choice, we improve the memory bandwidth during the gap-junction processing and allow the HLS-tool scheduling techniques to maximize parallelism. This would not be possible if both compartment and gap-junction logic shared the same memory banks.

With all input data ready, the next state of each neuron is computed. Each hardware neuron-processing module executes in parallel. It is worth noting that all three compartments (dendrite, soma, axon) within a neu-

ron module could execute in parallel, as they have no dependencies with each other. In practice, the axon and soma execute sequentially (soma first, axon second) to save on resources, while the dendrite compartment executes concurrently with the axon and the soma. That is due to the execution time of the dendrite which is longer than that of the axon and soma compartments combined. The final phase of the execution involves storing the newly produced IO-cell states in the BRAMs, to be used in the next simulation step, and streaming the output values needed by the experiment outside the accelerator. For our test cases, this output is the axon voltage of each neuron for every simulation step, representing the IO-network response. Presumably, the output could be any part of the neuron states, depending on the requirements of the neuroscientific experiment.

4.2.2 Time-multiplexing execution

For the accelerator to achieve real-time performance, each simulation step must be completed within the same time window of 50 μsec. Obviously, such a "real-time" constraint does not have a counterpart in biological neurons (as they operate in continuous fashion). It is imposed by our IO network simulator which is a self-contained, fixed-timestep, transient simulator – similar to most HH-based simulators – with a constant step $\Delta t = 50\mu sec$ in our case. Respecting this time-step duration is essential for generating biologically-plausible signals that can be interfaced to living tissue.

Of course, our hardware neuron network (and hardware modules in general) runs significantly faster than the real-time constraint at hand. We exploit this ***latency slack*** by using our hardware resources more efficiently and maximizing the number of simulated neurons by ***time-multiplexing*** of hardware blocks. More precisely, we use the same hardware neuron-processing module multiple times within a simulation step to compute states of different simulated neurons. As illustrated in Figure 4.4, each hardware neuron-processing module evaluates multiple simulated neurons that together comprise the total simulated network. By adjusting online the number of simulated cells each hardware neuron is simulating (i.e. the time-multiplexing factor), the network size can be altered without re-synthesizing the hardware kernel, even during the simulation, if experiments indeed require it (for instance, to emulate synaptic plasticity). This is achieved by storing different input vectors and cell states for each simulated neuron evaluated in each hardware neuron-processing module. However, the input vectors and cell states need to be stored in the BRAM; this ultimately means that the maximum network size shall be ***constrained by***

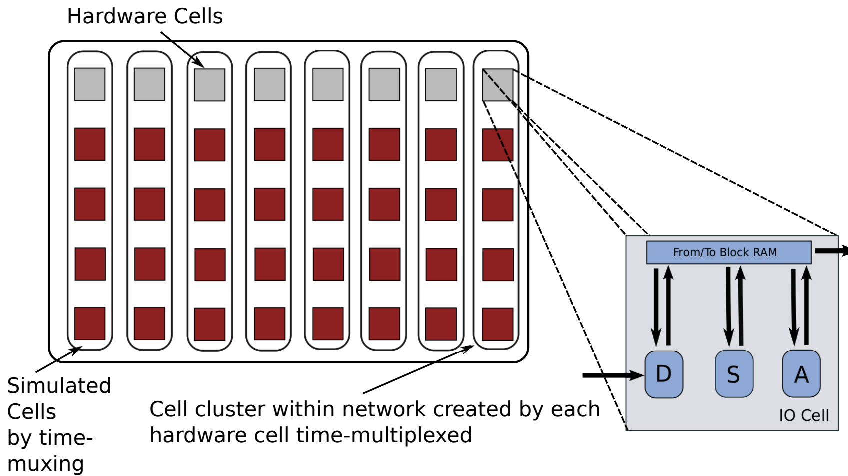


Figure 4.4: *Time multiplexing of hardware neurons.*

the amount of available of on-chip memory. The BRAMs are statically allocated before synthesis to support the maximum number of possible simulated cells at runtime.

4.2.3 HLS C-Code optimizations

A number of optimizations for increasing the efficiency and performance of the hardware design were implemented in the C code, motivated by code inspection and the profiling information presented in Section 4.1.3. According to profiling results of the reference C code, the most computationally intensive compartment in the model is the dendrite, more specifically, the gap-junction computations. These are responsible for accumulating the influence of all other neurons in the network and include complex arithmetic operations such as FP exponents and divisions performed for every other cell state, as shown in Listing 4.1. In such an all-to-all interconnected network, the amount of gap-junction computations increases ***quadratically*** with the network size.

Without changing the actual functionality described in Listing 4.1, we rewrote and simplified the gap-junction code. As shown in Listing 4.2, we removed from the for-loop any operations that are common for all iterations, thus reducing the required computations substantially. In short, we removed computations simulating the total gap-junction influence (Ic) from the accumulation loop, saving three multiplications and one addition per for-loop iteration. In the optimized code, the gap junctions accumulate only the input parameters of Ic and compute the total influence only once, after the accumulation has been completed. This modification yielded a

notable increase in the network size supported by our design for real-time simulations.

Listing 4.1: *Original gap-junction code [50].*

```
for (i=0; i<IO_N_INPUT; i++) {
  V = prevVdend - neighVdend[i];
  f = 0.8 * exp(-1 * V * V/100) + 0.2;
  Ic = Ic + (CONDUCTANCE * f * V);
}

return Ic ;
```

Listing 4.2: *Optimized gap-junction code.*

```
for (i=0; i<IO_N_INPUT; i++) {
  V = prevVdend - neighVdend[i];
  f_new = V * exp(-1 * V * V/100);
  F_acc =+ f_new;
  V_acc =+ V;
}

Ic = CONDUCTANCE * (0.8*F_acc + 0.2*V_acc); return Ic ;
```

A second modification in the original code that helped increase both performance and area efficiency was the replacement of any division-by-constant with an arithmetic equivalent (but less computationally intensive) multiplication-by-constant (e.g. $\frac{A}{100} \Leftrightarrow A * 0.01$). In computer arithmetic, the above modifications can introduce precision error in the computations performed; in the evaluation section we measure the effect of our optimization in the quality of the simulations. As shown next, the IO-model computations have a large number of divisions-by-constant operations, the replacement of which can influence both area and performance without introducing a significant precision error that would affect correct model behavior. This optimization had to be performed manually as the HLS tool does not support it automatically so as to avoid introducing potential precision error without the developer's consent.

In Table 4.2, we can see the performance and area benefits for the application, for each code modification. Opt1 denotes the gap-junction code modifications. The other two optimizations refer to the replacement of divisions-by-constant with multiplications-by-constant. Opt 2 replaces divisions only in the slowest part of the model (dendrite compartment), while Opt3 in the entire code. We initially attempted replacing the divisions only in the dendrite, since our main concern is performance while making sure that the arithmetic error would not be significant. As Opt1 and Opt2 had only favored the dendrite/gap-junction compartments, Opt3 was eventually

Design	Area	Real-Time Network Size	One Cell Latency
Baseline	99% of LUTs	48 cells	603 cycles
Opt1	99% of LUTs	84 cells	347 cycles
Opt2	96% of LUTs	96 cells	333 cycles
Opt3	91% of LUTs	96 cells	323 cycles

Table 4.2: *Synthesis Estimation for each optimization case with Vivado HLS 2013.2 for a Virtex 707 evaluation board. Opt1: Gap-junction calculations' optimizations. Opt2: Division-by-constant replacement in dendritic compartment. Opt3: Division-by-constant replacement in all 3 compartments. LUT: look up tables.*

also deemed useful as the balance changed and it lead to an extra performance benefit.

Overall, these modifications achieved an almost 50% decrease in single-neuron execution latency, doubling the maximum network size able to be simulated at real-time speed. There is also some area improvement which is not substantial due to the fact that both multiplications and divisions use – in most cases – the same number of DSP slices.

4.3 Evaluation of the Vivado HLS implementation

We evaluate, next, the performance and area cost of our proposed approach and measure its speedup compared to a software implementation. Moreover, we estimate the precision error after our modifications and, finally, discuss the efficiency of our approach compared to other related works.

4.3.1 Experimental methodology

The development of the Inferior-Olive design, as previously mentioned, was performed using the Xilinx Vivado High Level Synthesis Tool (HLS v2013.2). The tool gives the ability to describe hardware IPs using a subset of ANSI C and then automatically handles production of the IP control logic, hardware scheduling of the operations and translation of the described design in SystemC, VHDL or Verilog code. Vivado HLS also supports algorithm validation using the C code, as well as integration with RTL simulators for validation of the produced HDL code. The tool actually provides the RTL simulation with the correct input vector according to C test-benches. This allows for explicit RTL hardware validation with testbenches simulating the complete CPU/IP system operation.

Design	Speed-up
C Code – Double Floats	×58.64
C Code – Single Floats	×60.82
FPGA Accelerator	×731.23

Table 4.3: *Speed-up of C implementations and the FPGA-based accelerator compared to original Matlab code for simulating a real-time network of 96 neurons.*

The IO-network design was translated to VHDL code using HLS and validated using QuestaSim 10.1 in RTL. Our testbench highlighted the basic behavior of the IO-model. All neurons are initialized with identical states, and left without any outside stimuli, remain synchronized with their axon voltage values oscillating. After 20,000 simulation steps, evoked current signals are issued to all neurons for 500 simulation steps. The IO neurons respond to these stimuli by producing a complex spike as seen in Figure 4.8(a) before returning back to their oscillating steady state. The testbench simulates 6 seconds of real brain time, taking 120,000 simulation steps to complete.

4.3.2 Experimental results

The accelerator achieves ***real-time execution*** for a 96-neuron network with 100% (full) interconnection ratio at an operating frequency of 100 MHz[1] using a Virtex 7 XC7VX485T FPGA. In Table 4.3 we can see a performance comparison of the C code and the hardware accelerator against the original Matlab implementation; both the C-code and Matlab model run on a Xeon 2.66GHz machine with 20GB RAM. The double-FP C implementation is about ×58 faster that Matlab, while the use of single-FP arithmetic gives a speedup of almost ×61. The FPGA IO-network kernel achieves an impressive ×731 speedup compared to the Matlab version and ×12.5 compared to the C implementation.

The on-chip memory (BRAM) resources available allow for ***maximally*** simulating a 14,440-cell network (non-real-time). Figure 4.5 plots the execution time of our designs for different network sizes. It can be observed that the execution time scales with the network size slightly worse than linearly due to the gap-junction computations which increase quadratically with the network size for an all-to-all interconnected network. However, this is still significantly better than execution-time trends in software. This point is better illustrated in Figure 4.6 which plots the FPGA-based

[1]The operating frequency is limited by the Xilinx IP blocks used in the design.

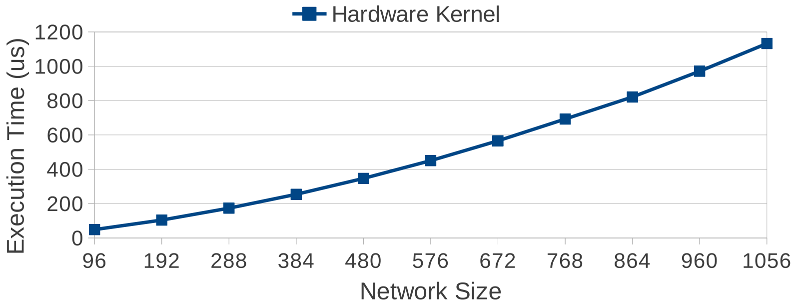


Figure 4.5: *Accelerator step execution time for different network sizes.*

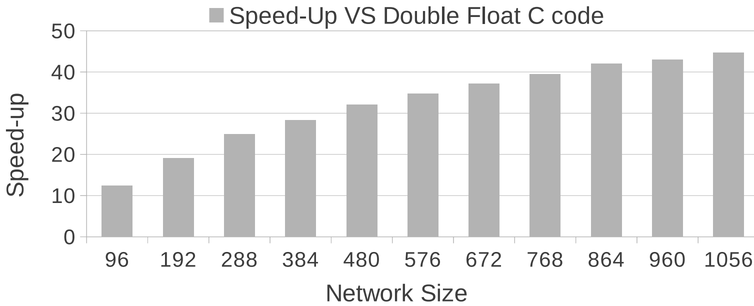


Figure 4.6: *Accelerator performance comparison to double-FP C implementation.*

speedup compared to the double-FP C implementation. As the network size increases above 96 cells, our FPGA-based simulation becomes slower than real-time, however it achieves an increasingly better speedup compared to the C implementation. This shows that the increasing gap-junction computations scale more gracefully in our parallel FPGA-based solution than in software. For a network of 96 cells, the speedup is about $\times 12.5$ compared to the C code implementation and goes up to $\times 45$ for a network size of 1,056 neurons.

Finally, the initialization delay also increases for higher network sizes, but in a linear fashion (Figure 4.7). It reaches a little over 100 μsec for a 1,056-neuron simulation. It should be noted that this time becomes proportionally smaller and even negligible for longer simulation times. Naturally, it also represents the time penalty incurred for re-initializing the cell-states at runtime.

Place-&-Route area results are retrieved using Vivado IDE 2013.2 (Table 4.4). Our accelerator has been designed to utilize the maximum of the FPGA resources; in practice, it uses 83% of available LUT logic, 78% of

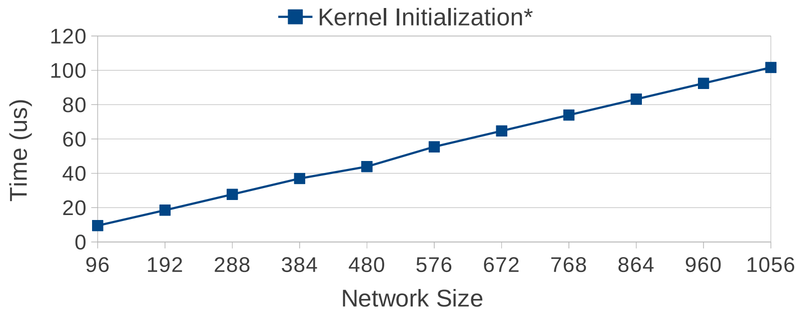


Figure 4.7: *Initialization delay for different network sizes.*

Area Component	Utilization	% of Available
LUTs	251485	83%
BRAMs	804	78%
FF	162217	27%
DSPs	1600	57%

Table 4.4: *Area utilization for the Virtex 707 evaluation board.*

BRAMs, 27% of Flip-Flops and 57% of the available DSPs on the FPGA chip.

4.3.3 Error estimation

As previously mentioned, the original IO-network model performed all computations with double-FP precision. The main reason was that its modelers (as so many peers in the neuromodeling field) have arbitrarily opted for double-FP precision since this is the highest intrinsically supported precision in many modern programming languages (here: Matlab). However, early in our design effort, we realized that double-FP precision would tax the FPGA with such high performance and area costs that no significant acceleration of the application could be achieved. We, therefore, resorted to switching to single-FP precision calculations for the hardware version of the IO-network model.

To make such a decision, we had to first make sure that single-FP precision would be sufficient for the application at hand. Due to the fact that correct, "reference" brain-simulation traces do not exist (in fact, this is one of the goals of model simulators like the one we are porting in this work), only empirical metrics of correctness can be given by neuroscientists at the moment. That is, over a practically infinite amount of simulation time – for instance 1 day of real-time brain simulation (amounting to approx. 1.73

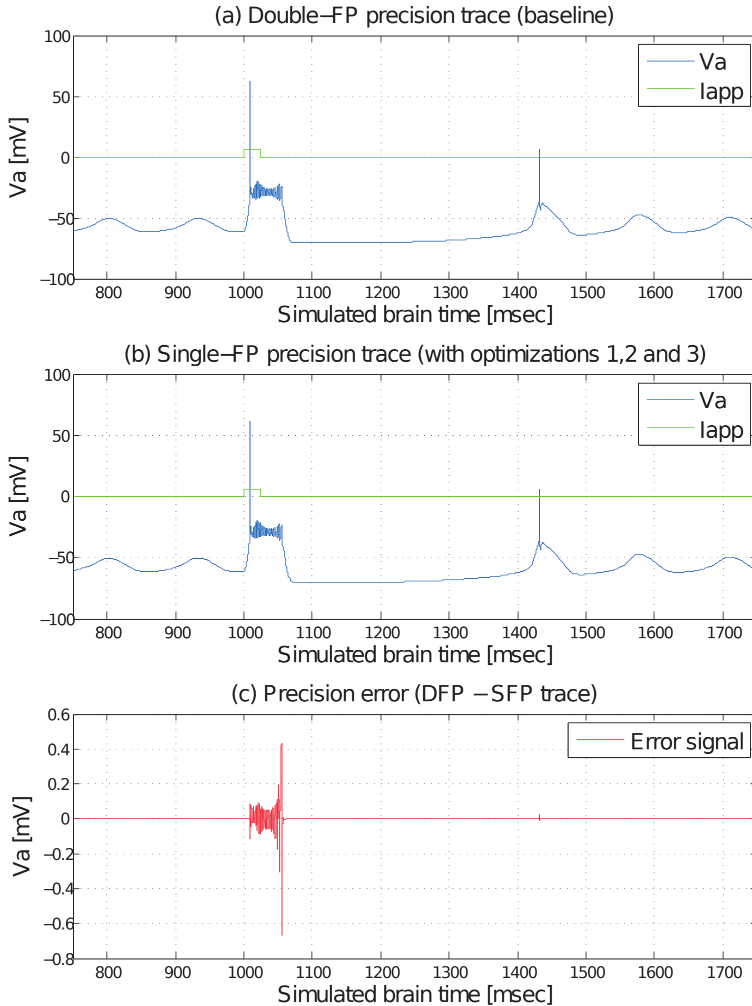


Figure 4.8: *Graphical comparison of numerical-precision error. Externally evoked input current (Iapp) in green, axonal voltage in blue and error signal in red (Va). (a) Reference trace in double-FP precision. (b) The same trace generated with single-FP precision and all three code optimizations. (c) The error signal (i.e. difference) between the two traces. Observe the amplitude units of the error.*

billion simulation steps) – the double-FP and single-FP simulation traces should not exhibit any biophysically different results.

Multiple tests have been run. As a simple illustration, in Figure 4.8 both the dynamic (complex spike) and steady-state (subthreshold oscillations) behavior of a single IO cell between 700 and 2000 msec of a simulation trace

have been captured. Figures 4.8(a) and (b) illustrate runs with double-FP and single-FP precision, respectively. In the single-FP case are also included the 3 code optimizations discussed in Section 4.2.3 which contribute an additional precision error. In Figure 4.8(c), the error signal (i.e. difference of the two signals) is plotted over the same simulation period. Analysis of the error reveals that there is ***no phase error***. A ***very low amplitude error*** is observed which ranges from 0.0%, at cell resting state (when most internal cell variables change slowly), to about 2.1%, at cell firing state. Such a low error signal does not affect the simulator functionality, especially since the model itself cannot guarantee such high accuracy to the real biological system. In conclusion, computations in single-FP precision along with the 3 performed optimizations are considered to ***not*** compromise the IO-network simulation correctness and are permanently adopted. It is, of course, conceivable that a more constrained numerical range could also be used (i.e. fixed-point precision), but extensive precision analysis of the mathematical model would be required to identify such an (integer) range with certainty.

4.3.4 Comparison to related work

We discuss next the efficiency of our design and related SNN FPGA-based approaches and attempt to analyze and compare them. A direct comparison is not possible as different works consider different neuron-models with radically different characteristics, which potentially change completely the requirements of each design. Moreover, despite its complexity, each model type has its own merits for neuroscience and potential usefulness in applications. Depending on application constraints or the subject of simulation experiments different models can be of use.

Table 4.5 summarizes a few of the best performing related FPGA-based brain-modeling works. The Izhikevich neuron models, as expected with their simplicity, allow for the implementation of very large network sizes in FPGA devices, as described in [85], achieving sizes already significant to actual brain subsystems (tens of thousands of neurons). Each Izhikevich neuron model requires 13 operations per 1ms. On the other hand, biophysically-meaningful models such as the HH models used in [1], are one to two orders of magnitude more costly in terms of operations, which is one of the main reasons why real-time network sizes achievable in such designs are much smaller. Another interesting observation is that HH models have a much shorter simulation timestep (tens of μseconds) compared to Izhikevich models (1 msec), which increases their complexity and their accuracy. The IO-model used in the present work has a simulation step of

Design	[19] [85]	[1]	IO Design
Model	Izhikevich	HH	Extended HH
Time Step (ms)	1	-	0.05
Real-Time Network Size	64000	4	96
Arithmetic Precision	Fixed Point	Floating Point	Floating Point
Operations per Neuron in 1ms	>13	<1200	22200
Neuron Model OPs * Net. Size (MFLOPS)	>832	4.8	2131.2
Interconnectivity Density	1.5% (1000 per neuron)	100%	100%
Speed-up vs. CPU		x12 (C Code)	x12.5 (C Coce) x731.23 (Matlab)
FPGA Chip	Virtex 6 SX475T Maxeler Machine	Spartan 3 XC3SD1800a	Virtex 7 XC7VX485T
Device Capacity (LUTs/ALMs)	297600 6-input LUTs	33280 4-input LUTs	303600 6-input LUTs
Performance density (FLOPS/LUT)**	2796*	576	7019

* Fixed-point operations ** 6-input LUTs

Table 4.5: *Overview of FPGA SNN Implementations on achievable real-time network sizes. For [1] compared to a Pentium 4 3GHz/3GB RAM and for the IO design to a Xeon 2.66GHz/20GB RAM.*

50 μsec, $\times$2-$\times$20 shorter than other models, making it have the tightest real-time constraint among the related works reported.

Especially for the IO-cell model considered in this chapter, the complexity is even greater than the other HH models. The design accurately models 3 compartments and the gap junctions that account for more than 22,000 FP operations per 1 msec, about $\times$19 more than the second most complex related model. Moreover, the use of simpler models to increase efficiency is not an option when modeling the Inferior Olive. Izhikevich and IaF models only have two basic output responses for their neuron: resting and firing states. The biological behavior of the Inferior Olive requires greater resolution, since neurons are constantly oscillating even in their resting state and have the property to synchronize. Such behavior could not be simulated with such simpler models.

A strategy that improved performance in some of the related works, however not applicable in the IO-model, is the event driven execution, e.g.,

in [85]. Another important performance advantage that the designs of simpler models have is the use of fixed-point arithmetic. In such models, precision errors can be insignificant for correct behavior. That is not self-evident in HH models as discussed earlier.

Although the above approaches are radically different, in Table 4.5 we attempt to quantify their complexity and evaluate their efficiency. We take into account the amount of computations per neuron in 1 msec and the network size to estimate the performance of each work in FP operations per second (FLOPS) and properly marking those that use fixed-point. It must be noted that estimations for the computing capabilities of each design are based on data presented in [6] and cannot account for the computations due to the extra custom-made characteristics in the network models of each design, as we do not have this information available. We assume that the majority of the computations come from the simulation of the main neuron model. Our design achieves 2,131.2 MFLOPS, when matching the real-time constraints. [85] supports 832 million fixed-point operations per second mostly due to the size of the simulated networks. Taking into account the area resources used in each work, we define a metric for performance density and measure operations per second per unit area (LUT). Our design has the highest performance density, with second best being at least $\times 2.5$ lower (without taking into account the difference between fixed- and floating-point) [85]; the higher number of DSP-slices in our FPGA device (2,800 vs. 2,015) is however in our advantage. Finally, our design achieves a $\times 731.23$ speedup compared to the original Matlab code and $\times 12.5$ compared to the double-FP C code. This speedup reaches almost $\times 45$ for higher network sizes.

To summarize although our accelerator implements an IO-network which is $\times 19$ more computationally intensive and has $\times 2$-$\times 20$ tighter real-time constraints compared to related models, it achieves at least $\times 2.5$ better performance density supporting 2.13 GFLOPS with a single FPGA device. The empirical precision-error analysis revealed that using our optimizations and single-FP arithmetic create a very slim amplitude error and no phase errors, preserving the correct biological behavior while benefiting in performance. Our design, implemented in a Virtex 7 XC7VX485T FPGA, can maximally support a 14,400-cell network with online parameter configurability for neuron state and network size. Although this design offered considerable speedup over a reference CPU implementation, it was still unable to fully exploit the parallelism of the model, which essentially is a dataflow application that can benefit from finer-grain parallelism which

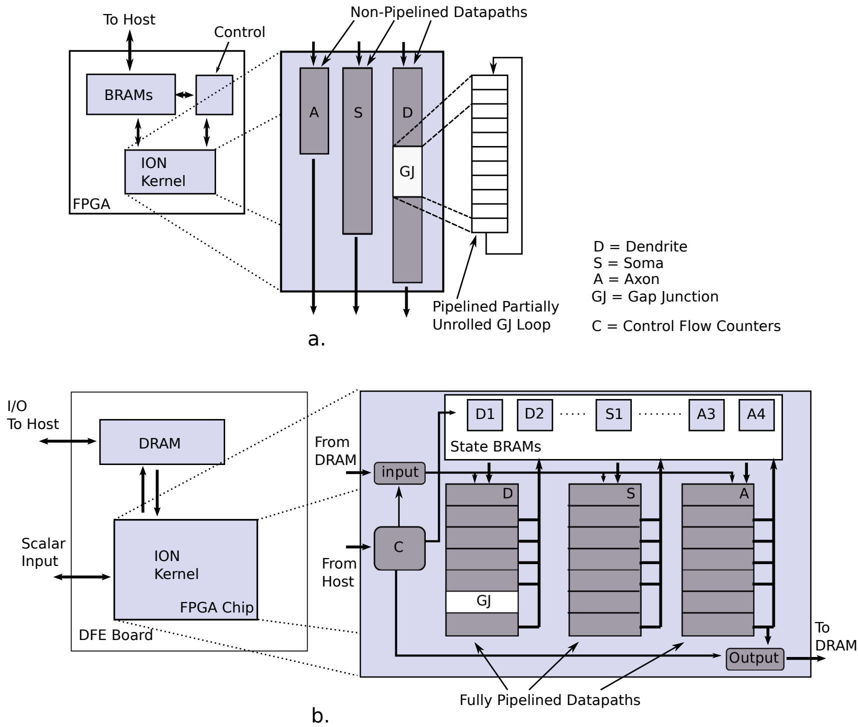


Figure 4.9: *Illustration of (a) A single instance of the FPGA IO Kernel (b) A single instance of the DFE IO Kernel.*

needed considerable restructuring of the initial model that the HLS infrastructure did not readily allowed for.

4.4 DFE-based inferior olive implementation

The ***data-flow engines*** (DFEs) design presented in this section was a continuation of the work of previous section's accelerator [3]. The FPGA kernel using the control-flow methodology that the Vivado HLS infrastructure provides cannot fully exploit the parallelism of an essentially dataflow application.

On the other hand, a Maxeler Dataflow Computing Machine [135], based on DFEs, has the ability to better exploit the inherent parallelism of the model and has the potential to achieve even greater speed-ups with minor changes in the model architecture. The device is essentially a reconfigurable hardware-based HPC accelerator, specializing on dataflow applications.

Its tool flow is designed and optimized to accommodate the acceleration of dataflow applications; that is, applications with the bulk of their implementation using purely raw computations with the absence (partially or totally) of branching execution or feedback paths. The Maxeler tools can exploit the nature of dataflow application to implement very fine-grain pipelined designs, maximizing the throughput and overall performance. The DFE boards also incorporate high-speed design for the communication between the reconfigurable chip and the on-board memory resources. The DFEs can also accommodate several GBs of on-board, high-speed RAM making it ideal for scientific applications manipulating large amounts of data. What makes Maxeler DFEs stand out from the rest of the FPGA-based solutions is the excellent high-level programming language employed for kernel coding (Java with Maxeler-related extensions) and the ability to form scaled up, multi-DFE platforms in a seamless (i.e. user-transparent) manner [135].

4.4.1 The IO-kernel DFE architecture

The DFE implementation of the IO network can be seen in Figure 4.9(b). The design incorporates 3 internal pipelines one for each part of the cell (Dendrite, Soma, Axon), executing the respective computations. The cell states consist of 19 FP values. Each parameter for each neuron is stored on its own BRAM block, for fast read/update of the network state. Since every new cell state is dependent only on the network state of the previous simulation step, only one copy of each neuron state is required. The input stream of the DFE kernel comes from the on-board RAM and represents the evoked inputs (one value for each neuron per simulation step) used in the dendritic computations comprising the network input. Only for the first simulation step the initial state and neighboring (gap-junction) influence are also streamed-in from the on-board memory as each neuron begins its first simulation step. The network output (represented by the axonal voltage) is also streamed to the on-board memory at the same point as it is updated on its respective BRAM block.

Due to the dataflow paradigm followed, the DFE kernel executes the complete simulation run when activated, as opposed to the control-flow-based FPGA kernel that only executes the simulation step by step, under the supervision of a MicroBlaze core. As such, the DFE kernel additionally receives scalar input parameters, denoting the simulation duration and the network size to be simulated. Program flow is monitored using hardware counters monitoring gap-junction loop iterations, neurons executed and the number of simulation steps concluded. All scalar parameters, activation of

the kernel, input-data preparation before execution and output visualization after execution is handled by an off-board host processor. The data flows through the DFE pipelines with each kernel execution step (or ***tick***), consuming the respective input or producing the respective output and state. Each kernel tick represents the completion of one gap-junction loop iteration. As a result, the DFE execution naturally pipelines not only the gap-junction loop iterations but the execution of different neurons within one simulation step as well, as opposed to the FPGA kernel that was capable of only the former (Figure 4.9(a)). Simulation steps are not pipelined in an all-to-all network, as every neuron must have the previous state of all other neurons ready for its gap-junction computations before a new step begins. The DFE pipeline is, thus, flushed before a new simulation step begins execution. The execution of a single simulation step requires N^2 ticks to be completed, where N is the network size.

4.4.2 Additional design optimizations

There are two straightforward ways to speed up execution of the DFE kernel. One is to use multiple instances of the kernel in a single DFE, if the DFE spare resources allow it, thus doubling the network size achievable within a certain time frame. The other is to unroll the gap-junction loop by replicating the single-iteration hardware logic, essentially executing multiple iterations of the loop per kernel tick. If U is the unroll factor of the loop, the number of ticks required for a network simulation step is $N*N/U$, denoting potentially a considerable speed-up. Both of these techniques are subject to area but also timing constraints. Loop unrolling, in particular, could cause extra pressure in the routing of the hardware, limiting the maximum achievable frequency of the DFE kernel.

4.5 Evaluation of the DFE implementation

The design was implemented on a Maxeler device using the Vectis architecture. The Vectis boards include a Virtex 6 SX475T FPGA in their DFEs. The maximum frequency that an IO kernel achieved on the DFE board was 150MHz. This design could be optimized by either using a second kernel instance within the same DFE or unrolling the gap-junction loop. Unfortunately, the spare resources did not enable the use of both optimizations simultaneously. As a result, 2 versions of the design were tested, one with 2 IO kernels within the DFE and one with a single kernel and loop unrolling. The maximum unroll factor achieved for the frequency of 150MHz was 8. One last design was also evaluated. By reducing the DFE frequency to 140 MHz, the unroll factor could be raised to 16 expecting to balance out

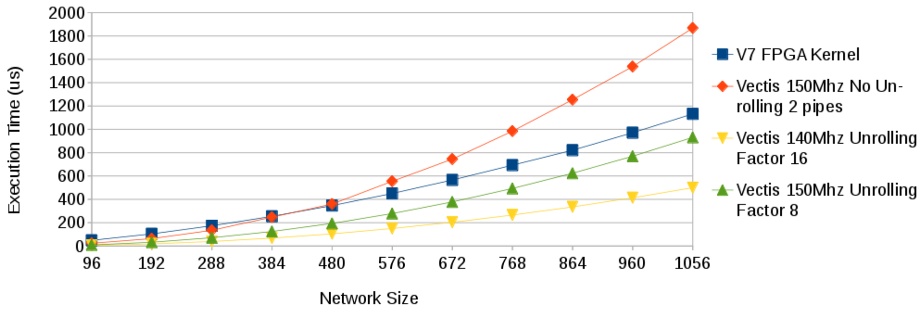


Figure 4.10: *Simulation step execution time for the DFE kernels and the FPGA kernel of section 4.2.1.*

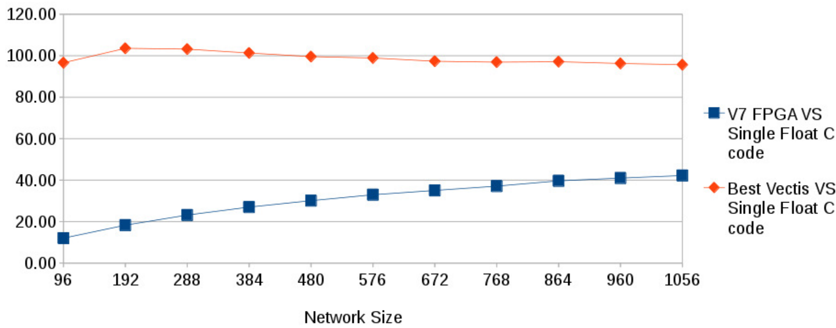


Figure 4.11: *Speed-up vs network size of best DFE and FPGA implementations compared to single-FP CPU-based execution on an Intel Xeon 2.66GHz with 20GB RAM.*

the performance loss due to the lower frequency. Larger unroll factors were not achievable due both to timing and area constraints.

In Figure 4.10 we can see the execution time for all the DFE-based designs and the FPGA-kernel version (deployed on a Virtex 7 XC7VX485T device running at 100 MHz[2]) which includes 8 instances of the IO kernel shown in Figure 4.9(a). Indeed, the best-performing Vectis implementation is the one with the lowest frequency but the highest unroll factor. The gain of loop unrolling supersedes the gains of using the extra kernel instance or the higher frequency. The FPGA implementation incorporates also unrolling optimizations but of lower factor (4) and, combined with its coarser-grain pipelining and lower operating frequency, performs worse than the DFE implementation. In effect, the DFE can simulate up to 330 Inferior-olivary cells at real-time speed (within the 50-μsec deadline) and is

[2]For fairness in comparisons, the Maxeler DFE and the Xilinx FPGA board contain similar resources.

Design	Cheung et al. [85]	IO of section 4.2.1	DFE
Model	Izhikevich	Extended HH	Extended HH
Time Step (ms)	1	0.05	0.05
Real-Time Network Size	64000	96	330
Arithmetic Precision	Fixed Point	Floating Point	Floating Point
Neuron Model OPs * Net. Size (MFLOPS)	> 832*	2131.2	24684
Speed-up vs. CPU	-	x12.5 (C Code)	x92 - x102 (C Code)
FPGA Chip	Virtex 6 SX475T Maxeler Machine	Virtex 7 XC7VX485T	Virtex 6 SX475T Maxeler Machine
Device capacity (LUTs)	297,600 6-input LUTs	303,600 6-input LUTs	297,600 6-input LUTs
Computation density (FLOPS/LUT)	2,796*	7,019	82,943

* Fixed-point operations

Table 4.6: *Overview of current and related work SNN Implementations on achievable real-time network sizes. CPU Speed-up for the IO designs is compared to a Xeon E5430 2.66GHz/20GB RAM.*

$\times$7.7 to $\times$2.26 faster than the FPGA implementation. In terms of speed-up compared to single-core CPU execution [3], the fastest DFE implementation has a speed-up of $\times$92 to $\times$102 compared to the single-FP C implementation (Figure 4.11). It uses about 74% of the total logic available in the DFE hardware; more specifically, about 64% of LUTs, 60% of FF, 30% of DSPs and 41% of on-chip BRAMs. The maximum network size that can be simulated is 7,680 IO neurons before we run out of resources.

To quantify the computation density of the design, we use the same method of calculating FP operations per second (FLOPS) per logic unit (LUTs) as in [3]. To the best of our knowledge at the time, the only other SNN implementation on a Maxeler DFE is the one by Cheung et al. [85]. A comparison of this work to the FPGA-based kernel and our new Maxeler-based design can be seen on Table 4.6. The Maxeler-based design can achieve 24.7 GFLOPS when executing its maximum real-time network (330 cells) and has a computation (performance) density of 82,943 FLOPS/LUT

[3]We use the single-core C implementation as a reference point. It would be possible and interesting explore a multi-core implementation and assess the speedups, but this subject is out of the scope of this work.

(6-input LUTs), as opposed to 2.1 GFLOPS and 7,019 FLOPS/LUT for the conventional FPGA kernel, respectively.

To summarize, the dataflow implementation of the model of the Inferior-olivary Nucleus has achieved significant speed-ups compared to the CPU implementation of the same model and related works. The inherent application parallelism was exploited to a much greater extent when implemented in a single DFE of a Maxeler Dataflow Machine. This, alongside with the higher operating frequency, led to a significant improvement over the previous design implemented on a conventional FPGA board. The fastest DFE implementation achieved a real-time network of 330 neurons – $\times 3.4$ larger than the FPGA one – and achieved almost $\times 2$-$\times 8$ greater speed-ups compared to the FPGA port. The real-time network supports about 24 GFLOPS and has almost $\times 11$ greater computation density than the conventional FPGA.

The larger real-time-network size, as well as the high modeling accuracy, have the potential to enable deeper exploration of the Olivocerebellar microsection behavior compared to the previous FPGA implementation. Besides the speedup, this DFE-based implementation has also opened new possibilities for future simulation-based brain research: The Maxeler DFE platforms offer extended capabilities and significantly higher programming ease compared to conventional FPGAs. Such capabilities include the use of the large DRAMs located on the DFE boards, fast network connections directly to the DFE fabric and the ability to combine multiple DFE boards for running massive-scale network simulations.

4.6 Satellite efforts on FPGA implementations

Additionally to the main effort there were other side-efforts to the HPC development exploring other aspects of the FPGA environment. This resulted into two basic parallel threads:

- An Embedded-HPC implementation of the Inferior Olive referred as ZedBrain
- A manually non-HLS based FPGA implementation to assess the benefit of custom FPGA design.

4.6.1 Embedded-HPC inferior olive - ZedBrain

Additionally to the main effort around accelerating the IO application, embedded-HPC solutions are also relevant. Mostly related to brain-rescue prototyping or artificial experiment setups, such embedded, cost-efficient

implementations can see considerable use on the research field. ZedBrain [136, 137] is using relatively cheap heterogeneous boards such as Avnet's Zynq 7020-based ZedBoard, building a standard HPC framework using Message Passing Interface (MPI). The previous approaches follow the straightforward strategy of deploying the whole neuronal model on the FPGA's resources, even though the size of the network has a linear impact on memory capacity. Additionally, due to the basic routing programmability of a typical, RAM-based FPGA, FPGAs are not capable of clocking at the maximum speed provided by the CMOS node in which they are fabricated. For example, the ARM core on a Zynq 7020 runs at 667MHz, while designers will consider themselves lucky at achieving more than 200MHz for the FPGA fabric.

ZedBrain presents a heterogeneous hardware-software co-designed implementation of the inferior olivary nucleus. Experimentation on partitioning the algorithm and its implementation and evaluating the use of a bare-metal approach versus the use of an operating system (OS), has been conducted.

The processing region of the Zynq (denoted as PS) includes a dual core ARM A9 CPU, with NEON SIMD capabilities, plus several I/O controllers. The integrated Artix-7 FPGA fabric (called the PL region) is interconnected via several channels to the PS. The problem consists now in determining which processing elements could take care of which sections of the model in a more efficient way. Distributing tasks is the first obvious answer (for instance, using MPI), as it would provide flexibility and some scalability. Secondly, based on a prior complexity analysis, the GJ operations were ported to the system on chip (SoC)'s PL, keeping the rest of the operations embedded in the PS. An overview of the proposed architectural solution is given in Figure 4.12. The GJs are mapped via AXI-Lite to the PS for programming and monitoring. Calculation data is transferred via AXI4-DMA from the DDR3 RAM.

The final proposed system offers:

- A balanced HW/SW co-design that takes advantage of several features of the Zynq platform (DMA and cache access from the FPGA fabric, programming and data downloading via TCP-IP and hardware resources administration via OS).

- A C++ flexible design that is portable to other Zynq platforms with more processing power.

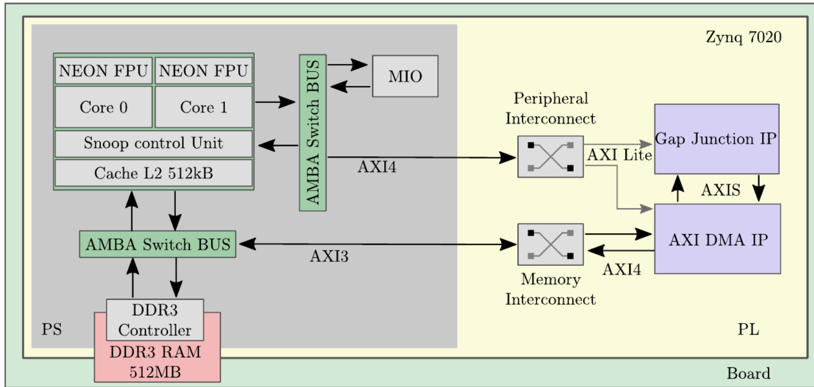


Figure 4.12: *Overall system architecture of Zedbrain. The GJ is managed by one thread of the ARM A9, via AXI4, while the other thread resolves software calculations and I/O outside of the ZedBoard.*

- A noticeable improvement in processing speed (over 18 times when compared with the same algorithm, optimized with NEON instructions, running single-threaded on the Zynq's ARM A9) [137].

4.6.2 Custom (non-HLS-based) acceleration of the inferior olive

In this work [138] the custom design of the IO kernel is explored. All subsequent efforts have used HLS infrastructure for the IO implementations. For practical considerations, HLS is required as custom designing the hardware on the HDL can lead to non-scalable design effort. But as HLS synthesis and place & routing are based on heuristics, it cannot be optimal and does not reveal the full potential of the FPGA as a platform.

This work proposes an efficient multi-chip dataflow architecture for the IO neuron cell and its subsequent interconnected network, which exploits data locality and minimizes network communications over one or multiple FPGA devices. Operationally, the neuron network needs to compute and communicate simulated IO responses to their neighbors and the axon. Both operations are executed concurrently, and separate hardware architectures for computation (based on the multi-compartmental extended HH) and communication are devised. A neuron computation unit is referred as a physical cell (PhC). Within a PhC, the topology-dependent (i.e., incorporating the neighbors coupling) dendrite calculation, and topology-independent Axon+Soma ($a + s$) calculation run in parallel (Figure 4.13).

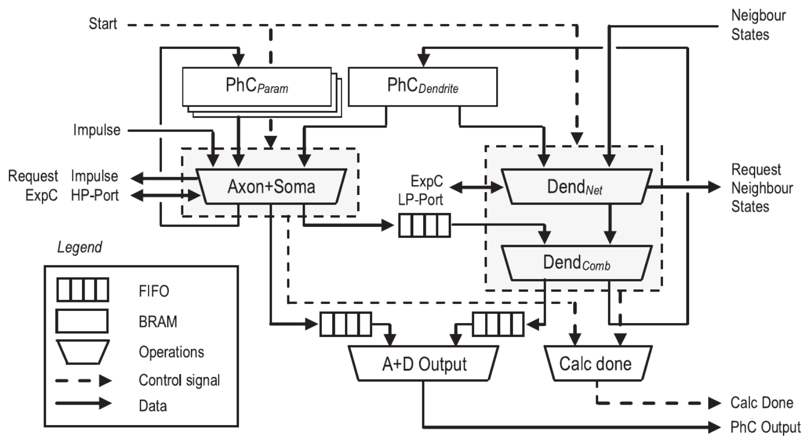


Figure 4.13: *Dataflow of a PhC.*

The Axon+Soma computational unit computes the axon and soma state, and updates a set of cell parameters, based on the current cell compartment states and cell parameters. Internally, the dendrite calculation is dependent on the result of the Axon+Soma calculation to calculate the new dendrite state. Externally, both calculations use the same exponent co-processor (ExpC). The exponent operations, compared to standard operations, require relatively more resources and cycles to complete. To reduce the required amount of resources without adverse effect on the calculation latency, a single exponent instance over multiple neuron calculations in a Kahn process network is utilized.

The neuron cells are connected with decreasing probability the further they are apart. The individual computation units, i.e., physical cells that are in a close proximity to each other are placed within a confinement of a (neighbor) cluster. The amount of clusters k implemented in the FPGA is based on the critical resources. The cluster controller relates new values to the calculation architecture when requested, and store and route their responses. Each cluster controller is designed around several parallel running hardware architectures, that are synchronized by FIFO's. In Figure 4.14, an example is given of a cluster controller with two connected PhCs.

The final contributions of this work are:

- Close to linear growth in the communication cost: with proposed data localization scheme and the resulting linear growth in communication cost, x31 to over x200 more neurons could be simulated in comparison to many of prior work, which are limited by the exponential growth in the communication cost [138].

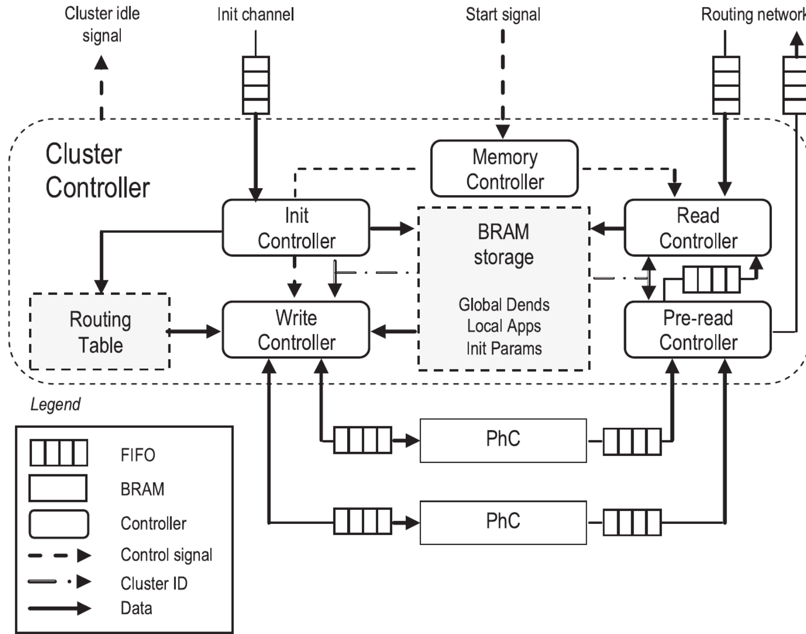


Figure 4.14: *A diagram showing how the controllers are housed within the cluster controller*

- A high run-time configurability, which reduces the need for resynthesizing the system. Additionally, adaption of routing tables, and changes to the calculation parameters are also possible. In this way, the system reduces the time required for experiments with biophysically accurate neurons.

- A custom-made simulator designed for high precision spiking neuron network simulations, but flexible enough to be used for smaller neural networks. The simulator features configurable on- and off-chip communication latencies as well as neuron calculation latencies. All parts of the system are generated automatically based on the neuron interconnection scheme in use.

4.7 The Problem of traditional FPGA workflow

The previous works have demonstrated a clear potential for FPGA-based technology and reconfigurable hardware in general to serve the research effort around computational neuroscience. But the experience of developing the accelerated application of the Inferior Olive using the FPGA toolflows

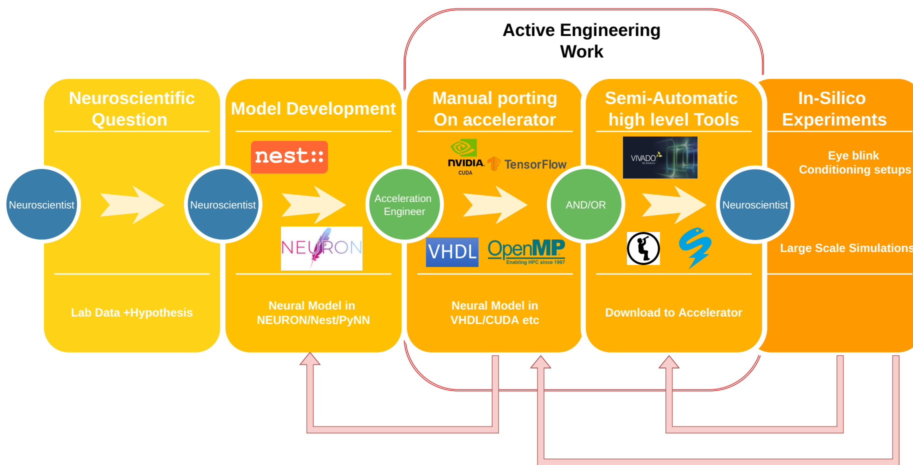


Figure 4.15: *Typical workflow for accelerating neuron applications using HPC devices.*

revealed a certain weakness for the platform when it comes to serving such a dynamic and diverse research field.

Initially, the process begins with the neuroscientific question and the lab/biological measurements that give the data required to formulate the neuroscientific computational model. In the case of this thesis, the subject was the research of the Inferior Olive. Then the computational neuroscienist develops the model using a familiar environment. Typical examples include MATLAB, NEURON, PyNN etc. Then for the acceleration to be implemented, the acceleration engineer needs to take over, in a sense mediate between the scientist and the HPC hardware. In the case of FPGA development, this means porting the initial model to VHDL/HLS descriptions, as the designs described in this chapter. This is essential as it is unrealistic for most neuroscience experts to be expected to acquire the knowledge and expertise for programming HPC hardware, let alone something as specialized as FPGA hardware.

Generally, computational neuroscientists tend to prefer their own familiar coding tools. Experience from custom made intermediate languages created to bridge the gap between neuron modelling and FPGA description, like DYNAMO, implies that unless the scientist is exposed to tools with already familiar constructs for development, wider adoption of the acceleration paradigm will be limited. In the absence of such familiar tools the mediation of the engineer is thus required (Figure 4.15).

The main issue especially, for FPGA development, though is the additional time to implement the accelerated application in combination with

Design	Relative FLOPS	Dev. Time (months)
Semi automatic porting [2]	×1	6
Xilinx HLS tools [3]	×159	4
Maxeler DFE HLS tools [98]	×1083	5
Fully custom Design [138]	×1277	12

Table 4.7: *Performance in FLOPS and development time for the various IO FPGA-based works presented in this chapter. Performance is normalized to the design of [2] which was a preliminary exploratory design before the implementation of section 4.2.1 was made. It achieved to simulate a single IO neuron and its immediate neighbouring GJ connections in real-time.*

the constant re-evaluation that the neuroscientific studies often require. The development does not stop after initial porting. The usage of neuron models on simulations or experiment requires the constant exploration and update of the models (model fitting) after initial porting. Updates than can often change the behavior of the application and its optimality on the FPGA hardware, that might require considerable additional development effort from the engineering side. Sometimes to the extend that any benefit gained from the greater speed of the application, can be lost in the intermediate work that the engineer might require to do. In Table 4.7 we can see a relative comparison between performance benefits and development time for the designs related to the FPGA-based acceleration of IO model that has been the topic of this chapter.

In all cases the development took several months, adding significant overhead to the neuroscientific process. As a result practical use on real-life computational research, using this method of work, would require considerable manpower and can only really be functional on a small scale and on a case by case basis.

4.8 Summary

In this chapter, we presented the full FPGA-focused effort on accelerating the inferior olive application. As an extended-HH representation with advanced connectivity modelling, the acceleration of this application does not only serve the specific research on the cerebellum exploration, but it is also useful to assess the FPGA platform for neuroscientific applications. The model used is quite demanding both in terms of throughput and latency and able to take any HPC platform to its potential limit. The FPGA development leads to promising performance results not only with custom

designs but also HLS toolflows that significantly reduce development time. FPGAs have proven suitable for small-scale embedded-HPC uses as well. The various efforts though reveal a very specific weakness of FPGA development that has less to do with the silicon itself and more of its programming environment. The FPGA tools are very inaccessible to non-experts, thus any acceleration effort would require the engineer (and the FPGA development time) to be in the critical path of the research process. This is creating an extra hurdle of development that will issue extra delays to the research process, which could balance out some of the acceleration benefits. Although such a development method can be beneficial on a case-by-case basis, it provides a significant obstacle for wider adoption of FPGAs, and HPC technology in general, by the computational neuroscience field.

CHAPTER 5

Comparison with other HPC solutions

- **"Performance Analysis of Accelerated Biophysically-Meaningful Neuron Simulations"**, **G. Smaragdos**, G. Chatzikostantis, S. Nomikou,D. Rodopoulos, I. Sourdis, D. Soudris, C. I. de Zeeuw,and C. Strydis, 2016 IEEE International Symposium on Performance Analysis of Systems and Software Ispass 2016, pp. 111, 2016

- **"mCluster: A Software Framework for Portable Device-Based Volunteer Computing"**, D. Theodoropoulos, G. Chrysos, I. Koidis, G. Charitopoulos, E. Pissadakis, A. Varikos, D. Pnevmatikatos, **G. Smaragdos**, C. Strydis, and N. Zervos, 16th International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2016)

- **"Accelerating complex brain-model simulations on GPU platforms"**, H.A. Du Nguyen, Zaid Al-Ars, **Georgios Smaragdos**, Christos Strydis, Design, Automation, and Test in Europe, DATE 2015

In this chapter we will present efforts involving the porting of the Inferior Olive into different acceleration technologies, besides FPGA-based technology. Using the IO as a respective complex neuroscientific benchmark can allow us to derive conclusions on the viability off each acceleration technology explored and make comparisons to the aforementioned FPGA designs.

There are a number of candidates that could be explored as competitive technologies to FPGA-based platforms. A good alternative would be the execution of neuron models in GPUs. Since DSP applications have repetitive and quite parallel functions to compute, GPUs are more capable to efficiently run neuron models. Yet, in the cases of complex models or very large-scale networks, they may not be able to provide real-time performance, due to the high rates of data exchange between the computational elements of models.

Another alternative would be the use of supercomputer implementations. Although these systems can emulate the behaviour and parallelism of biological networks with sufficient speed, the sheer size and complexity of these solutions makes them useful only for behavioural simulations. Supercomputer systems require immense space, implementation, maintenance and energy consumption costs while lacking any kind of mobility. A solution using similar programming environments to supercomputers would be many-core-CPUs such as the XEON PHI or the Epyc Multi-core CPUs, which present a much more cost-efficient alternative.

An option that can possibly implement brain simulations on extreme scales with very little cost is also volunteer computing. Middleware systems such as BOINC [139] can allow for the exploitation of the cumulative processing power of the billions of mobile devices in the wild, by users volunteering their devices in idle time. Even though not suitable for low latency requirements, when the requirement is scale the seer size of the embedded mobile market can potentially provide it.

Based on the above, three platforms were explored to compare with the FPGA-based implementations:

- Volunteer Computing [140]
- CUDA-based GPGPU designs [141]
- The Intel XEON PHI many-core platform [142]

5.1 The inferior olive on volunteer computing

Volunteer computing is a form of distributed computing. It pertains to computer or mobile device owners (volunteers), who donate their processing

power and/or storage resources to scientific projects that need to execute computationally intensive software routines (tasks). Volunteer computing was first applied by the "Great Internet Mersenne Prime Search" project[1] in 1996. However, it was mainly established in the early 2000's, when the University of California, Berkeley released the BOINC middleware [139] to the public.

To date, BOINC is still the most widely used middleware that scientists and companies utilize to launch research projects on distributed networks of desktop computers and mobile devices (smartphones and tablets)[2].

Although many companies and universities have acknowledged the power of volunteer computing, it is still applied with limitations mainly due to the following reasons:

- Available middleware frameworks do not support inter-node task dependencies, hence limiting the nature of projects and applications that are applicable for volunteer computing. Task dependencies are only supported at intra-node level, where developers manually divide an application into sets of dependent tasks, and assign each set to a virtual machine hosted on a client node [143].
- Setting up available middleware frameworks and launching research projects based on volunteer computing requires (a) skills and experience in computer-science aspects, and (b) significant effort and time (the very same experience that this team had while carrying out this work). As a result, scientists from other research domains (e.g. astrophysics, seismology, biology, biomedical, financial) find very hard starting such projects, which also impacts volunteer computing.

To alleviate these issues, the mCluster software framework for IoT-based volunteer computing is proposed. It strives to facilitate the deployment of applications on distributed networks consisting of IoT devices. mCluster targets to minimize the enormous effort and knowledge currently required to successfully launch research projects powered by volunteer computing.

Towards this, mCluster adopts a task-based programming model that requires simple pragma-based annotations of the application software, in order to dynamically resolve task dependencies. In other words, scientists and developers need only to insert certain key-words in their original software to describe task dependencies, which mCluster automatically handles at runtime. The Inferior Olive application was one of the use cases employed to demonstrate the benefits of the mCluster.

[1] http://www.mersenne.org/
[2] https://boinc.berkeley.edu/

5.1.1 The mCluster framework and programming model

Similarly to widely adopted frameworks (e.g. OpenMP), mCluster adopts a task-based programming model that requires developers to simply pragma-annotate the application code to dynamically resolve task dependencies. Figure 5.1 illustrates an example of a simple code that processes 1-d vectors. There are three tasks, namely "v_add", "v_sub" and "v_mult".

To define application tasks, developers need to insert above each task the ***#pragma mcluster*** string, followed by the ***input*** keyword. This defines all task inputs, where developers insert within parenthesis each input type, starting address and size, all separated by commas. Similarly, developers can define all task outputs with the keyword ***output***, followed within parenthesis by each output type, starting address and size.

The framework also provides manual task synchronization with the ***mClusterSync()***. The latter essentially works as a sync barrier that suspends further application execution, until all currently dispatched tasks have returned their outputs. It should be noted also that the mCluster API imposes certain limitations, such as nested task annotation. In other words, developers are not allowed to annotate a task within an already annotated task.

5.1.2 mCluster implementation

Figure 5.2 illustrates the mCluster-BOINC implementation. Considering as example application the source code depicted in Figure 5.1, the workflow proceeds as follows:

1. ***Remove data hazards and generate BOINC-compatible back end files:*** The original annotated code is source-to-source translated, in order to (a) remove any WAW and WAR hazards among tasks, and (b) generate the required BOINC-compatible back end files for task generation / management. In order to tackle inter-node tasks dependencies and, at the same time, follow the strict project/application BOINC structure, we associate each task with a BOINC application [139]. Thus, the mCluster back end BOINC files (a) parse the original code to keep all task metadata (i.e. unique id, inputs/outputs parameters and their sizes) in private structures, (b) run the required BOINC scripts to generate a new full BOINC project (including its internal databases and other structures), and (c) automatically generate a unique BOINC application associated to each annotated task in the original code, excluding its Work Generator (WG) [139], thus suspending execution. As soon as all applications are created, every

```
//vectors addition task
void v_add(int *vA,int *vB, int *vC) {
    //vectors addition code
    ..
}
//vectors subtraction task
void v_sub(int *vA,int *vB, int *vC) {
    //vectors subtraction code
    ..
}
//vectors multiplication task
void v_mult(int *vA,int *vB, int *vC) {
    //vectors multiplication code
    ..
}
int main (){
    //vectors declaration
    ..
    //vectors initialization
    ..
#pragma mcluster input (int &v1[0],VECTOR_DIM,int &v2[0],
VECTOR_DIM) output (int &v3[0],VECTOR_DIM)
    v_add(&v1[0],&v2[0],&v3[0]);
#pragma mcluster input (int &v3[0],VECTOR_DIM,int &v5[0],
VECTOR_DIM) output (int &v4[0],VECTOR_DIM)
    v_sub(&v3[0],&v5[0],&v4[0]);
#pragma mcluster input (int &v4[0],VECTOR_DIM,int &v6[0],
VECTOR_DIM) output (int &v3[0],VECTOR_DIM)
    v_mult(&v4[0],&v6[0],&v3[0]);
    return 0;
}
```

Figure 5.1: *Application-task annotation with the mCluster pragma keywords. Developers need only to define the input/output starting addresses and sizes.*

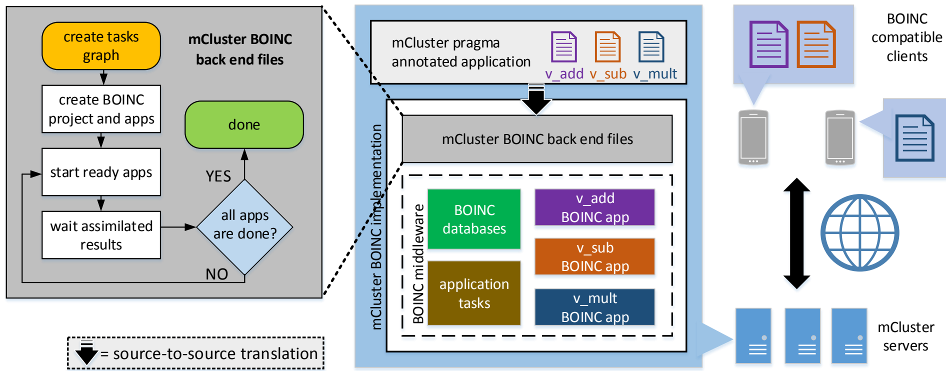


Figure 5.2: *The mCluster BOINC implementation: the original pragma-annotated code is source-to-source translated as described in the previous section and generates the BOINC-oriented back-end files, where each task is associated with a BOINC application and a task graph describes all task dependencies. At runtime, the mCluster starts all applications that have their inputs ready. When an application reports that assimilated results are ready, mCluster traverses deeper in the graph to start new applications that now have their inputs ready. Processing is finished when all applications have reported their assimilated results.*

client device that is registered to the BOINC project will receive all task executables and be ready to accept task instances for execution.

2. ***Start ready BOINC applications:*** mCluster traverses the generated task graph and starts all BOINC applications with ready inputs by automatically generating its WG.

3. ***Wait assimilated results:*** mCluster halts further execution of the back end files, until at least an application reports new results in its corresponding BOINC assimilator [139]. Based on the updated ready data, mCluster traverses further into the task graph to identify which BOINC applications have their inputs now ready, in order to proceed to step 2. mCluster also deletes all BOINC applications whose outputs are no longer needed, keeping at minimum resource utilization on the server side. If all BOINC applications are done, it exits.

It should be noted that the number of concurrently "active" BOINC applications depends on the server resources capacity (e.g. CPU power, available memory, etc). For this reason, mCluster has the option to con-

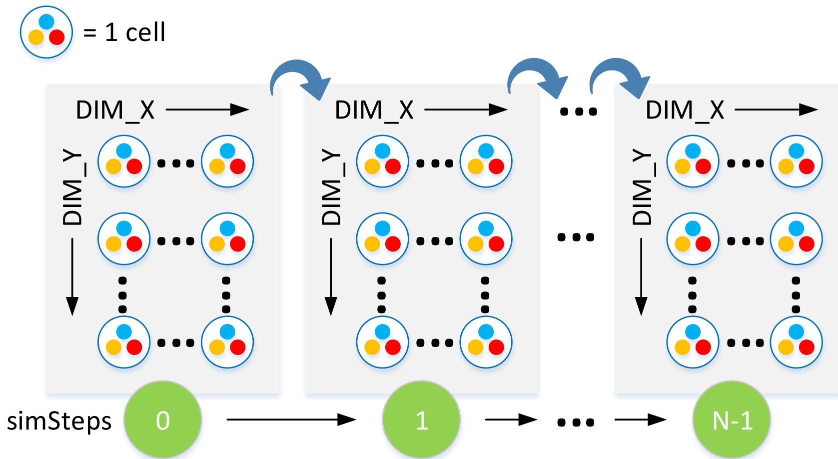


Figure 5.3: Illustration of the Inferior-Olive application execution procedure; in each simulation time step (simStep) each task calculates a cell's state within the 2-d cell grid with dimensions DIM_X × DIM_Y.

strain them up to a predefined limit, in order not to overload the server machine. Such a case can occur for example within for-loops iterated hundreds or thousand times that call independent task(s).

The IO application is tackled using the mCluster system by distributing neuron-cell computations in the virtual cluster of IoT devices (many of them based on commercial GPGPUs). Even though there would be a significant data-transfer penalty due to the wireless communication involved, data-transfer demands in the case of accurate-model simulations are low compared to the computational demands. By taking into account the fact that almost 200 million devices are sold to the public every quarter year, an mCluster implementation of this application shall give the ability to perform brain simulations (a) with biologically accurate neuron models, (b) in the realistic sizes of several millions of neurons, and (c) in a small fraction of the cost compared to investing on a CPU or GPGPU cluster.

In its current version, the application models, in time steps of 0.05 msec each, the state of a 2-d neurons grid (cells), all interconnected with their 8 neighbours. Figure 5.3 illustrates this procedure for N simulation time steps (simSteps). At every simStep, a task calculates a cell's state within the 2-d cell grid with dimensions DIM_X x DIM_Y, while all tasks can be executed concurrently.

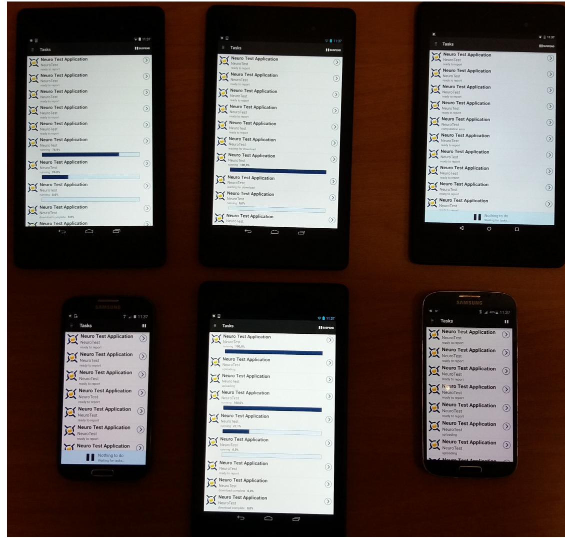


Figure 5.4: *Our experimental system consisted of four Asus Nexus 7 (2013) tablets and two Samsung Galaxy S4 smartphones. The BOINC client app is installed in each device.*

5.1.3 Evaluation of the IO on the mCluster

To verify the BOINC-compatible implementation, an experimental system consisting of one BOINC server machine and six client devices is deployed. The BOINC server is hosted on a 64-bit Ubuntu 12.04 LTS virtual machine (VM) with a single-core CPU at 2.1 GHz, 4GB RAM and 20GB storage.

The client machines are two Samsung Galaxy S4 smartphones and four Asus Nexus 7 (2013) tablets under different Android versions. Figure 5.4 shows the client machines while executing tasks of the Inferior-Olive application.

Figure 5.5 provides the average simStep execution time for 96, 192, 288 and 384 cell grid sizes, divided into the following segments: server work, download task data, task execution, and upload task results.

The server work segment represents an average 13% overhead introduced by (a) the mCluster framework that creates / deletes the required BOINC applications associated with the original tasks, and (b) run the BOINC middleware. On average, 67% of the overall time is spent on the tasks execution, thus indicating the benefit and need of having more processing resources available, in order to reduce the execution time. Finally, approximately 20% of the time is spent on exchanging data (upload and download) between the clients and the server.

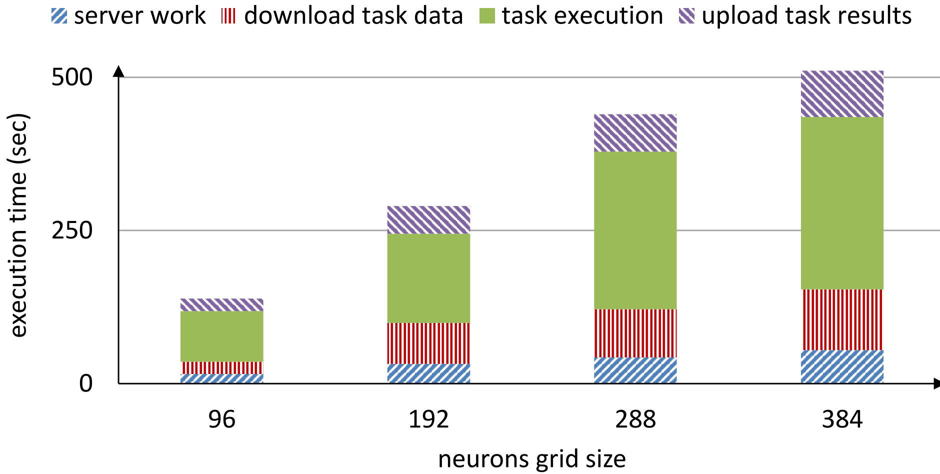
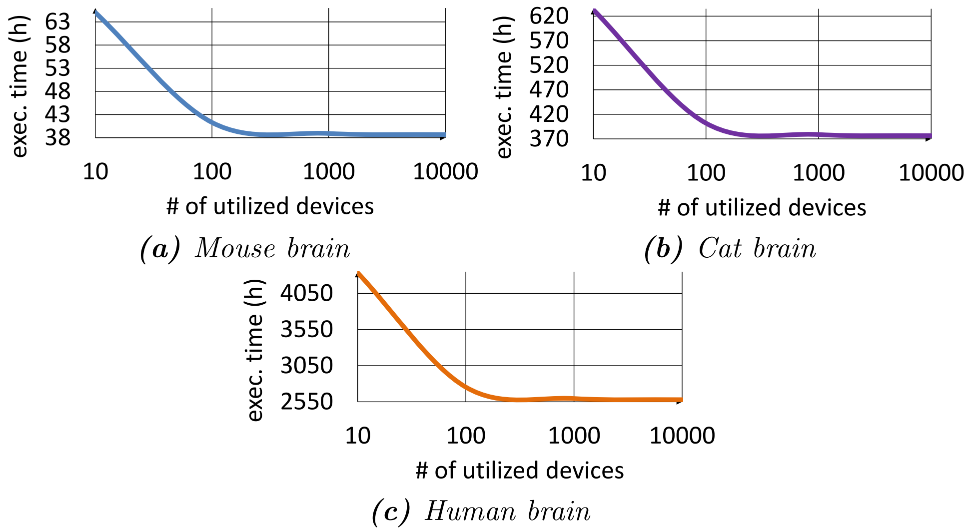


Figure 5.5: *Average simStep execution time of the Inferior-Olive application on all client devices. Overall time is divided into data download-/upload, execution and the mCluster scheduling overhead, such as BOINC application creation/deletion and graph traversing.*

According to our experiments, an Inferior-Olive task requires on average 1.46 sec, 0.93 sec and 4.87 sec for data transfers, server work (create and manage the corresponding BOINC application) and actual execution on a client device, respectively. In our current analysis we consider the mouse, cat and human Inferior-Olive nuclei, that consist on average of 15,000, 146,000 and 1,000,000 neurons. Figures 5.6a, 5.6b and 5.6c show the projected processing time for evaluating a single simStep when simulating the aforementioned different brain complexities.

Powered by volunteer computing, it can be assumed, within each simStep, up to 10,000 devices can be utilized to estimate concurrently each neuron's state. A conservative assumption as the typical participation in similar volunteer computing projects number in the hundreds of thousands of devices [144] and the total theoretical potential participants are estimated in the order of billions [145]. In other words, nowadays there are ample and available distributed processing resources, ready to be utilized for solving complex problems from various research domains.

Distributed-processing platforms based on the concept of volunteer computing, work on a "best-effort" approach. Processing nodes are unreliable in terms of availability and connectivity, thus execute tasks only when certain parameters and circumstances apply. As a result, they can not compete against high-performance (and dedicated) computing platforms.



(a) *Mouse brain*

(b) *Cat brain*

(c) *Human brain*

Figure 5.6: *Projected processing time of a single simStep when simulating different brain types using up to 10,000 IoT devices; execution time is lower bounded by the server work.*

Table 5.1 presents a simple comparison with the FPGA works from the previous chapter (Xilinx Virtex-7 FPGA platform [3] and a Maxeler Maia DFE [98]). The table reveals that both implementations can execute a single simStep at least 3 orders of magnitude faster compared to the mCluster system (we include the server-work and data-transfer overheads). However, [3] and [98] can process up to 14,400 and 7,840 neurons respectively due to their limited resources. In contrast, an mCluster-powered system is only limited in terms of resources by the available IoT devices in terms of processing resources and by the time budget.

In addition, commercial platforms introduce significant acquisition and maintenance costs. In contrast, mCluster-powered systems are solidly based on IoT devices owned by external users. Furthermore, as described in the beginning of this section, mCluster projects can be deployed even on VMs hosted on already available server machines, hence not imposing extra setup and maintenance costs.

5.2 The inferior olive on GPGPUs

The second platform explored was the GPGPU. This implementation is conducted quite early in the IO acceleration effort, thus a simpler instance

Table 5.1: Platform comparison table

platform	max. cell #	simStep exec. time	cost
FPGA Virtex-7 [3]	≤ 14,400	6.4 msec	$8,500
Maxeler Maia [98]	≤ 7,840	0.04 msec	>$25,000
mCluster (10k nodes)	1,000,000*	[7.26 sec – 2,578.80 hr]	$0**

* No real upper limit exists for mCluster; human-sized Inferior Olive selected here for comparison purposes.
** A small cost may be applied, in case mCluster is hosted to a dedicated server machine instead of a VM.

(8-way connectivity) of the application was used as a reference design to assess if the platform has potential to explore further.

The work is focused on simulations of large-scale networks and encompasses the following goals:

- To simulate successfully the IO-cell network for a relatively large number of cells while providing significant performance benefits compared to an execution on CPUs, as the FPGA counterparts are limited by their resource on the maximum supported network
- To present a comparison among multiple GPU platforms in order of effectiveness for simulating this model in particular, and complex neuron-model simulation in general.

5.2.1 The GPU implementation

In Figure 5.7a, a depiction of the original C application, which begins by initializing all cell parameters with a predefined value, is shown. After initializing all cell states and feeding inputs, three for-loops are employed to compute the model of the cells one by one and update new states at each time-step. The C implementation in Figure 5.7a includes one loop for 120,000 time-steps (in this particular modelling experiment) and two loops of visiting all elements of the network rows and columns. The compute-intensive part of the program is located within the two inner loops of simulation. A function in the third loop computes all cell parameters such as dendrite, soma and axon voltages. Thus, the CUDA implementation (as shown in Figure 5.7) focuses on resolving this critical part. Firstly, the two inner loops (2nd and 3rd loop) are mapped onto a 2-dimensional grid of CUDA threads. With this setting, every CUDA thread corresponds to an IO cell. Each thread computes parameters of the dendrite, soma and axon for every time step. Each time step corresponds to one iteration of the

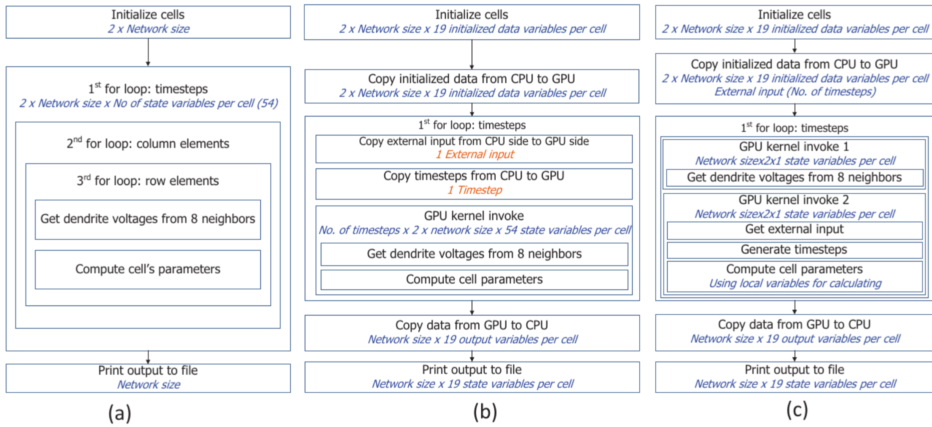


Figure 5.7: *a) C implementation; (b) Un-optimized CUDA implementation; (c) Optimized CUDA implementation.*

1st loop. The three compartments interact tightly with each other as the parameters of one compartment are used to compute other compartment parameters in the next iteration.

Three compartment parameters can be computed in parallel within one iteration. The synchronization across steps is carried out on the CPU side (host) while computing cell parameters at each step is performed on the GPU side (device). This implementation reveals a memory bottleneck. As the input size scales with the number of simulated neuron cells, the device (GPU) memory is not sufficient for a large input size. In addition, the bigger the amount of memory used, the more time-consuming memory fetches becomes. Hence, various optimizations should be applied on the implementation to reduce memory usage and access time.

Further optimization on the CUDA code were implemented as depicted in Figure 5.7c, by (i) coalescing global-memory accesses,(ii) eliminating branching divergence by using the texture memory.

5.2.2 GPU evaluation

A CPU platform (Intel Core i5-2450M (2.5 GHz) with 4GBs of RAM) is used as the baseline platform to run the simulation of the sequential implementation. The performance of the implementation is used to compare with that of the parallel implementation on GPU platforms. The GPU platforms chosen to investigate simulation speed-ups are four NVIDIA GPU platforms: Tesla K20c (Kepler), Tesla C2075 (Fermi), GeForce GTX480 (Fermi) and GeForce GT640 (Kepler). These platforms represent two

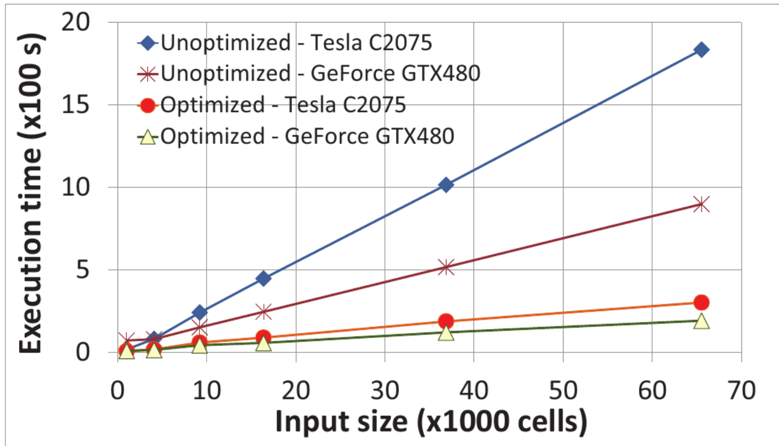


Figure 5.8: *Performance comparison between unoptimized vs optimized implementation (Fermi platforms - Block size 64).*

NVIDIA families in which Kepler has higher processing capabilities and energy efficiency than the Fermi architecture. The IO network model is ported onto each GPU both in single-floating-point and in double-floating-point precision (referred to as single- and double-precision simulation, respectively). Two of our exploration dimensions are the L1 cache usage and the optimal thread block size, which represents partly the GPU processing capability. The thread block size is the number of threads which are processed on the same GPU streaming processor. According to the GPU specifications, it should be a multiple of 32 for best benefits in memory transfers and fine-grain execution of all the threads. The idea behind choosing the thread block size is to maximize the occupancy of the GPU processors when executing the application. The bigger the thread block size is, the higher occupancy the application should get. Preliminary testing shows that the optimal block size for the this application is 64 and the the final performance results were taken using this block size.

In order to evaluate the optimization methods, the the unoptimized implementation on the Fermi platforms (Tesla C2075 and GeForce GTX480) is also simulated. Figure 5.8 reveals at least a 3x speedup of the optimized implementation as compared to the unoptimized one. The speed-up in comparison with CPU-based implementation reaches 9.2 times for the Tesla C2075 and 10.5 times for the GeForce GTX480. Apart from the high execution time, the unoptimized implementation also has drawbacks

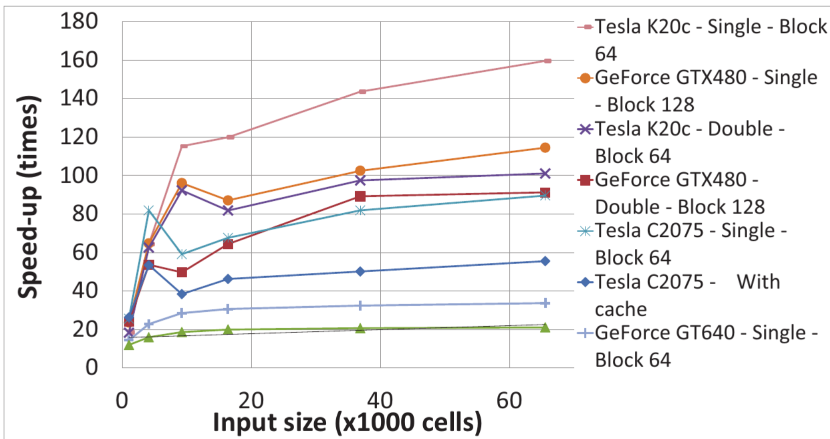


Figure 5.9: *Performance comparison to the CPU execution among all 4 GPU platforms.*

in term of high memory usage due to the large number of variables per cell. Hence, the simulation only facilitates small input sizes.

Figure 5.9 shows a comparison of maximum speed-ups achieved by the four platforms with respect to the reference CPU platform. The figure shows that speed-up is low for small input sizes and increases with increasing input size until it reaches a saturation point beyond the input size of 40,000 cells (except for Tesla K20c, which has the saturation point at a larger input size and was not shown in this figure). After reaching the saturation point, speed-up on different platforms stays constant. This is the point that the problem size is big enough to use the GPU parallel resources to their full extend. As expected previously, the performance of the single precision is always better than that of the double precision on the same platform. The Tesla K20c achieves the best performance for both single and double precision. The GeForce GTX480 performs the simulation significantly faster than the Tesla C2075. Even the double precision performance on the GeForce GTX480 is better than the single-precision performance on Tesla C2075. GeForce GT640 scores the worst performance, however, it still achieves speed-up (up to 20x) in comparison with the CPU platform.

Table 5.2 shows the maximum network size that can be simulated on each platform. This number depends on the global memory size on the GPU side. The optimized implementation achieves 3.3x larger network size than the unoptimized one by optimizing the number of required variables. Thus, the optimized version is able to simulate a realistic human IO network size

Table 5.2: Maximum achievable network size (cells) on different platforms of unoptimized and optimized implementation.

GPU	K20c	C2075	GT640	GTX480
Unoptimized	7,225,344	5,760,000	2,876,416	2,166,784
Optimized	23,658,496	18,939,904	9,437,184	7,054,336

that is approximately 1,000,000 cells. The number of simulated cells in all the GPU platforms is much higher than any biological IO network such as in rat (53,000 cells), chicken (21,600 cells), or cat (146,000 cells) [20]. Hence, GPUs are shown to be promising for HPC simulation of complex biologically accurate neural networks. Further exploration of the platforms capabilities on the field is quite relevant.

5.3 The inferior olive on a many-core Processor

In this section the goal was to analyse the class of the HH models on many-core processors and their direct comparison with the Maxeler DFE which produced the most promising results on FPGA-based platforms, using the IO as a benchmark. Before delving into the accelerator platforms, we start with a more detailed, platform-independent analysis of the application. Then, plausible IO use cases are defined and ported and evaluated on the two accelerator technologies. To the best of our knowledge, this work is the first to attempt a characterization of the class of HH models on state-of-the-art accelerator computing fabrics. The contributions of this work can be summarized as:

- We perform a detailed performance and scalability evaluation on two accelerating platforms: an Intel Xeon Phi and a Maxeler Vectis Data-Flow Engine (DFE).
- We compare and contrast the two platforms, in view of the three IO workload use cases defined in 3.3 and comment on the suitability and usability of each accelerator.

5.3.1 InfOli use cases

For our analysis, we use the three use cases defined in 3.3, which are representative of the memory and computational requirements in typical IO workloads.

The biology of each neuron is characterized by the internal conductances of the ion channels modelled in each compartment. In all use cases, the

user can set each neuron ion channel conductance separately with every experiment and for each cell, giving the greatest possible control over the biological behavior of the simulated network. Additionally, the application allows for the connectivity of the IO network to be programmable by the user before the simulation is deployed. The network connectivity is defined by an $N \times N$ ***connectivity matrix*** (where N is the network size) of FP values signifying the weight of each connection. The weight value is used in the GJ computations to calculate the connection impact on the neuron. A weight of 0.0 denotes the absence of the corresponding GJ connection. As a reminder the three use cases are focused around the biological complexity of the neuron connections:

1. **IO with Realistic Gap Junctions (RGJ)** – IO cells modelled with (biophysically) realistic GJ interconnectivity. The highest amount of detail is included in the GJ modeling.

2. **IO with Simplified Gap Junctions (SGJ)** – IO cells modelled with GJs replaced by simplified , passive connections. This constitutes a simpler implementation in comparison to the previous use case.

3. **IO with No Gap Junctions (NGJ)** – IO cells modelled without accounting for GJs and without any interconnectivity implementations. This is the simplest use case, whereby the neurons are modeled as separate computational nodes.

5.3.2 Quantifying neuron interconnectivity

While the IO application can implement any neuron interconnectivity (through a simple connectivity matrix), for the purposes of this work we feed the IO application with connectivity maps that were created ***a priori*** in order to reflect a certain connectivity density. Thereby, inter-neuron communication (and its associated computing complexity and memory overheads) can be manipulated for profiling purposes. To fully enable this configuration, we have implemented a ***connectivity generator*** which prepares the input of the IO application. The network to be simulated is, in the general case, expressed as a three-dimensional mesh of neurons. Each neuron is assigned a set of three integers representing the neuron's normalized Cartesian coordinates. These sets can be used to calculate the distance between each neuron pair. The distance between neurons adjacent to each other is considered as the unit of distance measurement. Based on the distance of each neuron pair, the probability of them forming a synaptic connection

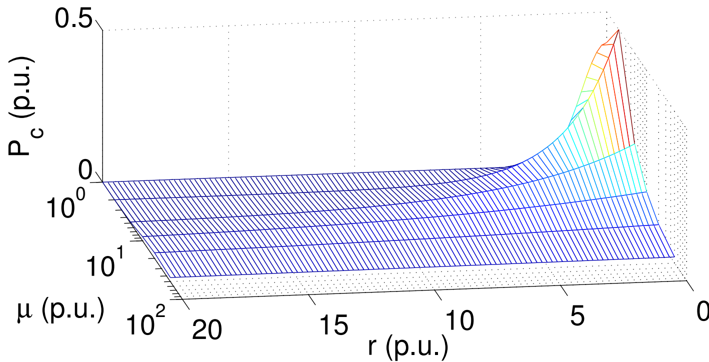


Figure 5.10: *Example of connection probability, using the exponential distribution with mean value μ and the distance between neurons (r). Both quantities are measured per unit (p.u.).*

can be computed, according to a pre-defined distribution. Figure 5.10 illustrates an example of the probability of a connection being formed (P_c) for a variety of distances between neurons (r) and a range of exponential distributions (differentiated by their average value μ). For each evaluated pair of neurons in the network, P_c is calculated and a random number x is produced between 0 and 1. Iff $x < P_c$, then we assume that a connection will be implemented for these two neurons. As a result, by tuning distribution parameters (μ in the example of Figure 5.10), a different intensity in the formation of neuron connections is achieved, and consequently different connectivity densities.

5.3.3 Target platforms

All three IO use cases have been evaluated on two platforms. More precisely a hardware-based implementation was developed to be ported to a Maxeler ***Vectis*** Data-Flow Engine (DFE) board as well as a software-based implementation for the Intel Xeon Phi 5110P system [146] (part of the Blue Wonder iDataPlex cluster hosted at STFC, UK [147]). The specifications' overview for both evaluation platforms is presented in Table 6.1.

The DFE used is a Maxeler HPC node based on reconfigurable hardware that was used in [98]. The DFE board used in our experiments is a 3rd-generation Vectis-DFE board, that includes a Xilinx Virtex-6 FPGA chip.

The Xeon Phi is a Many Integrated Core (MIC) architecture co-processor, which features 61 cores, each capable of supporting up to 4 instruction streams. The current generation of Phi cards, named Knights Corner, use

Table 5.3: *Specifications of Evaluation Platforms .*

Spec	Maxeler Vectis	Xeon Phi
On-Board DRAM	24 GB	8GB
RAM bandwidth	38.4 GB/s	320 GB/s
Memory streams/channels	15	16
On-chip memory	6.5 MB (FPGA BRAMs)	30 MB (L2 cache)
Number of chip cores	Not Applicable	61
Chip frequency	depends on design	1.053 GHz
Instructions set	fully configurable	64 bit
Power consumption	140 W	225 W

an Intel Xeon host processor which can offload work to the Phi, much like a GPU, using well-known programming models such as OpenMP and OpenCL. However, in contrast to GPU approach, the Phi can also be thought as a stand-alone processor in that it has its own Operating System. This allows for an application to run natively on the platform, which is what our IO implementation opts for.

5.3.3.1 IO implementations

The IO application on the Intel Xeon Phi co-processor, depicted in Figure 5.11, uses a shared-memory programming model by utilizing the OpenMP library. The model first accepts a user-defined map detailing connections and size of the desired neuron network to be simulated. This map is processed once and a network is dynamically generated, forming connections between the dendrites (Gap Junctions) as dictated by the map. Each neuron's GJ allocates arrays of floats. These arrays need to always store up-to-date information for a subset of the entire network, specifically the most recent dendritic membrane voltage potentials of the neurons on the other side of the GJ connections. Thus, for each simulation step, the algorithm of the model is summarized as refreshing the information of these arrays and, then, performing the necessary computations for each of the three neuron compartments (dendrite, soma and axon). In each algorithmic step, the neurons are divided into subsets as evenly as possible. Each subset is then handled by an OpenMP thread. Since the Xeon Phi 5110P can support up to $61 \times 4 = 244$ instruction streams, the application can use up to 244 OpenMP threads, splitting the network in just as many subsets. Therefore, for network sizes below 244 neurons, the maximum number of threads used (and thus the degree of parallelism) is limited by the number of simulated neurons. This is because a single neuron cannot be split into multiple threads. It has also been observed that dividing the network into very small subsets does not yield better performance, since each thread ends up

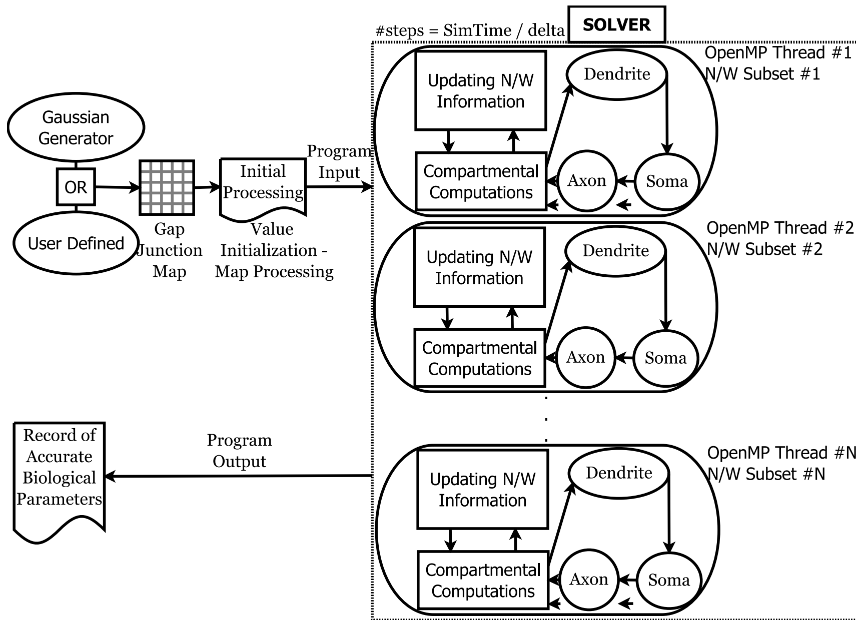


Figure 5.11: *Xeon-Phi architecture of the IO kernel.*

Table 5.4: *Logic utilization for the Virtex 6 FPGA chip on the Vectis.*

RGJ	SGJ	NGJ
257409 / 297600 (86.49%)	268458 / 297600 (90.20%)	167147 / 297600 (56.16%)

having low workload, disproportional to its overheads. Therefore, it is not always efficient to maximize the number of OpenMP threads, particularly for small networks.

The DFE implementation is similar to the one in [98] with small upgrade in order to accommodate the connectivity topology configurations as described in previous sections. The connectivity matrix weights are sent to the DFE at every simulation step, for the use cases that include programmable connectivity. Using a memory-based connectivity matrix allows us to avoid the time-consuming process of resynthezing a new DFE for updating the connectivity density of an experiment. The program flow is tracked again using hardware counters which monitor the neurons executed, the number of simulation steps completed, as well as the GJ loop iterations (where applicable). The data flows through the DFE pipelines at each kernel execution tick, consuming an input set and producing the output and a new neuron state. Simulation steps are not independent from each other and thus are not parallelizable. That is because every neuron requires the previous state of all other neurons to compute its GJ (in the

RGJ or the SGJ case) before a new step begins. As a consequence, the DFE pipeline is flushed before a new simulation step. This dependency is removed when 0% connectivity density is imposed on a simulation. In experiments with random connectivity, a constant number of ticks is spent by the DFE for computing each GJ connection whether it actually exists or not. This is because the kernel is designed with a fixed pipeline for computing GJ so as to be reusable for experiments with different connectivity parameters. In doing so, re-synthesizing the DFE kernel is avoided saving setup time for each experiment. Conversely, this DFE implementation cannot get a performance benefit from lower connectivity densities. However, using the same DFE for experiments of different connectivities offers predictable, guaranteed performance.

The resource utilization of the FPGA device of the DFE is reported in Table 5.4. A single computation kernel (DFE) fits on the FPGA device for each use case. SGJ has reduced computations allowing a higher degree of unrolling the GJ computation loop yielding added performance benefits.

5.3.4 Performance evaluation

In this section, we evaluate the performance of the use cases on the Xeon-Phi and the Vectis-DFE nodes. All use cases were executed using a simple experiment, borrowed from corresponding biological experiments, designed to produce a typical response from the IO model: a complex spike at the neuron output stage (axon). The experiments simulate 6 seconds of brain time. The complex spike is produced by evoking a small 6.0 mA pulse as input to all IO cells at the same point after program onset for about 500 simulation steps (or 25 ms, in brain time).

A standard procedure for experimenting with SNN models typically begins with an extensive, initial parameter-space exploration using small- or medium-sized networks. Having fine-tuned all model parameters, real experiments can then commence by simulating either small- to medium-sized networks (10s to 100s of cells) for exploring ***real-time, closed-loop control*** such as Brain-Computer Interfaces [148] (TYPE 1), or large-scale networks (>1000s of cells) for mounting ***behavioral experiments*** [17] (TYPE 2). In this section we present performance results only for the evaluation for the TYPE 1 experiments. The reason for that is that the measurements, after the publication of the results, were found to be partly contaminated for the large scale measurements. This was validated by the repetition of the measurements when the error was detected and also validated from later work presented in detail on the next chapter. Thus

only an analysis of the TYPE 1 experiments will be mentioned on this section.

5.3.4.1 Measurement methodology

Timing measurements on the Vectis DFE were taken measuring the kernel time within the host code using timestamps before and after the kernel call. The CPU host code is blocking, thus, only the DFE kernel is active during the measurement. The time includes the kernel execution (processing and DRAM data-exchange delay) and the activation delay of the FPGA device. This activation takes about 1 ms, which is negligible compared to the overall execution time that takes several seconds to several minutes in our experiments. The execution time of a single-simulation-step is derived from the total execution time divided by the number of simulation steps. The DRAM communication delay can be estimated by the amount of data exchanged between the FPGA device and the on-board DRAM per simulation step, considering the DFE DRAM bandwidth.

All measurements concerning the Xeon Phi have been carried out with Intel's profiling and analysis tool Vtune Amplifier XE 2015. Information for the DRAM accesses and average bandwidth used by the application, was obtained by performing bandwidth-mode analysis. This analysis further provides insights related to the program execution time and CPU utilization. The profiler was installed on the Intel Xeon host (since the Phi card has a minimal OS) and launched with directives for collecting information from the accelerator platform (*-target-system=mic-native* flag).

5.3.4.2 Execution time vs. network size vs connectivity density

Next, we evaluate the performance of the two platforms for different network sizes and connectivity densities. All execution results are presented per simulation time step.

The performance results of the three use cases on the Maxeler Vectis-DFE platform are depicted in Figure 5.12, for small-to-medium network sizes. Connectivity density does not affect the execution time of the DFE implementations. That is because the design statically supports all-to-all connections (100% connectivity density) in a fixed dataflow pipeline. As discussed earlier, this avoids re-synthesis of the design whenever connectivity parameters change, reducing the time needed to set-up an experiment.

As illustrated in Figure 5.12, for small/medium network sizes, the DFE's fine-grain parallelism yields good performance scaling to the network size, even for the demanding case of IO neurons with realistic GJs (RGJ). Execution time ranges from 6 *us* to a little over 500 *us* for 1,056 neurons.

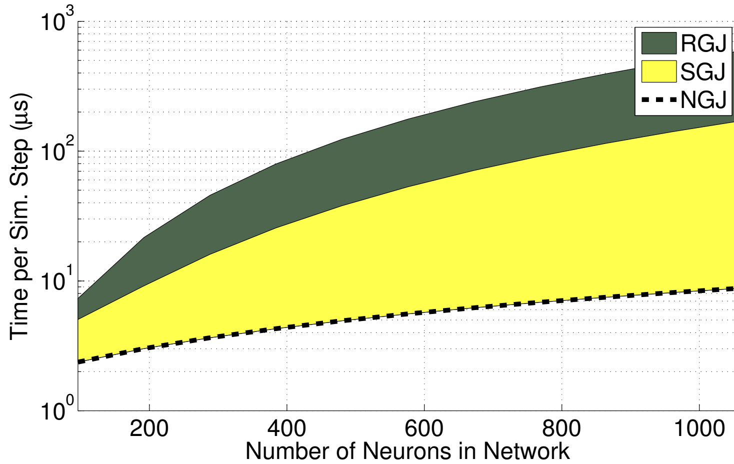


Figure 5.12: *Execution time per time step for small networks for Vectis DFE. Colored areas correspond to the range of possible execution-time values due to different connectivity densities (0%-100%). (Note: Graph areas are unstacked.)*

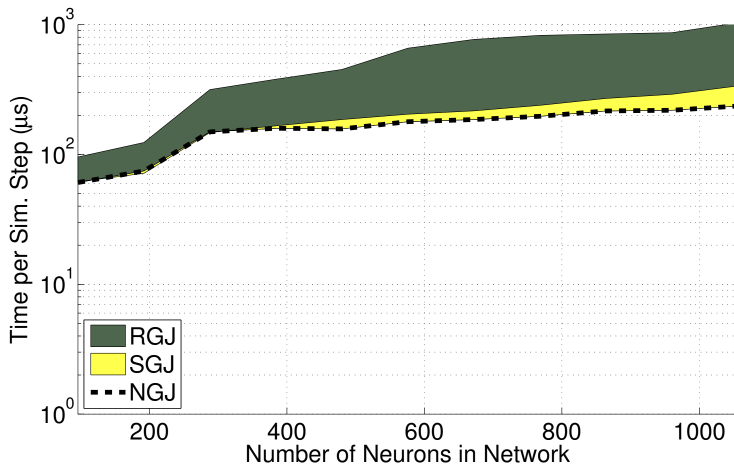


Figure 5.13: *Execution time per time step for small networks for XEON PHI. Colored areas correspond to the range of possible execution-time values due to different connectivity densities (0%-100%). (Note: Graph areas are unstacked.)*

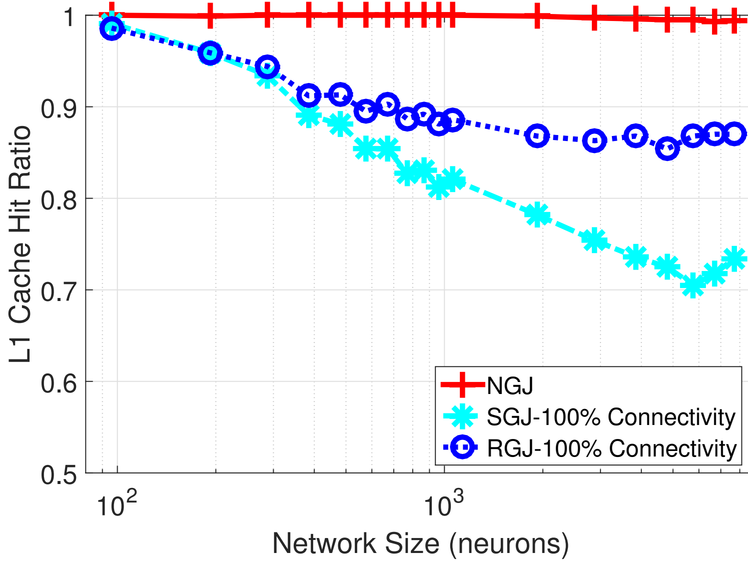


Figure 5.14: *L1 cache hit rates ratio for each use case for 100% connectivity networks of the Xeon Phi implementation.*

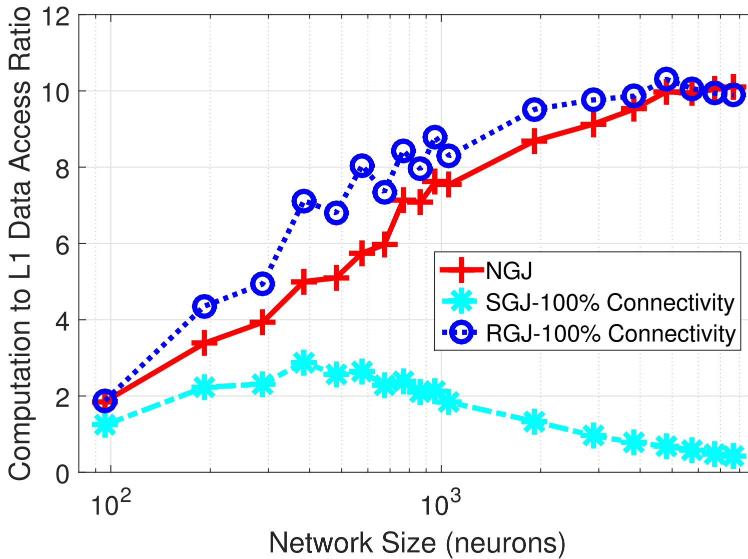


Figure 5.15: *L1 compute-to-data ratio for each use case for 100% connectivity networks of the Xeon Phi implementation.*

Table 5.5: *Real-time achievable network for each use case on each platform.*

Vectis-DFE Implementation	**Real-time Network Size**
RGJ	300
SGJ	550
NGJ	7,200
Xeon-Phi Implementation	**Real-time Network Size**
RGJ (100% connectivity)	24
SGJ (100% connectivity)	54
NGJ	54

Although the maximum size of a simulated network is important for TYPE-2 experiments, achieving (brain) real-time speed is critical for TYPE-1 experiments. Table 5.5 presents the ***real-time*** capabilities of each use case. For the RGJ use case, the Vectis DFE can simulate 300 neurons at real-time speed, while for the SGJ case the real-time network is 550 neurons. As expected, the NGJ case exhibits a linear increase in execution time; this results in a capacity to execute 7,200 neurons in real-time on the DFE.

Application behavior is significantly different on the Xeon-Phi implementation. Here, the design can actually benefit from lower connectivity densities but, at the same time, cannot provide a performance guarantee for every problem size in each use case. Different problem sizes and connectivity distributions can produce quite varied connectivity densities that affect performance.

In general, the Xeon-Phi performs well in simulating small-to-medium scale networks. Still, it is about 50% (medium-scale networks) to almost an order of magnitude (in small-scale networks) worse than the DFE implementation in the RGJ use case. The gap is even greater for SGJ and NGJ as illustrated in Figure 5.13.

To give a better insight on Xeon Phi's performance we discuss in more detail some of its performance metrics. Figure 5.14 illustrates the hit ratio of the Xeon Phi's L1 caches when simulating networks of various sizes and connectivity patterns. Large networks without GJs require small amounts of data and exhibit good data locality, leading to high hit ratios. On the other hand, simple and regular GJs have large memory footprints, leading to lower L1 hit rates. SGJ has a lower L1 hit ratio than RGJ as its GJs are less complex and exhibit less data reuse during processing, compared to the realistic GJs. Once the core fetches the GJ's data to its L1 cache, processing an a realistic GJ will yield more L1 hits than in the simpler GJ. Moreover, Figure 5.15 presents the ratio of CPU cycles spent in computations per

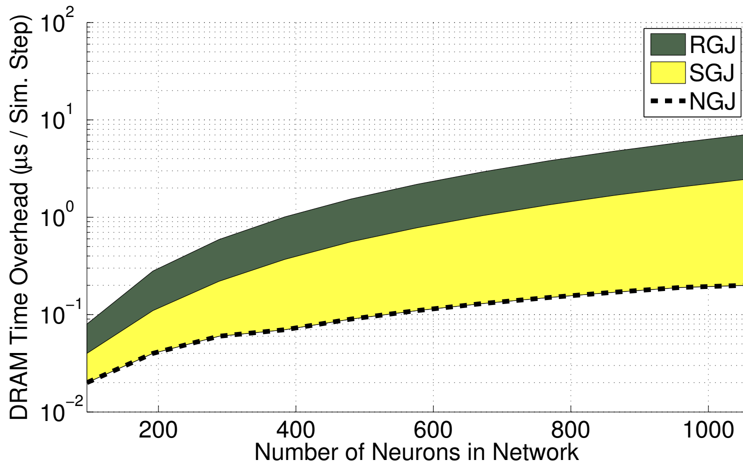


Figure 5.16: *DRAM data-transfer overheads for small network on the Vectis. It is clear that the platform has enough provisions and is not DRAM-bound for any of the explored use cases. Colored areas correspond to the range of possible timing-overhead values due to different connectivity densities (0%-100%). (Note: Graph areas are unstacked.)*

CPU cycles spent in accessing L1 cache and leads to similar observations. NGJ and RGJ achieve more computations per L1 access compared to SGJ which appears to wait overall longer for the memory. This implies that the XEON PHI platform should provide much greater performance the larger the network size would become.

Aiming for real-time performance, Xeon Phi can simulate 24 neurons in RGJ experiments. The real-time achievable network for the SGJ and NGJ is 54 neurons. All cases are substantially lower than Maxeler DFE as shown in Table 6.2. This is explained by the fact that the Xeon Phi resources cannot be used efficiently for such small networks sizes.

5.3.4.3 DRAM data-transfer overhead

Besides analyzing the overall execution time, it is interesting to also measure the fraction of overall time spent in processing versus the time spent in waiting for the DRAM data to arrive. Typically, in this class of applications, where complex biophysically accurate models are simulated, processing dominates the overall execution time. So, the data-transfer timing overheads are expected to be small compared to the computation times.

In the Vectis-DFE implementations, fast on-FPGA BRAM blocks are used to store the neuron states. BRAMs increase the overall input/out-

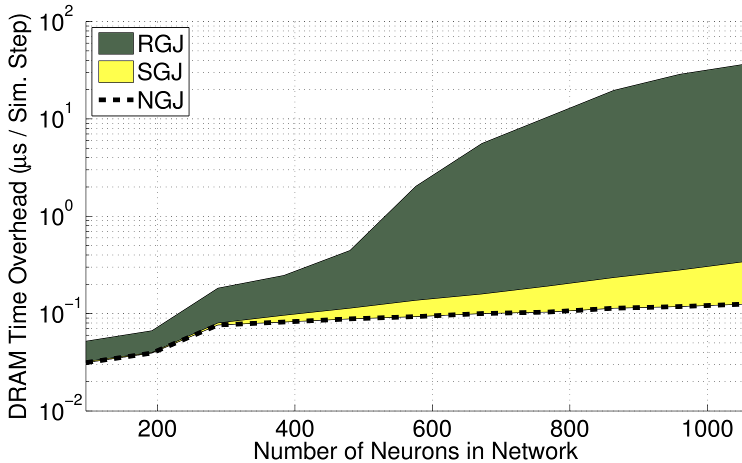


Figure 5.17: *DRAM data-transfer overheads for small network on the XEON PHI. It is clear that the platform has enough provisions and is not DRAM-bound for any of the explored use cases. Colored areas correspond to the range of possible timing-overhead values due to different connectivity densities (0%-100%). (Note:Graph areas are unstacked.)*

put bandwidth of the DFE and require in practice a single cycle to access. Neuron states are the most frequently used data both for the neuron-compartment processing as well as for the GJs processing. As a result, the Vectis-DFE implementation has negligible data-transfer overheads. The fraction of time spent waiting for off-chip data in small-to-medium networks, depicted in Figure 5.16, is consistently under 0.02% in the most memory-demanding RGJ case. Even for NGJ, where computations scale linearly to problem size, the time spent waiting for DRAM access is slightly over 0.02%, showing that the provided DRAM bandwidth is sufficient for feeding the DFE engine.

The Xeon-Phi offers an order of magnitude higher memory bandwidth than the DFE, achieving 320 GB/s versus 38.4 GB/s. As can be observed in Figure 5.17, for the most data-intensive RGJ, the DRAM overhead in execution time does not scale well above 500 neurons. Still, the overall performance penalty is low due to the high memory bandwidth of the Xeon Phi. When simpler or no GJs are employed (SGJ, NGJ), the DRAM data-transfer overhead is negligible and remains so for all tested problem sizes.

Table 5.6: *Comparison between tested platforms using the IO RGJ as benchmark (full GJ with all-to-all communication). Speed-up compared to single thread C implementations used in [3].*

Platform	MaxNeurons	RT Network Size	Speed-up	Effort
GPU	Millions	0	$\times 3 - \times 22$	Medium
mCluster	1,000,000*	0	$< \times 1$	Low
Xeon PHI	52,535	24	$\times 1.2 - \times 80$	Medium
DFE	7864	300	$\times 90 - \times 100$	High

*No real upper limit exists for mCluster - depends on the availability of participant IoT devices; human-sized Inferior Olive selected here for comparison purposes.

5.4 The problem of complex workload diversity

Just looking into cell model, HH models comprise challenging systems of differential equations that are essentially ***embarrassingly parallel*** computational problems that can be solved with a typical divide-and-conquer strategy. In such cases, each neuron model effectively becomes a free-running oscillator, the execution of which can be parallelized independently of its neighbours. The more powerful the processing nodes employed, the higher the speedups achieved.

However, complementing cell models to include Gap-Junction (or any type of complex-connection) modeling leads to extended-HH models which not only feature increased computational complexity (due to the GJ calculations) but, additionally, cease to exhibit an embarrassingly parallel nature. The reason, of course, is that – with a rising connectivity density among neurons in a network – dependencies among differential equations also rise, leading to computational problems that are increasingly difficult to parallelize across simulation time steps. In effect, in such cases ***coupled oscillators*** are formed that need to be co-simulated in strict lockstep among them. This requirement, in turn, enforces the use of cycle-accurate, transient simulators where simulation steps are hardly compressible and all neuron states need to be completely updated at each simulation step. Thus such applications provide significant challenge for any HPC platform.

In the part of volunteer computing, FPGA-based implementations can execute a single simStep at least an order of magnitude faster compared to the mCluster system. However they can process up to some tens of thousands of neurons due to their limited resources. In contrast, an mCluster-powered system is only limited by the available IoT devices in terms of processing resources. Additionally maintenance costs are reduced immensely as the system uses IoT user resources already in the wild. GPGPUs also

show good scalability, very high maximum network capacities and tractable programming effort.

The Maxeler Vectis-DFE implementation shows impressive performance for small-to-medium scale networks and also achieves higher real-time networks. This makes the DFE a suitable platform for speeding up experimentation on small-to-medium size neuron networks, that are often used for parameter-space exploration of neuron models. The DFE is also suitable for experimentation with real-time setups (TYPE 1) as it can achieve networks of meaningful sizes at real-time speeds. Finally, it can also provide predictable performance for any kind of network size or input and connectivity, a crucial factor when planning lengthy experiments and a feature that software-based solutions cannot easily provide. The Vectis DFE does have limitations, though, in performance when the computational demands increase above the parallelization capabilities that the DFE can provide. By the implications of the L1 cache analysis and because of the high memory bandwidth, it can be expected that the XEON Phi has the potential to provide comparable performance under specific workload circumstances (larger networks and less than 100% connectivity density).

Besides, despite the concessions Maxeler makes through use of its MaxelerJava programming language, the Xeon Phi remains a much more straightforward platform to program on, as it is a software-based solution. Furthermore, it can exploit different connectivity densities in terms of performance efficiency, but this aspect also results in its inability to provide predictable performance. Additionally, the resources on the Xeon Phi can support a larger maximum network population than the Vectis DFE. However, when it comes to real time model execution, the Xeon Phi falls short of the $50 - \mu sec$ timing constraint for achieving real-time simulations, regardless of the network size used.

A general comparison between platforms can be seen on Table 5.6. Even with just using a narrow benchmark within the IO possible uses (that of RGJ), we can already see that each technology seems to excel on different areas, all relevant to the computational research but also presenting high variability in the effort required by the engineer to implement. This heavily implies that a single platform cannot optimally serve all relevant neuroscientific needs. And this with only a single complex instance of the IO application. The application instance itself can vary, changing workload characteristics significantly, thus also how platforms compare. Taking into account the variability just within one application and the vast diversity within the computational neuroscientific field in general, there is a clear

message that the field as a whole can only be sufficiently served by a heterogeneous system.

5.5 Summary

In this chapter, we present a thorough performance analysis for various state-of-the-art HPC alternatives to FPGA-based solutions and make brief comparisons with each one. We target a number of accelerator technologies that have demonstrated great potential, especially given their integration and clustering capabilities: Volunteer computing, GPGPUs and the Intel Xeon Phi. GPGPUs are shown to be promising for HPC simulation of complex biologically accurate neural networks. Further exploration of the platform's capabilities on executing the IO is required. Volunteer computing although providing extreme scale at low cost cannot compete in terms of raw performance. The Xeon Phi, on the other hand, appears more suitable for large-scale simulations, with many neurons and dense interconnectivity between them. We substantiate, in all cases, that the target neuron simulator scales gracefully in terms of DRAM utilization as well. Results heavily imply that a true solution for HPC acceleration of neuroscientific models can only be served by a heterogeneous platform.

CHAPTER 6

The BrainFrame Platform

- **"BrainFrame: a node-level heterogeneous accelerator platform for neuron simulations"**, **Georgios Smaragdos**, Georgios Chatzikonstantis, Rahul Kukreja, Harry Sidiropoulos, Dimitrios Rodopoulos, Ioannis Sourdis, Zaid Al-Ars, Christoforos Kachris, Dimitrios Soudris, Chris I De Zeeuw and Christos Strydis in IOP Science, Journal of Neural Engineering, volume 14, Number 6, 2017.

- **"Multinode implementation of an extended Hodgkin–Huxley simulator"**, Giorgos Chatzikonstantis, Harry Sidiropoulos, Christos Strydis, Mario Negrello, **Georgios Smaragdos**, Chris I. De Zeeuw, Dimitrios Soudris in Elsevier Neuroscomputing,Volume 329, Pages 370-383, February 2019.

- **"flexhh: A flexible hardware library for hodgkin-huxley-based neural simulations"** Rene Miedema, **Georgios Smaragdos**, Mario Negrello, Zaid Al-Ars, Matthias Möller, and Christos Strydis, IEEE Access, vol. 8, 2020

- **"Exploring Complex Brain-Simulation Workloads on Multi-GPU Deployments"** Michiel A. van der Vlag, **Georgios Smaragdos**, Zaid Al-Ars, and Christos Strydis, ACM Trans. Archit. Code Optim. 16, 4, Article 53 (December 2019)

As established by the findings in the previous chapter, a better approach for neuroscientists to exploit HPC technology is to provide them with an acceleration platform that has the ability to adjust to the aforementioned variety of workload requirements. A heterogeneous system that integrates multiple HPC technologies, instead of just one, would be able to provide this.

As described in Section 4.7, the typical FGPA workflow can lead to a lot of extra delays in the research progress. This is the case also for other HPC technologies, such as high multi-threading and GPGPUs, even if less pronounced since they are less exotic platforms than FPGAs. Support for heterogeneity will add additional delays using the typical approach. For all integrated technologies the ability to select a different accelerator, depending on availability, cost and performance desired, must be provided.

Thus a different workflow framework is required to ensure the efficient service of the research effort. In this different framework, the engineering should not be mediating between the neuroscience model development and the production of scientific results. Instead, the user/neuroscientist is using basic and generic modeling IP blocks to develop his model. These blocks would be ported into the accelerator technology by the engineer beforehand and only actively maintained during the computational research process. If the library blocks provided are generalized enough, the neuroscientist is able to immediately employ the HPC technology as the neuron model is developed, without active porting effort by the engineer. The active engineering would be limited into improving and extending the block library for extension of support and can be done in parallel to the neuroscientific experiments (Figure 6.1).

In addition to this, a framework for a heterogeneous system needs to be using a popular (to the neuroscientist) user interface. It, thus, requires a front-end which should provide two crucial features:

- An easy and commonly used interface known by neuroscientists so that they can employ the platform without the constant support of an engineer.
- A front-end that can reuse the vast amount of models already available to the community.

In this chapter, we propose a framework for a ***heterogeneous acceleration platform*** for computationally challenging neuroscientific simulations called ***BrainFrame***. By using this system, we demonstrate the effect of model characteristics on performance and thus make a concrete

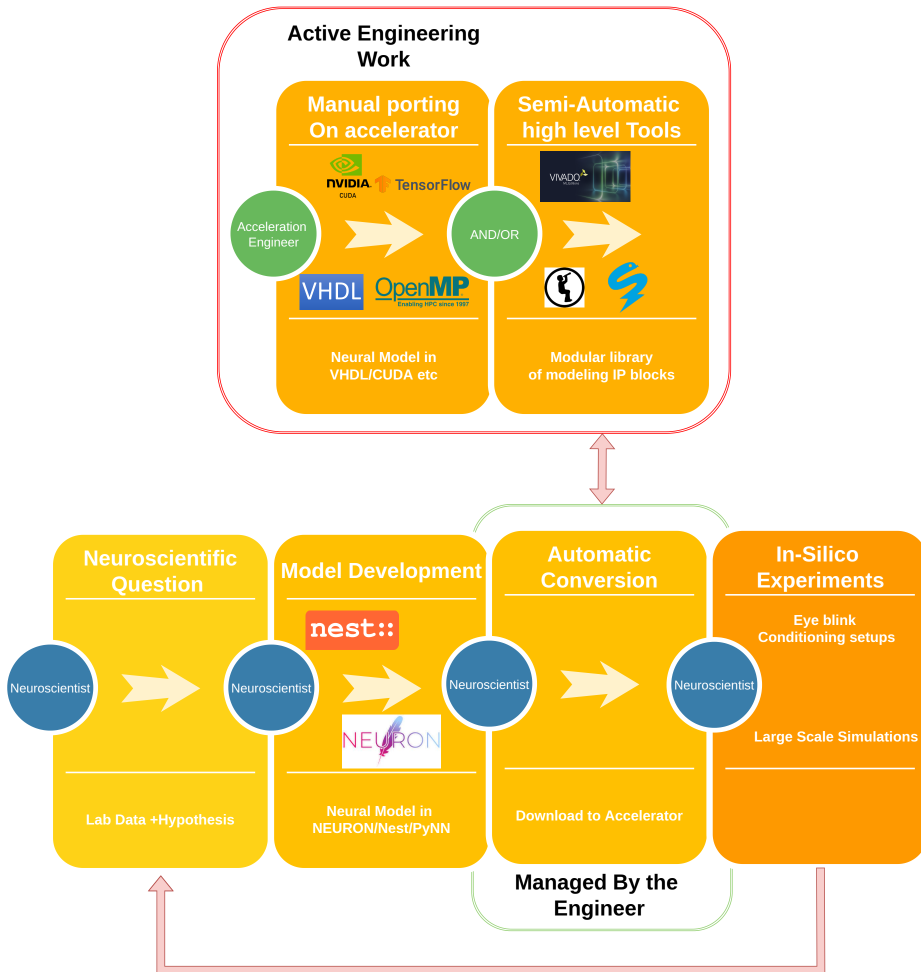


Figure 6.1: *Workflow framework proposed to support the scientific process through a heterogeneous HPC platform.*

case for the significance of employing heterogeneity in HPC systems used in the field of computational neuroscience. To this end, we use the IO as an extended Hodgkin-Huxley use case. As established before, it is a representative workload of such neuron representations, and thus it can work as a Proof-of-Concept benchmark for HH-based IP block libraries that the full platform would employ.

Depending on the desired model characteristics, we again assume two general types of simulations that are relevant in neuroscientific experiments (as in the previous chapter). The first one has to do with highly detailed (biophysically accurate and even accurate to the molecular level) models of smaller-sized networks that require ***real-time*** or close to real-time performance (TYPE-I experiments). The second type involves the simulation of large- or very large-scale networks in which the level of detail can often be relaxed. These experiments attempt to simulate network sizes and connection densities closely resembling their biological counterparts (TYPE-II experiments). Finally we also propose a front-end for the platform based on the PyNN language [27]. PyNN has been widely adopted by the computational-neuroscience community and has direct integration with many other well-known neuron modeling frameworks, covering both aforementioned features that such a front-end would require.

6.1 Methods

6.1.1 Application use cases detailed profiling

For our analysis, we employ the three use cases drawn from Section 5.3.1, which are representative of the memory and computational requirements of the IO workload. They can also be considered as plausible instances of multi-compartmental modeling using HH models with various cases of modeled inter-neuron connectivity.

For many complex experiments, it is not the number of connections but, rather, the connectivity density (C) that is indicative of neuron interconnectivity. The application allows for the connectivity of the IO network to be programmable by the user before the simulation is deployed. Network connectivity (when present) is defined by an $N \times N$ ***connectivity matrix*** (where N: Network size) of FP weights signifying the weight of each connection. Weights are used in the GJ computations to calculate the connection impact on each neuron.

In Figure 6.2, we see the amount of FP operations per use case for various interconnectivity densities, based on the aforementioned profiling of the IO application. From the same profiling run we can derive the compute (in

Table 6.1: *Specifications of the accelerator fabrics used.*

Spec	DFE (Maia)	Phi CPU (5110P)	GPU (Titan X)
On-Board DRAM	48 Gb	8 Gb	12 Gb
RAM bandwidth	76.8 Gb/s	320 Gb/s	336.5 Gb/s
On-chip memory	6 Mb	30 Mb	3 Mb
Number of chip cores	–	61	3072 CUDA Cores
Chip frequency	Kernel-defined	1.053 GHz	1 GHz
Instructions set	n/a	64 bit	32 bit
(TDP)	140 W	225 W	250 W
IC process	65nm	22nm	28nm

FLOPS) to memory (in single-FP memory accesses) ratio for the application, that reveals whether each use case is computation- or memory-bound (Figure 6.3).

The compute-to-memory-access ratio (from Figure 6.3) of the RGJ suggests also that this use case is computation-bound for all connectivity cases: With increasing problem sizes, the effects of the I/O increase but still the application retains its compute-bound nature. For the SGJ case the computations per simulation step are now $859 * N + 4 * N^2 * C$. The accumulation is parameterized using the weight that is assigned to each connection between two neurons, thus the connectivity information needs to be accessed the same way as is in the RGJ case. The actual FP operations are reduced by about one order of magnitude compared to the previous use case (see Figure 6.2). Yet, the connectivity aspect still disrupts the pure dataflow nature of the model. A positive compute-to-memory ratio is seen here as well. For the NGj case is the case where the application becomes purely dataflow and can achieve the greatest parallelism possible. The processing requirements scale almost linearly with the network size and, compared to the other use cases, fewer computations are needed, as shown in Figure 6.2 (computations per simulation step : $859 * N$). As we can see in Figure 6.3, although the NGJ use case shows that computation is still the most important aspect of the application, both computation and memory access scale linearly and at a similar pace.

6.1.2 HPC fabrics and Implementation

Our heterogeneous platform incorporates three accelerator fabrics; a Maxeler ***Maia*** Data-Flow Engine (DFE) board [149], an Intel Xeon Phi 5110P CPU [146] and a Maxwell-based Titan X GPU by NVidia [150] (Table 6.1).

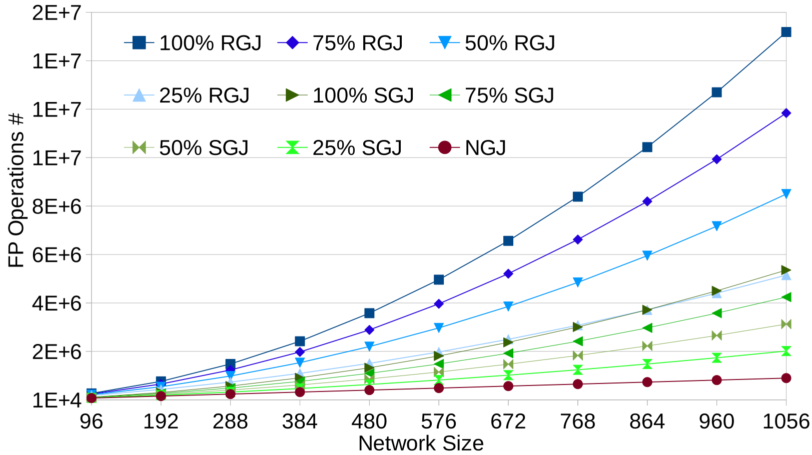


Figure 6.2: *Floating-point operations required per simulation step of the IO model for each use case and for different connectivity density percentages (%).*

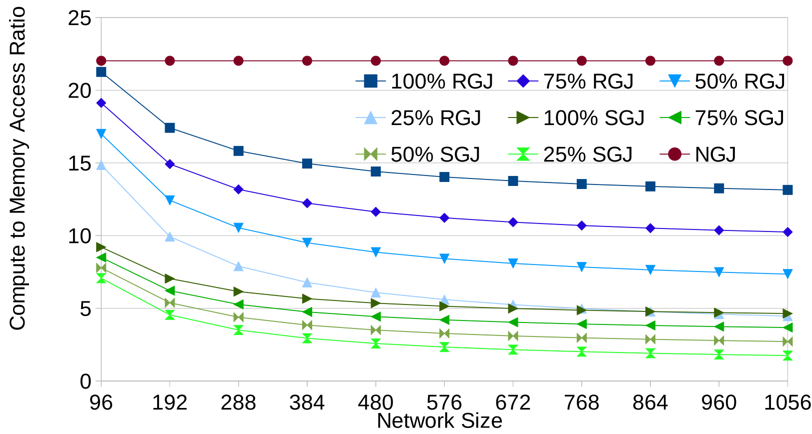


Figure 6.3: *Compute-to-Memory-Access Ratio per simulation step of the IO model for each use case and for connectivity density percentages (%).*

All there boards are PCIe-based which is how they communicate with the host system. The use of PCIe interfaces ensures that composition of BrainFrame-enabled machines can been easily tailored on a per-case basis depending on the availability of funds and hardware resources of a research laboratory. Different types and mixes of PCIe-based accelerators can be selected.

The Maia DFE is a Maxeler HPC technology based on reconfigurable hardware. The Maia DFE boards also incorporate a high-bandwidth, multichannel, highly parallel, customizable interface to the onboard DRAM memory resources (up to 96 GBs) making it ideal for scientific applications. The DFE board used in our experimental setup is a 4th-generation Maia-DFE board implemented using an Altera Stratix V 5SGSD8 chip.

The Xeon Phi is the same model as used in Section 5.3, the 5110p Knight's Corner board. The Titan X GPU includes 3,072 CUDA microcores, which are used to parallelize computation execution, and 12 GB of on-board RAM. GPU implementations also benefit from the generally good adoption of the NVidia CUDA-library open environment that allows porting of applications with similar ease to the Phi OpenMP and OpenCL frameworks. GPUs also come at a relatively lower cost than the other two accelerator types. However, as opposed to the Xeon Phi, a GPU cannot act as its own host increasing communication delays between host and accelerator during execution.

Lastly, it must be noted that BrainFrame is to be used in scientific research that is very dynamic and fast-paced. The goal is not to over-optimize the different accelerator implementations, but to propose and maintain a balance between the programming effort and optimization needed, resulting in shorter development times for cutting-edge research tools.

6.1.2.1 IO on the Maia DFE

The DFE implementation of the IO application can be seen in Figure 6.4 and is a more advanced version of the work described in Section 4.4. New features include the addition of programmable connectivity and programmable neuron state by the user between experiment runs without the need to re-synthesize the design. The design implements three pipelines on the DFE hardware to accelerate the application, one for each part of a neuron (Dendrite, Soma, Axon), executing the respective computations. The state parameters for each neuron are stored on separate BRAM blocks for fast reading/updating of the network state, as they are the data that are most used throughout the experiment execution. Since every new neuron state is dependent only on the network state of the previous simulation

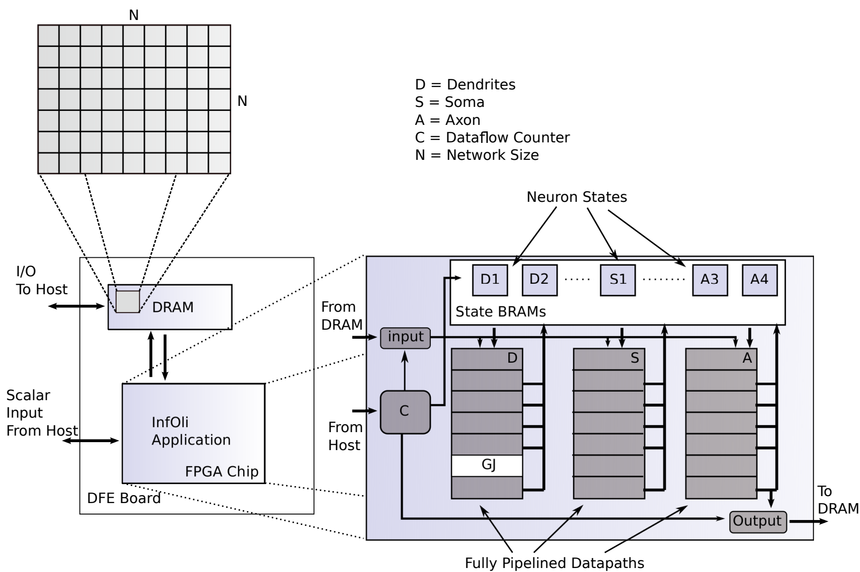


Figure 6.4: *DFE implementation of the IO application.*

step, a single copy of each neuron state is required at any point during execution. The input stream to the DFE kernel originates in the on-board DRAM and represents the evoked (external) inputs, used in the dendritic computations comprising the network input. The initialization data are also streamed in from the on-board memory only once at the start of execution. The size of the connectivity matrix makes it impossible to store on the on-chip memory. It is, thus, placed on the on-board RAM and streamed in batches dictated by the computations. The kernel output is streamed back to the on-board memory and – at the same time – is updated in the (on-chip) BRAM blocks of the DFE.

Simulation steps are not themselves directly parallelizable (as in the previous work), as every neuron must have the previous state of all other neurons available for its GJ computations (only in the RGJ or SGJ cases) before a new step begins. The DFE pipeline is, thus, flushed before a new simulation step begins execution. This dependency is lifted when in the NGJ case. Additionally, in use cases where programmable connectivity is included, the ticks for the evaluation and execution of a GJ connection are always spent regardless of whether a connection actually exists or not. Thus, this implementation cannot benefit from a smaller connectivity density in terms of performance.

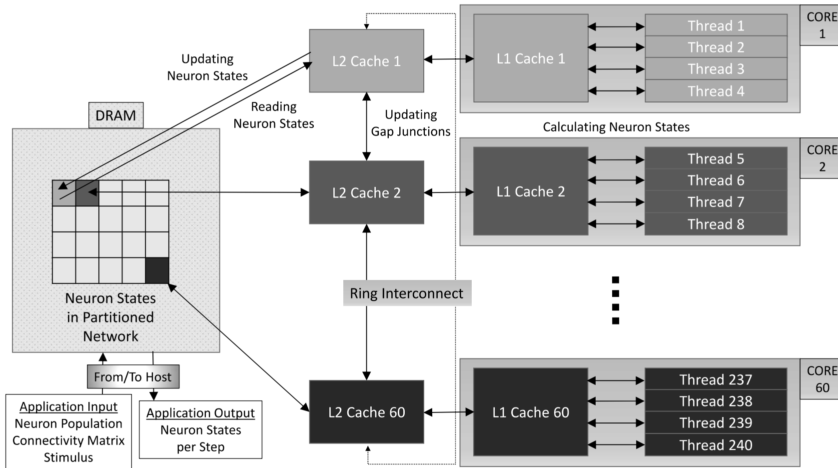


Figure 6.5: *Xeon-Phi implementation of the IO application.*

6.1.2.2 IO on the Xeon Phi

The IO application on the Intel Xeon Phi co-processor, depicted in Figure 6.5, is based on a typical shared-memory implementation. The application uses the OpenMP library to spawn threads, which can work in parallel. As the Xeon Phi 5110P uses one core to handle OS-related tasks and each core features multithreading technology that can service up to 4 instruction streams simultaneously, the IO application on the Xeon Phi uses up to $60 \times 4 = 240$ OpenMP threads. Each thread is programmed to handle a part of the neuronal network (sub-network), which is partitioned as uniformly as possible to prevent workload imbalances. In each simulation step, every OpenMP thread computes its sub-network's state. Each OpenMP thread accesses memory space shared by all threads so as to collect data from other neurons, with the purpose of re-evaluating the state of its sub-network's GJs. In this task, shared-memory accessing can cause stalls in thread operations due to issues such as memory contention. It should be noted that the described implementation assumes that the entire network is large enough to be partitioned in 240 parts. When dealing with smaller networks, the implementation utilizes less than the maximum amount of the platform's assets, since it is designed to require to assign one neuron on each OpenMP thread.

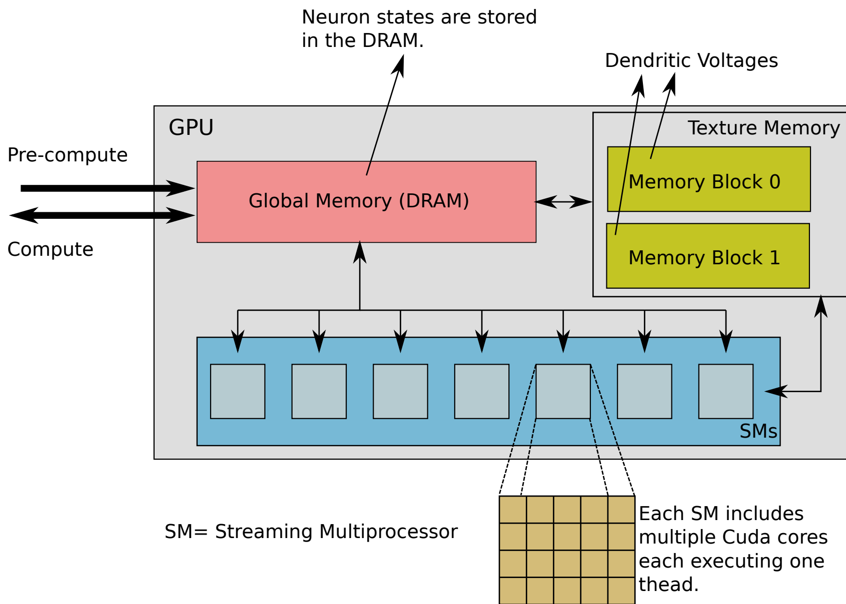


Figure 6.6: *GPU implementation of the IO application. Pre-compute and compute operations are issued by the host.*

6.1.2.3 IO on the Titan X GPU

In Figure 6.6, we can see the IO implementation on the GPU. The execution flow includes two stages, a pre-compute and a compute stage.

In the pre-compute stage, the host initializes the neuron states and the external input currents for the entire simulation duration. It allocates global memory on the device to store the current-step neuron states, next-step neuron states and the external input currents. At the end of this stage, the host copies the required data for simulation onto the GPU. Similarly to the other two accelerator implementations, the current-step dendritic voltages of all cells are accessed frequently as they are used to determine the GJ influence. To reduce memory latency, they are bound to the GPU texture memory. The texture memory is a cached memory on the GPU used to reduce memory latencies when the application has specific memory-access patterns. Writes to texture memory, during the compute stage, are conducted only after all computations of a simulation step have finished. It must be also noted that after the pre-compute stage, no data is transferred from the host to the GPU; the GPU contains all necessary information for the simulation.

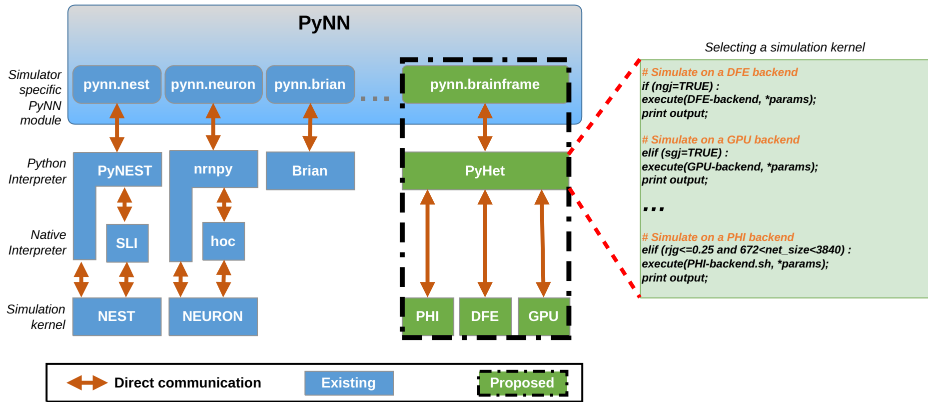


Figure 6.7: *PyNN architecture and the proposed BrainFrame framework.*

During the compute stage, the neuron calculations are performed and the new states are persistently stored throughout the simulation duration. To compute the new states for a single simulation step, the host launches a CUDA kernel on the GPU device. Before simulation, the kernel is configured for a particular use case (RGJ, SGJ or NGJ) and inter-neuron connectivity scheme (if applicable). The kernel is executed by a two-dimensional grid of CUDA threads on the device. Threads are executed in parallel by the CUDA micro-cores of the GPU. Every IO cell of the model is mapped to a corresponding thread that calculates the states of the neuron. On kernel completion, the host receives the calculated result of the simulation step from the device. The host uses two operation streams to issue the kernel execution and data-transfer operations to the GPU. A kernel in one stream is launched only when the kernel in the other stream has completed. Thus, when one stream is computing the currently executing simulation step, the other stream is performing the necessary data transfers to the host from the GPU. Since the texture memory is updated only after the kernel completes execution, data coherency is maintained. Thus, computation of the current-step neuron states and data transfer of the previously computed states overlap, effectively hiding Host-to-GPU transfer delays.

6.1.3 BrainFrame & the PyNN front-end

PyNN is a Python package that facilitates the interchangeability and the study of different simulation environments within the computational neuroscience community [27]. It allows for simulator-independent specification of neuronal-network models and already supports many popular simulators like NEURON, NEST , Brian, and so on.

The PyNN API supports modeling at multiple levels of abstraction, both at the neuron level and the network level. It provides a library of standard neuron, synapse, synaptic-plasticity models and a set of commonly-used connectivity algorithms while also supporting custom user-defined connectivity in a simulator-independent fashion.

We integrated the three accelerator fabrics as back-ends on the BrainFrame system using PyNN as a front-end. The PyNN integration provides the neuroscientific community with easy access on the accelerators without constant mediation from the acceleration engineer while also providing an interface for the already established models to be used with the new heterogeneous acceleration back-end. These characteristics of PyNN can have decisive impact on the adoption of BrainFrame by the community.

As a proof of concept for the front-end of the BrainFrame platform, we have added the IO model in the library of standard PyNN models. Following the PyNN paradigm, the user initially selects the simulator – in our case our BrainFrame simulator – and then proceeds to select the neuron model, in our case the Inferior-Olive model. A population of neurons using the chosen model is then generated, determining the inter-neuron connectivity type and, finally, a projection of the specified neuronal network is created.

The main difference between the proposed PyNN-backend substrate and the typical simulator back-ends within the PyNN environment is an additional selection step. In this step, a decision about which of the three alternative acceleration fabrics will be used for a specific experiment is made, based on the available hardware and the characteristics of the simulated neural network.

A conceptional view of the architecture of the PyNN BrainFrame module is shown in Figure 6.7. For the simulator kernels to communicate with the PyNN frontend, an intermediate BrainFrame-specific PyNN module (pynn.brainframe) is required that implements and extends common methods and objects like the neuron models, synapse models, projection methods and objects. In the case of the proposed BrainFrame module, we implemented objects and methods: i) for the initialization of the simulator, ii) for the description of the neuronal network in PyNN, and iii) for controlling the simulation execution. In some cases, an additional interpreter module is needed to translate these Python objects and parameters to each simulator's native parameters and language. For our system, we developed PyHet – the BrainFrame-specific Python interpreter – which serves the aforementioned role and also implements the accelerator selection.

The final BrainFrame System will be implementing more generic kernel libraries that will be used by the PyNN front-end to simulate user defined models. That way, the accelerator implementation will be completely transparent to the user and predictions can then be made based on the analysis of the individual kernels that can guide the selection algorithm.

6.2 Results

In this section, we present a thorough performance analysis of our heterogeneous BrainFrame platform. The goal is to evaluate the platform and give a clear view on how each accelerator performs when running various instances of the IO use cases, validating the usefulness of a heterogeneous HPC simulation framework for computational neuroscience. The performance analysis also acts as a guide for proposing an ***accelerator-selection algorithm***.

To validate the correct functionality of the separate accelerator implementations, we use a simple experiment that recreates a typical response that is found in the inferior-olive network (axon response). In this experiment, each cell produces a so-called complex spike, seen in Figure 4.1c. 6 seconds of brain time are simulated, which translates to 120,000 simulation steps. The complex spike is produced by applying a small current pulse as input to all IO cells at the same instance after program onset, for about 500 simulation steps (or 25 ms, in brain time).

As mentioned in the introduction, we identify two distinct tracks that can be followed in conducting neuroscientific experiments, both covered in this evaluation. We perform one batch of measurements ranging from 96 to 960 neurons representing small-scale, real-time TYPE-I experiments, and a second batch ranging from 960 to 7,680 neurons representing larger-scale TYPE-II experiments. We consider, by consulting our neuroscience experts, the minimum meaningful network size for experiments to be around 100 neurons, thus our measurements for TYPE-I experiments begin at 96 neurons. The evaluation is focused on the performance of single-node accelerators, thus a network-size cap is set by the smallest maximum network supported by each of the three accelerator fabrics: in this case, the DFE fabric limits network sizes to 7,680 cells.

6.2.1 Performance evaluation

All performance measurements concerning the Xeon Phi have been carried out through the VTune Amplifier XE 2015 profiling and analysis tool by Intel. Timing measurements on the Maia DFE were taken by measuring the

DFE-kernel time inlined within the host code using timestamps before and after the kernel call. Since the host code (in the CPU) is blocking, only the DFE kernel is active during measurements. The time includes the kernel execution (processing and DRAM data-exchange delay) and the activation delay of the FPGA device. This activation takes about 1 ms, which is negligible compared to the overall execution time that takes several seconds to several minutes in our test experiment. GPU kernel-time measurements were taken using the CUDA Event API.

6.2.1.1 TYPE-I experiments

Starting with the analysis for TYPE-I experimentation, in Figure 6.8 we plot the execution time of a single simulation time-step (50 μsec) for the most demanding use case, that of the RGJ with 100% connectivity density. Even though still not the most common case, a brain-simulation platform must support such high interconnectivity densities for certain TYPE-I experiments. The DFE exhibits the best performance for all tested network sizes. The Xeon Phi is a close second due to the local-memory delays and the less efficient use of its parallel threads: these network sizes are not large enough to provide sufficient parallelism for the Phi threads to be fully utilized. The GPU, on the other hand, has difficulties to cope with the computational intensity of the GJs, which involve mostly division and exponent FP calculations. Since each CUDA thread executes a single neuron, it cannot exploit any potential parallelism in the GJ calculation. This, alongside the fact that the CUDA threads are underutilized at such network sizes, impacts performance drastically.

The inefficiency of the Titan X GPU in performing the realistic GJ computations is clearly revealed in the SGJ case, next (see Figure 6.9). In this use case, which the most demanding GJ calculations are dropped, the GPU presents excellent scalability as the problem size increases, compared to the RGJ case. The Xeon Phi, on the other hand, still suffers from core-to-local-memory synchronization delays even though the actual calculations are much simpler now. The DFE needs to spend the same amount of operation ticks as in the RGJ case to evaluate the connection influence, even though it does enjoy gains in performance because of the simpler calculations involved (achieving higher operation frequencies, larger GJ computation parallelism and shorter pipelines). As a result, both latter accelerators show similar scaling properties to the RGJ case. In contrast, the GPU scores performance benefits in the SGJ case compared to the robust DFE for network sizes above 480 neurons.

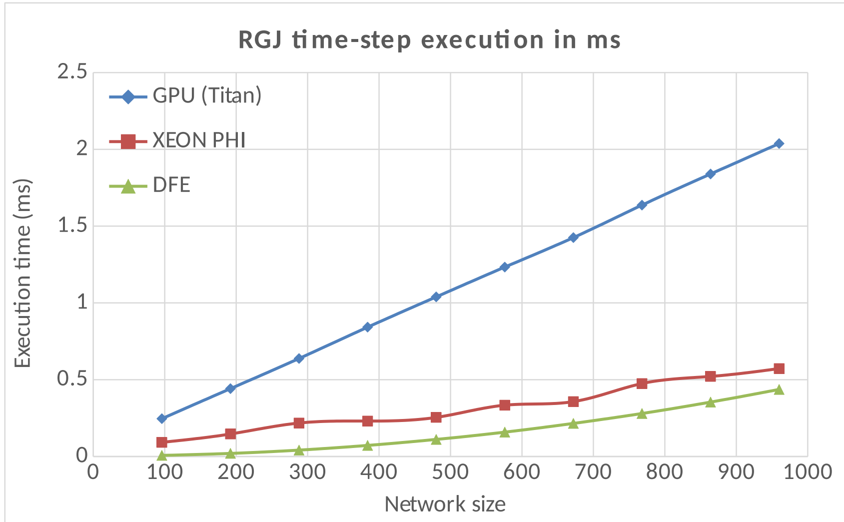


Figure 6.8: *RGJ execution time (TYPE I, 100% connectivity).*

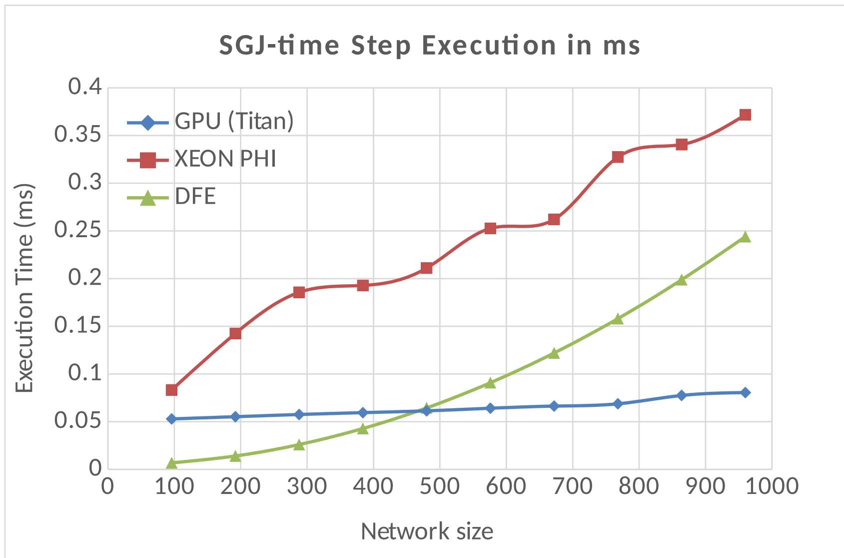


Figure 6.9: *SGJ execution time (TYPE I, 100% connectivity).*

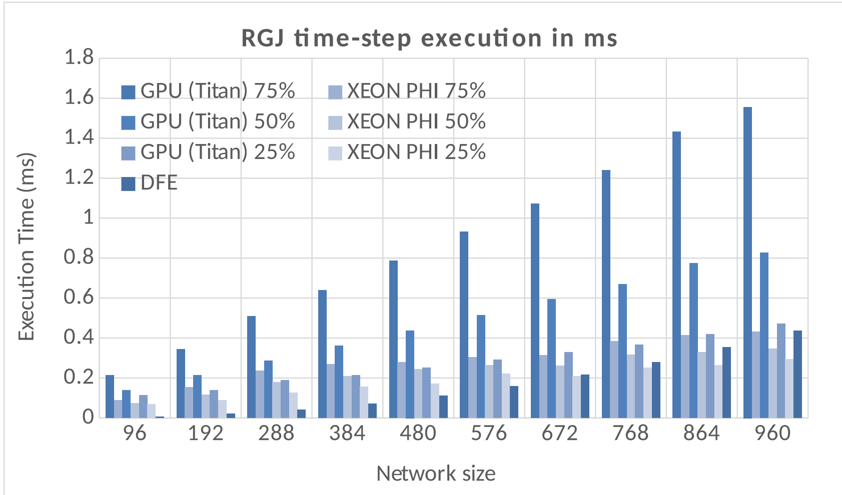


Figure 6.10: *RGJ execution time (TYPE I, <100% connectivity).*

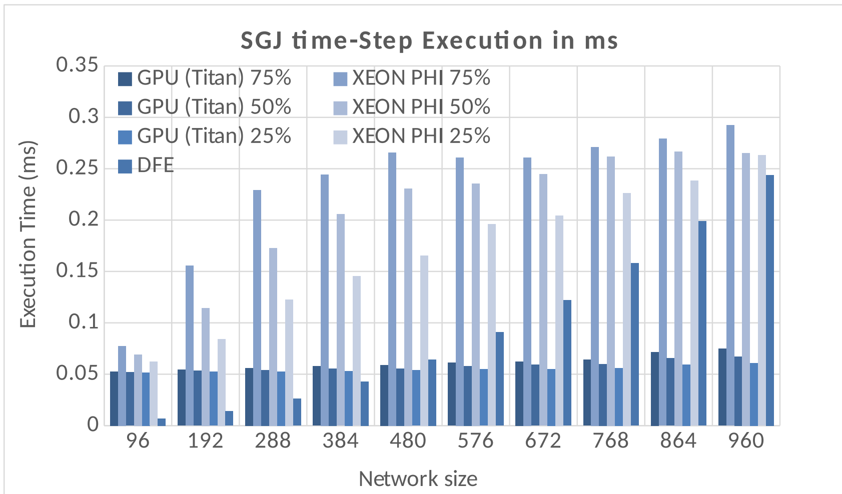


Figure 6.11: *SGJ execution time (TYPE I, <100% connectivity).*

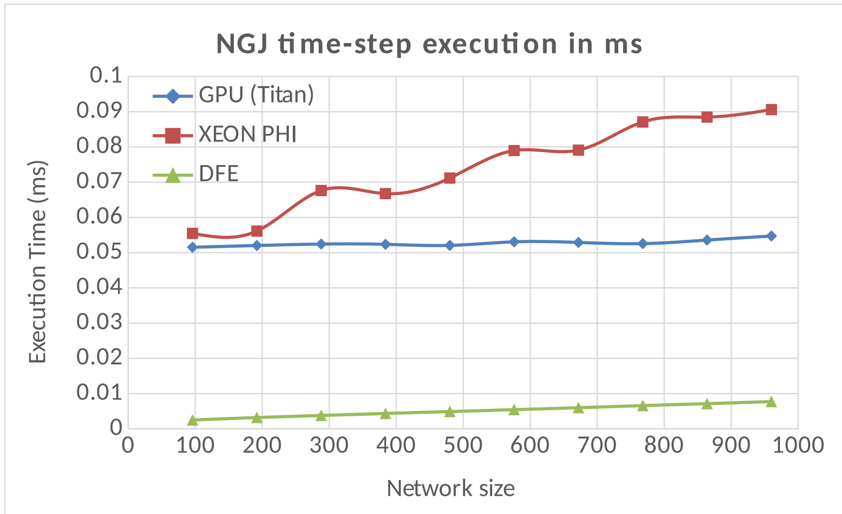


Figure 6.12: *NGJ execution time (TYPE I, no connectivity).*

Next, it is interesting to evaluate the three accelerators for connectivities of lower than 100%. Although not relevant for the DFE which maintains the same implementation for any connectivity density, smaller densities can influence the Xeon Phi and the GPU performance considerably. In Figure 6.10, we plot the execution time of a single simulation time-step for 25%, 50% and 75% connectivity densities, under the RGJ case. The GPU delivers significant gains but the inefficient GJ execution still causes it to perform worse than DFE, even though the latter operates as in a 100%-density simulation. The Xeon Phi, on the other hand, manages to achieve enough performance gains to become faster than the DFE for sufficiently large problem sizes; that is, sizes ≥960 neurons for 75% density, ≥864 neurons for 50% density and ≥672 neurons for 25% density.

Under the SGJ use case (Figure 6.11), we see similar trends as for the 100% SGJ use case: The GPU exhibits great scalability and is the best option for network sizes higher than 480 neurons. Besides, the DFE remains the most beneficial option for networks smaller than 480.

Under the NGJ case (no connectivity), for TYPE-I experiments, the results point to the DFE as the uniformly best option. In the complete absence of inter-neuron connectivity, the application becomes a purely dataflow workload, fully compatible for acceleration on a DFE, which is tailor-made for such cases, providing significant benefits over both the Xeon Phi and the GPU (see Figure 6.12).

Lastly, recall that for TYPE-I experiments, real-time speeds are often desired. Table 6.2 presents the real-time achievable networks for each use

Table 6.2: *RT-achievable network size (#cells) for each use case.*

Use case	DFE	Xeon Phi	GPU
RGJ (100%)	310	-	-
RGJ (75%)	310	-	-
RGJ (50%)	310	-	-
RGJ (25%)	310	-	-
SGJ (100%)	400	-	-
SGJ (75%)	400	-	-
SGJ (50%)	400	-	96
SGJ (25%)	400	-	96
NGJ	7,680	96	500

case. The results show that, for real-time experimentation, the DFE accelerator is the best option across the board. In contrast, and as mentioned in our previous analysis, the GPU and Xeon-Phi parallel threads tend to be underutilized at such small network sizes, even though most of the memory delays of using them are present. Thus the DFE – using fine-grain super-pipelined kernels – can achieve meaningful network sizes at real-time speeds under all use-case instances, according to the objective set in the introduction ($\geq$ 100 cells). For low ($\leq$ 50%) or zero densities, the GPU and Xeon Phi come close to the real-time objective.

6.2.1.2 TYPE-II experiments

For TYPE-II experiments, the trends under the RGJ case with 100% connectivity change significantly (see Figure 6.13). Here, the massive explosion of the GJ computations begins to stress the parallelization capabilities of both the Xeon Phi and the DFE. The DFE's efficient parallelization of the GJs relies mostly on its ability to unroll the GJ loop on the FPGA hardware, allowing for more iterations to finish per operation tick. However, the achievable unrolling factor is limited by the available chip area. For network sizes above 1,000 neurons, the DFE compiler is forced to reuse a lot of resources in time (as the unrolling factor is reduced with increasing network sizes). In effect, the dataflow paradigm gradually degenerates to a sequential execution, making the application less scalable on the DFE. The Xeon Phi follows a similar trend, as the communication overhead between cores (which are interconnected through a moderately efficient ring topology [32]) increases, leading to similarly diminished scalability. Opposite to these accelerators, GPU scalability is largely improved. The GPU is underutilized until all CUDA cores are used (3,072) simultaneously, so

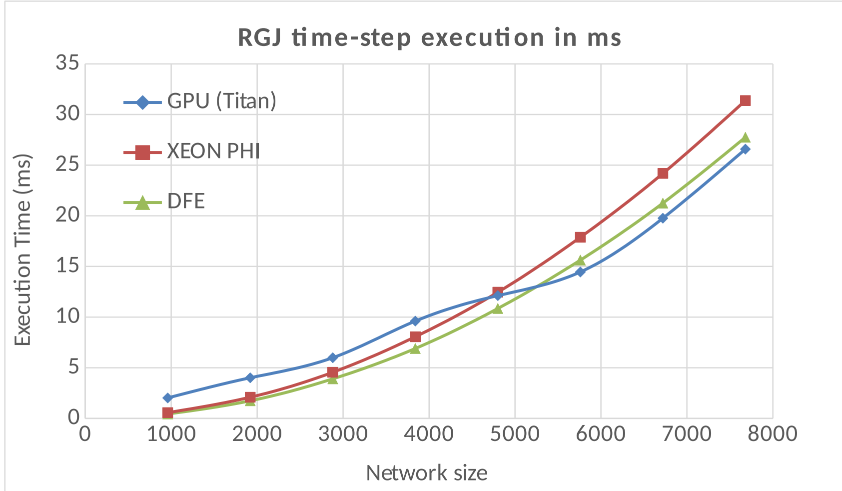


Figure 6.13: *RGJ execution time (TYPE II, 100% connectivity).*

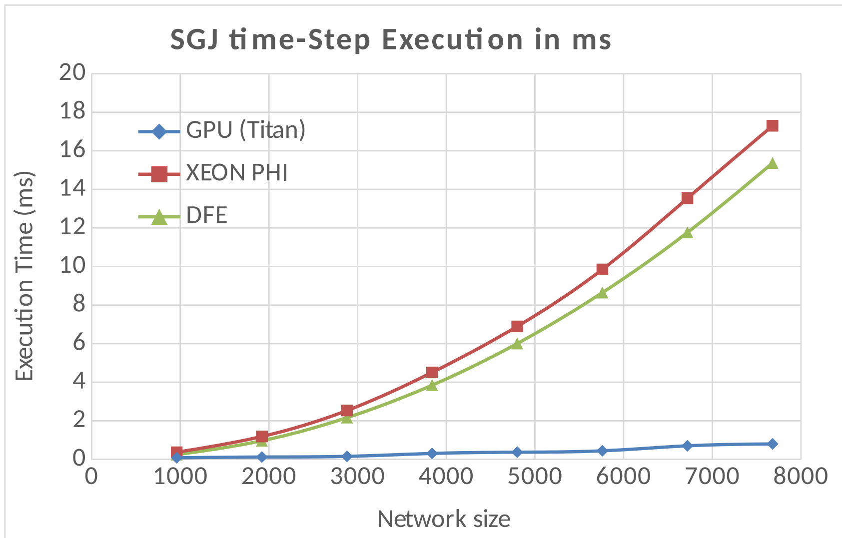


Figure 6.14: *SGJ execution time (TYPE II, 100% connectivity).*

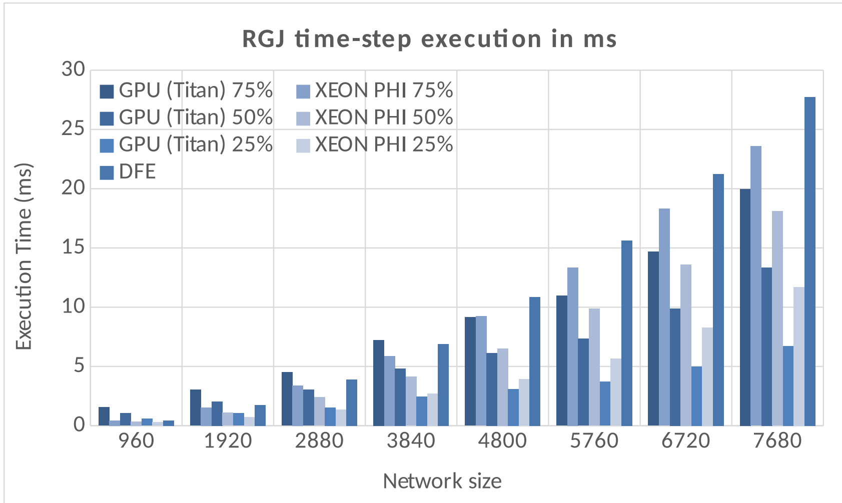


Figure 6.15: *RGJ execution time (TYPE II, <100% connectivity).*

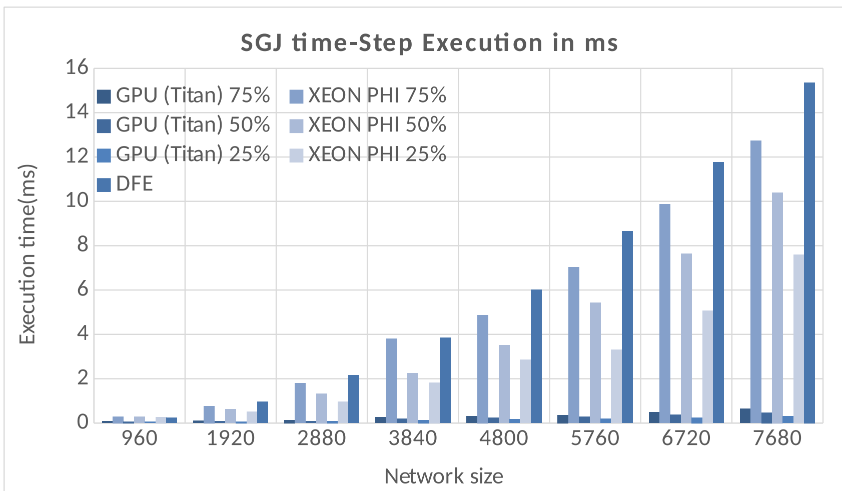


Figure 6.16: *SGJ execution time (TYPE II, <100% connectivity).*

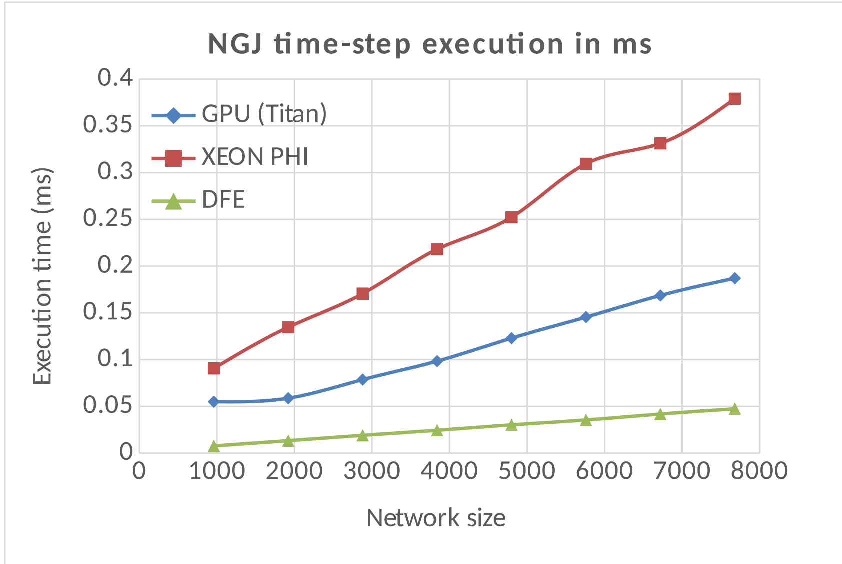


Figure 6.17: *NGJ execution time (TYPE II, no connectivity).*

for experiments over 3,000 neurons scalability is gradually improving. As a result, the GPU becomes the better performing solution (surpassing the DFE) for network sizes of 4,800 neurons and above.

For lower connectivity densities under the RGJ case, we observe similar trends, although the Xeon-Phi scalability is slightly better because of the lower interconnectivity (see Figure 6.15). Thus, the Xeon Phi retains the advantages it has for lower than 100% densities, compared to the DFE. Still, the effect of the inter-core communications is present, allowing for the GPU to overtake the Xeon Phi for network sizes above 4,800 neurons (for densities of 50% and 75%) and above 3,840 neurons (for 25% density).

Under the SGJ case, the DFE and Xeon Phi follow similar trends, although they are less pronounced (see Figures 6.14 and 6.16). As in the RGJ case, the GPU maintains its lead over the other two accelerator types for all tested network sizes and connectivity densities. Finally, in the NGJ case, the situation is the same as with TYPE-I experiments: The purely dataflow nature of the application allows the DFE to once more score the best performance across the board (Figure 6.17).

6.2.2 Accelerator-selection algorithm

The performance analysis discussed above can now be used to formulate a simple accelerator-selection algorithm for BrainFrame, automatically choosing the best-suited accelerator fabric based on the problem parame-

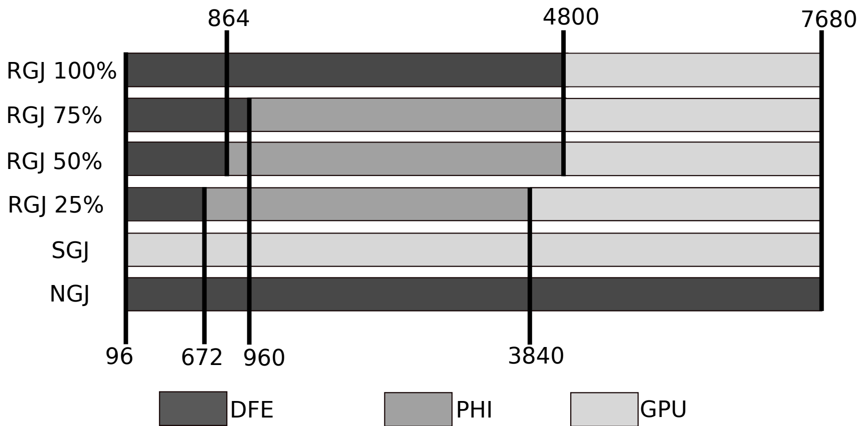


Figure 6.18: *BrainFrame accelerator-selection map for TYPE-II experiments. Selection is heavily (dependent on the experiment, involving all three accelerator fabrics. For TYPE-I experiments, the DFE is always the optimal choice (not shown).*

ters: mainly, connectivity detail (biophysically realistic: RGJ, simple: SGJ and not present: NGJ), density, and network size. Figure 6.18 shows the selection for our use-case instances. The RGJ case selection, which presents the most complex case in terms of accelerator choice, shifts between all three options depending on the connectivity density. For the SGJ case, the GPU is always the accelerator of choice, while for the NGJ case the DFE yields optimal results under all experiment parameters. Lastly, if the experiment is flagged as a real-time experiment, the algorithm exclusively chooses the DFE to accelerate the application, as it is the only clearly viable accelerator for real-time experiments.

As a simple example of how this selection can speed up experiments, we can assume a scenario where several batches of RGJ experiments need to be executed for various network sizes. Let us assume that each batch includes 5 experiments, each with gradually decreasing connectivity density (100%-75%-50%-25%-0%) and that each experiment in a batch simulates 40 seconds of brain time. The time saving in this example by using the BrainFrame system compared to homogeneous systems that integrate ***only a single*** accelerator type can be seen on Table 6.3.

The BrainFrame system can achieve significant benefits compared to the single-fabric systems that can range up to 86% execution-time reduction. On average, assuming the total runtime of all batches, the BrainFrame system can achieve 40% reduction compared to a DFE-only system, a 10.7%

Table 6.3: *Time savings (in minutes) with BrainFrame for the tested experimental RGJ batch scenario compared to three homogeneous-accelerator systems. The % savings are shown in parenthesis.*

	BrainFrame vs.		
Network Size	**DFE -only**	**Titan X -only**	**Phi -only**
384	0.0 (0.0%)	24.2 (86.2%)	8.6 (68.7%)
960	3.2 (13.8%)	45.8 (69.5%)	3.0 (12.8%)
5,760	1.9 (43.4%)	54.5 (27.0%)	10.7 (6.8%)
7,680	591.7 (40.0%)	1.9 (0.2%)	246.6 (21.7%)
All batches	707.7 (40.0%)	126.4 (10.7%)	268.9 (20.2%)

Table 6.4: *Energy savings with BrainFrame for the assumed experimental scenario compared to three homogeneous-accelerator systems. We assume nominal (TDP) power figures (see Table 6.1).*

	BrainFrame vs.		
Network Size	**DFE -only**	**Titan X -only**	**Phi -only**
384	0.0%	91.4%	82.5%
960	38%	86.4%	64.9%
5,760	51.3%	60.9%	55.1%
7,680	23%	20.4%	43.8%
All batches	27.3%	32.6%	45.9%

reduction to a GPU-only system and a 20.2% reduction compared to a Phi-only system.

If we consider the nominal scenario of TDP power consumption we can also present an estimation of the energy benefits of using BrainFrame compared to the single node accelerators for our example(Table 6.4). The energy saving for specific batches on the example are between 20.4% to even about 91.4%. For the all experimental batches the energy saving is between 27% to 45.9%. Reduction in energy consumption can greatly reduce operation and maintenance cost especially within a datacenter environment.

Although these figures will vary based on the particular accelerator instances used for the experiments, they give a rough estimate of the time savings that can be obtained by carefully selecting the accelerators for the various experiments. This selection can be easily extended/updated as new features and more generalized model libraries are added for acceleration (making the selection predictive for general cases) or as each acceleration technology is updated in the future.

6.3 Multi-node potential of BrainFrame back-ends

Even the most sophisticated models in computational neuroscience tend to include at most a few brain areas at a time due to both computational constraints and biological complexity. Nevertheless, there is wide consensus that most high-level functions of the brain require the integration of a large number of areas, which is very time sensitive. During functional behavior, the concurrent activity of multiple brain areas is widely and asynchronously exchanged. Coherent integration of time-sensitive streams is a fundamental problem in neuroscience. Complex and coordinated behaviors such as multi-limbed sensorimotor control depend imminently on this integration of multiple activity streams across multiple brain areas. This means that the understanding of biological brain function demands an understanding of time sensitive integration of multiple processing streams. Motor, cognitive and autonomic systems are brought together in tandem, and are highly contextualized. Even more, embedded and embodied models will eventually need to be computed in the body-environment loop.

As the field progresses in its task of modeling the brain, large-scale models with multiple areas to be integrated become feasible and even tenable. We might safely acknowledge that large-scale brain modeling attempts are scientific steps in the long road towards understanding brain function.

However, for the production of dynamical models of the functioning brain, there remain substantially unaddressed computational challenges. Centrally, the computational challenges of coordinating multiple brain areas

will be dependent on our ability of simulate the activity of multiple brain areas simultaneously, not excluding the centers at the core of the brain. A glimpse at modeling databases [151] will show that very few of the midbrain and hindbrain centers in the brain that have been integrated in large-scale models of brain dynamics. These centers often exhibit architectonic and physiological features that render them veritable bottlenecks in the future of highly plausible large scale brain simulation.

The constant issue with running such simulations is the computational demands that stem from their massive scale [152]. Traditional simulators using single node acceleration, although quite mature and powerful, very often come short in terms of performance, making experiments (often prohibitively) time-consuming unless massive (in both size and cost) supercomuters are utilized. Even using a single HPC accelerator is not enough to support the scale required for massive networks, if highly accurate models are used.

These limitations mean that the ideal HPC platform for large-scale brain simulations requires not only technology heterogeneity and programming constructs familiar to the neuroscientist that are portable and accessible, but also multi-HPC device support. Thus, it is highly relevant that we explore the potential of BrainFrame back-ends for multi-device support.

6.3.1 Mutli-DFE implementation and evaluation

The main challenges of the multi-DFE implementation is the synchronization and communication between the DFE-cards. The simulated network is split equally between the DFEs. The calculations of the network subsets are largely independent from each other, except for the dendrite voltages of the neurons that are inter-connected through gap junctions and are partitioned on different cards.

These need to be exchanged as they are produced between the DFEs, to be ready for use on the next simulation step. Since the FPGA hardware is statically allocated to avoid synthesis between different experiments and all-to-all interconnectivity needs to be supported, the system and data exchange is designed assuming all-to-all interconnectivity even if fewer than all gap junction connections are present in a given experiment. The transfer is implemented using the MaxRing, infrastructure on the DFE board that provides a direct connection between DFEs, using the spare PCI-E lanes, bypassing the host that would issue extra delays if data would be exchange through it. When 2 DFEs are used, the operation ticks required for the data transfer are equal to the ticks required to finish the simulation step computation. Thus, data exchange is overlapped with the actual execution

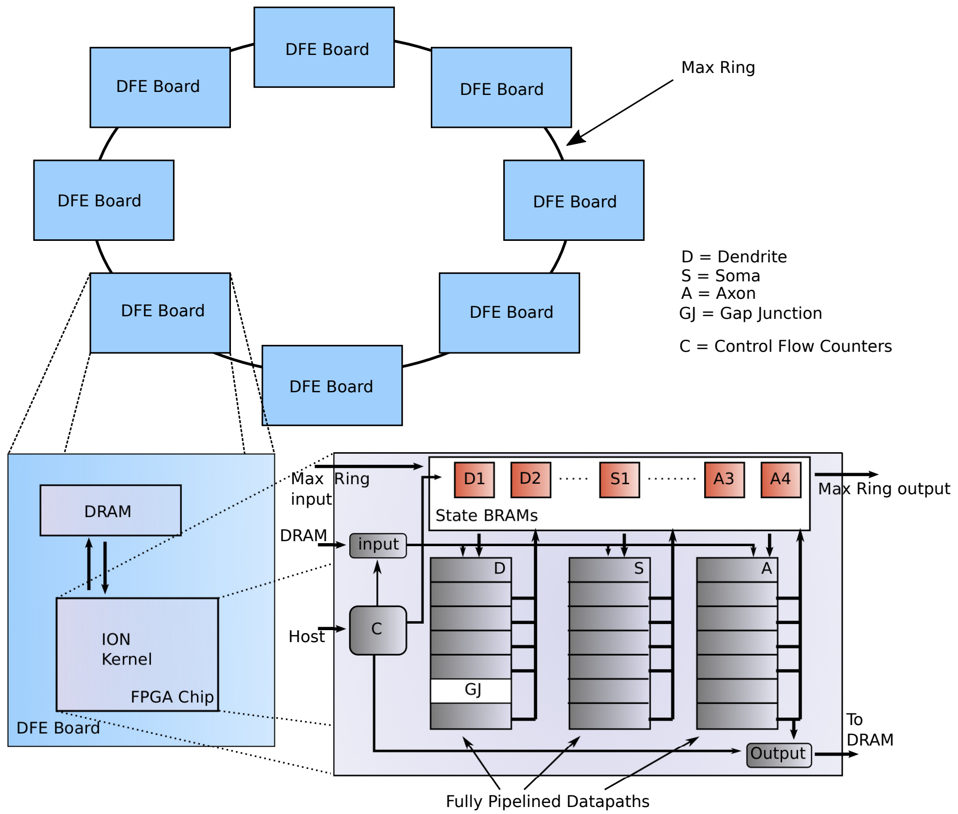


Figure 6.19: *Architecture of 8-DFE design.*

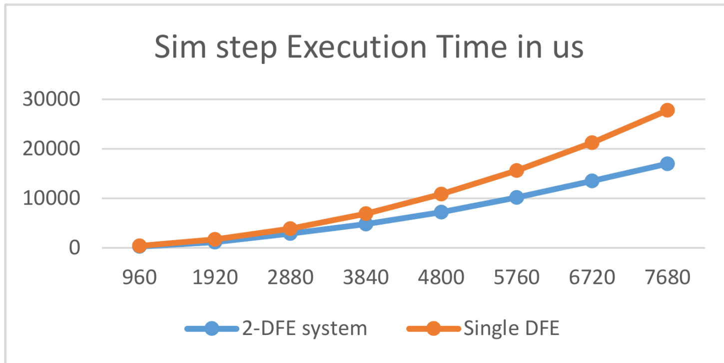


Figure 6.20: *Sim-step execution time for RGJ case vs network size (single vs. 2-DFE run).*

of the simulation step computations, keeping the overhead in performance small, especially for larger networks. Since only a small amount of extra BRAMs need to be used as buffers for the data values coming from the other DFE, the 2-DFE system still has enough on-chip RAM to be able to support 7680 neurons per card as in the single DFE version, increasing the maximum supported network to 15360 neurons.

For an 8-DFE system, things are be more challenging. Data between DFEs are transfered as a carousel through the DFE ring (Figure 6.19). As the ticks required for the data exchange between 8 DFEs are more than the actual computations done in parallel within each DFE, the 8-DFE system requires an extra data transfer phase after the computation to complete the data exchange before the next simulation step can begin.

The transfer from the immediate neighbor DFEs can be overlapped with computation but the rest of the data are exchanged at the same time as the DFE pipelines are being flushed. Depending on the problem size either the flushing or the data transfer phase might require more ticks. The extra operation ticks are added at the end of each simulation according to which of the two operations takes longer. Since now each DFE needs to have data buffers for the data coming from 7 other DFEs, the on-chip memory is stressed significantly more. As a result preliminary synthesis results in each DFE only supporting a maximum network of 3840 neurons, making the maximum supported network for the 8-DFE system at 30720 neurons.

We evaluate the multi-DFE device using a larger scale network and the RGJ and NGJ IO use cases as the worse and best case scenarios, respectively, in terms of computational complexity. In Figure 6.20, we can see

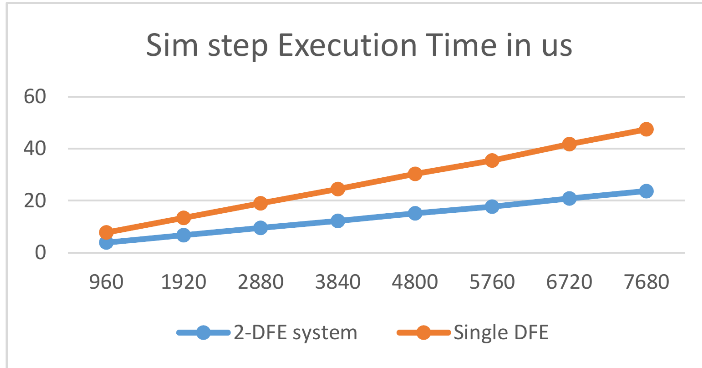


Figure 6.21: *Sim-step execution time for NGJ vs network size (single vs. 2-DFE run).*

the performance comparison between single and 2-node executions for the Maia (MAX4) DFE, all based on the RGJ case. The 2-DFE system shows increased efficiency for longer networks. The speedup of the 2-DFE system compared to the single node is between 1.45× to 1.68×. This is to be expected when higher problem sizes (i.e. network sizes) are demanded, whereby doubling the effective FPGA fabric provides significant performance enhancements. Comparing the same network sizes for the NGJ case (Figure 6.21), we can see an almost perfect scaling as we move to longer networks. As the NGJ has no connectivity modeled, there is no reason for communication between DFEs (over the MaxRing). Thus, performance benefits are due to the lack of inter-DFE communication delays, a shorter pipeline depth and no need for pipeline flushing.

The 8-DFE using the MaxRing (required for RGJ experiment execution) was developed but not validated due to the lack of (at the time) proper simulation and validation tools in the maxeler software infrastructure. The static nature of the FPGA hardware, though, allows us to make quite safe estimations on the synthesized design's performance on the RGJ case. In Figure 6.22 we can see that speedup between the 8-DFE estimated performance and 2-DFE configuration is maximally around 3.3x. Yet, once more, the larger DFE instance offers better scaling properties to the problem. The NGJ case again is devoid of such delays and the DFEs work practically independently from each other with perfect partitioning of the network between DFEs (thus these run could be fully validated and run on

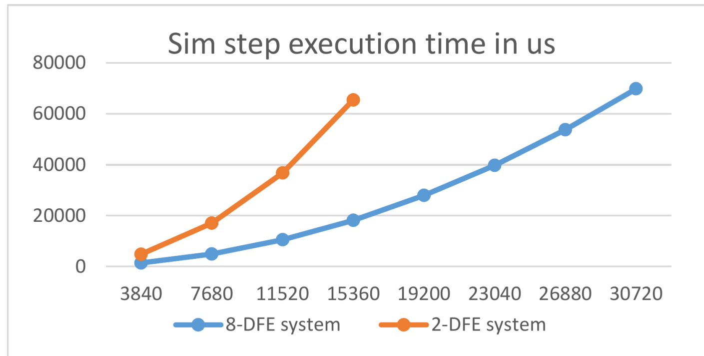


Figure 6.22: *Sim-step execution time for RGJ case vs network size (2-DFE vs. 8-DFE run).*

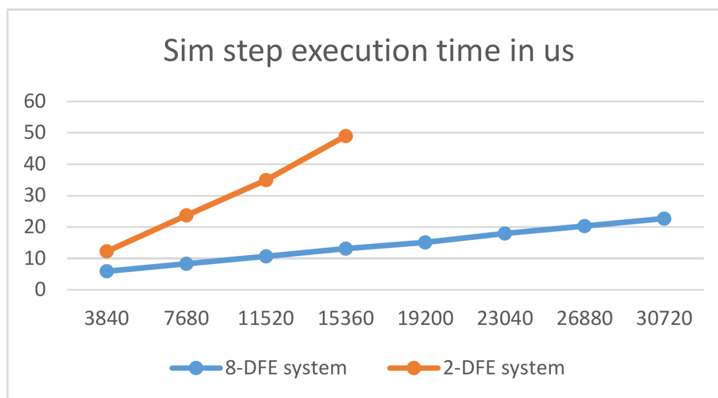


Figure 6.23: *Sim-step execution time for the NGJ case vs network size (2-DFE vs. 8-DFE run).*

the actual hardware), thus resulting in a speedup of about 3.8× compared to the 2-DFE design (Figure 6.23).

6.3.2 Mutli-node PHI implementation and evaluation

The exploration on the multi-PHI implementation was made in a more advanced Knight's Landing (KNL) models of Xeon PHIs. The KNL processors utilized in this implementation feature 72 cores, each able to dispatch up to 4 instruction streams simultaneously [153]. Thus, when employing n KNL processors, there is a degree of thread-level parallelism equalling $n \times 72 \times 4 = n \times 288$. In addition, it should be noted that each thread utilizes, when applicable, the AVX-512 instruction set; each core has two 512-bit-wide vector processing units (VPUs) which allow for vectorized instructions to operate on multiple data simultaneously. This Single-Instruction-Multiple-Data (SIMD) level of parallelism, combined with massive thread counts on each platform, enables strong computational performance when the target application is parallelized and vectorized properly for the KNL.

In the case of our implementation of the simulator (6.24), the thread-level computational capabilities of the ensemble of KNL processors are divided in groups and assigned to different MPI ranks [154]. This is necessary for the multi-KNL implementation; different processors do not share memory and thus, there is a need for communication between cores that co-operate on computing neuronal network states. Based on this fact, when employing n of processors, the minimum total number of MPI ranks the application needs to spawn is also n, one per processor or DRAM island; however this case has not proved optimal performance-wise in our studies.

Each MPI rank is responsible for the communication needs of a group of KNL threads; when spawning a total number of k MPI ranks and employing n KNL processors, each of the MPI ranks "holds" $n \times 288/k$ KNL threads. We have chosen to spawn $k = n \times 4$ MPI ranks, meaning each of the n KNL processors spawns 4 MPI ranks and each MPI rank handles the communication processes of 72 threads, which operate on the shared memory of a single KNL processor using the OpenMP library [155]. This middle-of-the-road approach to the ratio of MPI ranks to OpenMP threads coincides with previous decisions on the 1st generation Xeon Phi KNC [156] [157].

On an algorithmic level, OpenMP threads operate on different parts of the neuronal network. Each neuron in the network is assigned to a single thread in order to be processed. Each of the $n \times 288$ threads handles an equal number of neurons, in order for the computational workload to remain balanced. Thus, if the simulated network consists of a neurons and is being

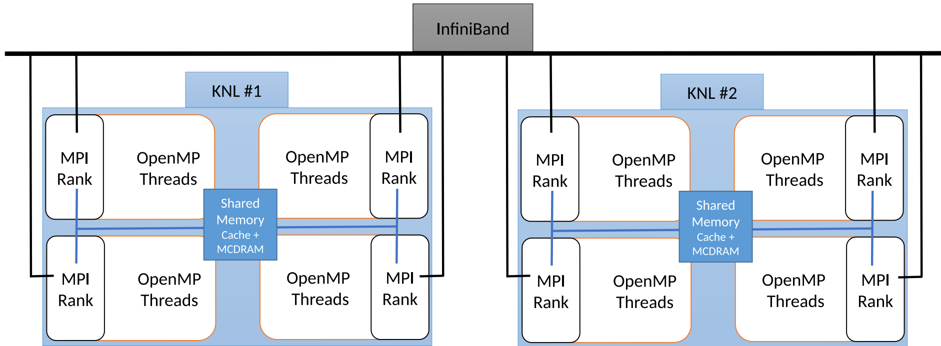


Figure 6.24: *Multi-KNL implementation.*

processed by n KNL processors, each thread handles the computational needs of approximately $a/(n \times 288)$ neurons.

In each simulation step, Gap Junctions need the dendritic membrane voltage levels of the participating neurons in the connection in order to be computed. In the worst-case scenario mentioned above, each MPI rank needs to broadcast the levels of dendritic voltage of each neuron they handle - a total of $a/(n \times 4)$ floating-point values. This is achieved by using MPI's broadcast function (**MPI_Bcast**). The MPI rank packs the necessary values in a buffer after its assigned OpenMP threads have calculated the up-to-date voltage levels. In the case of less than 100% connectivity density, the buffer omits values that are not necessary to any of the other $n \times 4 - 1$ MPI ranks.

After the MPI rank completes its **MPI_Bcast** function, it receives the rest of the $n \times 4 - 1$ MPI ranks' broadcasts. It then processes the contents of each received dendritic voltage buffer and dispatches their values to each of its assigned 72 threads. In the worst-case scenario, each of the 72 threads needs access to the full content of the received $n \times 4 - 1$ buffers; in this case, each rank essentially gets updated on the entirety of the rest of the network in every simulation step. The buffers are accessed by the OpenMP threads and the calculation of their Gap Junctions can be completed. The OpenMP threads of each MPI rank can then process the rest of neurons' data, such as their somatic and axonal ion channels, in order to complete all necessary computations for the given simulation step. The process begins anew in the next simulation step with a **MPI_Bcast** function from the MPI rank.

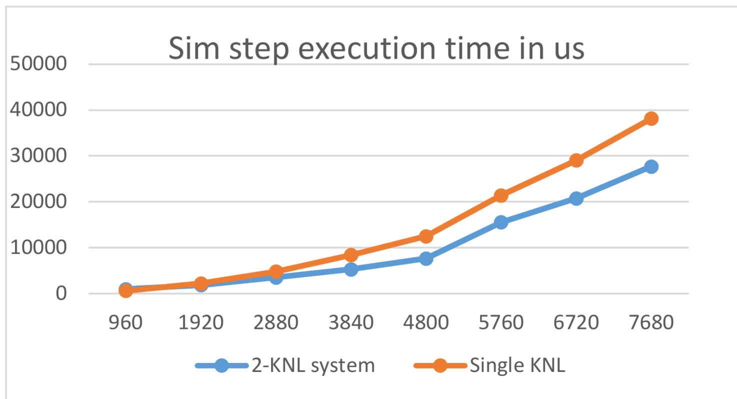


Figure 6.25: *Sim-step execution time for RGJ case in PHI KNLs (single vs. 2-KNL run).*

We repeat the above experiments using the Knights Lightning (KNL) version of Intel Xeon-Phi platform. It must be noted that the PHI KNL can support simulations of several millions of neurons. Additionally, the network connectivity density can have an effect on the performance, as opposed to the DFE where everything is statically allocated. Thus, in order for this KNL evaluation to be comparable to the DFE measurements we assume all-to-all connectivity between neurons and only measure for the same network sizes as the DFE measurements, even though the KNLs can support up to 2 million neurons.

In Figure 6.25, we can see the execution time for the RGJ case when using single and 2-node KNL configurations. The 2-node speedup reaches up to 1.4×. Although it is expected to improve for higher problems sizes, the constant and non-deterministic nature of the intercore communication on each KNL leads it to have poorer scalability than the DFE (e.g. compare to Figure 6.20).

In the NGJ case, the KNL – although showing good scalability is greatly outperformed by the DFEs (Figure 6.26; compare to Figure 6.21). The DFEs are specifically designed for such problems as the NGJ case, while in the PHI case – even though communication for application data is not present – delays for inter-core coordination and memory transfers are still being suffered. Similar dynamics can be observed when comparing the 2-KNL with the 8-KNL system. PHI scalability is greatly improved with a maximum speedup of 3.13× for the largest comparable problem size (Fig-

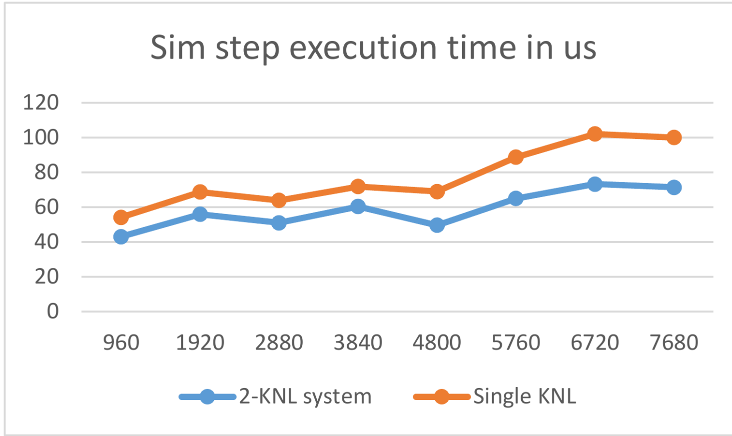


Figure 6.26: *Sim-step execution time for NGJ case in KNL system vs network size (single vs. 2-KNL run).*

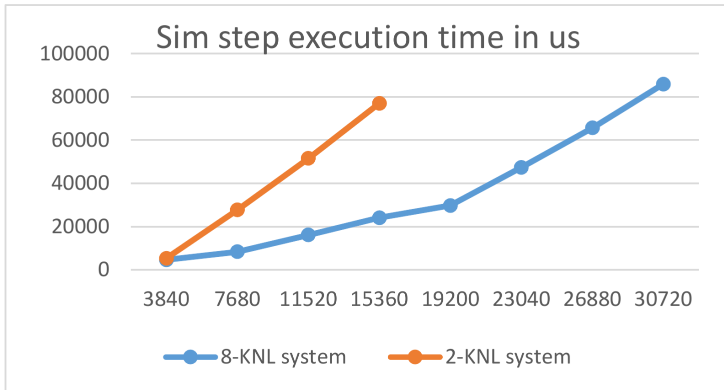


Figure 6.27: *Sim-step execution time for RGJ case vs network size (2-KNL vs. 8-KNL run).*

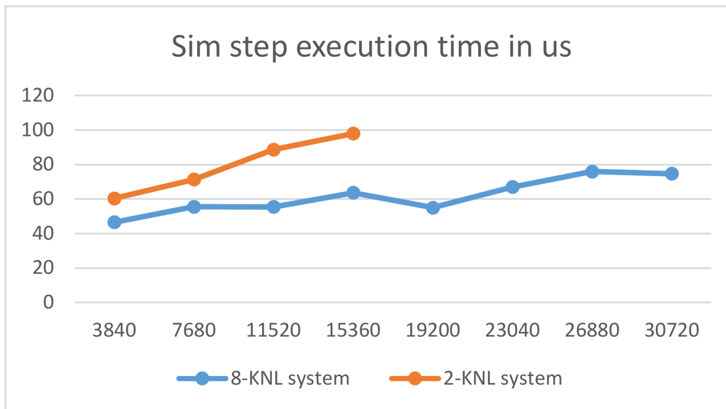


Figure 6.28: *Sim-step execution time for NGJ in KNL multi-node systems (2-KNL vs. 8-KNL run).*

ure 6.27), while for the NGJ case the situation is greatly in favor of the DFE (Figure 6.28).

To show the KNL support for larger problem sizes than the DFE, we conduct experiments for populations of millions of neurons, as well. An important distinction in how a neuronal network is implemented in hardware and software can be observed when evaluating how a non-fully-connected network operates in either case. As mentioned above, it has a clear effect on KNL performance: Figure 6.29 and Figure 6.30 depict scalability plots for the case of PHI KNL processors handling networks of varying sizes and a set number of synapses per neuron.

The difference between the two figures lies in how the synapses are distributed throughout the network. In Figure 6.29, synapses follow a Gaussian distribution pattern in which neurons are most likely to form bonds with other neurons that are physically proximal to one another. In Figure 6.30, synapses do not follow this distribution and are created uniformly throughout the network, ignoring the parameter of neuron proximity in the network. The KNL processors handle the case of Gaussian distributions in a much more efficient manner than the case of uniformly distributed synapses. This can be explained by considering that the synapses present the communication bottleneck between the KNL cores, as well as between the different KNL dies in multi-KNL configurations. When synapses connect proximal neurons, data locality is increased and the communication bottleneck is a smaller obstacle to overcome. For uniform distributions,

however, the bottleneck becomes a much more important factor and hinders good simulator scalability and overall performance.

As a result, Figure 6.30 shows a severe increase in execution times when compared to Figure 6.29 by almost an order of magnitude. Furthermore, the case of the uniform distribution shows poor gains from employing 4 or 8 KNL processors over utilizing 1 or 2, particularly when considering networks of large size and 1,000 synapses per neuron (bright yellow line in the figures). These differences would not be observed in the case of hardware implementation, since synapse placement is already assumed to be at 100% density for any network, thus eliminating any considerations caused by connected-neuron proximity.

6.3.3 Mutli-node GPU implementation and evaluation

The multi-GPU exploration was conducted on the Cartesius cluster implemented by SURFsara [158]. The GPU accelerator island consists of 66 Bullx B515 GP-GPU accelerated nodes. Each one of these nodes consists of 2 × 8-core 2.5 GHz Intel Xeon E5-2450 v2 (Ivy Bridge) CPUs/node, 2 × NVIDIA Tesla K40m GPUs/node, and 96 GB of memory/node. The Tesla K40m is a server-grade PCIe accelerator board consisting of a single GK110B GPU with NVIDIA Kepler architecture, which fully supports Remote Direct Memory Access (RDMA). From the node topology, it became clear that the GPUs are not connected to the same PCI root complex. Each node contains two CPUs which have their own root complex. Each CPU has a GPU connected to it, separating the memory address space. Intra-node communication between nodes using the improved RDMA technology on Cartesius is, thus, not possible. Luckily, this does not affect our core experiments which are focused on RDMA inter-node communication. Also, from reported benchmarks on Cartesius, it is observed that, even for larger messages, OpenMPI has the lowest latency and could acquire the highest bandwidth. The implementation uses the GPUDirect infrastructure.

GPUDirect is a software technology providing high-bandwidth and low-latency communications directly among multiple (disaggregated) GPUs in a cluster. The main two technologies under this umbrella are Peer-to-Peer (P2P) for intra-node and Remote DMA (RDMA) for inter-node communication. P2P transfer technology allows for fast intra-node transfers but does nothing for inter-node memory transfers. It allows for buffers to be copied directly between the memories of GPUs. It employs Unified Virtual Addressing (UVA), which enables the host memory and memory of all GPUs to be combined into one large virtual address space [16]. A prerequisite for P2P is that the source and destination device need to be

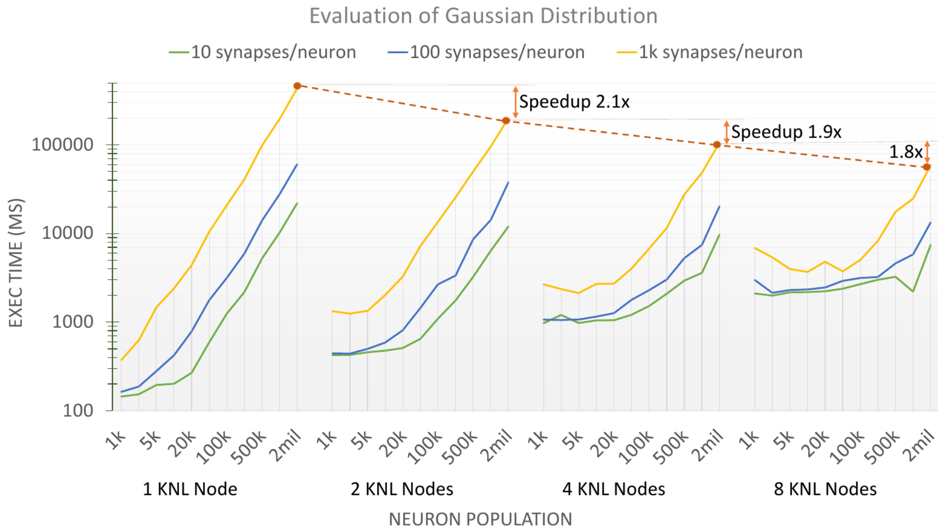


Figure 6.29: Gaussian distribution in formed synapses throughout the neuronal network, for different neuron counts and synapses per neuron. The evaluation shows scalability for 1, 2, 4 and 8 Xeon-Phi KNL configurations. High scalability is shown for the heaviest network workload of 2 million neurons/1,000 synapses per neuron.

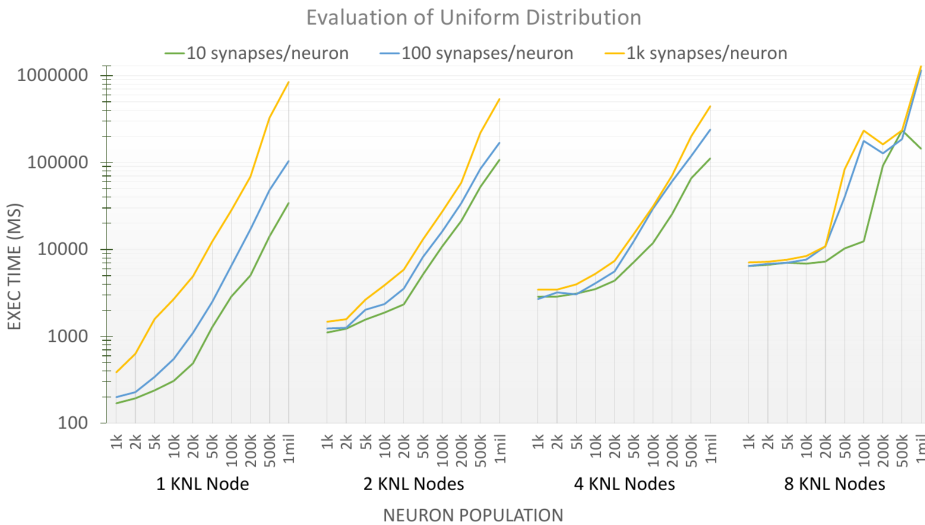


Figure 6.30: Uniform distribution in formed synapses throughout the neuronal network, for different neuron counts and synapses per neuron. The evaluation shows scalability for 1, 2, 4 and 8 Xeon-Phi KNL configurations. Scalability is observably lower, particularly for 8-KNL configurations.

attached to the same PCIe root complex, the application needs to be 64 bit, and the operating system must enable the Tesla Compute Cluster option. RDMA allows memory transfers between GPUs in a cluster over PCIe or Infiniband. The GPU is enabled to send data directly to a remote system without any intervention from the CPU. This technology is available for the Tesla accelerator cards, starting from the Kepler architecture onward.

The multi-GPU design was based on the single GPU design of Section 6.1.2.3. The execution requires 4 distinct phases:

- Connectivity-network Generation: The process of generating the network topology for the simulation. For the purposes of this design, the design was tested with Gaussian and Uniform network topologies.
- Connectivity-network Dispersal: The target cell for which neighbor cells are selected always resides on a local GPU. Neighbor cells may reside on the local GPU or on remote GPUs. Once the connectivity network with the neighbor cells connected through GJs has been set up, the remote connections need to be dispersed among the GPU-world in such a way that the computed dendritic values in each time step are sent to the correct GPUs.
- Cell Computation: When all the cell connections have been dispersed, we can enter the actual simulation, which is a transient process. Now, the axon, soma, and dendrite voltages of each neuron are computed.
- Dendrite Communication: In every simulation step, the dendrite voltages of the neighboring cells on remote GPUs are required by the local GPU to perform the neuron computations. Thus, after each cell-computation step, the newly calculated dendrite voltages are communicated across the GPU-world.

Execution time is used as the performance metric for our evaluation (Figure 6.31). For both uniform- and Gaussian-connected networks, performance is calculated by comparing results (i) for 1M cells when scaling GPU-world size from 1 to 32 GPUs and per neuron connections from 10 to 1k (weak-scaling experiments) and (ii) for a high network connectivity of 1k when scaling network sizes from 65k to 4M cells and for GPU-world sizes from 1 to 32 GPUs (strong-scaling experiments). The results mentioned for 4M cells have been extrapolated based on the ratio of the number of neighbors to execution time and have been added for qualitative comparison. Both network connectivity types are compared to highlight the different impacts on performance.

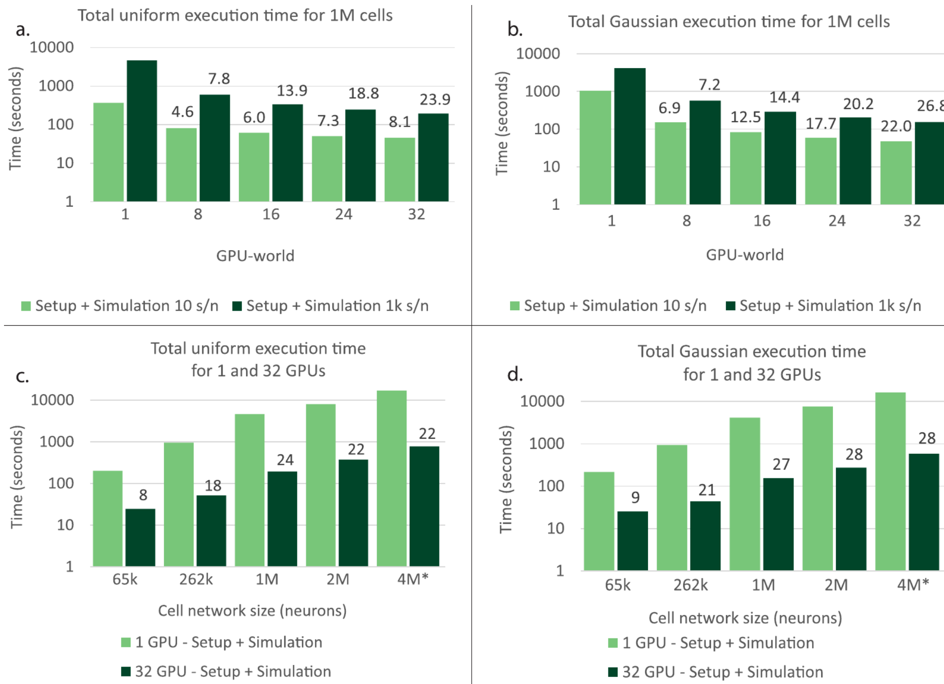


Figure 6.31: *Obtained total execution times when scaling GPU-world size for 1M neurons for densities of 10 and 1k synapses/neuron (s/n). For (a) uniform and (b) Gaussian connectivity networks. (c) Obtained total execution times when scaling neuron network size with a per neuron connectivity of 1k synapses/neuron (s/n) for 1 and 32 GPU(s) for uniform and (d) Gaussian connectivity networks. The numbers in the graphs are the speedups of the results in relation to the single GPU-world execution. * 4M cell measurements are extrapolated.*

From the evaluation of the multi-GPU implementation it is observed that, for larger densities, a uniform network displays a lower overall speedup when compared to its Gaussian counterpart. The dendrite-communication phase becomes the performance bottleneck of the simulation process as traffic among GPU nodes increases in volume. However, this communication overhead does not dominate the overall execution while scaling network sizes is tractable. The IO-model workload has proven to execute on as many GPUs as available and no erratic behavior (as in the case of the multi-KNL design) has been noticed. The multi-KNL design shows erratic behavior for large uniformly distributed, and even more for Gaussian-distributed networks, due to the performance penalties incurred for keeping the many CPU caches of the KNL architecture coherent.

6.4 flexHH: BrainFrame library for HH-based neural simulations

As mentioned in Section 6.1 for the BrainFrame framework to be useful in practice, neuroscientists must be able to develop their own models within BrainFrame using generic libraries. 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. Achieving such flexibility is very challenging especially on FPGA-based platforms, such as the DFE used in BrainFrame.

Thus, we develop flexHH [159] as the first prototype of such libraries (implementing HH neuron model libraries), validating against a reference design, and evaluate it. The library consists of five HH-model variants. Each of the five implementations supports a different number and type of features which can be user-specified at simulation startup; i.e. not at design time, at marginal performance cost compared to hard-coded designs. We begin with the DFE back-end as the most challenging platform for such libraries compared to software-based solutions. The contributions of this work are as follows:

- A scalable hardware library of accelerated, parameterizable and NeuroML-compliant [160] HH-model implementations which offer high performance gains. NeuroML is a XML based description language that aims to provide a common data format for defining neuron models and is compatible with a number of other simulation packages like PyNN and NEURON.

Table 6.5: *Overview of supported model features per flexHH kernel.*

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

- A set of crucial model features: custom ion gates, gap-junctions connectivity and multi-compartmental neurons.

6.4.1 The DFE implementation

This first version of flexHH targets a single DFE node. Besides the standard HH-model, three crucial extensions are added: user-defined ion gates, gap-junction interconnectivity, and support for multiple compartments. Each feature introduces a hardware-resource overhead that is subsequently translated to a performance overhead on the DFE technology. As a result, flexHH provides *five* different instances (or kernels), each incorporating more or less a superset of features compared to its predecessor (Table 6.5). The library thus provides the user with the choice of using simpler model instances (if all features are not required) with a benefit in performance or maximum network capacity. The simplest flexHH kernel (*HH*) supports the basic HH-model. The *HH fully featured (HH+custom+multi+gap)* kernel supports all extended HH features needed to simulate the complete IO-model use case.

In case multiple cell compartments are not supported, a single compartment coincides with a whole cell, and the terms are interchangeable. The ODE systems implemented are represented by state variables comprising membrane potentials of the compartments ($V[i]$) and gate-activation variables ($Y[i]$)[1], where i is the index of the variable. The index can be a combination of multiple integers; e.g. to represent gate h of compartment k of cell j, the index (j, k, h) can be used. Those state variables – which are single-precision floating-point variables – are updated as described in Algorithm 1.

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

Algorithm 1 HH-model evaluation

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

For each simulation, the solver is invoked for updating the neural network for a predefined number of steps N_{steps} and with a time step dt. For each gate (in N_{gates}) of each compartment (in N_{comps}) of each cell (in N_{cells}) – across simulation steps –, an `updateY` function is called which iteratively updates the values of the gate-activation variables $Y[i]$. For each compartment, a second function `updateV` updates the compartment's membrane-potential value $V[i]$.

Because of the resource limitations of the FPGA hardware, there is a maximum number of cells or compartments ($N_{comps,max}$) and a maximum number of gates per compartment ($N_{gates,max}$) that a kernel can support. Thus, the resources are *statically* allocated to those maximum values and any model instantiation with less or equal requirements can be used without the need to re-synthesize the kernel. Static allocation has the added benefit of providing performance guarantees, a crucial property for modern HPC deployments.

All equations in `updateV` and `updateY` are described by parameters which are kept constant during a single experiment. As flexHH has been designed to support heterogeneous neural networks, the constants can vary per level of *neuron modeling hierarchy*. Depending on the kernel instance, the functionality of the equations differs according to the supported features.

The computing model of the DFE consists of a host CPU which establishes in-/out- data streams to the DFE board, as shown in Figure 6.32 (top view). The DFE comprises on-chip, fast BRAM memory and on-board, slower DRAM memory. To adhere to the dataflow paradigm, the `updateV` and `updateY` parameters are streamed to the DFE kernel at appropriate times. However, the amount of data needed to be transferred is too large to fit into the FPGA BRAMs and, therefore, those parameters are on the on-board DRAM. In contrast, the state variables are updated frequently and, therefore, are stored in the BRAMs to reduce transfer latencies. Additionally, as the state variables describe the behavior of the

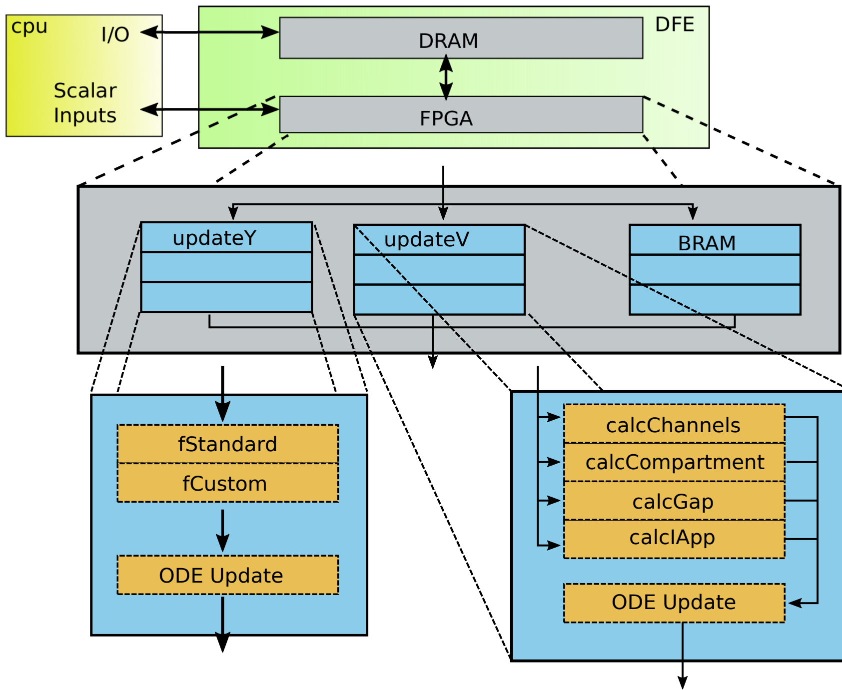


Figure 6.32: *Schematic overview of implementation on the DFE.*

model, all the state variables are sent to the on-board DRAM as output. Input/Output data transfers between board and host only occur before and after a simulation run.

A schematic overview of the HH kernel, as deployed on the DFE, can be seen in Figure 6.32 (bottom view).

The temporal derivative of the voltage Vi of an HH compartment i is calculated through Equation (6.1) where the currents $I_{mc,i}$ and/or $I_{gap,i}$ are the currents received from the inter-compartment connections and from the gap junctions, respectively. These parameters may be omitted if the executed instance does not support the respective features. $I_{channels,i}$ is the current received from the leakage current plus the sum of all currents of all ion channels. The final current $I_{app,i}$ is the applied current to the respective cell or compartment, representing outside input to the network/cell. It takes the form a single pulse of applied current per compartment. C is the membrane capacitance.

$$\frac{dV_i}{dt} = \frac{I_{channels,i}, +I_{app,i} + I_{mc,i} + I_{gap,i}}{C} \tag{6.1}$$

Table 6.6: *Specifications of the h/w used for performance measurements*

Specification	Maia DFE	Intel Core i7-4870H
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
IC process	65nm	22nm

6.4.2 Experimental setup and evaluation

The performance measurements for the accelerated kernels are done on a Maia DFE. The Maia-DFE specifications are shown in Table 6.6. The accelerated kernels are compared against C implementations executing on an 2.5-GHz Intel Core i7-4870HQ CPU (of a similar process generation to the Maia DFE).

The main parameters affecting resource usage are $N_{Comps,max}$, $N_{gates,max}$, and uf (unroll factor). As the maximum values $N_{Comps,max}$, $N_{gates,max}$, and uf are interdependent, there is a performance trade-off between maximally achievable network size and performance of the kernel based on the values of these design parameters. Besides conferring with experts, we also polled 10% of the 660 realistic single-neuron models in ModelDB [151]. We found that 10 channels per compartment cover 89% of all cases, therein we have chosen to restrict the maximum number of gates ($N_{gates,max}$) to 10 per compartment, as a reasonable ceiling for modeling custom ion gates. Fixing this at compile time bounds the DFE resource requirements while still retaining potential for a wide variety of experiments. We have explored (but do not include here for brevity) and derived the most viable pairs of these parameters for each one of the five flexHH kernels, also taking into account the memory I/O-bandwidth restrictions of the DFE. The objective was to increase performance while still providing support for experiments with network sizes of at least 20K compartments. The kernel configurations that resulted from this exploration are summarized in Table 6.7. These configurations are used for the evaluation.

The evaluation measurements have been done using simple neuron-model experiments of several thousands of simulation steps. The *HH* and *HH+gap* kernels were tested using the standard HH-model [36]. The *HH+custom*-kernel test simulates soma compartments from the IO-model. Finally, both the *HH fully featured* and *HH+custom+multi* kernels are

Table 6.7: *Optimized flexHH-kernel configurations, used for evaluation and the speedup against the CPU.*

model	uf	$N_{comps,max}$	$N_{gates,max}$	Speedup vs. CPU
HH	4	53,248	10	×35.49
HH+gap	24	24,576	10	×36.36
HH+custom	3	53,248	10	×15.88
HH+custom+multi	4	28,672	10	×14.33
HH fully featured	16	24,576	10	×23.98

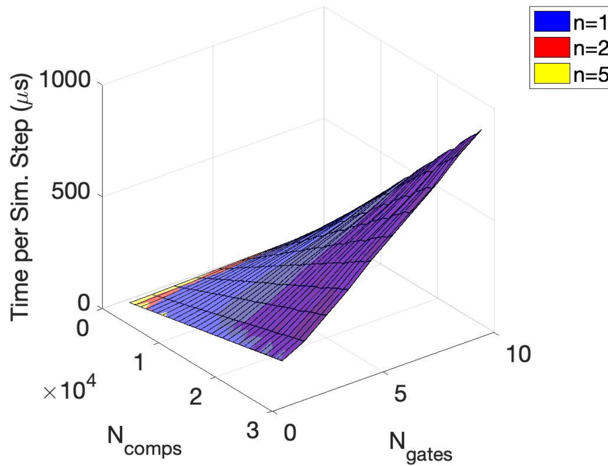


Figure 6.33: *Execution Time per step for the HH+custom+multi. N=Number of compartments.*

tested using IO cells, with each cell consisting of three compartments as in the original model description. Kernel times include both compute and on-board DRAM communication latency.

To derive performance speedups compared to the single-threaded version of the kernels, we simulate runs of 23,040 compartments. Using the same problem size for each kernel gives a good basis for comparing the overheads of the different flexHH features. For the HH+custom+multi+gap kernel, that is tested by simulating IO neurons, 23,040 compartments account for 7,680 cells, as a single IO cell consists of 3 compartments. This gives us a basis for comparing the performance of our generic library with

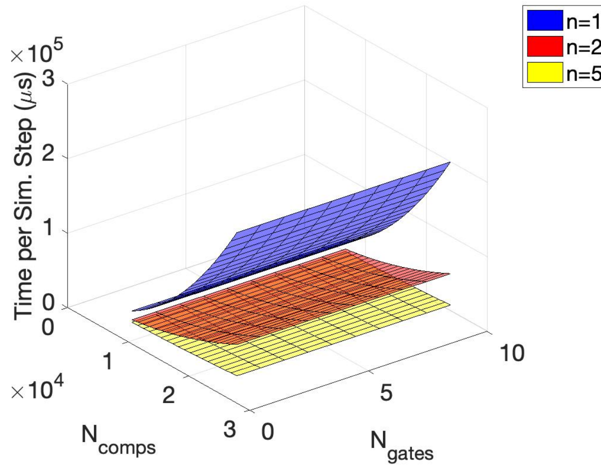


Figure 6.34: *Execution Time per step for the HH+custom+multi+gap. N=Number of compartments.*

the previously reported hard-coded DFE-version of the IO in the BrainFrame proof-of-concept which maximally supports 7,680 cells.

In Table 6.7, we can also see the speedup results of our five DFE instances compared to the single-threaded C versions. All flexHH kernels are deployed to take advantage of as much FPGA area as possible. The observed speedup is between 14× and 36×. It must be noted that the C version already provides significant performance benefits compared to the established NEURON simulation environment, resulting in a cumulative DFE speedup of 1,065× for the simple *HH* kernel compared to NEURON. An interesting observation is that the most complex kernel in the flexHH library actually has higher speedups compared to the simpler *HH+custom+multi* kernel (that also simulates an IO cell). This can be attributed to the existence of gap junctions. As seen in [161], including gap junctions to the model can dominate computational requirements. Employing loop unrolling on the gap-junction loop in hardware yields greater benefits compared to the CPU version, making the DFE version more efficient for this problem case. The same trend can be observed between the *HH* and *HH+gap* kernels. The presence of the gap junctions severely limits the maximum compartment count.

When comparing the simulation-step execution time of our most complex kernel with the hard-coded IO kernel seen in the BrainFrame proof-of-concept evaluation, a speedup of 1.36× is observed, a result making

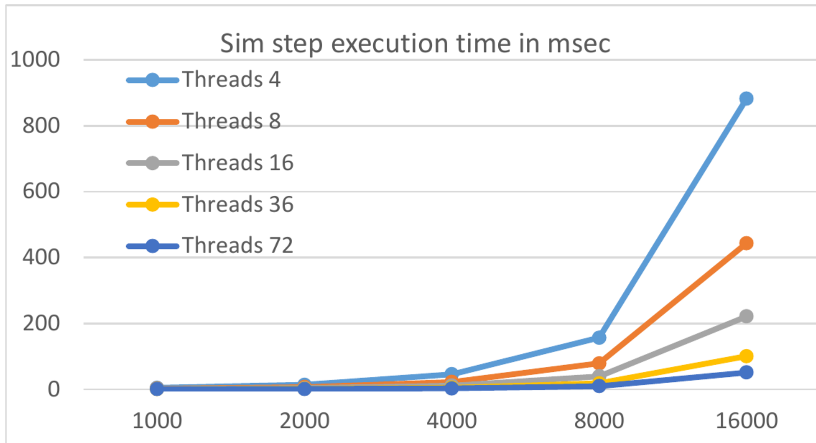


Figure 6.35: *Execution Time per step vs network size for the HH+custom+multi+gap.*

the flexHH competitive also against the Xeon-Phi and GPU platforms of BrainFrame. The flexHH speedup is partly caused by a higher operational frequency (180 MHz vs 150 MHz) and partly because of the structure of the flexHH kernel that is more compact and suffers less from pipeline-flushing overheads.

In terms of performance scalability, the simulation-step execution time scales linearly in relation to the number of gates for all cases. In terms of the compartment number, on the other hand, instantiating gap junctions changes things significantly: even though execution time scales linearly with compartment count when gap junctions are not present, the relation becomes quadratic when they are included; see Figure 6.33 and Figure 6.34. Additionally, for the *HH+custom+multi+gap* kernel, the amount of compartments per cell have a direct effect on performance scalability. As gap junctions dominate execution in higher problems sizes, the more compartments per cell an experiment has, the less gap-junction connections will be present for the same amount of total compartments. This allows for the execution time to scale more gracefully the more compartments that are included within a cell (Figure 6.34).

6.4.3 flexHH on Intel Xeon CPUs (on Amazon Cloud)

Performance experiments for the simplest flexHH-library model (HH) have also been conducted on Xeon implementations, run on Amazon AWS infrastructure. Specifically, the following instances have been tested: c5.xlarge,

c5.2xlarge, c5.4xlarge, c5.9xlarge, and c5.18xlarge – with 4, 8, 16, 36 and 72 CPU cores, respectively. The CPUs used for the simulations were Intel Xeon Platinum 8124M at 3.0 GHz 18 cores, each. We tested the worst-case scenario when it comes to computational requirements with 16 compartments per neuron, 8 gates per neuron and 100% connectivity density. Experiments were simulating 100 msec of brain time. In Figure 6.35, looking at Sim-step execution time vs network size, we can see the performance scalability for each thread case. The implementation presents great scalability for every case up to 8000 neurons. After that point, the lower-thread-count instances seem to bottleneck, as there are not enough parallel threads to cope with the increased computations. Yet, for larger problem sizes the cases above 16 threads still show excellent scalability.

6.5 Summary

In this chapter, we have proposed BrainFrame, an heterogeneous acceleration platform to serve computational-neuroscience studies in conducting the variety of real experimentation often required for the study of brain functionality. We have focused our analysis on biophysically-accurate neuron models, as such models are considered essential for the deeper understanding of the system properties of biological brain networks. In order for the BrainFrame system to cope with the demand for high ease of programming use as well as the computational requirements of the field, we have presented a proof-of-concept HPC platform that integrates three accelerator technologies already proven in brain simulations. The performance analysis of the system employing use cases that take into account connectivity density and modeling complexity, has revealed that all three fabrics are essential within such a powerful simulation platform so as to optimally serve all possible experimentation cases. The platform, thus, achieves efficient large-network experiments as well as real-time performance for meaningful network sizes ($\geq$ 100 cells).

BrainFrame is complemented, finally, with a PyNN front-end so as to tackle the much sought usability objective. The PyNN front-end makes the heterogeneous platform immediately accessible to a multitude of prior modeling works, which is an essential strategy for the wide adoption of complex HPC platforms in the neuroscientific community. This can be accomplished provided that more development is conducted into supporting the basic blocks of typical HH modeling, as the proof-concept-system only supports the IO model. Furthermore, building on the elegant PyNN infrastructure, a simple accelerator-selection algorithm has also been developed for automatically identifying the most suitable HPC fabric (Xeon

Phi, GPU, DFE) per neuroscientific experiment and has been integrated in BrainFrame. Last but not least, all accelerators use PCIe slots to connect to the host system, which greatly amplifies the platform flexibility and permits adjusting the platform hardware depending on the funds and hardware resources available to a research lab wishing to use BrainFrame. BrainFrame code repositories can be found in [162, 163].

Additionally to the proof-of-concept, the multi-device potential of the BrainFrame back-ends was explored giving a clear indication on the scalability of the acceleration technology used in BrainFrame. Lastly, the prototype IP libraries for HH-based modeling and their performance evaluation for DFE and multi-threading HPC substrates ware presented.

CHAPTER 7

Conclusions

Biophysically accurate models of biological systems, such as the ones using the HH formalism that has been the main focus of this thesis, are comprised mostly of a set of computationally challenging deferential equations often implementing an oscillatory behavior. If neurons are simulated as independent computational nodes (NGJ case), then dependencies between the equations do not arise, allowing divide-and-conquer, data-flow and event-driven acceleration strategies to be used very efficiently. The moment interconnectivity between oscillating neurons is also modelled (like GJs, input integrators, STDP synapses etc), the cells become coupled oscillators. The embarrassingly parallel and dataflow nature of the application is then broken. All neuron states need to be completely updated at each simulation step to retain correct functionality. This requirement, in turn, enforces the use of cycle-accurate, transient simulators and forbids event-driven implementations. As a result, a single HPC fabric cannot be a universal solution to the problem since it is unable to cover all the aforementioned requirements efficiently, as our analysis also reveals.

The above difficulties strongly hint on why most of the computational-neuroscience community has so far avoided employing HH models and multi-compartmental models with complex connections on large problem sizes using conventional computing machines. The eventual use of biophysically plausible neurons and connections on a larger scale is anticipated to contribute substantially in explaining biological behavior. Additionally, most related works seem to suffer from a limited re-usability value due to their (often inexistent) user interface. They ignore the challenge of the neuroscientific community adopting the proposed platform and very few propose solutions to that end. Beuler et al. [97] developed a graphical interface alongside their FPGA-based simulator. Although it does provide ease of use in experiments, it is still confined to only one platform and only one application with limited flexibility to be the basis of a more widely adopted system. Weinstein et al. [130] [164] took the approach of developing their own language to interface to their acceleration library, the DYNAMO compiler. Despite the limitation of using only FPGAs as the back-end platform, the DYNAMO compiler is a technically complete solution. Unfortunately, it failed to achieve wide adoption by the scientific community as it requires learning a new language and, additionally, the non-trivial process of porting all existing neuron models to the new coding paradigm.

PyNN has also been used in the past to tackle the issue of user interface. The most promising solution, both in terms of usability and computational ability, was proposed by Cheung et al. [123] with NeuroFlow.

In this work, the researchers integrated PyNN to their DFE-based hardware library. Neuroflow also provides a very complete library of IPs in the back-end, covering a great portion of possible applications. Yet, the system is still integrating a single acceleration platform. What is more, the performance and efficiency analysis is only presented for a single use case of a generally simpler model (Izhikevich) and with connectivity modeling of medium complexity (STDP) and relatively lower density (about 10%). The behavior and performance of the system for the rest of the supported features is not self-evident and is expected to be significantly different to the Izhikevich case, especially for accurate modeling such as the HH and with high connectivity densities, as shown by our performance analysis on the DFE platform. Furthermore, many of the performance benefits are accomplished using event-driven simulations (neurons are evaluated only when their inputs are triggered), that cannot always be employed, as discussed earlier.

To the best of our knowledge, no prior work has considered an heterogeneous acceleration system for coping with the variability of the applications in the field. Additionally, the PyNN front-end provides a familiar interface to the neuroscientific community, thus future integration of BrainFrame to the main PyNN code would make it a complete solution for a node-level heterogeneous system. The BrainFrame paradigm is primary designed to support multi-node setups. Such setups can be facilitated in the now up-and-coming heterogeneous datacenters, provided crucial aspects such as low-latency interconnects are tackled. Such a development would lead to a dramatic increase in the size of network populations supported at tractable simulation times, while also providing a way for small-medium-sized labs to use BrainFrame as a service, thus, enabling them to exploit the benefits of such an HPC platform without suffering the cost of creating and maintaining a local setup.

7.1 Scientific contributions

The main contributions of this thesis can be summarized as follows:

- The production of a literature review of FPGA-based acceleration efforts for SNNs and a comprehensive analysis of prior art on the field. The literature review reveals the potential that FPGA designs have on the field and their viability for real-time experimentation. Finally, the literature review shows that there is a severe lack of accessibility and re-usability in the vast majority of prior work beyond one-off, experiment-specific application (chapter 3).

- The acceleration of a novel, Hodgkin-Huxley based, state-of-the-art model of the inferior olive, an important subsystem of the olivocerebellar brain system using various FPGA-based HPC technologies. The designs explored employed traditional FPGA and dataflow engines providing considerable speed-up compared to the traditional programming environments used in the neuroscietific field. The FPGA-focused research also lead to the development of an embedded-HPC, spin-off design: ZedBrain (chapter 4).
- The evaluation of the FPGA-based acceleration effort and a comprehensive comparison with other HPC technologies implementing the same application. Specifically exploring volunteer computing, GPGPU and multithreading technology, their comparison highlighted the need for heterogeneity (chapter 5).
- The proposal and development of the BrainFrame HPC platform, supporting the inferior olive and standard Hodgkin-Huxley models. The platform proof-of-concept makes a clear case for the benefits of a heterogeneous system because of the high diversity of experimental use cases; benefits that involve both performance and energy efficiency (chapter 6).
- An exploration of the potential of BrainFrame back-ends (DFE and Xeon PHI) for multi-device/multi-node support that is essential if computational experiments are to tackle workload sizes that come close to the size of real biological networks (chapter 6).
- The implementation and performance evaluation of the first version of HH-based generic IPs for BrainFrame for both DFE and OpenMP multi- and many-core) environments (chapter 6).

7.2 Future work

The thesis purpose was to explore the challenges of accelerating neuron simulations. Although an important challenge in itself, the issue of HPC adoption in the field has been proven to be far more complex than just the acceleration part. Equally important is the development tool-flow and the familiarity/accessibility of the technology by the end user. BrainFrame, in its current state, makes a first step towards this goal, but for this effort to be truly fruitful, considerable additional research and technical development is required.

Cloud-based deployment of the system: BrainFrame is designed with both on-premise and cloud deployment in mind. Accessibility is a

major aspect for wide adoption. Thus, deployment as a cloud-HPC live service is essential. Additional engineering work is required for this to be accomplished. Robust containerization of the whole framework is required in order for the previous goals of accessibility and portability to be fulfilled. A complete cloud-HPC live service requires a collection of a vast amount of specialized knowledge. These, including the support of the acceleration back-ends, security and credential management, web-service architecture, middleware support to connect front- and back- ends, resource orchestration, UX design, neuron model validation and standardization both in the front- and back-end.

Extension of flexHH: Currently, the flexHH library supports DFE and OpenMP-based back ends. Its extension to more platforms like GPUs and its upgrade to support multi-device and multi-node setups would be an important step for optimal support of HH simulations.

Extension of model support: Only one type of neuron model and one synapse model are currently supported by BrainFrame. The support of a more extended collection of models is an essential long term goal for a platform like BrainFrame. Potential next natural steps would be the support of Izhikevich and AdEx neuron models and STDP and general modular Hebbian synapses, resulting in covering (alongside flexHH) a vast subset of the computational neuroscience field. Ideally, a mature system should utilize a neurosimulator able to generate code automatically for the different back-ends, making HPC development fully transparent to the model development.

Smart orchestration and selection: One of the major advantages of BrainFrame is the use of automatic selection of acceleration so that the user can exploit heterogeneity without having the specialized knowledge to identify which is the optimal platform for thier use case. Currently selection of resource allocation is based on a simple linear regression algorithm. But the great diversity of the neuron model field and the constant advancement of HPC technology can change the optimality of each use case dynamically. An automatic way of selecting acceleration and resource allocation using artificial intelligence and machine learning can serve the goal of accurate performance efficiency estimation (which is the main criteria for back-end selection).

Advanced visualization and data analysis automation: Improved usability for a simulation platform can be served with advanced automation for analysis and visualization of results. The ability for users to generate typical methods of visualization of neuron simulation results and the ability to customize types of data to be visualized can accelerate the research pro-

cess considerably. Additionally, automated processes for typical activities during experimentation, like Design Space Exploration automation, would increase the value for the scientist/user greatly. Such automation should be accompanied with professional UX design on the front-end as well.

Bibliography

[1] Y. Zhang, J. P. McGeehan, E. M. Regan, S. Kelly, and J. L. Nunez-Yanez, "Biophysically Accurate Floating Point Neuroprocessors for Reconfigurable Logic," ***IEEE TRANSACTIONS ON COMPUTERS***, vol. 62, no. 3, pp. 599–608, march 2013.

[2] I. Sourdis, C. Strydis, A. Armato, C. S. Bouganis, B. Falsafi, G. N. Gaydadjiev, S. Isaza, A. Malek, R. Mariani, S. Pagliarini, D. N. Pnevmatikatos, D. K. Pradhan, G. Rauwerda, R. M. Seepers, R. A. Shafik, G. Smaragdos, D. Theodoropoulos, S. Tzilis, and M. Vavouras, "Desyre: On-demand adaptive and reconfigurable fault-tolerant socs," in ***Reconfigurable Computing: Architectures, Tools, and Applications***, D. Goehringer, M. D. Santambrogio, J. M. P. Cardoso, and K. Bertels, Eds. Cham: Springer International Publishing, 2014, pp. 312–317.

[3] G. Smaragdos, S. Isaza, M. F. van Eijk, I. Sourdis, and C. Strydis, "FPGA-based Biophysically-meaningful Modeling of Olivocerebellar Neurons," in ***Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays***, ser. FPGA '14. New York, NY, USA: ACM, 2014, pp. 89–98. [Online]. Available: http://doi.acm.org/10.1145/2554688.2554790

[4] J. del Valle, J. Ramirez, M. Rozenberg, and I. Schuller, "Challenges in materials and devices for resistive-switching-based neuromorphic computing," ***Journal of Applied Physics***, vol. 124, p. 211101, 12 2018.

[5] J. Walinga, ***Introduction to Psychology - 1st Canadian Edition***. Royal Roads University, 2010.

[6] E. Izhikevich, "Which Model to Use for Cortical Spiking Neurons? ," ***IEEE Transactions on Neural Networks***, vol. 15, no. 5, 2004.

[7] A. Cassidy, A. G. Andreou, and J. Georgiou, "A Combinational Digital Logic Approach to STDP," ser. IEEE International Symposium on Circuits and Systems (ISCAS), May 2011, pp. 673–676.

[8] C.I. De Zeeuw, F.E. Hoebeek , L.W.J. Bosman, M. Schonewille, L. Witter, and S.K. Koekkoek, "Spatiotemporal firing patterns in the cerebellum," ***Nat Rev Neurosci***, vol. 12, no. 6, pp. 327–344, jun 2011.

[9] M. E. Sorensen, "Functional Consequences of Model Complexity in Hybrid Neural-Microelectronic Circuits," Ph.D. dissertation, School of Biomedical Engineering Georgia Institute of Technology, Mar. 2005.

[10] S. P. Johnston, G. Prasad, L. Maguire, and T. M. McGinnity, "AN FPGA HARDWARE/SOFTWARE CO-DESIGN TOWARDS EVOLVABLE SPIKING NEURAL NETWORKS FOR ROBOTICS APPLICATION," ***International Journal of Neural Systems,***, vol. 20, no. 6, pp. 447–461, 2010.

[11] M. Pearson, A. G. Pipe, C. Melhuish, B. Mitchinson, and T. J. Prescott, "Whiskerbot: A Robotic Active Touch System Modeled on the Rat Whisker Sensory System," ***Adaptive Behavior - Animals, Animats, Software Agents, Robots, Adaptive Systems***, vol. 15, no. 3, pp. 223–240, Sep. 2007. [Online]. Available: http://dx.doi.org/10.1177/1059712307082089

[12] W. McColloch and W. Pitts, "A Logical Calculus of the Ideas Immanent in Nervous Activity," ***Bulletin of Mathematical Biophysics***, vol. 5, pp. 115–133, 1943.

[13] F. Rosenblatt, "A Probabilistic Model for Information Storage and Organization in the Brain," ***Psychological Review***, vol. 65, no. 6, pp. 368–408, 1958.

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

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

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

[17] B. Glackin, J. A. Wall, T. M. McGinnity, L. P. Maguire, and L. J. McDaid, "A spiking neural network model of the medial superior olive using spike timing dependent plasticity for sound localization," ***Frontiers in Computational Neuroscience***, vol. 4, no. 18, 2010. [Online]. Available: http://www.frontiersin.org/computational_neuroscience/10.3389/fncom.2010.00018/abstract

[18] F. Akopyan, J. Sawada, A. Cassidy, R. Alvarez-Icaza, J. Arthur, P. Merolla, N. Imam, Y. Nakamura, P. Datta, G.-J. Nam, B. Taba, M. Beakes, B. Brezzo, J. B. Kuang, R. Manohar, W. P. Risk, B. Jackson, and D. S. Modha, "Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip," ***IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems***, vol. 34, no. 10, pp. 1537–1557, 2015.

[19] K. Cheung, S. R. Schultz, and P. H. W. Leong, "A Parallel Spiking Neural Network Simulator," in ***International Conference on Field-Programmable Technology, FPT 2009***, Dec. 2009, pp. 47–254.

[20] S. W. Moore, P. J. Fox, S. J. Marsh, A. T. Markettos, and A. Mujumdar, "Bluehive — A Field-Programable Custom Computing Machine for Extreme-Scale Real-Time Neural Network Simulation," in ***IEEE 20th International Symposium on Field-Programmable Custom Computing Machines***, Spring 2012, pp. 133–140.

[21] N. A. E. of (nae.edu), "Grand Challenges for Engineering," ***www.engineeringchallenges.org***, 2010.

[22] J. R. De Gruijl, B. Paolo, G. de Jeu Marcel T., and D. Z. C. I., "Climbing Fiber Burst Size and Olivary Sub-threshold Oscillations in a Network Setting," ***PLoS Comput Biol***, vol. 8, no. 12, 12 2012.

[23] T. Berger, A. Ahuja, S. Courellis, S. Deadwyler, G. Erinjippurath, G. Gerhardt, G. Gholmieh, J. Granacki, R. Hampson, M. C. Hsaio, J. Lacoss, V. Marmarelis, P. Nasiatka, V. Srinivasan, D. Song, A. Tanguay, and J. Wills, "Restoring lost cognitive function," ***IEEE Engineering in Medicine and Biology Magazine***, vol. 24, no. 5, pp. 30–44, 2005.

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

[25] J. M. Bower and D. Beeman, ***The Book of GENESIS***, 2003.

[26] E. Kandel, J. Schwartz, and T. Jessel, ***Principles of Neural Science***. Elsevier, 2000.

[27] A. P. Davison, ***PyNN: A Python API for Neural Network Modelling***. New York, NY: Springer New York, 2022, pp. 2947–2948. [Online]. Available: https://doi.org/10.1007/978-1-0716-1006-0_261

[28] T.-S. Chou, H. J. Kashyap, J. Xing, S. Listopad, E. L. Rounds, M. Beyeler, N. Dutt, and J. L. Krichmar, "Carlsim 4: An open source library for large scale, biologically detailed spiking neural network simulation using heterogeneous clusters," in ***2018 International Joint Conference on Neural Networks (IJCNN)***, 2018, pp. 1–8.

[29] J. C. Knight, A. Komissarov, and T. Nowotny, "Pygenn: A python library for gpu-enhanced neural networks," ***Frontiers in Neuroinformatics***, vol. 15, 2021. [Online]. Available: https://www.frontiersin.org/articles/10.3389/fninf.2021.659005

[30] A. Perera Molligoda Arachchige, "The blue brain project: pioneering the frontier of brain simulation," ***AIMS Neuroscience***, vol. 10, pp. 315–318, 11 2023.

[31] EPFL, "Blue brain faq," https://www.epfl.ch/research/domains/bluebrain/frequently_asked_questions/.

[32] G. Chatzikonstantis, D. Rodopoulos, C. Strydis, C. I. D. Zeeuw, and D. Soudris, "Optimizing Extended Hodgkin-Huxley Neuron Model Simulations for a Xeon/Xeon Phi Node," ***IEEE Transactions on Parallel and Distributed Systems***, vol. 28, no. 9, pp. 2581–2594, Sept 2017.

[33] I. S. Han, "Mixed-signal neuron-synapse implementation for large-scale neural network," ***Neurocomputing***, vol. 69, no. 16, pp. 1860–1867, 2006, brain Inspired Cognitive Systems. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231206001445

[34] W. Van Geit, M. Gevaert, G. Chindemi, C. Rössert, J.-D. Courcol, E. B. Muller, F. Schürmann, I. Segev, and H. Markram, "Bluepyopt: Leveraging open source software and cloud infrastructure to optimise model parameters in neuroscience,"

Frontiers in Neuroinformatics, vol. 10, 2016. [Online]. Available: https://www.frontiersin.org/articles/10.3389/fninf.2016.00017

[35] A. Javanshir, T. T. Nguyen, M. A. P. Mahmud, and A. Z. Kouzani, "Advancements in Algorithms and Neuromorphic Hardware for Spiking Neural Networks," ***Neural Computation***, vol. 34, no. 6, pp. 1289–1328, 05 2022. [Online]. Available: https://doi.org/10.1162/neco_a_01499

[36] 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.

[37] R. Brette and W. Gerstner, "Adaptive exponential integrate-and-fire model as an effective description of neuronal activity," ***Journal of Neurophysiology***, vol. 94, no. 5, pp. 3637–3642, 2005, pMID: 16014787. [Online]. Available: https://doi.org/10.1152/jn.00686.2005

[38] G. Smith, C. Cox, S. Sherman, and J. Rinzel, "Fourier Analysis of Sinusoidally Driven Thalamocortical Relay Neurons and a Minimal Integrate-and-Fire-or-Burst Model," ***Neurophysiology***, vol. 83, pp. 588–610, 2000.

[39] E. Izhikevich, "Resonate-and-fire Neurons," ***Neural Networks***, vol. 14, pp. 883–894, 2001.

[40] G. B. Ermentrout, "Type I membranes, phase resetting curves, and synchrony," ***Neural Computation***, vol. 83, pp. 979–1001, 1996.

[41] G. B. Ermentrout and N. Kopell, "Parabolic Bursting in an Excitable System Coupled With a Slow Oscillation," ***SIAM Journal on Applied Mathematics***, vol. 46, pp. 233–253, 1986.

[42] E. Izhikevich, "Simple Model of Spiking Neurons," ***IEEE Transactions on Neural Networks***, vol. 14, no. 6, 2003.

[43] R. FitzHugh, "Impulses and physiological states in models of nerve membrane," ***Biophysical Journal***, vol. 14, pp. 445–466, 1961.

[44] H. Wilson, "Simplified Dynamics of Human and Mammalian Neocortical Neurons," ***Theoretical Biolology***, vol. 200, pp. 375–388, 1999.

[45] V. Booth and J. Rinzel, "A Minimal, Compartmental Model for a Dendritic Origin of Bistability of Motoneuron Firing Patterns,"

Journal of Computational and Neuroscience, vol. 2, pp. 299–312, 1995.

[46] D. O. Hebb, ***The Organization of Behavior: A Neuropsychological Theory***. Wiley, 1949.

[47] G. Q. Bi and M. Poo, "Synaptic modifications in cultured hippocampal neurons: dependence on spike timing, synaptic strength, and postsynaptic cell type," ***Journal of Neuroscience***, vol. 18, pp. 10 464–10 472, 1998.

[48] S. Song, K. Miller, and L. Abbott, "Competitive Hebbian learning through spike-timing-dependent synaptic plasticity," ***Nature Neuroscience***, vol. 3, no. 9, pp. 919–926, 2000.

[49] S. P. Johnston, G. Prasad, L. Maguire, and T. M. McGinnity, "A Hybrid Learning Algorithm Fusing STDP with GA based Explicit Delay Learning for Spiking Neurons," ser. IEEE Conference on Intelligent Systems, Sep. 2006, pp. 632–637.

[50] J. R. De Gruijl, T. M. Hoogland, and C. I. De Zeeuw, "Behavioral Correlates of Complex Spike Synchrony in Cerebellar Microzones," ***Journal of Neuroscience***, vol. 34, no. 27, pp. 8937–8947, 2014. [Online]. Available: http://www.jneurosci.org/content/34/27/8937

[51] T. M. Hoogland, J. R. D. Gruijl, L. Witter, C. B. Canto, and C. I. D. Zeeuw, "Role of Synchronous Activation of Cerebellar Purkinje Cell Ensembles in Multi-joint Movement Control," ***Current Biology***, vol. 25, no. 9, pp. 1157–1165, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0960982215003231

[52] Z. Gao, B. J. van Beugen, and C. I. D. Zeeuw, "Distributed synergistic plasticity and cerebellar learning," ***Nature Reviews Neuroscience***, vol. 13, pp. 619–635, Sep. 2012.

[53] B. Glackin, J. A. Wall, T. M. McGinnity, L. P. Maguire, and L. McDaid, "A spiking neural network model of the medial superior olive using spike timing dependent plasticity for sound localization," ***Frontiers on Comput. Neurosci.***, vol. 4, no. 18, 2010.

[54] V. Braitenberg and A. Schüz, ***Cortex: statistics and geometry of neuronal connectivity***. Springer Science & Business Media, 2013.

[55] ***Differential and Difference Equations***. John Wiley and Sons, Ltd, 2016, ch. 1, pp. 1–53. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781119121534.ch1

[56] ***Numerical Differential Equation Methods***. John Wiley and Sons, Ltd, 2016, ch. 2, pp. 55–142. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781119121534.ch2

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

[58] C. Borgers 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.

[59] S. Henker, J. Partzsch, and R. Schüffny, "Accuracy evaluation of numerical methods used in state-of-the-art simulators for spiking neural networks," ***Journal of computational neuroscience***, vol. 32, no. 2, pp. 309–326, 2012.

[60] I. Peng, R. Pearce, and M. Gokhale, "On the memory underutilization: Exploring disaggregated memory on hpc systems," in ***2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)***, 2020, pp. 183–190.

[61] M. Shantharam, M. Tatineni, D. Choi, and A. Majumdar, "Understanding i/o bottlenecks and tuning for high performance i/o on large hpc systems: A case study," in ***Proceedings of the Practice and Experience on Advanced Research Computing***, ser. PEARC '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3219104.3219120

[62] K. Kauth, T. Stadtmann, V. Sobhani, and T. Gemmeke, "neuroaix-framework: design of future neuroscience simulation systems exhibiting execution of the cortical microcircuit model 20× faster than biological real-time," ***Frontiers in Computational Neuroscience***, vol. 17, 2023. [Online]. Available: https://www.frontiersin.org/articles/10.3389/fncom.2023.1144143

[63] A. Ghani, T. M. McGinnity, L. P. Maguire, and J. Harkin, "AREA EFFICIENT ARCHITECTURE FOR LARGE SCALE IMPLE-

MENTATION OF BIOLOGICALLY PLAUSIBLE SPIKING NEURAL NETWORKS ON RECONFIGURABLE HARDWARE," ser. International Conference on Field Programmable Logic and Applications (FPL '06), Aug. 2006, pp. 1–2.

[64] A. Cassidy, S. Denham, P. Kanold, and A. G. Andreou, "FPGA Based Silicon Spiking Neural Array," in ***IEEE Biomedical Circuits and Systems Conference, BIOCAS 2007***, Nov. 2007, pp. 75–78.

[65] Blair Hugh T, Cong Jason, and Wu Di, "FPGA Simulation Engine for Customized Construction of Neural Microcircuits," ***ICCAD / IEEE/ACM International Conference on Computer-Aided Design. IEEE/ACM International Conference on Computer-Aided Design***, vol. 2013, pp. 229–229, apr 2013. [Online]. Available: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC4288851

[66] B. Schrauwen and J. V. Campenhout, "Parallel hardware implementation of a broad class of spiking neurons using serial arithmetic," in ***IN PROCEEDINGS OF ESANN'06***, 2006, pp. 623–628.

[67] B. Glackin, T. M. McGinnity, L. P. Maguire, Q. X. Wu, and A. Belatreche, "A Novel Approach for the Implementation of Large Scale Spiking Neural Networks on FPGA Hardware," ***Computational Intelligence and Bioinspired Systems, Lecture Notes in Computer Science***, vol. 3512, pp. 1–24, 2005.

[68] H. Shayani, P. Bentley, and A. M. Tyrrell, "Hardware Implementation of a Bio-plausible Neuron Model for Evolution and Growth of Spiking Neural Networks on FPGA," in ***NASA/ESA Conference on Adaptive Hardware and Systems***, Jun. 2008, pp. 236–243.

[69] ——, "A Cellular Structure for Online Routing of Digital Spiking Neuron Axons and Dendrites on FPGAs," in ***ICES '08 Proceedings of the 8th international conference on Evolvable Systems: From Biology to Hardware***, 2008, pp. 273–284.

[70] R. Guerrero-Rivera, A. Morrison, M. Diesmann, and T. C. Pearce, "Programmable Logic Construction Kits for Hyper-Real-Time Neuronal Modeling," ***Neural Computation***, vol. 18, pp. 2651–2679, 2006.

[71] H. H. Hellmich and H. Klar, "An FPGA based simulation acceleration platform for spiking neural networks ," in ***47th Midwest Symposium on Circuits and Systems, MWSCAS '04***, vol. 2, 2004, pp. 389–392.

[72] ——, "SEE: a Concept for an FPGA based Emulation Engine for Spiking Neurons with Adaptive Weights," ***WSEAS Trans. on SYSTEMS***, vol. 3, no. 2, Apr. 2004.

[73] H. H. Hellmich, M. Geike, P. Griep, P. Mahr, M. Rafanelli, and H. Klar, "Emulation engine for spiking neurons and adaptive synaptic weights ," in ***IEEE International Joint Conference on Neural Networks, IJCNN '05***, vol. 5, Summer 2005, pp. 3261–3266.

[74] L. Wan, Y. Luo, S. Song, J. Harkin, and J. Liu, "Efficient neuron architecture for fpga-based spiking neural networks," in ***2016 27th Irish Signals and Systems Conference (ISSC)***, June 2016, pp. 1–6.

[75] H. Rostro-Gonzalez, J. H. Barron-Zambrano, C. Torres-Huitzil, and B. Girau, "Low-cost hardware implementations for discrete-time spiking neural networks," in ***Cinquieme conference pleniere francaise de Neurosciences Computationnelles***, 2010.

[76] A. Cassidy, A. G. Andreou, and J. Georgiou, "Design of a One Million Neuron Single FPGA Neuromorphic System for Real-time Multimodal Scene Analysis," in ***45th Annual Conference on Information Sciences and Systems (CISS)*** , Mar. 2011, pp. 1–6.

[77] R. M. Wang, C. S. Thakur, and A. van Schaik, "An fpga-based massively parallel neuromorphic cortex simulator," ***Frontiers in Neuroscience***, vol. 12, p. 213, 2018. [Online]. Available: https://www.frontiersin.org/article/10.3389/fnins.2018.00213

[78] E. Kousanakis, A. Dollas, E. Sotiriades, I. Papaefstathiou, D. N. Pnevmatikatos, A. Papoutsi, P. C. Petrantonakis, P. Poirazi, S. Chavlis, and G. Kastellakis, "An Architecture for the Acceleration of a Hybrid Leaky Integrate and Fire SNN on the Convey HC-2ex FPGA-Based Processor," ***2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)***, pp. 56–63, 2017.

[79] M. L. Rosa, E. Caruso, L. Fortuna, M. Frasca, L. Occhipinti, and F. Rivoli, "Neuronal dynamics on FPGA: Izhikevich's model," in

Spei conferance on Bioengineered and Bioinspired Systems II, vol. 5839, Jun. 2005.

[80] A. Cassidy and A. G. Andreou, "Dynamical Digital Silicon Neurons," in ***IEEE Biomedical Circuits and Systems Conference, BioCAS 2008*** , Nov. 2008, pp. 289–292.

[81] D. Just, J. F. Chaves, R. M. Gomes, and H. E. Borges, "An Efficient Implementation of a Realistic Spiking Neuron Model on an FPGA," in ***IJCCI***, 2010.

[82] C. M. Niu, S. Nandyala, W. J. Sohn, and T. Sanger, "Multi-scale hyper-time hardware emulation of human motor nervous system based on spiking neurons using fpga," in ***Advances in Neural Information Processing Systems 25***, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 37–45.

[83] M. Wildie, W. Luk, S. R. Schultz, P. H. W. Leong, and A. K. Fidjeland, "Reconfigurable Acceleration of Neural Models with Gap Junctions," in ***International Conference on Field-Programmable Technology, FPT 2009***, Dec. 2009, pp. 439–442.

[84] F. G. Sanchez and J. Nunez-Yanez, "Energy proportional streaming spiking neural network in a reconfigurable system," ***Microprocessors and Microsystems***, vol. 53, pp. 57 – 67, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0141933117301321

[85] K. Cheung, S. R. Schultz, and W. Luk, "A large-scale spiking neural network accelerator for FPGA systems," in ***Proceedings of the 22nd international conference on Artificial Neural Networks and Machine Learning - Volume Part I***, ser. ICANN'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 113–120. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33269-2_15

[86] G. Trensch and A. Morrison, "A system-on-chip based hybrid neuromorphic compute node architecture for reproducible hyper-real-time simulations of spiking neural networks," ***Frontiers in Neuroinformatics***, vol. 16, 2022. [Online]. Available: https://www.frontiersin.org/articles/10.3389/fninf.2022.884033

[87] M. Nouri, G. Karimi, A. Ahmadi, and D. Abbott, "Digital multiplierless implementation of the biological fitzhugh–nagumo

model," ***Neurocomputing***, vol. 165, pp. 468 – 476, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S092523121500466X

[88] E. L. 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. 417–436, 2004.

[89] J. Luo, K. Nikolic, B. D. Evans, N. Dong, X. Sun, P. Andras, A. Yakovlev, and P. Degenaar, "Optogenetics in silicon: A neural processor for predicting optically active neural networks," ***IEEE Transactions on Biomedical Circuits and Systems***, vol. 11, no. 1, pp. 15–27, Feb 2017.

[90] K. Akbarzadeh-Sherbaf, B. Abdoli, S. Safari, and A.-H. Vahabie, "A scalable fpga architecture for randomly connected networks of hodgkin-huxley neurons," ***Frontiers in Neuroscience***, vol. 12, p. 698, 2018. [Online]. Available: https://www.frontiersin.org/article/10.3389/fnins.2018.00698

[91] S. Yang, J. Wang, B. Deng, C. Liu, H. Li, C. Fietkiewicz, and K. Loparo, "Real-time neuromorphic system for large-scale conductance-based spiking neural networks," ***IEEE Transactions on Cybernetics***, 4 2018.

[92] Y. Zhang, J. Nuñez-Yañez, J. McGeehan, E. Regan, and S. Kelly, "A BIOPHYSICALLY ACCURATE FLOATING POINT SOMATIC NEUROPROCESSOR ," in ***International Conference on Field Programmable Logic and Applications. FPL 2009.***, Aug. 2009, pp. 26–31.

[93] J. C. Moctezuma, J. P. McGeehan, and J. L. Nunez-Yanez, "Biologically compatible neural networks with reconfigurable hardware," ***Microprocessors and Microsystems***, vol. 39, no. 8, pp. 693 – 703, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S014193311500143X

[94] A. Podobas and S. Matsuoka, "Designing and accelerating spiking neural networks using opencl for fpgas," in ***2017 International Conference on Field Programmable Technology (ICFPT)***, Dec 2017, pp. 255–258.

[95] S. Yaghini Bonabi, H. Asgharian, S. Safari, and M. Nili Ahmadabadi, "FPGA implementation of a biological neural

network based on the Hodgkin-Huxley neuron model," ***Frontiers in Neuroscience***, vol. 8, p. 379, 2014. [Online]. Available: https://www.frontiersin.org/article/10.3389/fnins.2014.00379

[96] P. Pourhaj and D.-Y. Teng, "FPGA based pipelined architecture for action potential simulation in biological neural systems ," in ***23rd Canadian Conference on Electrical and Computer Engineering (CCECE)***, May 2010, pp. 1–4.

[97] M. Beuler, A. Tchaptchet, W. Bonath, S. Postnova, and H. A. Braun, "Real-Time Simulations of Synchronization in a Conductance-Based Neuronal Network with a Digital FPGA Hardware-Core," in ***Artificial Neural Networks and Machine Learning – ICANN 2012***, September 2012.

[98] G. Smaragdos, C. Davies, C. Strydis, I. Sourdis, C. Ciobanu, O. Mencer, and C. I. De Zeeuw, "Real-time olivary neuron simulations on dataflow computing machines," in ***Supercomputing***, J. M. Kunkel, T. Ludwig, and H. W. Meuer, Eds. Cham: Springer International Publishing, 2014, pp. 487–497.

[99] G. J. Christiaanse, A. Zjajo, C. Galuzzi, and R. van Leuken, "A real-time hybrid neuron network for highly parallel cognitive systems," in ***2016 38th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)***, Aug 2016, pp. 792–795.

[100] T. S. Mak, G. Rachmuthx, K. P. Lam, and C.-S. Poon, "Field Programmable Gate Array Implementation of Neuronal Ion Channel Dynamics," in ***2nd International IEEE EMBS Conference on Neural Engineering*** , Mar. 2005, pp. 144–148.

[101] L. TIGĂERU and G. BONTEANU, "A Neuron Model for FPGA Spiking Neuronal Network Implementation," ***Advances in Electrical and Computer Engineering***, vol. 117, no. 4, 2011.

[102] R. Agis, E. Ros, J. Dıaz, R. Carrillo, and E. M. Ortigosa, "Hardware event-driven simulation engine for spiking neural networks," ***International Journal of Electronics***, vol. 94, no. 5, pp. 469–480, 2007.

[103] R. Agis, J. Dıaz, E. Ros, R. Carrillo, and E. M. Ortigosa, "Event-Driven Simulation Engine for Spiking Neural Networks on a Chip," ***Reconfigurable Computing: Architectures and Applications Lecture Notes in Computer Science***, vol. 3985, pp. 36–45, 2006.

[104] E. Ros, R. Agis, R. Carrillo, and E. M. Ortigosa, "Postsynaptic Time-dependent Conductances in Spiking Neurons: FPGA Implementation of a Flexible Cell Model," in ***Proceedings of the 7th International Work-Conference on Artificial and Natural Neural Networks: Part II: Artificial Neural Nets Problem Solving Methods***, ser. IWANN '03. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 145–152. [Online]. Available: http://dx.doi.org/10.1007/3-540-44869-1_19

[105] E. Ros, E. M. Ortigosa, R. Agís, R. Carrillo, A. Prieto, and M. Arnold, "Spiking neurons computing platform," in ***Proceedings of the 8th international conference on Artificial Neural Networks: computational Intelligence and Bioinspired Systems***, ser. IWANN'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 471–478. [Online]. Available: http://dx.doi.org/10.1007/11494669_58

[106] E. Ros, E. M. Ortigosa, R. Agis, R. Carrillo, and M. Arnold, "Real-Time Computing Platform for Spiking Neurons (RT-Spike)," ***IEEE Transactions on Neural Networks***, vol. 17, no. 4, pp. 1050–1063, july 2006.

[107] C. Torres-Huitzil and B. Girau, "Massively Distributed Digital Implementation of a Spiking Neural Network for Image Segmentation on FPGA," ***Neural Information Processing : Letters and Reviews***, vol. 10, no. 4-6, pp. 105–114, Apr. 2006.

[108] B. Girau and C. Torres-Huitzil, "Massively distributed digital implementation of an integrate-and-fire LEGION network for visual scene segmentation," ***Neurocomputing***, vol. 70, pp. 1186–1197, 2007.

[109] K. Rice, M. Bhuiyan, T. Taha, C. Vutsinas, and M. Smith, "FPGA Implementation of Izhikevich Spiking Neural Networks for Character Recognition," in ***International Conference on Reconfigurable Computing and FPGAs, ReConFig '09.*** , dec. 2009, pp. 451–456.

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

[111] A. Zuppicich and S. Soltic, "FPGA implementation of an evolving spiking neural network," in ***Proceedings of the 15th international conference on Advances in neuro-information processing - Volume Part I***, ser. ICONIP'08. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 1129–1136. [Online]. Available: http://dl.acm.org/citation.cfm?id=1813488.1813639

[112] B. Schrauwen, M. D'Haene, D. Verstraeten, and J. V. Campenhout, "Compact hardware liquid state machines on FPGA for real-time speech recognition," ***NEURAL NETWORKS***, vol. 21, no. 2-3, pp. 511–523, 2008.

[113] L.-C. Caron, F. Mailhot, and J. Rouat, "FPGA implementation of a spiking neural network for pattern matching," in ***IEEE International Symposium on Circuits and Systems (ISCAS)***, may 2011, pp. 649–652.

[114] C. Lammie, T. Hamilton, and M. R. Azghadi, "Unsupervised character recognition with a simplified fpga neuromorphic system," in ***2018 IEEE International Symposium on Circuits and Systems (ISCAS)***, May 2018, pp. 1–5.

[115] D. Neil and S. Liu, "Minitaur, an event-driven fpga-based spiking network accelerator," ***IEEE Transactions on Very Large Scale Integration (VLSI) Systems***, vol. 22, no. 12, pp. 2621–2628, Dec 2014.

[116] D. Roggen, S. Hofmann, Y. Thoma, and D. Floreano, "Hardware spiking neural network with run-time reconfigurable connectivity in an autonomous robot," in ***NASA/DoD Conference on Evolvable Hardware***, july 2003, pp. 189–198.

[117] M. Pearson, A. Pipe, B. Mitchinson, K. Gurney, C. Melhuish, I. Gilhespy, and M. Nibouche, "Implementing Spiking Neural Networks for Real-Time Signal-Processing and Control Applications: A Model-Validated FPGA Approach," ***IEEE Transactions on Neural Networks***, vol. 18, no. 5, pp. 1472–1487, sept. 2007.

[118] M. Pearson, C. Melhuish, A. Pipe, M. Nibouche, L. Gilhesphy, K. Gurney, and B. Mitchinson, "Design and FPGA implementation of an embedded real-time biologically plausible spiking neural network processor," in ***International Conference on Field Programmable Logic and Applications***, aug. 2005, pp. 582–585.

[119] M. Pearson, I. Gilhespy, K. Gurney, C. Melhuish, B. Mitchinson, M. Nibouche, and A. Pipe, "A Real-Time, FPGA Based, Biologically Plausible Neural Network Processor," in ***Artificial Neural Networks: Formal Models and Their Applications – ICANN 2005***, ser. Lecture Notes in Computer Science, W. Duch, J. Kacprzyk, E. Oja, and S. Zadrozny, Eds. Springer Berlin / Heidelberg, 2005, vol. 3697, pp. 755–756, 10.1007/11550907_161. [Online]. Available: http://dx.doi.org/10.1007/11550907_161

[120] M. Mokhtar, D. M. Halliday, and A. M. Tyrrell, "Hippocampus-Inspired Spiking Neural Network on FPGA," in ***Proceedings of the 8th international conference on Evolvable Systems: From Biology to Hardware***, ser. ICES '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 362–371. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-85857-7_32

[121] R. Weinstein, "TECHNIQUES FOR FPGA NEURAL MODELING," Ph.D. dissertation, School of Electrical and Computer Engineering, Georgia Institute of Technology, Dec. 2006.

[122] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," ***SIGARCH Comput. Archit. News***, vol. 23, no. 1, p. 20–24, mar 1995. [Online]. Available: https://doi.org/10.1145/216585.216588

[123] 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. [Online]. Available: https://www.frontiersin.org/article/10.3389/fnins.2015.00516

[124] A. Upegui, C. A. Pena-Rayes, and E. Sanchez, "An FPGA platform for on-line topology exploration of spiking neural networks," ***Microprocessors and Microsystems***, vol. 29, pp. 211–223, 2005.

[125] A. Ghani, L. McDaid, A. Belatreche, and W. Ahmed, "Neuro inspired reconfigurable architecture for hardware/software co-design," in ***IEEE International SOC Conference, SOCC 2009.***, sept. 2009, pp. 287–290.

[126] B. Glackin, J. Harkin, T. M. Mcginnity, and L. P. Maguire, "A Hardware Accelerated Simulation Environment for Spiking Neural Networks," in ***Proceedings of the 5th Interna-***

tional Workshop on Reconfigurable Computing: Architectures, Tools and Applications, ser. ARC '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 336–341. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-00641-8_38

[127] N. Mehrtash, D. Jung, H. Hellmich, T. Schoenauer, V. Lu, and H. Klar, "Synaptic plasticity in spiking neural networks (SP2INN): a system approach," ***IEEE Transactions on Neural Networks***, vol. 14, no. 5, pp. 980–992, sept. 2003.

[128] S. Yang and T. M. McGinnity, "A biologically plausible real-time spiking neuron simulation environment based on a multiple-FPGA platform," ***SIGARCH Comput. Archit. News***, vol. 39, no. 4, pp. 78–81, Dec. 2011. [Online]. Available: http://doi.acm.org/10.1145/2082156.2082176

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

[130] R. Weinstein and R. Lee, "Design of High Performance Physiologically-Complex Motoneuron Models in FPGAs," in ***2nd International IEEE EMBS Conference on Neural Engineering***, march 2005, pp. 526–528.

[131] 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, 2006. [Online]. Available: http://stacks.iop.org/1741-2552/3/i=1/a=003

[132] R. Weinstein, M. Reid, and R. Lee, "Methodology and Design Flow for Assisted Neural-Model Implementations in FPGAs," ***IEEE Transactions on Neural Systems and Rehabilitation Engineering***, vol. 15, no. 1, pp. 83–93, march 2007.

[133] 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, pMID: 10444678. [Online]. Available: https://doi.org/10.1152/jn.1999.82.2.804

[134] C. Feenstra, "A Memory Access and Operator Usage Profiler Framework for HLS Optimization: Using the Lucas Optical Flow Algorithm as Case Study," Master's thesis, EEMCS, Circuits and Systems, TuDelft, 2011.

[135] Maxeler Technologies, "www.maxeler.com/products/."

[136] G. Smaragdos and C. S. Garcia, "Zedbrain," https://github.com/cadriansalazarg/Final_Version_Demo_OpenHW_2017_Xilinx, 2017.

[137] K. Alfaro-Badilla, A. Chacón-Rodríguez, G. Smaragdos, C. Strydis, A. Arroyo-Romero, J. Espinoza-González, and C. Salazar-García, "Prototyping a biologically plausible neuron model on a heterogeneous cpu-fpga board," in ***2019 IEEE 10th Latin American Symposium on Circuits and Systems (LASCAS)***, 2019, pp. 5–8.

[138] A. Zjajo, J. Hofmann, G. J. Christiaanse, M. van Eijk, G. Smaragdos, C. Strydis, A. de Graaf, C. Galuzzi, and R. van Leuken, "A real-time reconfigurable multichip architecture for large-scale biophysically accurate neuron simulation," ***IEEE Transactions on Biomedical Circuits and Systems***, vol. 12, no. 2, pp. 326–337, April 2018.

[139] David P. Anderson, Eric Korpela, Rom Walton, "High-Performance Task Distribution for Volunteer Computing," in ***International Conference on e-Science and Grid Computing***, 2005, pp. 196–203.

[140] D. Theodoropoulos, G. Chrysos, I. Koidis, G. Charitopoulos, E. Pissadakis, A. Varikos, D. Pnevmatikatos, G. Smaragdos, C. Strydis, and N. Zervos, "mcluster: A software framework for portable device-based volunteer computing," in ***2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)***, 2016, pp. 336–341.

[141] H. D. Nguyen, Z. Al-Ars, G. Smaragdos, and C. Strydis, "Accelerating complex brain-model simulations on gpu platforms," in ***2015 Design, Automation and Test in Europe Conference and Exhibition (DATE)***, 2015, pp. 974–979.

[142] G. Smaragdos, G. Chatzikostantis, S. Nomikou, D. Rodopoulos, I. Sourdis, D. Soudris, C. I. De Zeeuw, and C. Strydis, "Performance analysis of accelerated biophysically-meaningful neuron simu-

lations," in ***2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)***, 2016, pp. 1–11.

[143] Gary A. McGilvary, Adam Barker, Ashley Lloyd, Malcolm Atkinson, "V-BOINC: The Virtualization of BOINC," in ***IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing***, May 2013, pp. 285–293.

[144] D. Anderson, "Boinc: A platform for volunteer computing," ***Journal of Grid Computing***, vol. 18, 03 2020.

[145] GSMA Intelligence, "The Mobile Economy 2015," 2015.

[146] J. James and J. Reinders, ***Intel Xeon Phi coprocessor high-performance programming.***, 2013.

[147] "Science&Technology Facilities Council, The Hartree Centre."

[148] T. Yamazaki and J. Igarashi, "Realtime cerebellum: A large-scale spiking network model of the cerebellum that runs in realtime using a graphics processing unit," ***Neural Networks***, vol. 47, pp. 103–111, 2013, computation in the Cerebellum. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0893608013000348

[149] O. Pell, O. Mencer, K. H. Tsoi, and W. Luk, ***Maximum Performance Computing with Dataflow Engines***. New York, NY: Springer New York, 2013, pp. 747–774. [Online]. Available: http://dx.doi.org/10.1007/978-1-4614-1791-0_25

[150] NVidia Corporation, "www.geforce.com."

[151] "https://modeldb.science/."

[152] H. Markram ***et al.***, "Reconstruction and Simulation of Neocortical Microcircuitry," ***Cell***, vol. 163, no. 2, pp. 456–492, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0092867415011915

[153] J. Jeffers, J. Reinders, and A. Sodani, ***Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition***. Morgan Kaufmann, 2016.

[154] M. Snir, ***MPI–the Complete Reference: The MPI core***. MIT, 1998.

[155] L. Dagum and R. Enon, "OpenMP: an industry standard API for shared-memory programming," ***IEEE CSE***, vol. 5, no. 1, pp. 46–55.

[156] G. Chatzikonstantis, D. Rodopoulos, S. Nomikou, C. Strydis, C. I. De Zeeuw, and D. Soudris, "First impressions from detailed brain model simulations on a Xeon/Xeon-Phi node," in ***ACM Computing Frontiers***, 2016.

[157] G. Chatzikonstantis, D. Rodopoulos, C. Strydis, C. I. De Zeeuw, and D. Soudris, "Optimizing Extended Hodgkin-Huxley Neuron Model Simulations for a Xeon/Xeon Phi Node," ***IEEE Transactions on Parallel and Distributed Systems***, 2017.

[158] "Surfsara. 2016. cartesius: the dutch supercomputer. retrieved from https://userinfo.surfsara.nl/systems/cartesius."

[159] R. Miedema, G. Smaragdos, M. Negrello, Z. Al-Ars, M. Möller, and C. Strydis, "flexhh: A flexible hardware library for hodgkin-huxley-based neural simulations," ***IEEE Access***, vol. 8, pp. 121 905–121 919, 2020.

[160] 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, 2014.

[161] G. Smaragdos, G. Chatzikonstantis, R. Kukreja, H. Sidiropoulos, D. Rodopoulos, I. Sourdis, Z. Al-Ars, C. Kachris, D. Soudris, C. I. D. Zeeuw, and C. Strydis, "BrainFrame: a node-level heterogeneous accelerator platform for neuron simulations," ***Journal of Neural Engineering***, vol. 14, no. 6, p. 066008, nov 2017. [Online]. Available: https://doi.org/10.1088%2F1741-2552%2Faa7fc5

[162] NCL, "Brainframe repository," https://gitlab.com/c7859/neurocomputing-lab/Inferior_OliveEMC, 2017.

[163] G. Chatzikonstantis, "infoli on xeon phi," https://github.com/GeorgeChatzikonstantis/infoli, 2017.

[164] R. K. Weinstein, "Techniques for FPGA neural modeling," Ph.D. dissertation, 2006.

Acknowledgments

As with any multi-year project, this thesis would also not be possible without the invaluable support of others.

First and foremost, i would like to thank Christos, my co-promotor. First for giving me the chance to do this PhD and also for trusting me with all the other activities within the lab and CUBE. I hope I have managed to justify the trust given to me. I have grown a lot under your supervision and mentorship and without the chances you have given me, my professional life would have been in a very different and probably worse place.

Dear Gianni, my other co-promotor. Your guidance and support, even if remote, has been incredible. You were a constant source of stability, order and sound advice in a process that often was, by nature, very chaotic. Thank you for everything.

And of course I would like to thank my promotor Chris. Very few department heads would take the gamble of adding a team of engineers into a neuroscience department at the time that you did it, long before the value of such a move was clear and most did not really know what we were actually doing. This PhD would not have been a reality otherwise. You were a constant source of inspiration and motivation and I hope our contributions to the department have managed to match or exceed your expectations.

I would also like to thank Bas Koekkoek. Your endless wave of ideas was a constant source of motivation to learn and do more. It was a joy to work with you in all the CUBE, Neurasmus and general department infrastructure projects. Thank you to all present and past CUBE and NCL lab members, Mario, Pieter, Harry, Rene, River, Stephanie, Chagajeg, Sotiris, Elias, Stefanos, Farnaz, Ali, Luuk, Lennart, Max, Angelo, Michiel, Micheal, Frits, Jan and especially Robert that we have shared so much time in our small office. And of course my paranymphs, Sadaf and Bas G. You cannot ask for better friends and colleagues. You made and are making the lab one of the most excellent places to work in.

Special thanks also to the current and past members of the rest of the department/theme that we cooperated and socialized over the years: Aleksandra, Gerard, Anna, Laurens, Vincenzo, Lieke, Dick (sorry for the noise), Bas H., Elise, Mandy, Devika, Nils, Loes.

Big thanks all the colleagues outside of EMC that have worked with over the years, who some I am also proud to call friends, Georgi, Stavros, George Ch., Dimitris T., Alirad, Aggelos, Sebastian I., Dimitris S, Craig.

Finally an immense thank you to my friends and family. To my parents that I owe everything in every possible way. To friends close by, Antonis S., Sofia (thank you for the great cover), Antonis L., Eva, George Ma., Michaela, George Mo., Dafni, George R. and to friends from afar, Sotiria, Vasilis, Alekos, Roula, Dimitris R., Nikos V., Babis, Panos, Tasos. None of what I have done all this time would have been possible without your support.

Curriculum Vitae

Personal information

Date of Birth : 03/07/1983
Place of Birth : Thessaloniki, GR

Residence: 's-gravelandseweg 631, 3119XT Schiedam, The Netherlands

E-Mail : g.smaragdos@pm.me

Research Interests

Reconfigurable hardware, HPC Computing, Scientific Computing, Computer Architecture, Fault-Tolerant Computing, Embedded Systems.

Professional Activities in Erasmus MC

August 2012 - Present

Research Analyst (in parallel with PhD Studies) at the Neuroscience Department at the **Erasmus Medical Center**.

August 2019 - Present

System Administrator of CUBE (https://ultrasoundbrainimaging.com/) at the Neuroscience Department at the **Erasmus Medical Center**.

Teaching Tasks

NB3016 Current topics in Nanoboilogy: A primer on High-speed Scientific simulations

- Self-Study guidance, June 2015, June 2016, June 2017
- Exam Evaluation, June 2015, June 2016, June 2017
- Lab creation, June 2015, June 2016, June 2017

Master Projects (Co-)Supervision

- Modeling of Olivocerebellar Neurons using SystemC and High-Level Synthesis, Van Eijk M. F., 2014
- Towards Real-Time Detection and Tracking of Spatio-Temporal Features: Rodent Whisker Tracking, Ma Y., 2017
- FlexHH: A flexible hardware library for Hodgkin-Huxley-based neural simulations, Miedema R., 2019
- Multi-GPU Brain: A multi-node implementation for an extended Hodgkin-Huxley simulator, van der Vlag M., 2019
- Towards Real-Time Olivary Neuron Modeling, Nicou N., 2020
- Beamforming FPGA acceleration project (Official Title TBD), Kypriotis V., 2023

Other Research Tasks

Participation in EU research Projects

- DeSyRe - Grant agr. no.: 287611
- VINEYARD - Grant agr. no.: 687628
- EuroEXA - Grant agr. no.: 754337

Research Development

Main Control of ErasmusLadder Prototype: The ErasmusLadder is an instrument that consists of two goal boxes with a horizontal ladder in between. The touch-sensitive rungs of the ladder make it possible to measure step durations and step types. Over several sessions, the ErasmusLadder software records the motor performance and the motor learning ability of mice. In the second phase of testing, a sudden object – a rung raised above stepping surface – challenges the mouse. How the animal learns to cope with this object is a measure of reflexive motor learning and a reflection of cerebellar functioning. The main control was programmed in C using an Arduino Due micro-controller.

Honors and Grants

- Best Paper Nomination - ISPASS April 2016
- EuroLab-4-HPC Cross-Site Collaboration Grant - May 2016

Peer reviewing (Reviewer or Sub-Reviewer)

Conferences

- International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS): 2013, 2016, 2017, 2019.
- International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART): 2013, 2015.
- International Conference on Field Programmable Logic and Applications (IEEE FPL): 2013, 2014, 2015, 2019, 2020, 2021, 2022, 2023.
- Design Automation Conference (DAC): 2014.
- International Conference on Application-specific Systems, Architectures, and Processors (IEEE ASAP): 2022.
- International Conference on Hardware/Software Co-design and System Synthesis (CODES + ISSS): 2014.
- IEEE International Symposium on Circuits and Systems (ISCAS): 2015.
- HiPEAC Cryptography and Security in Computing Systems Workshop (CS2): 2015, 2016.

Journals

- ACM Transactions on Architecture and Code Optimization (ACM TACO): 2012.
- ACM Transaction on Embedded Computing (ACM TECS): 2014.
- IOP Science Journal of Neural Engineering : 2016.
- IEEE Transactions on Nuclear Science: 2016.
- IEEE Transactions on Parallel and Distributed Systems: 2023

Free Software Contributions

Scientific Computing

- Simple Hardware Accelerated Inferior Olive Simulation Demo. Source Code part of the Maxeler AppGallery. Published under 3-clause BSD license.
  Source in NCL lab gitlab:
  https://gitlab.com/c7859/neurocomputing-lab/Inferior_OliveEMC

Other Interests

Cinema, Traveling, Photography (***http://turin231.daportfolio.com/***), FLOSS software

List of Publications

Publications included in this thesis

1. ***"BrainFrame: a node-level heterogeneous accelerator platform for neuron simulations"***, ***Georgios Smaragdos***, Georgios Chatzikonstantis, Rahul Kukreja, Harry Sidiropoulos, Dimitrios Rodopoulos, Ioannis Sourdis, Zaid Al-Ars, Christoforos Kachris, Dimitrios Soudris, Chris I De Zeeuw and Christos Strydis in IOP Science, Journal of Neural Engineering, volume 14, Number 6, 2017.

2. ***"Performance Analysis of Accelerated Biophysically-Meaningful Neuron Simulations," G. Smaragdos***, G. Chatzikostantis, S. Nomikou,D. Rodopoulos, I. Sourdis, D. Soudris, C. I. de Zeeuw,and C. Strydis, in 2016 Ieee International Symposium on Performance Analysis of Systems and Software Ispass 2016, pp. 1–11, 2016.

3. ***"Real-Time Olivary Neuron Simulations on Dataflow Computing Machines"***, ***Georgios Smaragdos***, Craig Davies, Christos Strydis, Ioannis Sourdis, Catalin Ciobanu, Oskar Mencer and Chris I. De Zeeuw, Int'l Supercomputing Conference (ISC) - Also Appearing in LNCS 8488 proceedings, Leipzig, Germany, June 2014.

4. ***"FPGA-based Biophysically-Meaningful Modeling of Olivocerebellar Neurons"***, ***Georgios Smaragdos***, Sebastian Isaza, Martijn Van Eijk, Ioannis Sourdis and Christos Strydis, 22nd ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), Monterey, California, February, 2014.

5. ***"flexhh: A flexible hardware library for hodgkin-huxley-based neural simulations"*** Rene Miedema, ***Georgios Smaragdos***, Mario Negrello, Zaid Al-Ars, Matthias Möller, and Christos Strydis, IEEE Access, vol. 8, 2020

6. ***"Multinode implementation of an extended Hodgkin–Huxley simulator"***, Giorgos Chatzikonstantis, H Sidiropoulos, Christos Stry-

dis, Mario Negrello, ***Georgios Smaragdos***, CI De Zeeuw, DJ Soudris in Elsevier Neuroscomputing,Volume 329, Pages 370-383, February 2019.

7. ***"Exploring Complex Brain-Simulation Workloads on Multi-GPU Deployments"*** Michiel A. van der Vlag, ***Georgios Smaragdos***, Zaid Al-Ars, and Christos Strydis, ACM Trans. Archit. Code Optim. 16, 4, Article 53 (December 2019)

8. ***"A Real-Time Reconfigurable Multichip Architecture for Large-Scale Biophysically Accurate Neuron Simulation"***, Amir Zjajo, Jaco Hofmann, Gerrit Jan Christiaanse, Martijn F. van Eijk, ***Georgios Smaragdos***, Christos Strydis, Alexander de Graaf, Carlo Galuzzi, Rene van Leuken in IEEE Trans. Biomed. Circuits and Systems, Volume: 12, Issue:2 , April 2018.

9. ***"Prototyping a Biologically Plausible Neuron Model on a Heterogeneous CPU-FPGA Board"***, Kaleb Alfaro-Badilla, Alfonso Chacón-Rodríguez, ***Georgios Smaragdos***, Christos Strydis, Andrés Arroyo-Romero, Javier Espinoza-González, Carlos Salazar-García in IEEE 10th Latin American Symposium on Circuits and Systems (LASCAS), February 2019.

10. ***"mCluster: A Software Framework for Portable Device-Based Volunteer Computing"***, D. Theodoropoulos, G. Chrysos, I. Koidis, G. Charitopoulos, E. Pissadakis, A. Varikos, D. Pnevmatikatos, ***G. Smaragdos***, C. Strydis, and N. Zervos, , 16th International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2016).

11. ***"Accelerating complex brain-model simulations on GPU platforms"***, H.A. Du Nguyen, Zaid Al-Ars, ***Georgios Smaragdos***, Christos Strydis, Design, Automation, and Test in Europe, DATE 2015.

12. ***"DeSyRe: On-Demand Adaptive and Reconfigurable Fault-Tolerant SoCs"***, I. Sourdis, C. Strydis, A. Armato, C.S. Bouganis, B. Falsafi, G.N. Gaydadjiev, S. Isaza, A. Malek, R. Mariani, D.K. Pradhan, G. Rauwerda, R.M. Seepers, R.A. Shafik, ***G. Smaragdos***, D. Theodoropoulos, S. Tzilis, M. Vavouras, S. Pagliarini, and D. Pnevmatikatos, 10th Int'l Symp. on Applied Reconfigurable Computing (ARC), Vilamoura, Algarve, Portugal, LNCS 8405, pp. 312–317, 2014.

13. ***"ESL Design of Customizable Real-Time Neuron Networks"***, Martijn van Eijk , Carlo Galuzzi , Amir Zjajo , ***Georgios Smaragdos***, Christos Strydis and Rene van Leuken, IEEE Biomedical Circuits and Systems Conference (BIOCAS 2014).

Other publications

1. ***"Towards Real-Time Whisker Tracking in Rodents for Studying Sensorimotor Disorders"***, Yang Ma, Prajith Ramakrishnan Geethakumari, ***Georgios Smaragdos***, Sander Lindeman, Vincenzo Romano, Mario Negrello, Ioannis Sourdis, Laurens W.J. Bosman, Chris I. De Zeeuw, Zaid Al-Ars, Christos Strydis in International Conference On Embedded Computer Systems: Architectures, Modeling And Simulation (SAMOS 2017), June 2017.

2. ***"Resilient CMPs with Mixed-grain Reconfigurability"***, Ioannis Sourdis, Danish Anis Khan, Alirad Malek, Stavros Tzilis, ***Georgios Smaragdos***, Christos Strydis, accepted in IEEE Micro, special series on "Harsh Chips", 2015.

3. ***"Reducing the Performance Overhead of Resilient CMPs with Substitutable Resources"***, A. Malek, S. Tzilis, D.A. Khan, I. Sourdis, ***G. Smaragdos***, C. Strydis , International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS 2015).

4. ***"Efficient On-line Testing of an Array of Reconfigurable RISC Processor"***, S. Pagliarini S. Pontarelli, J. Mathew, D.K. Pradhan, I. Sourdis, D.A. Khan, A. Malek, S. Tzilis, ***G. Smaragdos***, C. Strydis, 4th MEDIAN Workshop, 2015.

5. ***"A Probabilistic Analysis of Resilient Reconfigurable Designs"***, A. Malek, S. Tzilis, D. A. Khan, I. Sourdis, ***G. Smaragdos***, C. Strydis, International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS 2014).

6. ***"A Dependable Coarse-grain Reconfigurable Multicore Array"***, ***Georgios Smaragdos***, Danish Anis Khan, Ioannis Sourdis, Christos Strydis, Alirad Malek, Stavros Tzilis, 21st Reconfigurable Architectures Workshop (RAW 2014), Phoenix, Arizona, May, 2014.

Summary

Among the various methods in neuroscience for understanding brain function, in-silico simulations have been gaining popularity. Advances in neuroscience and engineering led to the creation of mathematical models of networks that do not simply mimic biological behaviour in an abstract fashion but emulate it in significant detail, even to the level of its biophysical properties. Such an example is the Spiking Neural Network (SNN) that can model a variety of additional behavioural features, like encoding data and adapting according to a spike train‘s amplitude, frequency and general precise pattern of arrival of spiking events on a neuron. As a result, SNNs have higher explanatory power than their predecessors, thus brain simulations based on SNNs become an attractive topic to explore. In-silico simulations of SNNs can have beneficial results not only for neuroscience research but breakthroughs can also potentially benefit medical, computing and A.I. research. However, SNNs come with computationally demanding workloads that traditional computing might struggle to handle. Thus, the use of High Performance Computing (HPC) platforms in this application domain becomes desirable. This dissertation explores the topic of HPC-based in-silico brain simulations.

Initially, the effort focuses on custom hardware accelerators, due to their potential in providing real-time performance alongside support for large-scale non-real-time experiments and specifically Field Programmable Gate Arrays (FPGAs). The nature of FPGA-based accelerators provides specific benefits against other similar paradigms like digital Application Specific Integrated Circuit (ASIC) designs. Firstly, we explore the general characteristics of typical SNNs model types to identify their computational requirements in relation to their explanatory strength. We also identify major design characteristics in model development that can directly affect its performance and behaviour when ported to an HPC platform. Subsequently, a detailed literature review is made on FPGA-based SNN implementations.

The HPC porting effort begins with the implementation of an extended-Hodgkin-Huxley model of the Inferior-olivary nucleus featuring advanced connectivity. The model is quite demanding and complex enough to act as a realistic benchmark for HPC implementations, while also being scientifically relevant in its own right. FPGA development shows promising performance results not only when doing custom designs but also using High-level synthesis (HLS) toolflows that significantly reduce development time. FPGAs have proven suitable for small-scale embedded-HPC uses as well. The various efforts, though, reveal a very specific weakness of FPGA development that has less to do with the silicon itself and more with its programming environment. The FPGA tools are very inaccessible to non-experts, thus any acceleration effort would require the engineer (and the FPGA development time) to be in the critical path of the research process.

An important question to be answered is how the FPGA platform would compare to other popular software-based HPC solutions such as GPU- and CPU-based platforms. A detailed comparison of the best FPGA implementation with GPU and manycore-CPU ports of the same benchmark is conducted. The comparison and evaluation shows that, when it comes to real-time performance, FPGAs have a clear advantage. But for non-real-time, large scale simulations, there is no single platform that can optimally support the complete range of experiments that could be conducted with the inferior olive model. The comparison makes a clear case for BrainFrame, a platform that supports heterogeneous HPC substrates.

This dissertation, thus, concludes with the proposal of the BrainFrame system. The proof-of-concept design supports standard and extended Hodgkin-Huxley models, such as the original inferior-olive model. The system integrates a GPU-, CPU- and FPGA-based HPC back-end while also using a standard neuroscientific language front-end (PyNN) that can score best-in-class performance, alleviate some of the development hurdles and make it far more user-friendly for the typical model developer. Additionally, the multi-node potential of the platform is being explored. BrainFrame provides both a powerful heterogeneous platform for acceleration and also a front-end familiar to the neuroscientist.

Samenvatting

Van alle verschillende neurowetenschappelijke methodes om hersenfuncties te begrijpen, zijn in-silico simulaties steeds populairder aan het worden. Vooruitgang in de neurowetenschappen en engineering heeft geleid tot wiskundige modellen van netwerken die niet simpelweg een abstractie van biologisch gedrag opleveren, maar het gedrag daadwerkelijk in detail kunnen nabootsen, tot op het niveau van de biofysische eigenschappen. Een voorbeeld hiervan is een Spiking Neural Network (SNN), die een aantal additionele eigenschappen kan modelleren, zoals het encoderen van data of adaptatie op geleide van de amplitude, frequentie en het precieze patroon van het aankomstmoment van spike-events op een neuron. Als gevolg hiervan hebben SNN's een hogere verklarende kracht dan hun voorgangers, waardoor hersensimulaties gebaseerd op SNN's een aantrekkelijk onderwerp worden om te verkennen. In-silico simulaties van SNN's kunnen niet alleen gunstige resultaten opleveren voor onderzoek in de neurowetenschappen, maar ook leiden tot mogelijke doorbraken in de medische, computer- en AI-onderzoeksgebieden. SNN's zijn echter computationeel zware operaties, die traditionele computers moeilijk kunnen verwerken. Daarom wordt het gebruik van High Performance Computing (HPC)-platforms in dit toepassingsgebied steeds aantrekkelijker. Dit proefschrift verkent het onderwerp van op HPC gebaseerde in-silico hersensimulaties.

Allereerst richt dit proefschrift zich op custom hardwareversnellers, die zowel real-time prestaties kunnen bieden alsook grootschalige niet-real-time experimenten – in het bijzonder Field Programmable Gate Arrays (FPGA's) – kunnen ondersteunen. FPGA-gebaseerde versnellers bieden specifieke voordelen ten opzichte van andere vergelijkbare paradigma's zoals Application Specific Integrated Circuit (ASIC)-ontwerpen. Eerst verkennen we de algemene kenmerken van typische SNN-modeltypen om hun computationele vereisten te identificeren relatief tot hun verklarende kracht. We identificeren ook belangrijke ontwerpeigenschappen in modelontwikkeling die direct van invloed kunnen zijn op prestaties en gedrag bij de overgang

naar een HPC-platform. Vervolgens wordt een gedetailleerd literatuuronderzoek uitgevoerd naar op FPGA gebaseerde SNN-implementaties.

Het HPC-porting proces begint met de implementatie van een uitgebreid Hodgkin-Huxley-model van de inferior olive nucleus met complexe connectiviteit. Het model is veeleisend en complex genoeg om te fungeren als een realistische benchmark voor HPC-implementaties, en tegelijkertijd wetenschappelijk relevant. FPGA-ontwikkeling toont veelbelovende prestatieresultaten, niet alleen bij het maken van op maat gemaakte ontwerpen, maar ook bij het gebruik van High-level Synthesis (HLS)-toolflows die de ontwikkeltijd aanzienlijk verminderen. FPGA's zijn ook bewezen geschikt voor kleinschalig embedded-HPC-gebruik. De verschillende inspanningen onthullen echter een zeer specifieke zwakte van FPGA- ontwikkeling die minder te maken heeft met het silicon zelf, en meer met de programmeeromgeving. De FPGA-tools zijn zeer ontoegankelijk voor niet-experts, waarbij elk inspanning om verder te kunnen versnellen medeafhankelijk is van een ingenieur en de FPGA-ontwikkeltijd.

Een belangrijke vraag die moet worden beantwoord, is hoe het FPGA-platform zich verhoudt tot andere populaire op software gebaseerde HPC-oplossingen, zoals GPU- en CPU-gebaseerde platforms. Er wordt een gedetailleerde vergelijking gemaakt van de beste FPGA-implementatie met GPU- en manycore-CPU-poorten van dezelfde benchmark. De vergelijking en evaluatie laten zien dat, als het gaat om real-time prestaties, FPGA's duidelijk een voordeel hebben. Maar voor niet-real-time, grootschalige simulaties is er geen enkel platform dat optimaal de complete reeks experimenten kan ondersteunen die met het model van de inferior olive kunnen worden uitgevoerd. De vergelijking pleit duidelijk voor BrainFrame, een platform dat heterogene HPC-substraten ondersteunt.

Dit proefschrift eindigt met het voorstel van het BrainFrame-systeem. Het proof-of-concept ontwerp ondersteunt standaard en uitgebreide Hodgkin-Huxley-modellen, zoals het originele inferior olive-model. Het systeem integreert een GPU-, CPU- en FPGA-gebaseerde HPC-back-end en een standaard neurowetenschappelijke taal front-end (PyNN), die best-in-class perfomance kan scoren, ontwikkelingsuitdagingen kan verlichten en het veel gebruiksvriendelijker kan maken voor de typische modelontwikkelaar. Bovendien wordt het multi-node potentieel van het platform onderzocht. BrainFrame biedt zowel een krachtig heterogeen platform voor versnelling als een front-end die vertrouwd is bij de neurowetenschapper.