Instituto Tecnológico de Costa Rica

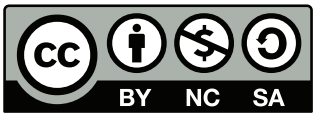
Escuela de Ingeniería Electrónica

Design of a multi-FPGA system for biologically plausible neural networks based of heterogeneous computing

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Electronics, Major in Embedded Systems

Jason Kaled Alfaro Badilla

Cartago, August 31



I declare that this thesis document has been made entirely by me, using and applying literature about the topic and introducing my own knowledge and experimental results.

In cases in which I have used references, I proceed to indicate the sources from corresponding references. Therefore, I assume full responsibility for the thesis work done and for the contents on this document.

Jason Kaled Alfaro Badilla

Cartago, September 25, 2022

Céd: 2-0739-0325

Instituto Tecnológico de Costa Rica
Escuela de Ingeniería Electrónica
Maestría Académica en Electrónica
Trabajo Final de Graduación
Tribunal Evaluador

Tesis de maestría defendida ante el presente Tribunal Evaluador como Requisito para optar por el título de Máster en Ingeniería Electrónica Grado Académico de Magister Scientiae

Miembros del Tribunal

CARLOS ADRIAN SALAZAR GARCIA
Firmado digitalmente por CARLOS ADRIAN SALAZAR GARCIA
Fecha: 2022.09.26 16:28:49 -06'00'

M. Sc. Carlos Salazar García
Profesor Lector

Firmado por RONNY GIOVANNI GARCIA RAMIREZ (FIRMA)
PERSONA FISICA, CPF-01-1137-0229.
Fecha declarada: 26/09/2022 07:06 PM
Esta representación visual no es fuente de confianza. Valide siempre la firma.

Dr. Ronny García Ramírez
Profesor Lector

JUAN JOSE MONTERO RODRIGUEZ (FIRMA)
Digitally signed by JUAN JOSE MONTERO RODRIGUEZ (FIRMA)
Date: 2022.09.27 22:37:47 -06'00'

Dr. Juan José Montero Rodriguez
Profesor Lector

ALFONSO CHACON RODRIGUEZ (FIRMA)
PERSONA FISICA, CPF-01-0702-0796.
Fecha declarada: 26/09/2022 03:07:08 PM
Razón: Apruebo
Lugar: Cartago Contacto: alchacon@tec.ac.cr

Dr. Alfonso Chacón Rodríguez
Profesor Asesor

Los miembros de este Tribunal dan fe de que la presente tesis de maestría ha sido aprobada y hace constar que cumple con las normas establecidas por la Unidad Interna de Posgrados de la Escuela de Ingeniería Electrónica del Instituto Tecnológico de Costa Rica.

Cartago, 31 de Agosto

Abstract

Today neuroscience is vastly specialized such that computational neuroscience tries to bridge the gaps of knowledge between the theory and the experiments. In-silico experiments are computer simulations with complete control over the scenario; this techniques try to decode the functionality of the biological neural networks and the biophysical dynamics which this cells inherent. This work explores a way to improve biological-precise spiking neural networks simulations with FPGA acceleration. Our approach focuses with creating a hardware acceleration for one cell compartment using a system-on-chip, this serves a proof-of-concept to value how flexible is the platform to accelerate similar simulations using the hybrid hardware-software methods. The work described in this thesis is a implementation of the inferior olivary nuclei model implemented with a extended Hodgkin-Huxley neural model. The development platform was the Xilinx's Zynq-7000 and the Vivado Hardware Design suite.

Results obtained in this work shows that the hybrid computing is more performance efficient in using the FPGA resources. Also proves a more flexible platform unlike other authors similar work. Finally, the use of a shared DRAM between the CPU and FPGA fabric showed a bottleneck for the design, its noted that it would be preferable to separate if possible the main DRAM between both systems.

Keywords: SNN, FPGA, SoC, Neural network, Hodgkin-Huxley, cluster, HPC

Acknowledgements

This work required a considerable amount of work. Much of this investigation would not be possible if not the help of close friends and laboratory assistants. Thankful to share time with them, and the company that served my pets.

Jason Kaled Alfaro Badilla

Cartago, August 31 2022

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Decoding the brain is one of the most outstanding goals in humankind because of the implying benefits in advances in medicine and artificial intelligence; sadly, still, we know little information. Motivation in neuroscience is endless; hence, the field has snowballed over the years. Research such as how to unravel effective treatments for prevalent psychiatric severe disorders such as Alzheimer's disease and Schizophrenia [1] is expected to bring new life standards. Therefore, people from many interdisciplinary fields unite forces to tackle these hard problems, and new consortium groups are established and funded by the government's investments, such as the European Human Brain Project (HBP) [2] and the USA's BRAIN initiative [3]. This transition led room to create innovative specializations in the field, and conceive novel theoretical and experimental analysis for studying the brain [4]. The theoretical analysis targets the intrinsic features of the brain cells, such as its electrophysiological properties, its anatomical structure, its region classifications, and many more.

Computational neuroscience becomes essential during recent years because of the advances in the semiconductor industry, computers become faster every year and can do computing analysis more accessible, hence increasing the productivity in research (i.e., with computer-assisted statistical analysis). The research area promises to bridge the gaps of knowledge between the laboratory and the theoretical models [5]. Some popular topics from the field are neuroinformatics, medical informatics, neurorobotics, and brain simulation [2].

This thesis focuses on brain simulation performance and its close relationship with hardware specialization.

1.1 Computational neuroscience

1.1.1 In-silico experiments

Unlike other fields, neuroscience studies one of the main organs of living beings; thus, invasive experiments are desired but expensive to maintain and rarely available for ethical reasons. Therefore, *in-silico* experiments come with many advantages over the *in-vivo* or *in-vitro* counterparts, such as not requiring surgical procedures and complete control over the running experiment. A cubic millimeter of the brain cortex contains several tens of thousands of individual brain-cells; therefore, computer technology is a known limitation that prohibits the scale of the experiments. Nevertheless, a modern supercomputer has been catching up recently, thus making feasible simulations of neural networks comprised of millions of neurons [1].

Mechanistic modeling on the neuron, in the quantitative sense, gain attention since the established work of Hodgkin and Huxley [6]. They developed a model for the action-potential generation and propagation in giant squid axons [1]. Today, numerous detailed neural models are ready to use accessible online in public databases (see [7]), and it is encouraged to researches re-utilize them. Consider that the most basic neural model are the building blocks for creating more refined models, i.e., the Hodgkin-Huxley model, thus modeling each neuron of every region of the brain is a marvelous task [1].

1.1.2 Neural models

To grasp the function of neuron cells, we model them in quantitative representations; these are a set of equations that tailor to a specific neuron such as cells in the mammalian sensory cortex, the hippocampus, or the thalamus [1]. The brain, in its more basic form, is a complex network, interconnected in such a way where pattern recognition, thinking analysis and motor work is possible. Likewise, a neural network is a circuitry of individual instances of neurons (based on a model). Two main groups divide the studies of multiple topologies: first, the people who want to learn how the neurons learn, and the scientists which interest is to comprehend how the internal mechanics of the neuron interact in its processing capabilities. The first topic, usually, manages models where the resemblance is morphological to a neuron; because the computational cost by including sophisticated will make more difficult the task at large-scale. The second is more interested in models where its variables and parameters are associated to biological mechanism from brain neurons; these experiments follow the range from one single neuron towards the order if the thousands of cells. A perfect model won't exist ([1], [8]); thus, the artificial cell we simulate remains constrained.

The field of neural networks is a broad topic, but the set of neural networks of interest for this thesis are the spiking neural networks (SNN), **not** the artificial neural networks (ANN). The SNNs are the closest neural network that reassembles the biological ones [9].

Our research focuses on improving the computational performance in these biophysical inspired SNN-related simulations.

1.1.3 Spiking neural networks

Two important terms to achieve a biologically accurate representation of an SNN is the connectome and the synapse model. The connectome is the fixed arrangement in how regions of the brain grow routed. According to the dimensional scale, there are two classes: macro-connectomics (localization of brain activity under emotions or specific thoughts), or micro-connectomics (hierarchical structure of sub-networks on regions) [4]. Every physical connection is a channel to share information between cells (synapse). The synapses can occur by different mechanic means; in general, there are electric and chemical synapses.

The electrical synapses are the simplest synapse type and consist of intercellular channels that allow ions and small molecules to pass from one cell to the next. These synapses are also known as gap-junctions. Gap-junction does not distinguish between pre- and postsynaptic neurons [10]. Chemical Synapses sequence starts when the pre-synaptic signals travel via a release of neurotransmitters from the pre-synaptic neuron, which binds to receptors at the postsynaptic neurons [10]. The most crucial feature from the synapses is that they are a complex mechanism in which they can regulate networks and filter information. The learning process of the neural networks is believed to come from the plasticity of the connections given by the synapses (such as spiking-dependent dependent plasticity or STDP) [10].

Herz et al. [8], categorized single-cell models for SNN in five levels, each level represents a layer of abstraction where the model is simplified. For the level I models, they have great detailed compartmental models obtained from anatomical reconstructions and based on the spatial structure. i.e., a complex dendritic tree quickly needs more than 1000 compartments to capture the cell's specific electrotonic structure [8]. Level II models reduced the order of complexity by decreasing the number of compartments between two to four. These reduced models are often sufficient to understand soma-to-dendritic interactions that govern spiking or bursting [8]. Likewise, Level III reduces the compartments to a single one per neuron. These models have led to a quantitative understanding of many dynamical events, including phase-sync spiking, bursting, and spike-frequency adaptation [8]. Then, Level IV models are cascade models. They are a concatenation of mathematical primitives, such as linear filters, nonlinear transformations, and random processes. Limitations start when emulation of far downstream from the sensory periphery cannot synthesis precisely its behavior. Lastly, level V models only center around the signal-processing capabilities of a single neuron without considering its bio-physical machinery.

Fig. 1.1 shows the typical workflow in simulation-centric methodologies in neuroscience (according to [4]). The first stage is the data gathering from experiments; then, the

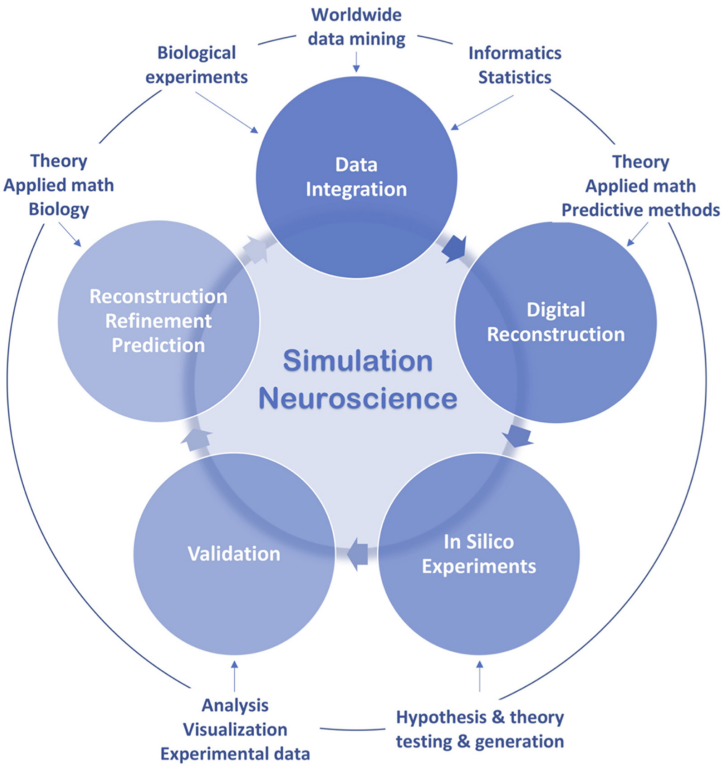


Figure 1.1: Simulation experiments typical workflow [4].

obtained data-set is processed and fitted to a particular SNN model, which is common in in-silico experiments. The validation stage analyses the output of the simulation; this is compared to the source data via visualization or statistical methods. Further refinements of the model go to in the final stage, which gives new hints for research.

Sometimes it is desired to use a neural model that is hardware-friendly to optimize and feasible to run a large-scale experiment. The SpiNNaker project [11] is a neuromorphic computing platform that is optimized to perform these types of models such as leaky integrate and fire (LIF) and Izhikevich model. Other projects like Bluehive ([12]) and Zedwulf ([13]) work on these models too. But, on this thesis, the goal focuses on simulations of Level II models that inherit the Hodgkin-Huxley model.

1.2 FPGA in High-Performance Computing

High-performance computing (HPC) is a discipline in computer science which searches for the most effective methods to tackle big problems to execute. In the area of research, the most common HPC technology stacks are of many-core CPU systems and GPUs accelerators [14]. Recently, Field Programmable Gate Arrays (FPGA) gains attention on the community by excelling in performance on specific applications such as bioinformatics and digital signal processing [14]. The FPGAs are flexible devices that can be designed as a custom ASIC chip but reconfigured internally at any moment. Their power relies

upon their innate design, where via an array of reconfigurable device primitives, it can fit almost every digital architecture. According to [15], the FPGAs' market was niche; but, with the rapid advances in the semiconductor industry, the FPGA market grew, by consequence, its applications increased. In [14], the FPGAs suits better for applications that can take advantage of custom data widths operands, combinational logic problems, finite state machines, and parallel MapReduce problems; but, GPUs and many-core CPUs are still preferable in intensive floating-point calculations.

Electronic design automation (EDA) tools had improved alongside the FPGA generations; thus, the FPGA design experience can be the most productive it has been; bringing down the barrier of entry (see [16] for more details on the FPGA's history). One feature that boosts productivity in FPGA projects is the high-level synthesis (HLS) tools. The HLS methodologies bypass the need to meticulously design every component of the system on a hardware description language (HDL), which takes a great effort to write down and test [17]. The HLS tools make the description of the desired system in a higher-level programming language (i.e., C/C++) into an HDL design [17].

The HLS tools is a subject on research because the quality of results produced in HLS are worse than manual register-transfer level (RTL) coding flows [17]. Nevertheless, HLS is currently a viable option for fast prototyping, according to [17]. Some of the HLS tools available on the market are LegUp, Catapult, Chisel, Vivado-HLS, and many more (check [17] for more HLS tools and their usage). Under the subject on HLS in HPC, it made the transition more comfortable for a software developer to use FPGA effectively, i.e., via OpenCL API [18].

1.2.1 Programmable System-on-Chip: opportunities for high-performance computing

Modern system-on-chip (SoC) development platforms often offer heterogeneous computing opportunities where an FPGA is integrated with a CPU through a fabric. As stated in [19], single-chip heterogeneous computing are a great deal in their energy efficiency. Energy is critical in embedded applications which are battery-powered but is not apparent the importance in HPC computing; technically, power requirements gives insight about the scalability of the system because the heat dissipation makes the implementation not practical to implement[19]. Heterogeneous SoC is a novel solution that could combine the forces of serial processing capabilities of many-core CPUs with massively parallel computing with reconfigurable fabric.

In [12] proved a low-cost FPGAs cluster could manage large-scale SNN simulations, an implementation that was more cost-effective than a high-end FPGA implementation. Although the neural model achieved was the Izhikevich model thus the biological meaning of the machine is lower. Taking similar steps from [12], how can we extend this design to a more challenging SNN model such as a multi-compartment cell. This kind of model usually is computed on single-precision floating-point, even on double precision in some

scenarios[1]. FPGA implementations could be not ideal by high-end GPU FP capabilities on this scenarios[14]. However, heterogeneous computing in FPGA programmable SoC can cover the FPGA's weakness because a many-core CPU system can handle most FP processing and the FPGA computes the most substantial workload of the SNN. With this approach, we see benefits in FPGA's resource savings, which imply a smaller footprint, and therefore it can fit in low-cost FPGAs. This thesis works further on this premise.

1.3 Scope of this thesis

The objective of this thesis is to develop a heterogeneous computing system for an SNN model derived from the HH model. The proposed method implements FPGA hardware acceleration to speedup its simulation runtime. Unlike other authors (same mentioned previously), the scheme follows an approach in programmable SoC (PSoC) computing, where the optimization of the SNN benefits from the architecture of the hardware. Due to the hard-work required to optimize one SNN in hardware, there is no guarantee that the model will be relevant in the long run but similar techniques can extend to other similar models.

1.3.1 Structure of this document

Three main chapters organize this document. The following chapters gather information and analysis from published work in [20] and [21] by the author of this thesis. Chapter 2 includes the profiling of the performance of the algorithm. Chapter 3 describes the HLS implementation on an FPGA to generate the HDL synthesizable hardware. And finally, chapter 4 details the software side of the implementation and it describes how vectorized code for the inferior-olivary model using NEON SIMD was implemented.

Chapter 2

The Inferior Olivary Nucleus eHH Model: Computational Load Analysis

2.1 The inferior olivary nucleus model

The inferior olivary nucleus (ION) forms an intricate part of the olivocerebellar system which is believed to be related to the timing of motor commands and learning [22]. The main feature of this cell is that it forms part of the densest brain region where its activity only gets triggered when multiple neurons are synchronized (and subsequently transmitting a short burst of spikes [22]). To effectively emulate this behaviour, De Gruijl *et al.* [22] developed a model based on a three-compartmental cell (dendrite, soma and axon) with GJ interactions between the dendrite-compartments of each neuron (see Fig. 2.1). The model is also called the extended Hodgkin-Huxley model (eHH). The operations are performed with single-precision floating-point (SPFP) representation; a summary of the required SPFP operations and data transfer needed for the different sections of the model are given in Table 2.1.

As discussed in [20], [23], [24], the eHH ION model has a computational complexity that is mainly determined by the GJ interactions, particularly when the network is densely interconnected, being the worst case an all-to-all connection, when the complexity becomes $O_{gj}(n^2)$, with n the number of dendritic connections for each cell. As an aside, since a required property of a biologically accurate cell model is that the neural network must be synchronized in order to guarantee the correct calculation of the dendritic phenomena, event-driven simulations are discarded [25], and the differential equations are usually solved via the Euler method [22].

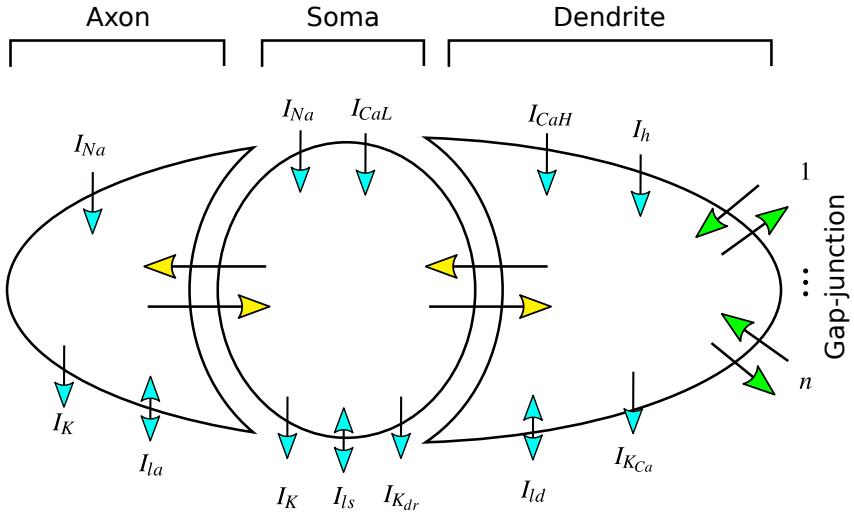


Figure 2.1: Schematic representation of the three-compartmental cell model proposed in [22]. Each arrow indicates an ion channel or electrical current direction.

Table 2.1: Single-Precision Floating-Point (SPFP) operations and transfers per simulation step, per neuron [23], [26]. Here, N means the total neuron population.

Operational compartment	**No. of SPFP operations**
Gap junction unit	$12\times N$
Cell compartment (Axon, soma, dendrite)	859
I/O and storage	**No. of SPFP transfers**
Neuron state (R/W)	19
Evoked input (R)	1
Connectivity vector (R)	$1\times N$
Neuron conductances (R)	20
Axon output (W)	1 (axon voltage)

2.2 FPGA approaches at simulating biologically accurate SNN models

Plenty of work has been carried out in porting SNN models to FPGAs, as the already mentioned cases of [9], [27]. But there are also other SNN implementations (single compartment unlike ION), as that of [12], [28], and some have also already used heterogeneous platforms as the one used in this work (see [13]), proposing a cluster of 32 boards in order evaluate the communication performance of a sparse graph-oriented application using MPI (in this case a sparsely distributed Iz based SNN). These sparse SNN models are, nonetheless, not able to support the ION model, which is one of the main objectives that motivated this work. As already discussed, a key aspect of the ION system is the dense interconnection among the cell population. Researchers in [9] and [27] center their strategy around multiple instances of execution units of neural cells called physical cells (PC). Each cell-state is associated with one PC, then each PC executes a set of cell-states each

simulation step. In both cases, their approaches are fast enough to achieve locked-step simulation at a 50 μs time step (necessary for the model's convergence, in order to have simulation timings equivalent to the brain's own real time response), but circumscribed to a small cell population in a all-to-all dendritic connection. This sizing restriction is mainly due to the inner gap-junction interactions within the dendrite compartment. These operations are modelled as a pair of nested `for` loops (see 2.1), that scale in a $O(n^2)$ fashion as the number of dendritic connections, n, increases. One could easily try to port to hardware via an HLS loop-unrolling directive that distributes into parallel hardware the arithmetic operations. But this direct approach, unfortunately, does not scale effectively because of hardware resources sharing among the rest of the PC's compartments, which in the case of [9], limit the total number of real-time simulated cells to 96. The work of [27], claims bringing up the total number of real-time running cells up to 768 for a single Zedboard (in what seems to an 8-way connection scheme among neurons, with no specific experimental timing results given for their proposed Multi-FPGA platform), at the cost of a customized structure, not readily ported to a different SNN model. Both [9], [27] claim the capacity of accommodating larger SNNs if this real-time constraint is removed, nonetheless.

2.3 An heterogeneous approach to SNN simulation accelerators

This thesis work aims at providing an efficient accelerated implementation of the eHH ION model, that is also flexible enough in case of requiring extensive modifications of the model used, without losing efficiency. The overall high level architecture used is shown in Fig 2.2, using Avnet's Zynq-7020 SoC ZedBoard with 512MB DDR3 RAM. The processing region of the Zynq (called PS) includes a dual-core ARM A9 CPU, with NEON SIMD capabilities, plus several I/O controllers such as an Ethernet interface. The integrated Artyx-7 FPGA fabric (called the PL region) is interconnected via several AXI4 bus channels to the PS. The simulation task is partitioned between the PS and PL regions: the soma, the axon, and the dendrite compartments described by the eHH equations are executed on the ARM cores. The gap-junction interactions are processed in the PL region. Note that the latter has limited local on-chip memory (technically called BRAM by Xilinx), making difficult for the whole conductance matrix to reside locally next to the gap-junction processing unit (here called GJU), if the said so matrix is over a few thousands elements. This entails a continues transfer of such matrix between the DDR3 RAM and the GJU, in each computation step (a handicap that may easily be overcome on FPGAs with bigger BRAMs).

To complement on the decision on what to port to the hardware, a profiling of the C code of the eHH was carried on, running on a single thread in the ZedBoard's ARM A9, at SPFP. Result s are shown in Fig. 2.3. Notice the gap-junctions computational load impact on the performance, as the network interconnection density grows. Without such

load, the dendrite's computation time falls under that of the other two compartments. For realistic networks sizes (over hundreds of cells), the GJ is the costliest function), with $O(N^2)$ complexity, while other compartment functions are $O(N)$. Results agree with reports in the literature (see [9]).

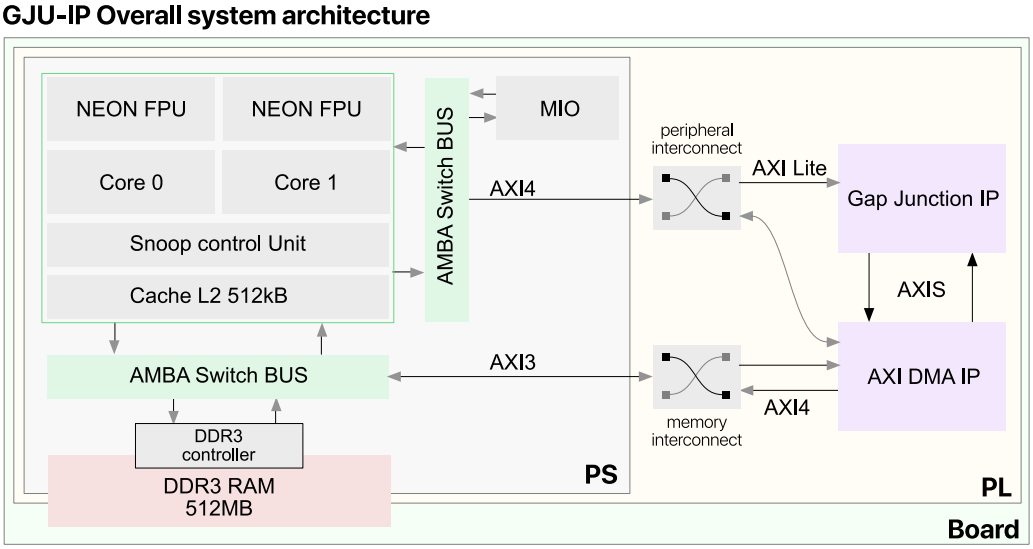


Figure 2.2: Overall system architecture. The GJU-IP is managed by one thread of the ARM A9, via AXI4, while the other thread handles the axon, soma and dendritic compartmental models, and takes care of the system's I/O.

Listing 2.1: Gap-junction pseudocode

```
float Vdend[N];
float Iout[N];
float Conn[N][C];
for(indxNeu=0; indxNeu<N; indxNeu++){
  float facc=0; float vacc=0;
  for(indxCon=0;indxCon<C;indxCon++){
    v=Vdend[indxNeu]-Vdend[indxCon];
    f=v*expf(-v*v*0.01);
    facc+=Conn[indxNeu][indxCon]*f;
    vacc+=Conn[indxNeu][indxCon]*v;
  }Iout[indxNeu]=0.8*facc+0.2*vacc;
}
```

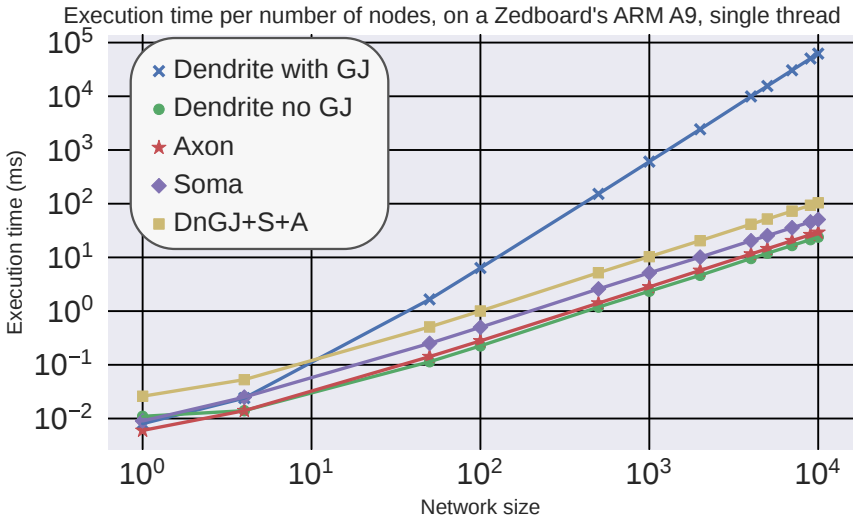


Figure 2.3: ZedBoard's ARM A9 execution time per compartmental model of the eHH, SPFP, single-threaded implementation. The *DnGJ+S+A* curve plots the sum of the three compartments, excluding the GJ. The soma compartment dominates, once the GJU is excluded from the dendrite compartment.

Chapter 3

Gap Junction Unit Implementation Using Data Flow HLS Optimizations

3.1 Defining GJU computation algorithm

As seen in Listing 2.1, the computation of the dendritic currents for each cell (`Iout[N]`, requires the traversing for each neuron of the network of a $N \times N$ conductance matrix `Conn[N][C]`). This matrix defines which cells in the $N \times N$ sized network share synaptic connections and thus computation also requires as inputs the dendrite voltages of each neighboring cell, `Vdend[N]`.

Now, instead of loading the complete conductance matrix along with the dendrites' output voltages into the GJU, some data restructuring is carried on. This subdivision of the data allows for more efficient DMA block transfers, and also makes hardware optimization of the GJ operations easier (as more logic, DSPs blocks, FFs and BRAM become available for data processing, instead of being occupied with data tables). Let the interaction among neurons be defined by a conductance matrix $\mathbf{C} \in \mathbb{R}^{\mathbf{N}\times\mathbf{N}}$ where $\mathbf{C} = [\mathbf{c}_1,\mathbf{c}_2,\ldots,\mathbf{c}_N]$, and with each dendrite's output voltage stored in a vector as $\mathbf{v} \in \mathbb{R}^{\mathbf{N}}$.

Let us define now a vector storing the potential difference between a particular dendrite and all of its neighbors,

$$\mathbf{v}_{\text{diff}}^{(i)} := \mathbf{v}[i]\vec{1} - \mathbf{v} \tag{3.1}$$

where $\vec{1} = [1,\ldots,1]$. For each i neuron, one can create a vector $\mathbf{f}^{(i)} \in \mathbb{R}^N$ where each element $\mathbf{f}^{(i)}[j]$ is paired with an element $\mathbf{v}_{\text{diff}}^{(i)}[j]$ as

$$\mathbf{f}^{(i)}[j] = \mathbf{v}_{\text{diff}}^{(i)}[j] \exp\left[-(\mathbf{v}_{\text{diff}}^{(i)}[j])^2/100\right] \tag{3.2}$$

One can now find the vector storing all the currents generated in each dendrite by the synaptic influence of its neighbors as defined by the eHH model, such that

$$\mathbf{i}_{\text{GJ}} = [I^{(1)}, I^{(2)}, \ldots, I^{(N)}] \in \mathbb{R}^{\mathbb{N}} \tag{3.3}$$

where

$$I^{(i)} = 0.8 f_{\text{acc}}^{(i)} + 0.2 v_{\text{acc}}^{(i)} \tag{3.4}$$
$$f_{\text{acc}}^{(i)} = \mathbf{f}^{(i)} \cdot \mathbf{c}'_i \tag{3.5}$$
$$v_{\text{acc}}^{(i)} = \mathbf{v}_{\text{diff}}^{(i)} \cdot \mathbf{c}'_i \tag{3.6}$$

Note that the dot product in (3.5) and (3.6) can be divided in subsets of products of some arbitrary number of M samples. If one defines the amount of sub-products as $K := \text{ceil}(N/M)$, then its recursive sequence for $k \in \{0, \ldots, K-1\}$ is

$$f^{(i)}(-1) = v^{(i)}(-1) = 0$$
$$f^{(i)}(k) = \mathbf{f}^{(i)}[kM : (k+1)M] \cdot \mathbf{c}'_i[kM : (k+1)M] + f^{(i)}(k-1)$$
$$v^{(i)}(k) = \mathbf{v}_{\text{diff}}^{(i)}[kM : (k+1)M] \cdot \mathbf{c}'_i[kM : (k+1)M] + v^{(i)}(k-1)$$
$$\Rightarrow f^{(i)}(K-1) = f_{\text{acc}}^{(i)}, v^{(i)}(K-1) = v_{\text{acc}}^{(i)}$$

Finally, note that the computing of M sub-products $f^{(i)}(k)$ and $v^{(i)}(k)$ share the same input $\mathbf{v}[kM : (k+1)M]$, then if $\mathbf{v}[iM : (i+1)M]$ is available, $\mathbf{V}_{\text{diff}}^{(iM:(i+1)M)}$ can be computed on the fly with each step k. Thus, each element in the sub-products can be executed in parallel and accumulated as

$$I^{(iM:(i+1)M)}(k) = 0.8 f^{(iM:(i+1)M)}(k) + 0.2 v^{(iM:(i+1)M)}(k) + I^{(iM:(i+1)M)}(k-1) \tag{3.7}$$

where $I^{(iM:(i+1)M)}(k)$ can be understood as the processing of sub-blocks $\mathbf{c}'_{(iM:(i+1)M)}[kM : (k+1)M]$ of size $M \times M$ and subsets $\mathbf{v}[kM : (k+1)M]$ of length M. Thus, after K sub-blocks and sub-vectors from $\mathbf{v}$ are processed, the final result are M values $I^{(iM:(i+1)M)}$. Some processing overhead is required in order to create the sub-vectors of the conductance matrix ($\mathbf{c}'_i$) and the dendrite's voltages, and sorting the final results. Besides, the DMA block size ought not be so small such that DMA transfers become impractical: an optimum block size need be determined in the design space. Also, the described algorithm is also known in other literature as a strip-mined optimization as a coding pattern (see [29]). A pseudo-algorithm code example for the GJ is shown in 3.1.

Listing 3.1: Gap-junction strip-mined optimization pseudocode (one strip)

```
VRow[STRIP_SIZE];
VCol[STRIP_SIZE];
popFirstV(input,VRow,VCol);
Facc[STRIP_SIZE]; Vacc[STRIP_SIZE];
while(count<ColBlockProc){
  for(row=0;row<STRIP_SIZE;row++){
    cond[STRIP_SIZE];
    popCond(input,cond);
```

```
   popV(input,VCol);
   for(col=0;col<STRIP_SIZE;col++){
   #pragma HLS UNROLL
    V = VRow[row] - VCol[col];
    F = V * expf(V * V * hundred);
    Facc[row] += F * cond[col];
    Vacc[row] += V * cond[col];
   }
  }count+=STRIP_SIZE
}
I[STRIP_SIZE];
for(row=0;row<STRIP_SIZE;row++){
 I[row]=0.8*Facc[row]+0.2*Vacc[row];
}pushI(output,I);
```

3.2 Applying Hardware Architecture Design Patterns for Improving the GJ Accelerator in HLS

An GJU was designed using C++ and compiled to RTL using Vivado HLS with the strip-mined loop transformation [29] previously detailed. Following the block diagram shown in 2.2: note that in listing 3.1, the `input` and `output`interfaces match the AXI-Stream interfaces, these ports and a configuration AXI4-Lite port will define the main interfaces for the GJU. The Xilinx's AXI-DMA IP feeds the voltage from the dendrite and the interconnect conductance values to the interface, then writes back to the RAM the generated current values.

Two specific considerations were carried out: the GJU input bus-width is set to 64 bit, and allocating the dendrite's voltage vector on BRAM while executing the GJ interactions (eliminating the cost of DRAM fetching). The design philosophy tried to follow a structured methodology to produce efficient C++ code that translate to optimal RTL in HLS. These guidelines were taken from [29]. This methodology tries to reproduce RTL stream blocks by representing explicit data-paths and its latency constraints.

3.2.1 Architectural design and coding methodology

As [29] indicates, two different but equivalent C++ portions of code producing the same results, will not necessarily be ported to the same hardware processing structure, let alone the most efficient. The inherent dataflow/paralellism nature of hardware must somehow be made explicit to the HLS, which means that a detailed block diagram of the intended architecture must be constructed in order to facilitate the best resulting code in terms of the final generated hardware. This is a mandatory practice in Register Transfer Level

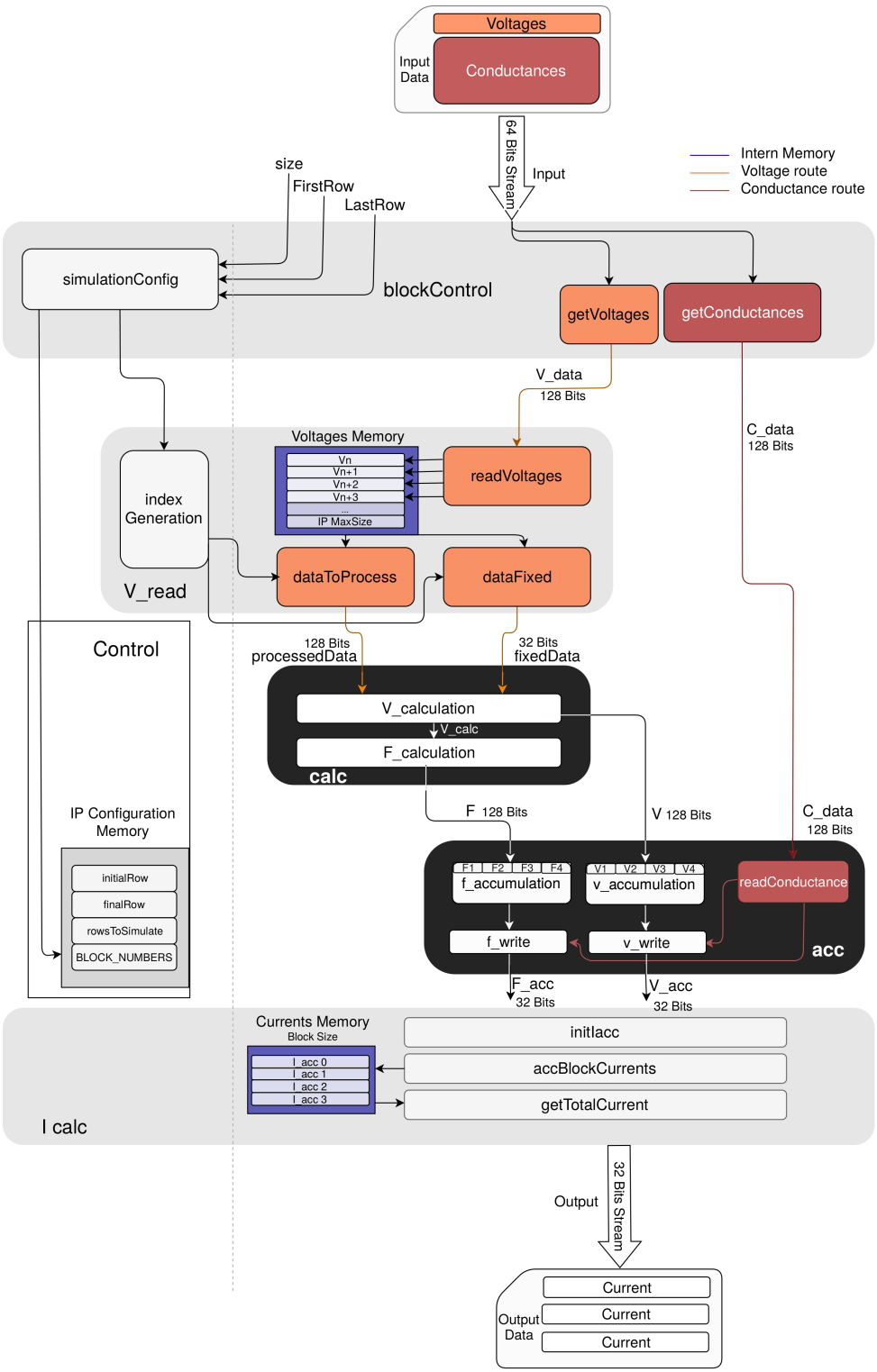


Figure 3.1: Overall GJU-IP architecture. The GJU-IP is composed of 5 block processes: `blockControl`, `V_read`, `calc`, `acc` and `Icalc`. Each process runs in parallel fashion and configured by control registers accessible via memory map AXI4-Lite interface.

hardware design, which requires at least high level architectural knowledge on behalf of the designer. Even though modern HLS tools ease the translation between the description of functional data processing algorithms and their hardware implementation, the timed, concurrent nature of hardware is still difficult to circumscribe using general programming constructs, which means that synthesis tools are still not capable of generating optimal solutions without guidance.

Therefore, an iterative study of the required GJU operations was first carried out, in order to discover all data dependencies and thus create an efficient datapath. The final block design, shown in Fig. 3.1, is the result of such iterative process. It is composed of five software-programmable and independent modules. Each module shares configurable parameters which allow for partial or full execution of the GJ interactions, given a maximum defined number of rows in the conductance matrix and a maximum fixed cell population (which in this case was limited to 10000 cells in an all-to-all connection, due to system requirements constraint at this value, nonetheless on the board could fit more cells). Each module is synchronized by FIFO interfaces. The `blockControl` module receives the data from the AXI-DMA stream. Meanwhile, data is packaged in 128-bit words and written through the `V_read` and `acc` FIFO interfaces. The `V_read` module manages the storage of the updated dendrite's voltages in the local BRAM and fetches the voltages according to the access pattern required (based on the conductance matrix row-column indexes). The `calc` module computes the values of `v` and `f` (the same name variables as those from listing 2.1) in data words of four V_j; therefore, the `acc` module accumulates each word and sends each block component to `I_calc` module.

The data parsing model is shown in Fig. 3.2; here, the conductance matrix is swept row-wise, each row being divided into sub-matrices (this particular matrix accommodation allows for later partitioning of the network among several boards). Pipelining is used in order to traverse local rows from each sub-matrix (each row is composed of 128-bit words). The `I_calc` module writes to the output-stream in a burst of four I_{out} vectors, when the main row is completed. The design's performance is bound by the input-stream throughput (the time it takes to read each dendrite's voltage and associated conductances matrix from DRAM).

In order to translate into hardware the design in Fig. 3.1, the C++ code must be now be written using the guidelines given in [29], by identification the appropriate constructs that would described the intended hardware structure. The architecture is functionally translated by dividing each module as a task in which each communicator matches to a FIFO interface during the synthesis process.

Dividing the project in multiple independent modules allows for individual module optimization tuning and testing, and makes therefore for a more maintainable base code. Later performance inspection provides important feedback in order to balance the each module's optimization tweaking, such that there is no module faster in latency than the others (which could complicate general synchronization). A wrapper function is used to contain each modules: code listings 3.2 and 3.3 show the integration of each module in

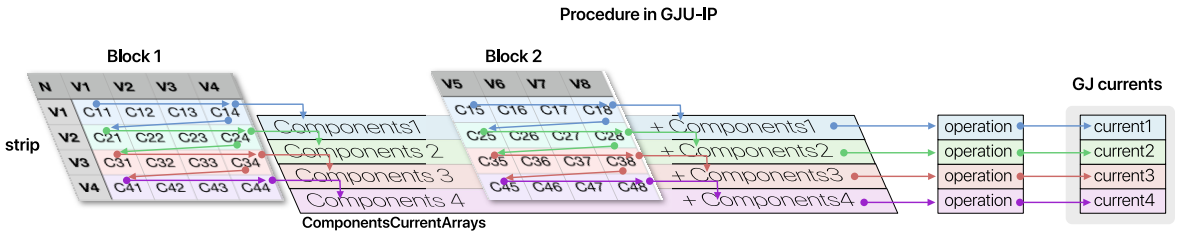


Figure 3.2: Representation of the execution procedure in the GJU-IP. The conductance matrix is divided in sub-matrices with 16 conductances each. Each row in the blocks is traversed in a pipelined fashion by the `calc` and `acc` modules. The results after processing each row are accumulated by the `I_calc` module which, after processing all data blocks in the sub-matrix, produces the results of four GJ currents.

order to form the GJU-IP main interfaces. Note that the specific `DATAFLOW` directive in listing 3.3 indicates to the HLS that processes are expected to execute concurrently.

Listing 3.2: Gap-junction unit wrapper pseudocode

```
void GapJunctionIP(
in64Bits &input, Stream &output,
int size,int FirstRow, int LastRow){
#pragma HLS INTERFACE axis port=input
#pragma HLS INTERFACE axis port=output
 Config simConfig;
 simulationConfig<Config>
     (simConfig,FirstRow,LastRow,size);
 execute(input,output,simConfig,size);
}
```

Listing 3.3: Gap-junction unit execution processes pseudocode

```
void execute(
in64Bits &input, Stream &output,
Config &simConfig,int size){
static 128bitStream Vdata("Vdata");
static 128bitStream Cdata("Cdata");
static 128bitStream pData("pData");
static Stream fData("fData");
static 128bitStream F("F");
static 128bitStream V("V");
static 128bitStream Facc("Facc");
static 128bitStream Vacc("Vacc");
#pragma HLS DATAFLOW
 blockControl(
   input,Vdata,Cdata,simConfig,size);
```

```
  V_read(
    Vdata, pData, fData, simConfig, size);
  calc(
    pData,  fData, F, V, simConfig);
  acc(
    F, V, Cdata, Facc, Vacc, simConfig);
  I_calc(
    output, Facc, Vacc, simConfig, size);
}
```

3.3 Results

Xilinx Vivado HLS is used for the implementation on a Zynq XZ7020 of the synthesized design. The GJU-IP, with all the required AXI4 interfaces runs at 120 MHz on the FPGA. The complete system performance is measured at this clock speed. The resources utilization is given in Table 3.1 an a comparison in Table 3.2, between the initial implementation [20] and the current's. Note that the current design needs about 41% less arithmetic primitives (multiply and accumulate units, DSPs in Xilinx lingo) and 43% less look-up tables (LUTs). Nonetheless, due the temporal storage of the dendritic voltages and the FIFO interfaces among the modules, about 14% more programmable LUTRAM and 600% more BRAM are needed. This is not so serious, any way, as total LUTRAM required is under 5% and BRAM at 30% of the XZ7020-1 resources count for each. Table 3.3 shows a comparison between the effective utilization of the FPGA resources required to solve SPFP operations for same ION model (although not the same connectivity scheme). The current work exhibits better FLOPS throughput per DSPs and LUTs.

Table 3.1: FPGA resource utilization summary based on the ZedBoard development platform. Note that room is still available if one were to fit another instance of the GJU-IP in order to parallelize simulations further, taking advantage of the four 64-bit AXI4-HP Bus channels available in the XZ7020-1.

Resources	This work	XZ7020-1	Total (%)
LUT	15 266	53 200	28.70
LUTRAM	846	17 400	4.86
FF	21 616	106 400	20.32
BRAM	42	140	30.00
DSP	91	220	41.36

Table 3.2: Comparison between initial version [20] and current's, in terms of FPGA resources utilization for the GJU. The use of LUTRAM and BRAM primitives increase because of the local storage of the dendritic voltages for the GJ execution. Still, major savings are noticeable.

Resources	Prior work	This work	Diff. (%)
LUT	26 877	15 266	↓43.20
LUTRAM	739	846	↑14.48
FF	27 468	21 616	↓21.30
BRAM	6	42	↑600.00
DSP	156	91	↓41.67

Table 3.3: Utilization of FPGA resources and performance capacity of SPFP operations executed on the FPGA fabric, compared against a all-to-all ION network [9] implementation, and a 8-way connectivity reported in [27]. The t_{step} column represent the execution time during one simulation step and the SimC column means the total of neuron population simulated. This work displays a more efficient performance density (ratio of FLOPS and FPGA resources) for the given DSPs and LUTs units.

Source	FPGA	f_{clk}	DSP	LUT	SimC	t_{step} (ms)	MFP opts	MFLOPS	MFLOPS /DSP	MFLOPS /LUT	FLOPS /(DSP $\cdot f_{clk}$)
This work	Zynq-7000	120MHz	91	15k	1056	4.8	13.38	2788	30.64	0.183	0.2532
					1188	6.03	16.94	2809	30.86	0.184	0.2551
Smaragdos [9]	Virtex7	100MHz	1600	251k	1056	1.1	14.29	12990	8.119	0.052	0.0812
Zjajo [27]	Virtex7	100MHz	1008	190k	1188	0.05	1.135	22690	22.51	0.012	0.2251

3.4 Conclusions

This chapter has reported the application of a hardware-oriented methodology based on HLS dataflow transformations, in order to improve FPGA-based HLS designs both in time performance and resources saved. As a study case, results on the acceleration of the simulation of a biologically accurate neural network on a heterogeneous SoC-FPGA platform have been presented. The final design consumes fewer resources and runs 10 to 4 times faster than a previous implementation

Chapter 4

Software Integration and SIMD Optimizations

4.1 Introduction

In this chapter, it is explained how it was integrated the software with the GJU to realize SNN-extended simulations. The block diagram for the complete system SoC architecture is shown in 2.2. Based of the system defined, its inferred that the software must be capable to handle the DMA transactions and maintain memory coherence between both devices. Also, it should configure the GJU registers according the simulation settings: `Network Size` and `Start`, via the AXI-Lite interface.

The execution of the eHH model should be distributed on two threads. One thread executes the soma and axon compartmental models, while the second manages the GJU-IP in order to carry out the dendrite compartment computations. There are multiple ways to cover this requirements with a software stack running in the ARM-A7 CPUs. The chosen option is to use an operating system (OS) image to facilitate the software development with high-level languages to integrate the usability of the GJU.

4.2 Software Stack

The operating system image created for this project is a composition of the following software technologies:

- **Operating System:** `linux-xlnx`, Linux forked project maintained by Xilinx to provide support for SoC-FPGA platforms such as Xilinx Zynq-7000 and Ultra-scale series.
- **Toolchain:** `GCC Linaro ARM32-hf toolchain`, GNU C/C++ compiler. Used to compile an image of the operating system and software developed for the platform.

- **Bootloader:** `Uboot`, bootloader image that fetches the location of the OS image into the RAM, and its parameters: disk location and device-tree.
- **File system image:** `Linaro Ubuntu image`, image that contains a file system release of Ubuntu which is compiled and distributed by Linaro. The file system image is paired with the toolchain used to compile them. For more details in how to flash a SD memory with the image see [20].
- **Linux Drivers:** there are two main drivers used to cover the functionality of the system:
  - Linux Userspace IO: driver framework to handle memory-mapped devices accessible to userspace. Drivers developed using this framework move the core logic to manage the device registers outside to the userspace instead of the kernelspace. This driver is used to handle the DMA transactions and the GJU configurations from the userspace software.
  - `udmabuf`: userspace mappable DMA buffer. It is used to handle the coherency between a DMA buffer mapped in RAM memory. This drivers takes a chunk of reserved memory from system DRAM (up to 128MB) and provide support to give a handle to the userspace application to use it and select when to rollback the cache data into the DRAM and vice-versa, therefore the DMA can read/write from the shared memory with the correct data. Original repository in [30].

4.2.1 SNN Simulator Application - ZedBrain

This application is developed in C++ languange. It creates simulations according the specification from a JSON configuration file. From the configuration file it loads the parameters required to unveil the execution of the simulation. It has three phases:

1. Phase 1: Load from configuration file the parameters, allocate the required memory according the need of the simulation, in function of the network size. Create the handles to the GJU and AXI-DMA devices.
2. Phase 2: Create a DMA buffer and load the constant conductances of the network.
3. Phase 3 — Execution loop: execute Soma and Axon calculations while Dedrites' GJ calculations finishes. From the DMA buffer is only required to update the dendrite voltages for each simulation step. When GJU is ready, it can be finalized the complete Dendrite computations. Finally, store the values of each Axon voltages for each cell.

4.3 Neon SIMD Engine

The Zynq-7000 processors are SIMD capable thanks to the Neon Engines. Each CPU core has one Neon compute unit. This engines are capable to compute with 64/128 bits words of data for both integer and floating point precision. The Neon engines are intended to improve performance for applications that do data processing, data conversions and memory accesses [31].

There is different ways to use this instructions, from a high-level, the compiler can makes efforts to auto-vectorize code sections according to a predetermined coding pattern [31] (support for this feature will vary between compilers and target hardware architecture). Another common practice is to forward code declarations in the code, also known as pragmas, to help the compiler to locate the auto-vectorization opportunities. Normally, this two methods are favored in standard code bases due to its portability and low investment in time-to-optimize.

If the constraints of the project requires to obtain the best performance possible, the programmer can accommodate in a assembly-like style the program by using intrinsics in the code [31]. These intrinsic are functions and data-types that serves as a wrapper to the programmer to manually select and use individual SIMD instruction. By using intrinsics, the programmer is the responsible to properly vectorize the code but without enter in such detail as writing assembly code. Its major drawback is to lose code portability.

4.4 Vectorization of eHH Inferior Olivary functions

The eHH computational model is divided in functions to calculate each ion current for each cell compartment (see Fig. 2.1). These functions are presented in the Table 4.1. The intrinsic functions and datatype selected to use are shown in Table 4.2, note that these intrinsics work with vector words of **four** FP values [31]. Rewriting the inferior olivary code to use NEON SIMDs is not a clear-cut task therefore, as it should be, the first step is to review how in the original code deal with the cell state variables in memory. In the code listing shown in 4.4, is shown an example of how is written the cell state in the original code; note that the internal variables of the cell are packed in memory consecutively and this causes issues by creating the NEON datatype `float32x4_t` because similar state variables should be gathered manually and probably it will cause several cache misses.

The main change to facilitate the vectorization of the functions is to modify the declaration of the cell state, by grouping the variables of each cell on a memory allocated space according its field. In this way, the load and store is more cache friendly and by having many local vector variables of the same state field, the vectorization is straight forward due to having to port directly each scalar operation inside the functions in Table 4.1 to a vector version from Table 4.2. An example of this kind of structure is shown in the code listing 4.4.

The final step to complete the vectorization, is to implement a NEON version of the exponential function (e^x). This was solved by using an open source implementation of the vector exponential function written by [32]. This vector function receives and returns the same datatype of `float32x4_t` as the other ones.

Table 4.1: The functions targeted to optimize with code vectorization. Each variable in **Inputs** and **Outputs** columns uses single precision FP data. For a neural network, during a simulation step each function is called once for each neural cell.

Functions to optimize	**Inputs**	**Outputs**
DendHCurr	*prevHcurrent_q, prevVdend*	*Hcurrent_q*
DendCaCurr	*prevCalcium_r, prevVdend*	*Calcium_r*
DendKCurr	prevPotassium_s, prevCa2Plus	*Potassium_s*
DendCal	*prevCa2Plus, prevI_CaH*	*Ca2Plus*
DendCurrVolt	*Ic, Iapp, Hcurrent_q, Calcium_r, Potassium_s, prevVdend, prevVsoma*	*Vdend, I_CaH*
SomaCalcium	*prevCalcium_k, prevCalcium_l, prevVsoma*	*Calcium_k, Calcium_l*
SomaSodium	*prevSodium_h, prevVsoma*	*Sodium_m, Sodium_h*
SomaPotassium	*prevPotassium_n, prevPotassium_p, prevVsoma*	*Potassium_n, Potassium_p*
SomaPotassiumX	*prevPotassium_x_s, prevVsoma*	*Potassium_x_s*
SomaCurrVolt	*prevVdend, prevVsoma, prevVaxon, Calcium_k, Calcium_l, Sodium_h, Potassium_n, Potassium_p, Potassium_x_s*	*Vsoma*
AxonSodium	*prevSodium_h_a, prevVaxon*	*Sodium_h_a, Sodium_m_a*
AxonPotassium	*prevPotassium_x_a, prevVaxon*	*Potassium_x_a*
AxonCurrVolt	*prevVsoma, prevVaxon, Sodium_h_a, Sodium_m_a, Potassium_x_a*	*Vaxon*

4.5 Results

To proof the improvements in the code, it was written a set of benchmarks using the Google Benchmark library [34]. The measurements consisted in a simple time execution comparison between the scalar function runtime four times versus the vectorized function

Table 4.2: Basic intrinsic functions and datatypes supported by ARM

Scalar DataType	**NEON Datatype**	**Description**
float	`float32x4_t`	Packed struct that holds four FP values. Datatype used in NEON Intrinsics.
Scalar Operation	**NEON Intrinsics**	**Description**
Addition	`vaddq_f32(float32x4_t a, float32x4_t b)`	FP vector addition.
Multiplication	`vmulq_f32(float32x4_t a, float32x4_t b)`	FP vector multiplication.
Negative	`vnegq_f32(float32x4_t a)`	FP vector negate sign.
Reciprocal	`vrecpeq_f32( float32x4_t a)` `vrecpsq_f32( float32x4_t a, float32x4_t b)`	The reciprocal operation is divided in two steps: reciprocal estimate and reciprocal step. The use of these intrinsics imply a Newthon-Rhapson method to calculate the reciprocal. See [33] for more information.
Load	`vld1q_f32(float *ptr)`	Read and load from memory a structure of four FP values.
Store	`vst1q_f32(float *ptr, float32x4_t a)`	Store from struct of four FP values to memory.

Listing 4.1: Pseudocode example of a scalar struct that holds the state variables of a cell

```
typedef struct Cell{
        float Var1;
        float Var2;
        ...
        float VarX;
}Cell;

Cell NeuralNetworkState[N];
```

Listing 4.2: Pseudocode example of a vector friendly struct that holds the state variables of a cell

```
typedef struct NeuralNetworkState{
        float *Var1;
        float *Var2;
        ...
        float *VarX;
}NeuralNetworkState;
```

with the NEON improvements; the benchmarks were run in hardware using the ZedBoard development board. The results are shown in Fig. 4.1, note that the functions `CompDend`, `CompSoma` and `CompAxon` are wrappers that internally make calls for the other optimized functions, therefore they are seen as an accumulative improvement with the other functions. `CompDend` improves by a factor of ≈ 4, `CompSoma` with a factor of ≈ 4.5 and `CompAxon` by ≈ 4.2.

To guarantee that the integrity of the precision is kept the same, the code was tested using multiple written tests using the Google test library [35]. The method consisted of comparing the outputs of the each of the functions versus its scalar implementations by bounding a tolerance of 0.0001%.

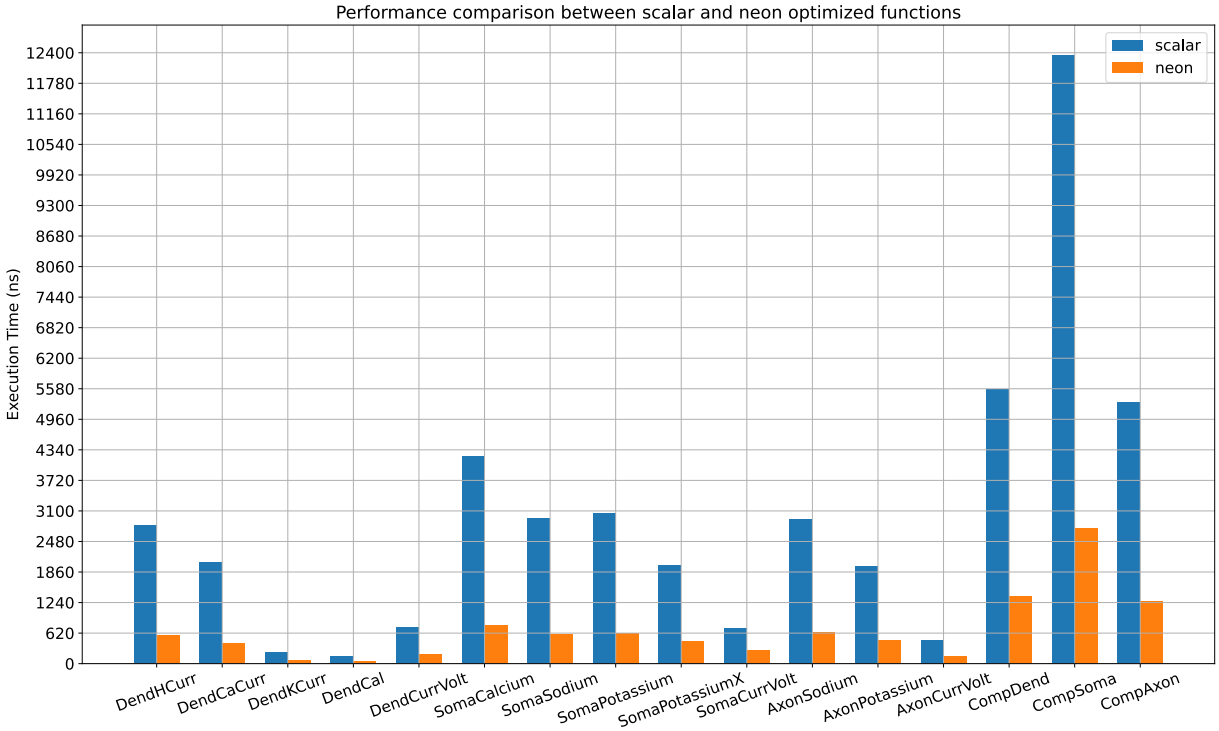
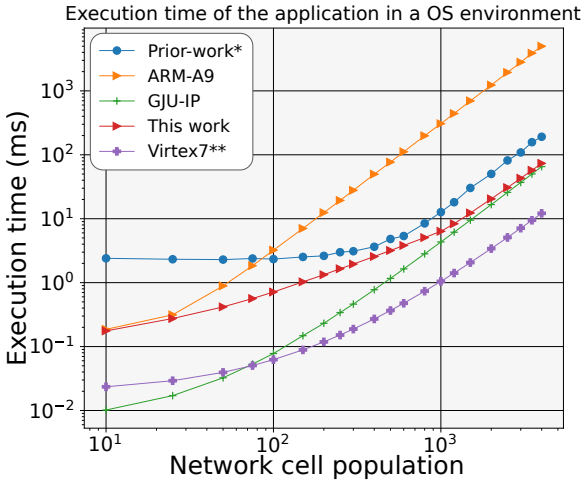


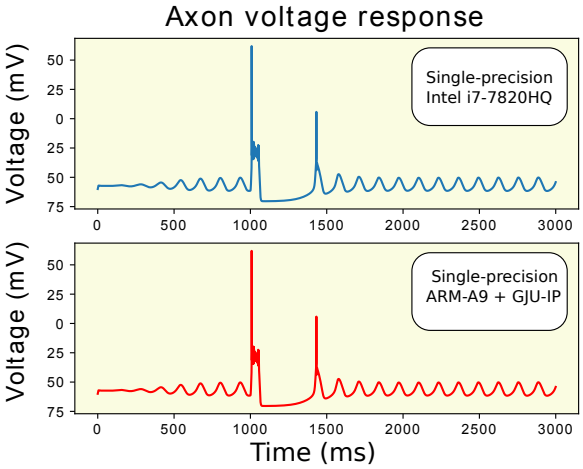
Figure 4.1: Performance comparison between the original scalar code versus the NEON version. The scalar execution time is measured by running the same function call four times.

4.5.1 System integration results

Figure 4.2a compares the execution-time of the proposed system (*sw hw accel*), for networks of different sizes, against C code running on the single-threaded 32-bit Zynq's ARM A9 @666MHz (*sw*), a single-threaded NEON SIMD optimized version on the same core (*sw simd*), a Quad Core Intel 64-bit i7-7820HQ @3.9GHz running the AVX2 SIMD optimized code on a single thread (*i7 sw simd*), and data from an eHH HLS version (*virtex hw*) completely ported to a Virtex XC7VX485T @100Mhz, taken from [9]. The improvement is of at least one magnitude order, when comparing the *sw hw accel* solution against the *sw simd* option, and it is equivalent to the *i7 sw simd* option. The implementation, nonetheless, underperforms by one magnitude order against what's reported in [9], but the latter running on a 10 times more expensive Virtex-7 board, and without scalability outside the board.



(a) Performance comparison between the implementations of the eHH model for different populations sizes. The GJU-IP curve points out the execution time latency of only the FPGA.



(b) Single neuron time response comparison against C++ golden reference software model for a 1000 ION eHH network, connected all-to-all.

Figure 4.2: Results from evaluating the ION eHH model, with the GJU-IP running on a ZedBoard's Zynq XZ7020-1. The soma, axon and dendrite compartments are executed on the Zynq PS, under Linux. The PL runs at a 120 MHz clock frequency. Average computation performance is given in sub-figure 4.2a for a single step. Sub-figure 4.2b shows that error ϵ is under 0.00001.

4.6 Conclusions

In this chapter it was determined how the overall system integration works, how the code modifications in the data structures and the use of NEON intrinsic functions reduced four times the execution time of the inferior olivary eHH model implemented; and the

performance obtained in the complete system.

Chapter 5

Conclusions

A heterogeneous implementation of the eHH model on an Avnet Zynq-7020 ZedBoard has been presented. A comparison of performance against previous work of this thesis author and a baseline model performance extrapolated from results from [9] is shown in Fig. 4.2a (logarithmic scale on both axes). Average resolution time for a computational step is improved here 4× against results in [20] for ION cell populations of 1000 cells and more, while reaching to 10× and more for ION cell populations under 1000 cells (in an all-to-all connection scheme). The system is now only an order of magnitude slower that the results given by [9] for a 100 cells ION simulation, on a much bigger FPGA. The simulation timing step, nonetheless, is still an order of magnitude over the required 50μs for converging to real brain timing. For reference, a comparison of a simulation output is shown in Fig. 4.2b. Note here how the error extracted from this simulation case is bounded at 0.00001 for the worst case.

The time required to generate three seconds of brain activity in this implementation takes about 9 minutes (PS@666MHz, PL@120MHz), while a multi-threaded PC implementation completes in 2:40 minutes (i7-7820HQ@3.9GHz). That's only a third of the speed for a much slower, cheaper alternative, both in cost and in power needs, the latter implied in the total power consumption reported by Vivado for this design: two Watts, against the 45 Watts reported for a i7-7820HQ@3.9GHz at full resources utilization (as reported in [36]).

In this project, the main bottleneck seen is the AXI4 buses that reads from the DRAM. Future work could treat the conductances as constant and work with them as binary values; also separate the conductances storage to another DRAM only accessible to the FPGA.

Bibliography

[1] G. T. Einevoll, A. Destexhe, M. Diesmann, S. Grün, V. Jirsa, M. de Kamps, M. Migliore, T. V. Ness, H. E. Plesser, and F. Schürmann, "The Scientific Case for Brain Simulations", *Neuron*, vol. 102, no. 4, pp. 735–744, 2019, ISSN: 10974199. DOI: 10.1016/j.neuron.2019.03.027.

[2] E. DAngelo, G. Danese, G. Florimbi, F. Leporati, A. Majani, S. Masoli, S. Solinas, and E. Torti, "The human brain project: High performance computing for brain cells Hw/Sw simulation and understanding", *Proceedings - 18th Euromicro Conference on Digital System Design, DSD 2015*, pp. 740–747, 2015. DOI: 10.1109/DSD.2015.80.

[3] C. L. Martin and M. Chun, "The BRAIN Initiative: Building, Strengthening, and Sustaining", *Neuron*, vol. 92, no. 3, pp. 570–573, 2016, ISSN: 10974199. DOI: 10.1016/j.neuron.2016.10.039. [Online]. Available: http://dx.doi.org/10.1016/j.neuron.2016.10.039.

[4] X. Fan and H. Markram, "A brief history of simulation neuroscience", *Frontiers in Neuroinformatics*, vol. 13, no. May, pp. 1–28, 2019, ISSN: 16625196. DOI: 10.3389/fninf.2019.00032.

[5] H. Sprekeler, G. Deco, and W. Gerstner, "Theory and Simulation in Neuroscience", *Science*, vol. 338, no. 6103, pp. 60–65, 2012, ISSN: 1095-9203. DOI: 10.1126/science.1227356. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/23042882.

[6] A. L. Hodgkin and A. F. Huxley, "A Quantitative Description of Membrane Current and its Application to Conduction and Excitation in Nerve", *Journal of Physiology*, vol. 1, no. 117, pp. 500–544, 1952, ISSN: 09237984. DOI: 10.1080/00062278.1939.10600645. arXiv: NIHMS150003.

[7] M. L. Hines, T. Morse, M. Migliore, N. T. Carnevale, and G. M. Shepherd, "ModelDB: A Database to Support Computational Neuroscience.", *Journal of computational neuroscience*, vol. 17, no. 1, pp. 7–11, 2004, ISSN: 09295313. DOI: 10.1023/B:JCNS.0000023869.22017.2e. arXiv: arXiv:1112.2903v1.

[8] A. V. Herz, T. Gollisch, C. K. Machens, and D. Jaeger, "Modeling single-neuron dynamics and computations: A balance of detail and abstraction", *Science*, vol. 314, no. 5796, pp. 80–85, 2006, ISSN: 00368075. DOI: 10.1126/science.1127240.

[9] G. Smaragdos, S. Isaza, M. F. van Eijk, I. Sourdis, and C. Strydis, "FPGA-based biophysically-meaningful modeling of olivocerebellar neurons", *FPGA*, pp. 89–98, 2014. DOI: 10.1145/2554688.2554790.

[10] M. Hennig, "Modelling Synaptic Transmission", *Modelling Synaptic Transmission*, pp. 1–18, 2005. DOI: 10.1088/1478-3975/4/1/001. [Online]. Available: http://homepages.inf.ed.ac.uk/mhennig/synaptic%7B%5C_%7Dtransmission.pdf.

[11] S. J. van Albada, A. G. Rowley, J. Senk, M. Hopkins, M. Schmidt, A. B. Stokes, D. R. Lester, M. Diesmann, and S. B. Furber, "Performance comparison of the digital neuromorphic hardware SpiNNaker and the neural network simulation software NEST for a full-scale cortical microcircuit model", *Frontiers in Neuroscience*, vol. 12, no. MAY, pp. 1–20, 2018, ISSN: 1662453X. DOI: 10.3389/fnins.2018.00291.

[12] 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", *Proceedings of the 2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines, FCCM 2012*, pp. 133–140, 2012, ISSN: 1467316059. DOI: 10.1109/FCCM.2012.32.

[13] P. Moorthy and N. Kapre, "Zedwulf: Power-performance tradeoffs of a 32-node Zynq SoC cluster", in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, May 2015, pp. 68–75. DOI: 10.1109/FCCM.2015.37.

[14] M. Véstias and H. Neto, "TRENDS OF CPU , GPU AND FPGA FOR HIGH-PERFORMANCE COMPUTING", *24th International Conference on Field Programmable Logic and Applications (FPL)*, 2012.

[15] T. Stephen, "Three Ages of FPGAs : A Retrospective on the First Thirty Years of FPGA Technology", *Proceedings of the IEEE*, vol. 103, no. 3, pp. 318–331, 2015.

[16] R. Kobayashi, Y. Oobata, N. Fujita, Y. Yamaguchi, and T. Boku, "OpenCL-ready High Speed FPGA Network for Reconfigurable High Performance Computing", *HPC Asia*, 2018. DOI: 10.1145/3149457.3149479.

[17] S. Lahti, P. Sjovall, J. Vanne, and T. D. Hamalainen, "Are We There Yet? A Study on the State of High-Level Synthesis", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 5, pp. 898–911, 2019, ISSN: 02780070. DOI: 10.1109/TCAD.2018.2834439.

[18] H. R. Zohouri, "High performance computing with fpgas and opencl", *CoRR*, 2018. [Online]. Available: http://arxiv.org/abs/1810.09773.

[19] E. S. Chung, P. A. Milder, J. C. Hoe, and K. Mai, "Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs?", *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, pp. 225–236, 2010, ISSN: 10724451. DOI: 10.1109/MICRO.2010.36.

[20] 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 Systems (LASCAS)*, Feb. 2019, pp. 5–8. DOI: 10.1109/LASCAS.2019.8667538.

[21] K. Alfaro-badilla, A. Arroyo-Romero, L. León-Vega, C. Salazar-garcía, J. Espinoza-gonzalez, F. Hernández-Castro, G. Smaragdos, C. Strydis, and A. Chacón-Rodríguez, "Improving the Simulation of Biologically Accurate Neural Networks Using Data Flow HLS Transformations on Heterogeneous SoC-FPGA Platforms", *Latin America High Performance Computing Conference (CARLA) 2019*, pp. 1–15, 2019.

[22] J. R. de Gruijl, P. Bazzigaluppi, M. T. de Jeu, and C. I. de Zeeuw, "Climbing Fiber Burst Size and Olivary Sub-threshold Oscillations in a Network Setting", *PLoS Computational Biology*, vol. 8, no. 12, 2012, ISSN: 1553734X. DOI: 10.1371/journal.pcbi.1002814.

[23] 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", *IOP*, vol. abs/1612.01501, 2016. arXiv: 1612.01501.

[24] 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 simulations", *ISPASS 2016 - International Symposium on Performance Analysis of Systems and Software*, pp. 1–11, 2016. DOI: 10.1109/ISPASS.2016.7482069.

[25] D. Soudris, I. Sourdis, Z. Al-Ars, H. Sidiropoulos, C. I. De Zeeuw, G. Chatzikonstantis, G. Smaragdos, C. Kachris, C. Strydis, R. Kukreja, and D. Rodopoulos, "BrainFrame: a node-level heterogeneous accelerator platform for neuron simulations", *Journal of Neural Engineering*, vol. 14, no. 6, p. 066 008, 2017, ISSN: 1741-2560. DOI: 10.1088/1741-2552/aa7fc5.

[26] G. Smaragdos, G. Chatzikostantis, S. Nomikou, D. Rodopoulos, I. Sourdis, D. Soudris, C. I. D. Zeeuw, and C. Strydis, "Performance analysis of accelerated biophysically-meaningful neuron simulations", *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 1–11, Apr. 2016. DOI: 10.1109/ISPASS.2016.7482069.

[27] 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, 2018, ISSN: 19324545. DOI: 10.1109/TBCAS.2017.2780287.

[28] 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, ISSN: 18792782. DOI: 10.1016/j.neunet.2017.09.011. [Online]. Available: https://doi.org/10.1016/j.neunet.2017.09.011.

[29] J. de Fine Licht, S. Meierhans, and T. Hoefler, "Transformations of high-level synthesis codes for high-performance computing", *CoRR*, vol. abs/1805.08288, 2018. arXiv: 1805.08288. [Online]. Available: http://arxiv.org/abs/1805.08288.

[30] K. Ichiro, *Udmabuf*, https://github.com/ikwzm/udmabuf, 2020.

[31] A. Company, "Introducing neon development article", 2009. [Online]. Available: http://www.arm.com.

[32] J. Pommier, *Neon_mathfun library*, http://gruntthepeon.free.fr/ssemath/neon_mathfun.h, 2011.

[33] A. Company, "Neon: Programmer's guide", 2013. [Online]. Available: http://www.arm.com.

[34] Google, *Google benchmark library*, https://github.com/google/benchmark/, 2022.

[35] ——, *Google test library*, https://github.com/google/googletest/, 2022.

[36] Intel, *Intel® Core™ i7-7820HQ Processor*. Online; accessed 24 May 2019, 2019. [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/97496/intel-core-i7-7820hq-processor-8m-cache-up-to-3-90-ghz.html.