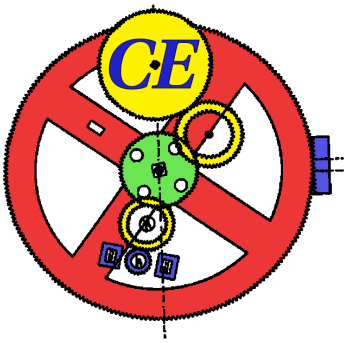


Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2020

MSc THESIS

Towards Real-Time Olivary Neuron Modeling

Nikolas Nicou

Abstract

The field of Computing has been a significant catalyst for innovation across various segments of our lives. Computational neuroscience keeps demanding increased performance to implement powerful simulators able to closely approximate brain behavior using complex mathematical models. This resulted in various High-Performance Computing systems able accelerate the above simulation workloads. One of the challenges is how these applications are being ported to massively parallel accelerators that requires significant time and effort for designing and debugging. This thesis primary task is to optimize an existing hardware library for neural simulation. The above library uses one of the most widely used biophysically-meaningful neuron models called Hodgkin-Huxley. The library optimizations will be performed while following a design methodology to accelerate applications on Maxeler's Data-Flow Engines (DFEs). A DFE is an FPGA-based accelerator incorporating a top-of-the-line reconfigurable device surrounded by high bandwidth, large capacity on-card memory. This work focused in the fully extended model that had room for performance improvements. The result, an optimized model that takes advantage of the FPGA capabilities and achieve up to 2.66x speed up over the previous implementation. They key to this speedup is the use of fixed-point arithmetic that provides 2x speed up compared to the optimized floating-point version. Additionally, the model is implemented in multiple kernels in such a way that can be scaled up using multiple DFEs to achieve even greater performance.

Q&CE-CE-MS-2020-12

TU Delft

Faculty of Electrical Engineering, Mathematics and Computer Science

Towards Real-Time Olivary Neuron Modeling

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Nikolas Nicou
born in Larnaca, Cyprus

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

Towards Real-Time Olivary Neuron Modeling

by Nikolas Nicou

Abstract

The field of Computing has been a significant catalyst for innovation across various segments of our lives. Computational neuroscience keeps demanding increased performance to implement powerful simulators able to closely approximate brain behavior using complex mathematical models. This resulted in various High-Performance Computing systems able accelerate the above simulation workloads. One of the challenges is how these applications are being ported to massively parallel accelerators that requires significant time and effort for designing and debugging. This thesis primary task is to optimize an existing hardware library for neural simulation. The above library uses one of the most widely used biophysically-meaningful neuron models called Hodgkin-Huxley. The library optimizations will be performed while following a design methodology to accelerate applications on Maxeler's Data-Flow Engines (DFEs). A DFE is an FPGA-based accelerator incorporating a top-of-the-line reconfigurable device surrounded by high bandwidth, large capacity on-card memory. This work focused in the fully extended model that had room for performance improvements. The result, an optimized model that takes advantage of the FPGA capabilities and achieve up to 2.66x speed up over the previous implementation. They key to this speedup is the use of fixed-point arithmetic that provides 2x speed up compared to the optimized floating-point version. Additionally, the model is implemented in multiple kernels in such a way that can be scaled up using multiple DFEs to achieve even greater performance.

Laboratory : Computer Engineering
Codenumber : Q&CE-CE-MS-2020-12

Committee Members :

Advisor:	Prof. Dr. Ir. Georgi N. Gaydadjiev, CE
Chairperson:	Prof. Dr. Ir. Georgi N. Gaydadjiev, CE
Member:	Dr. Ir. Christos Strydis, Erasmus MC
Member:	Dr. Ir. Chris Verhoeven, ELCA

Contents

List of Figures

List of Tables

List of Listings

Glossary

ASIC Application Specific Circuit.

BRAM Block Random-Access Memory.

CLB configurable Logic Block.

CPU Central Processing Unit.

DDR Double Data Rate.

DFE Data-Flow Engine.

DIMM Dual In-line Memory Module.

DSP Digital Signal Processor.

FIFO First In First Out.

FMem Fast Memory.

FPGA Field Programmable Gate Array.

GPU Graphics Processing Unit.

HH Hodgkin-Huxley.

HPC High Performance Computing.

IO Inferior Olive.

ION Inferior Olive Nucleus.

LMem Large Memory.

LUT Look up Table.

MAC Multiple and Accumulate.

PCIe Peripheral Component Interconnect express.

SLiC Simple Live CPU.

SLR Super Logic Region.

SLR Shift Register Logic.

SSI Stacked Silicon Interconnect.

TSV Through-Silicon via.

URAM Ultra Random-Access Memory.

Acknowledgements

I would like to thank my supervisor, Georgi Gaydadjiev, for the opportunity to work on this research project, and for his invaluable feedback and support throughout this long journey. Additionally, I would also thank Christo Strydis for being always available to talk and support me during this research project.

In addition, I would like to thank, George Smaragdos, Lukas Vermond, Rene Miedema, and especially Nils Voß for their technical help and support.

Last but not least I would like to thank my family and friends for supporting me and encouraging me during this chapter of my life.

Nikolas Nicou
Delft, The Netherlands
December 1, 2020

1 Introduction

During the last four decades, central processor units (CPUs) have made huge leaps in performance. The driving factor for this technological advancement is mainly due to Moore's Law that postulated that the number of transistors on a silicon chip doubles about every two years. Moore's prediction set the pace for our modern digital revolution [11]. In every new generation of CPUs, substantial improvements are made that unlocked the implementation of more complex and demanding applications. As a result, this expanded the user expectations and demands for the next generation of processors. During the last decade, the situation has changed, and this trend is gradually declining [12, 13, 14]. The ever increasing demands of High-Performance Computing (HPC) applications and Big Data, are not met with CPUs alone and alternative solutions are sought to meet this performance demands [15, 16].

The solution to this is the use of hardware acceleration that augments processor with application specific co-processors [17]. A good candidate that combines the right price, performance and power consumption for a co-processor is the Field Programmable Gate Array (FPGA) [15, 18]. FPGAs can be used for virtually any task due to their reconfigurable nature to speed up financial risk analytics, database acceleration, science, and engineering complex applications [19, 20, 21, 22, 23].

Building, biologically inspired systems, require computations with massive parallelism, and that is why FPGAs are an interesting option. One approach that balances power consumption, implementation complexity, and flexibility are Maxeler's Data-flow Engines (DFEs). A DFE is a special-purpose reconfigurable chip (FPGA) that can be reprogrammed at runtime [24].

In this thesis, the performance of the flexHH library used for neural simulations is improved by following the steps of the Maxeler's Technologies design process for HPC applications. This process prescribes how to implement complex applications using reconfigurable hardware and how to optimize these applications. The primary task is to minimize the time of development and the time spent on debugging. We start with the flexHH library analysis to estimate the performance and area requirements before implementation. The parts that will be accelerated on hardware are found, and the respective application's architecture is designed and then implemented. Different datatypes will be investigated during the application analysis and compare them to find the most fitting for our application. The encountered pitfalls and the decisions taken for each step are detailed explained. By using this acceleration process methodology, will help to optimize the performance of the flexHH library.

1.1 Motivation

The use of reconfigurable hardware accelerators means that hardware architecture is no longer fixed, and it can change during the application runtime. This benefits the use of FPGAs across any application, but it introduces many additional challenges. Having a variable hardware architecture requires a tremendous amount of time for its validation. Regardless of the time spent on validation, bugs, and architecture inefficiencies may be found during the late implementation stages, leading to a complete application redesign. Additionally, it is well known that FPGA development is very specialized and hence challenging, especially when full utilization of the FPGA capabilities are sought [25]. The time spent on going back and forth in a design leads to increased development time and effort. Furthermore, creating such applications in reconfigurable hardware without setting the requirements which come through application analysis may not lead to the expected outcome (e.g., runtime speedup). All aspects of the development cycle, application analysis, architecture analysis, implementation, and debugging are united in Maxeler's design methodology to program a DFE, an FPGA-based accelerator, and help overcome these challenges. This methodology considers all the possibilities and requirements before going to the implementation phase, providing a bug-free optimal architecture that maximizes system resource utilization.

The flexHH library simulates a part of the brain, which is considered a complex problem, and for this reason, hardware acceleration is employed. Brain simulations are important and can help us to understand further how the brain functions. In most cases, simulated mechanisms are based on hypotheses. Brain platforms than can simulate brain regions fast and in real-time could enable the neuro-scientific community to test these hypotheses. Furthermore, having a deeper understanding of brain functionality can lead to several critical practical applications. Achieving accurate enough real-time simulations can drive the development of **robotic prosthetics** and **implants** to restore lost brain functionality (**Brain rescue**). Moreover, understanding biological systems and having richer computational dynamics for their models can lead to more advanced **artificial-intelligence** and **robotics applications**. Finally, a better understanding of the brain workings can lead to new non-Von-Neumann **architectural paradigms** being used for the advancement of the computing field [24].

1.2 Thesis goal

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

- *Optimization in performance of an existing hardware library for Hodgkin-Huxley-based neural simulations*

As it is already mentioned, this thesis focuses on optimizing the performance of a hardware library used for neural simulations using Maxeler's design process that is useful and crucial to apply it while implementing HPC applications. All the steps needed to develop an HPC application that exploits the hardware resources to improve performance will be included. A fixed-point analysis will be conducted and the results will be compared

with floating-point. The goal is to prove that it is essential to consider fixed-point arithmetic and it will provide substantial improvements on performance over floating-point. Additionally, the performance and area estimations will be compared with the actual numbers resulting from the actual implementation. Ideally, the results will be as close as possible to the real numbers that will further leverage the use of Maxeler's design process.

1.2.1 Research Questions

The research questions of this thesis are as follows:

- Can fixed-point arithmetic provide substantial improvement on performance for flexHH library without losing substantial accuracy compared to floating-point;
- Can the models of the flexHH library be implemented in such a way to be scalable using multiple DFEs.

1.3 Thesis Organization

The arrangement of the rest of this thesis report is as follows. Chapter 2 presents a brief introduction regarding the brain, the data-flow paradigm, and FPGAs. In Chapter 3, the related work regarding the case study is briefly explained. Chapter 4 is emphasized in the design methodology process and all the analyses conducted. Chapter 5 describes the implementation and the modifications executed to reach the final application. The evaluation of the error in performance and resource usage estimated in our methodology and the comparison between floating-point and fixed-point applications are discussed in Chapter 6. Finally, Chapter 7 concludes this thesis with a discussion of the contributions and the proposed future work.

2 Background

In this chapter, the background information required to understand the remainder of this thesis is presented. In section 2.1, a brief description of the biological neuron is discussed. Parts of human brain are briefly explained in section 2.2, and more details about the cerebellum and the inferior olive nucleus are provided. Following, sections 2.3 and 2.4 introduce the Hodgkin-Huxley (HH) and the Inferior Olive (IO) models, respectively. In section 2.5, background knowledge regarding the technology used and its characteristics are discussed. Finally, in section 2.6 introduces dataflow computing and Maxeler's Dataflow Engines (DFEs).

2.1 The Biological Neuron

The human brain consists of billions of neurons also called (neural) cells within the nervous system that transmit, stores and process information throughout the human body. There are different types of neurons with varying kinds of morphology, as shown in Figure 2.1. However, a typical neuron can be described as three functionally distinct parts, also called compartments, the dendrites, the cell body or soma, and the axon, as illustrated in Figure 2.2. Each compartment has a membrane with ion channels. Depending on the state of the channels (open or closed), ions can flow in and out by changing the membrane's voltage potential. The change in the membrane potential can generate voltage signals, also called action potentials. Dendrites can be described as the input part of the neuron that collects signals from other neurons and transmits them to the soma. Next, the soma is where the processing is done; if the total input exceeds a certain threshold, it generates an output signal/action potential. Finally, the output signal is taken over by the axon that delivers the signal to other neurons. The junction between two neurons is called a synapse. Two kinds of synapses exist, the chemical and the electrical synapses. When an action potential arrives in a chemical synapse, it triggers a complex chain of biochemical processing steps. Specific channels open, and ions can flow into the cells that lead to a change of the membrane potential. Electrical synapses, or also called gap junctions, are specialized membrane proteins that directly connect two neurons. Details about the functional aspects of gap junctions are not known, but they are believed to be involved in the synchronization of neurons [2].

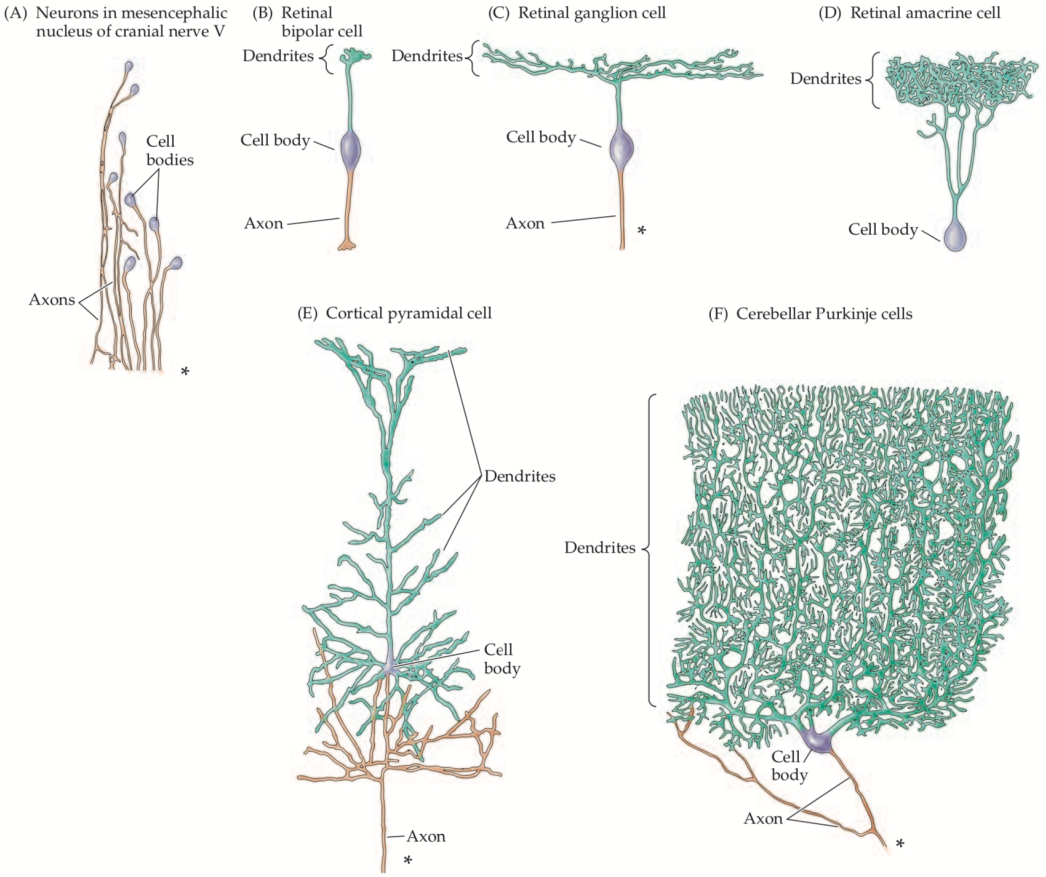


Figure 2.1: Examples of different types of nerve cells found in the human nervous system [1]

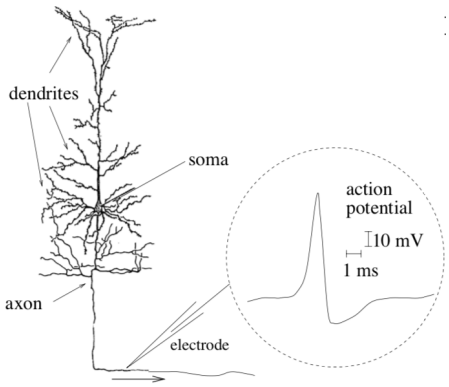


Figure 2.2: A single neuron in a drawing by Ramón y Cajal. Dendrite, soma, and axon compartments can be clearly distinguished. A spike (action potential) initiated by the axon is also shown [2]

2.2 The Human Brain

The brain is the overseer of the human body; it collects, processes, and coordinates all the information from the censoring network throughout the body, and it makes all the decisions. The brain is divided into three different regions, the **brainstem**, the **forebrain** and the **cerebellum** as depicted in Figure 2.3. The cerebrum and diencephalon together constitute the **forebrain**, the brainstem consists of the **midbrain**, the **pons**, and the **medulla oblongata** [3]. The spinal cord is a long, tube-like structure that starts at the end of the brainstem and continues down to the spine's bottom. It consists of nerves that convey incoming and outgoing messages between the brain and the nerves in the rest of the body.

Brainstem
All the signals between the spinal cord, forebrain, and the cerebellum pass through the brainstem. It receives and combines input from all the central nervous system (brain, spinal cord) and processes a great deal of neural information. Additionally, the brainstem is involved in motor functions, cardiovascular and respiratory control, and the mechanisms that regulate sleep, wakefulness, focus, and attention [3, 26, 27]. The forward-most portion of the brainstem is the **midbrain** and it serves essential functions in motor movement, particularly movements of the eye, and in auditory and visual processing [28]. The **pons** is located between the midbrain and the medulla oblongata, and it includes neural pathways that connect the medulla with the cerebellum. It is also the point of origin or termination for four cranial nerves that transfer sensory information and motor impulses to and from the facial region and the brain. Active functioning of the pons may also be fundamental to rapid eye movement (REM) sleep [29]. Finally, it works together with the medulla oblongata to serve an especially critical role in generating breathing's respiratory rhythm. **Medulla oblongata** is the lowest part of the brain and the lowest portion of the brainstem. The medulla oblongata plays a critical role in transmitting signals between the spinal cord and the higher parts of the brain. It controls autonomic activities, such as heartbeat and respiration [30].

Forebrain
The larger component of the forebrain, the **cerebrum**, includes all the higher mental functions such as thinking and human memory. It consists of two halves hemispheres; the right cerebral hemisphere that controls the left side of the body, and the left cerebral hemisphere that controls the right side of the body [26]. Both hemispheres are further divided into four areas, as shown in Figure 2.4, called lobes, and are explained below:

- **Frontal Lobe:**
  The frontal lobe is the largest lobe of the cerebrum and is responsible for many behavioral traits such as personality, problem-solving, decision-making, organization, and many more. The frontal lobe uses the information from the environment, memory, and emotions to make decisions, affecting the person's personality. To achieve these high cognitive functions, the frontal lobe needs to communicate with the rest of the brain and filter out the vast amount of information to what is important and relevant [26, 27].

- **Parietal Lobe:**
  The parietal lobe is vital for sensory perception, including the sense of touch, pressure, hearing, sight, and smell. Additionally, it integrates the information from our senses to provide functions such as spatial awareness, coordination of other parts of the body (coordination of hands, arms, and eye motions), the judgment of texture, weight, size, and shape [27, 31].

- **Occipital Lobe:**
  Occipital lobe processes visual signals that arrive from the eye's retina. While the retina is the part of the human body that detects colors, edges, and movement, the occipital lobe is responsible for informing us what we see [27].

- **Temporal Lobe**
  The temporal lobe is where the human memories are stored; this allows a human to distinguish sounds, smells, shapes, and people from one another. Additionally, emotions and language comprehension are also stored in the temporal lobe together with new information that is stored in the short-term memory [27, 26]

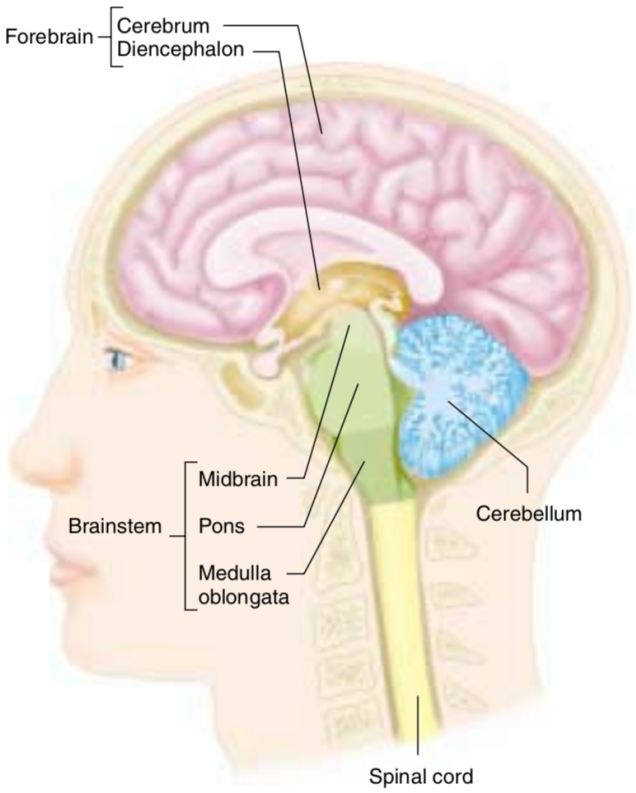


Figure 2.3: The divisions of the brain and the spinal cord [3]

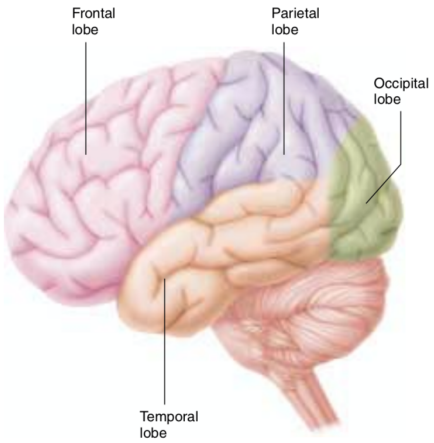


Figure 2.4: A lateral view of the brain divided in cerebrum's four lobes [3]

The second component of the forebrain is the **diencephalon** that contains two major parts: the **thalamus** and the **hypothalamus**. The thalamus's main function is to relay motor and sensory signals to the cerebral cortex (outer layer of the cerebrum). It also plays a key role in alertness and focused attention. Hypothalamus lies below the thalamus, and it is a tiny region that accounts for less than 1 percent of the brain's weight. Even so, it contains different cell groups and pathways that form the master command center for neural and endocrine coordination. The endocrine system is the collection of glands that produce hormones that regulate metabolism, growth and development, tissue function, sexual function, reproduction, sleep, and mood [3].

Cerebellum

The cerebellum is one of the smallest parts of the brain, but it contains over 50% of the total number of neurons, and it can be considered one of the most complex and dense regions of the brain. It does not initiate movement but influences the sensorimotor region that coordinates the body's activities and motor learning skills. Additionally, it plays a vital role in functions related to balance, posture, and sensing of rhythm that enables the handling of concepts such as music and harmony. Detecting errors in movement and adjusting the next set of movements to make them more accurate is also a cerebellum task. The cerebellum receives information from the muscles and joints, skin, eyes, ears, viscera, and the parts of the brain involved in controlling movement to carry out these functions. These activities are carried out automatically by this brain area and are not under a person's control. Until now, the cerebellum is not very well understood, and many studies are conducted to learn more about the purpose and its functionality [27, 26, 32, 33, 3, 24].

The communication between the cerebellum and other parts of the nervous system is conducted by three large pathways called cerebellar peduncles and are explained below:

1. **Superior cerebellar peduncle**
   The neurons that give rise to this pathway are in the deep cerebellar nuclei. Their axons project to the primary motor and premotor areas of the cortex through the

thalamus as shown in Figure 2.5 (b) [1].

2. **Middle cerebellar peduncle**
   The middle cerebellar peduncle is the largest; it connects the cerebellum to the pons and transmits information about the body parts' desired position. Most of the cells that create this pathway are in the base of the pons. These cells form the pontine nuclei that relay information from the cortex to the cerebellum, as illustrated in blue and yellow lines in Figure 2.5 (a) [34, 1].

3. **Inferior cerebellar peduncle**
   The inferior cerebellar peduncle brings sensory information about the actual position of the body parts. The vestibular nuclei and the spinal inputs (the dorsal nucleus of Clarke) provide the cerebellum with information. This information is coming from the labyrinth in the ear, from muscle spindles, and from other mechanoreceptors that monitor the body's position and motion. Finally, the entire cerebellum receives modulatory inputs from the inferior olive. Inferior olive nuclei evidently participate in the learning and memory functions served by cerebellar circuitry as depicted (green lines) in Figure 2.5 (a) [34, 1].

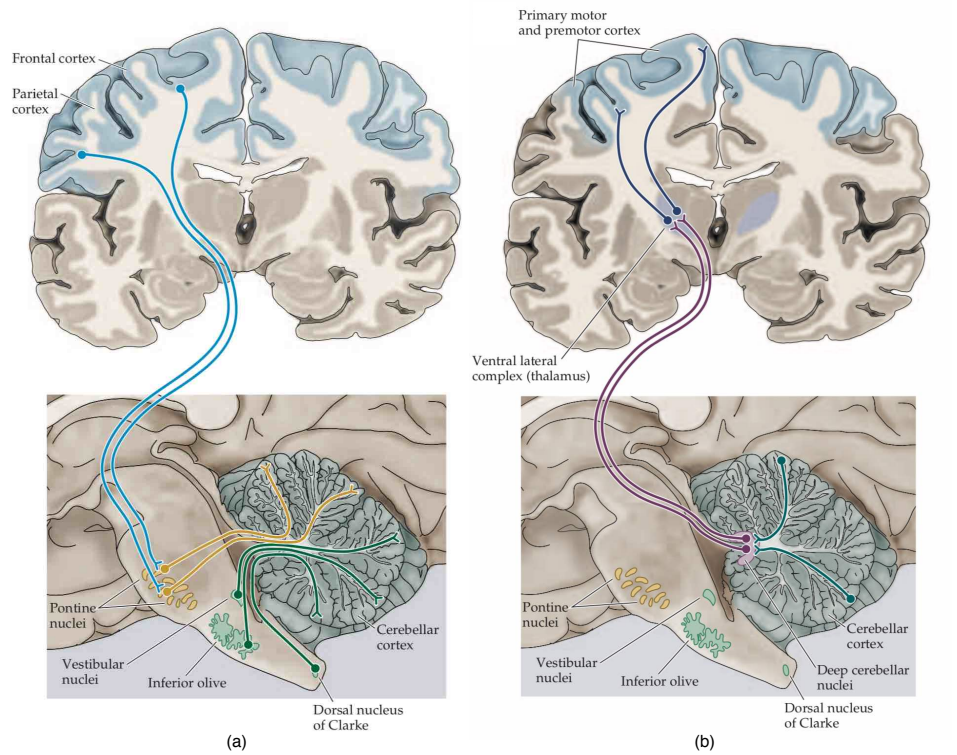


Figure 2.5: Cerebellum connections to the brainstem and cerebrum. (a) Illustrates the inputs to the cerebellum from the cortexes, spinal cord, and the brainstem. (b) Output from the cerebellum to primary and premotor-cortex [1]

Circuits within the Cerebellum

The final destination of the cerebellar cortex's inputs is a distinctive cell type called the Purkinje cell. The axons from the pontine nuclei and other sources are called mossy fibers that carry information from many brain regions, as shown in Figure 2.6. Mossy fibers synapse on granule cells that give rise to specialized axons called parallel fibers that ascend to the cerebellar cortex's molecular layer. The parallel fibers relay information via excitatory synapses onto the Purkinje cell's dendritic spines. Additionally, the Purkinje cells receive direct modulatory input on their dendritic shafts from the climbing fibers, which originates from the inferior olive. Climbing fibers carry information about sensory events such as a tactile sensation on a specific part of the skin.

Each Purkinje cell receives numerous synaptic contacts from a single climbing fiber. In most models that describe the cerebellum's functions, the climbing fibers regulate movement by modulating the effectiveness of the mossy and parallel fiber connections with the Purkinje cells. Finally, Purkinje cells project to the deep cerebellar nuclei being the only output cells of the cerebellar cortex. This primary circuit is repeated numerous times in the cerebellum and is considered a fundamental functional module of the cerebellum. Modulation of signal flow through these modules provides the basis for both real-time regulation of movement and long-term regulation changes that underlie motor learning [4, 1, 35].

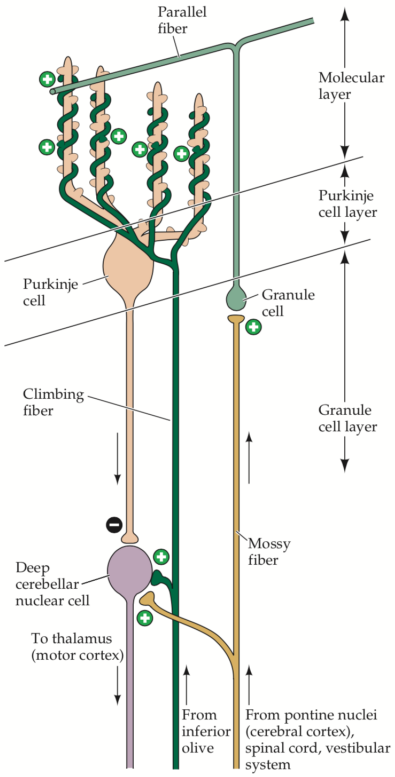


Figure 2.6: Olivocerebellar circuit [1]

A Purkinje cell receives a large number of parallel fiber inputs, which modulate its intrinsic rapid spiking behavior, and only one climbing fiber input. Nonetheless, when a climbing fiber action potential is fired, the ongoing Purkinje cell's rapid spiking behavior is interrupted as it generates a so-called complex spike, after which the Purkinje cell falls silent for approximately 20 ms as illustrated in Figure 2.7. Since the inferior olive nucleus gives rise to the climbing fibers, it plays a vital role in the cerebellum's functioning [4, 35]. The cells inside the inferior olive nucleus are also interconnected by purely electrical connections between their dendrites, called gap junctions. Gap junctions are considered important for synchronizing activity within the nucleus and, thus, greatly influencing movement and motor learning[33].

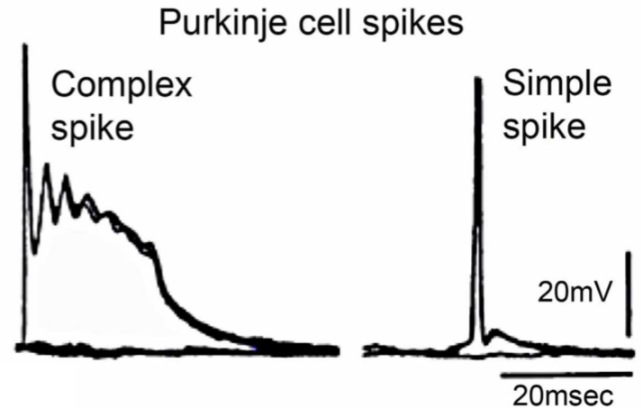


Figure 2.7: Intracellular recordings of a complex spike elicited by climbing fiber stimulation and a simple spike elicited by mossy fiber activation. [4]

2.3 The Hodgkin-Huxley model

Alan Hodgkin and Andrew Huxley performed experiments on the squid's axon and produced the Hodgkin-Huxley (HH) model published in 1952 [36]. HH has significant value on computational neuroscience, and as shown in [37], it is still the most biologically accurate model for a single neural cell. The HH model is a mathematical model that describes how action potentials in a neural cells are initiated and propagated. It includes a set of nonlinear differential equations that approximate the electrical characteristics of neural cells.

The Hodgkin-Huxley model is further explained using Figure 2.8. The cell membrane separates the interior of the cell from the exterior and acts as a capacitor. If an input current is injected into the cell, it may add a further charge on the capacitor or leak through the cell membrane's channels. Hodgkin and Huxley found three different ion currents; sodium (K), potassium (N_a), and a leak current. Specific voltage-dependent ion channels, one for sodium and another for potassium, control ions flow through the cell membrane. The leak current takes care of other channel types, which are not described explicitly [5].

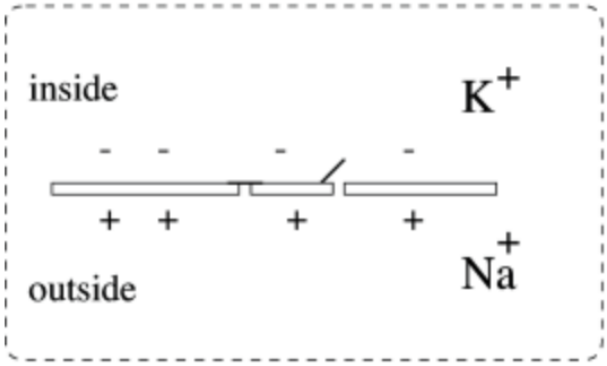


Figure 2.8: Schematic diagram for the Hodgkin-Huxley model. [5]

The mathematical description of the model can be seen in Equations 2.1 to 2.5. Equation 2.1 calculates the membrane voltage derivative and is the sum of the currents that flow through the membrane divided by the membrane conductance. I_{app} is the membrane external current, and any function can model it. The I_{leak} is the leakage current and is described by Equation 2.2, where g_l is the ionic conductance for the leakage current and V_l, is the voltage potential at which the "leakage current" due to chloride and other ions is zero [36]. $I_{channels}$ is the sum of the currents generated by the ion channels, as shown in Equation 2.3, where I_K and I_{NA} are the currents for each ion channel. These currents are calculated using Equations 2.4 and 2.5, V_K and V_{Na} are the voltage potentials of each ion gate and $g_K n^4$ and $g_{N_a} m^3 h$ express the ionic conductances. The g_K and g_{N_a} represents the maximum conductances and n,m and h corresponds to the gate activation variables. A gate-activation variable defines the probability that a single gate will be open, represented by the n^4 and $m^3 h$ in the ionic conductances. Each of the gate activation variable is calculated using Equations 2.6 - 2.8. In Equations 2.9 - 2.14 a_n, b_n, a_m, b_m, a_h and b_h are calculated that represents the rate of transfer of ions from outside to inside the membrane (a_n, a_m and a_h) and the opposite direction (b_n, b_m and b_h) which varies with voltage [36, 38].

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

$$I_{leak} = g_l(V - V_l) \tag{2.2}$$

$$I_{channels} = I_K + I_{NA} \tag{2.3}$$

$$I_K = g_K n^4 (V - V_k) \tag{2.4}$$

$$I_{Na} = g_{Na} m^3 h (V - V_{Na}) \tag{2.5}$$

$$\frac{\mathrm{d}n}{\mathrm{d}t} = \alpha_n(1-n) - \beta_n n \tag{2.6}$$

$$\frac{\mathrm{d}m}{\mathrm{d}t} = \alpha_m(1-m) - \beta_m m \tag{2.7}$$

$$\frac{\mathrm{d}h}{\mathrm{d}t} = \alpha_h(1-h) - \beta_h h \tag{2.8}$$

$$a_n = \frac{0.01(V+10)}{e^{\frac{V+10}{10}} - 1} \tag{2.9}$$

$$\beta_n = 0.125e^{\frac{V}{80}} \tag{2.10}$$

$$a_m = \frac{0.1(V+25)}{e^{\frac{V+25}{10}} - 1} \tag{2.11}$$

$$\beta_m = 4e^{\frac{V}{18}} \tag{2.12}$$

$$a_h = 0.07e^{\frac{V}{20}} \tag{2.13}$$

$$\beta_h = \frac{1}{e^{\frac{V+30}{10}} + 1} \tag{2.14}$$

2.4 The Inferior Olive model

Recall that Purkinje cells receive two inputs, the parallel fibers and the climbing fiber that originates from the inferior olive nucleus. Parallel fibers generate many simple spikes compared to climbing fibers inputs that generates a few so called "complex spikes" that overrides the current activity of the cells and then silencing them for a short time. It is well known that simple spikes are involved in motor control, where the role of complex spikes is more controversial [39]. Many studies are conducted in order to learn how the cerebellum is capable of motor learning and control despite the very low firing of the inferior olive inputs [40]. The findings in [35] showed that neural cells synchronization can have a big effect on the climbing fiber burst size through the gap junction coupling between IO neural cells.

The model used to simulate the IO cells is an extended HH model developed by De Gruijl [35]. This means that this model adds a few extensions to the original HH model to support the simulation of the IO model. The first extension is the addition of extra ion channels with different gates compared to the HH model. Therefore, some of the derivatives of the gate-activation variables are described by more complex functions. The second extension supports multiple cell compartments compared to the HH model that only describes the axon. The IO model describes the dendrites, the soma, and the axon compartments, and now each adjacent compartment exchange a current which is used to calculate the voltage derivative of each compartment. Finally, the gap junctions that connects the IO neural cells are included. A current flow through gap junctions, if the dendritic membrane potentials of the connected neural cells differs. This model is used by the neuroscientific community to learn how the inferior olive nucleus influence the cerebellum in motor learning skills and sensorimotor control. Detailed information about the model will be presented in the related work section.

2.5 Field-Programmable Gate Array (FPGA)

FPGAs are semiconductor devices that are based in a matrix of configurable logic blocks (CLBs) and on-chip memory connected via programmable interconnects [41]. Originally, FPGAs were used for validation and prototyping of application-specific integrated circuits (ASIC)s design before manufacturing. The technology advanced, the logic density increased, and other features were also added, such as embedded processors, digital signal processor (DSP) blocks along with improvements in the operating frequencies. Currently, FPGAs are used in a wide variety of applications and, more recently, in HPC. What motivates the use of FPGAs in HPC applications is the high performance that comes from FPGA's flexibility. This flexibility makes possible the implementation of highly customized application specific architectures.

Besides FPGAs, other technologies, such as multicore CPUs and GPUs, are used for HPC applications. The number of cores limits the parallelism in multicore CPUs, but the operating frequency is higher than FPGAs. The GPUs operational frequency ranges between FPGAs and CPUs, but GPU supports a larger number of cores. For both multicore CPUs and GPUs, the latency and power consumption associated with memory access and memory conflicts increase rapidly as data travels through the memory hierarchy [42]. On the other hand, FPGAs are well-known for their superior power efficiency when compared to GPUs and multicore CPUs. ASICs can also be considered since they outperform FPGAs in terms of performance and energy efficiency because they are fully customized circuits for a given application. However, ASICs have a significantly longer time to market since many of the back-end processes need to be taken care from the designer (physical layout, routing nodes, constraints) and are only fabricated when a design is thoroughly validated [43]. Moreover, the ASIC cost is exceptionally high except when a massive quantity is being produced, and once developed, the design cannot be changed. Based on the application's properties and user communities, any of the technologies may provide a suitable hardware platform. Still, the power consumption benefits and the in-field programmability enabling support of multiple applications deem

the FPGAs a suitable candidate for HPC applications.

2.5.1 Reconfigurable Fabric

The reconfigurable fabric of an FPGA consists of five main resource types; the **Configurable Logic Blocks (CLBs)**, the **Programmable Interconnect**, **Memory resources (Block RAM/Ultra RAM)**, the **DSP** units and the **Programmable I/O blocks (IOBs)** as shown in Figure 2.9. Each FPGA vendor has its own FPGA architecture but in general they are all a variation of the architecture shown in Figure 2.9. Details for each of the above mention FPGA's resource type will follow with focus on Xilinx's FPGAs.

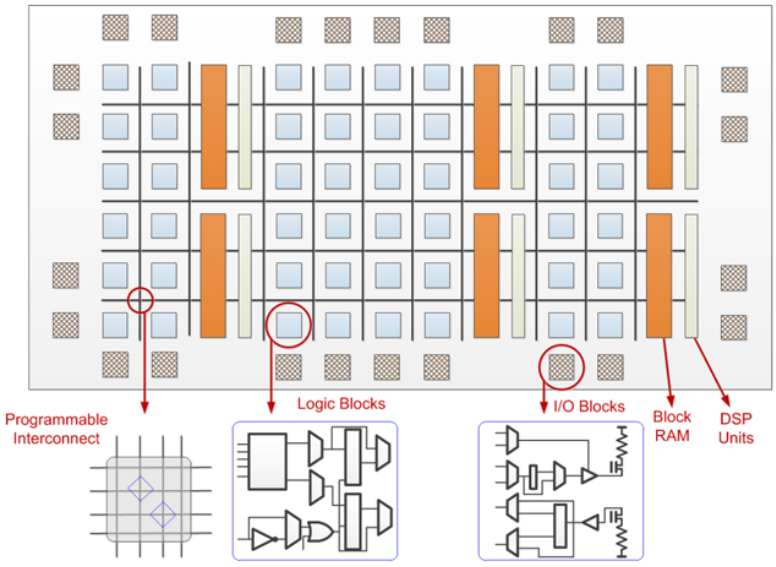


Figure 2.9: Basic architecture of an FPGA [6]

2.5.2 CLBs

The Configurable Logic Block is the backbone of the FPGA technology and the main resource needed to implement logical functions. Each CLB is made up of logic elements grouped in a slice, along with interconnections resources to connect these logic elements. Each slice consists of:

- Look-up Tables (LUTs);
- Storage elements that can be configured as Flip-Flops or latches;
- Dedicated high-speed carry logic for arithmetic functions;
- Multiplexers;
- Distributed memory;
- Shift register logic (SRL).

LUTs consists of 1-bit memory cells (SRAM) and a set of multiplexers. The values stored in these memory cells will be available at the LUT's output based on the input fed to the multiplexer's control lines. For example, a 4-input/1-output LUT can implement any 4-input boolean function.When the FPGA is configured, LUTs content is also configured based on the function that needs to be performed [44].

Dedicated carry logic improves the performance of arithmetic functions such as adders, counters, and multipliers. Carry logic is often used for smaller arithmetic functions [45].

Multiplexers inside slices can combine the LUTs to create more extensive functions without using another slice [45].

Distributed memory; In Xilinx's UltraSCALE$^+$ architecture, there are two types of slices, the SLICEL (logic) that only implements combinatorial functions and SLICEM (memory) that can also be configured as a distributed memory. Multiple SLICEM slices can be combined to form deeper or wider memories [45].

Shift register logic: Each LUT in a SLICEM can also be used as a 32-bit shift register. Combining all the LUTs in a slice allows construction up to a 256-bit SLR.

2.5.3 Programmable Interconnect

The programmable interconnect is a set of wires which can be wired together to connect any two blocks in an FPGA. As shown in Figure 2.9, these matrix-like interconnects enable arbitrarily logic networks to be constructed by the user [46].

2.5.4 Memory Resources

Besides the SLICEM memory logic, there are two more physical memory types, the BlockRAM (BRAM) and the UltraRAM (URAM).

BRAM is a block of 36Kb memory, which contains two independently controlled 18Kb memories. BRAM blocks can be used as one or two memory units and have flexible configuration regarding their depth and width. They consist of two reads and write ports; data can be written/read on either or both ports. These blocks of RAMs can be combined to enable deeper and wider memory implementations.[47]

URAM is a high density memory building block. One URAM module can store up to 288 Kbits of data and can be configured as a 4,096 x 72-bit memory block, about eight times larger than the BRAM. A URAM includes two ports, and it can independently perform a read or write per clock per port. As is for BRAMs, the 288Kb block can be cascaded to construct deeper and wider memory implementations.[47, 46]

2.5.5 Digital Signal Processors (DSPs)

A DSP unit supports many independent functions such as multiply, multiply and accumulate(MAC), multiply-add, barrel shift, four input add, bit-wise logic functions, and many others. Having a unit implemented to execute dedicated functions decreased area

cost and improved frequency. These units are optimized in speed/area compared to when implemented in programmable logic [46].

2.5.6 Input/Output Blocks

The programmable I/O blocks are designed to interface the fabric signals to the external world at the periphery of an FPGA.

2.5.7 Stacked Silicon Interconnect Technology (SSI)

Xilinx introduce the SSI technology in their newest FPGA platforms. As shown in figure 2.10, multiple Super Logic Regions (SLRs), which are active FPGA dies, are connected to the silicon interposer. Interposer is a passive layer that serves the purpose of power delivery, configuration connectivity, and the connectivity between SLRs and SLRs to package substrate via through-silicon vias (TSVs) [46]. TSVs are vertical electrical connections that pass-through silicon and provide high-performance interconnections to create stacked circuits. Micro bumps and C4 Bumps that are shown in Figure 2.10 there various different solder joints that are used to interconnect semiconductor devices to external circuitry chips. Using this approach, Xilinx can now deliver devices with higher logic capacity and more on-chip resources. [48].

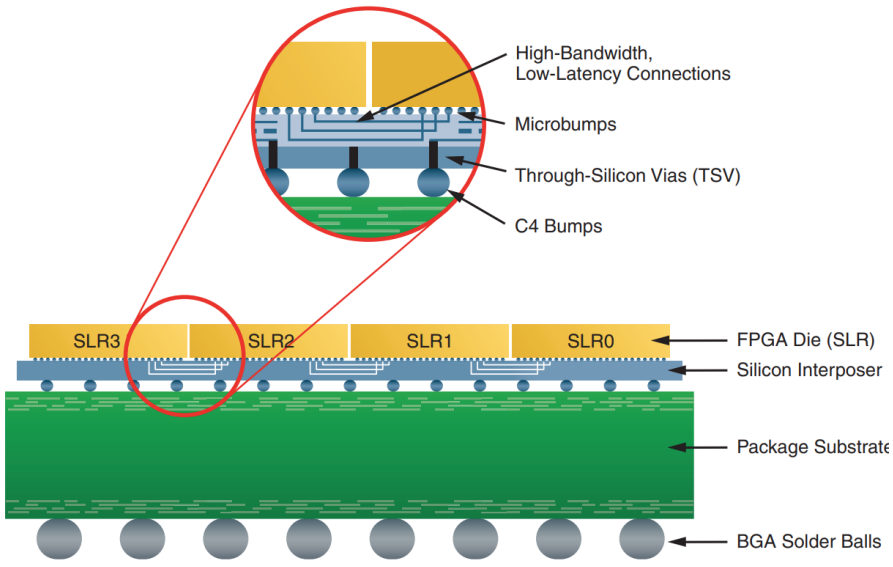


Figure 2.10: Representative SSI Device Construction [7]

2.6 Dataflow Computing

There are two computing paradigms found in today's computers, the controlflow and the dataflow paradigm. In the first case, the execution process is essentially slow because instructions have to be fetched, decoded, and then executed. Additionally, data for each instruction needs to be fetched from memory, used during the execution phase, and then

the output is stored back to memory. All these operations are extremely time-consuming [49]. Dataflow computing focuses on how data moves in a 2D-space, and control is triggered by data movement. This results in the elimination of the instructions necessary to control the computations. Because there are no instructions, there is no need for instruction-decode logic, instruction caches, branch prediction, or dynamic out-of-order scheduling. By eliminating the controlflow overhead, almost all resources of the chip area are dedicated to performing computations. On the other hand, special mechanisms are required for data-availability detection and orchestration of data streams. Figure 2.11 illustrates the difference between controlflow and dataflow computing paradigms. In diagram (a), to add A and B, you first need to fetch data from memory, do the computation and then store the result back to the memory. On the other hand, in diagram (b), all commands can be executed at once (all values are streamed at once) compared to the diagram (a) where everything will be executed serially.

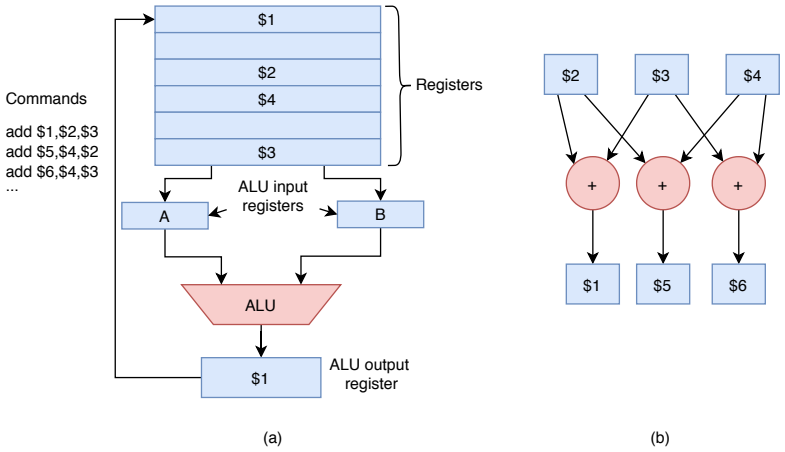


Figure 2.11: Computing Paradigms.(a) Controlflow (b) Dataflow

Dataflow computing paradigm focus on optimizing the movement of data in an application and utilize the massive parallelism of FPGAs. This provides benefits in performance, space, and power consumption compared to controlflow computing.

The concept of dataflow computing is not new [50, 51], but dataflow computing could not achieve commercial success because reconfigurable hardware technology and system software technology were not yet ready. Additionally, the applications of those days were not going to exploit the streaming capabilities of dataflow computing to result in performance superiority [52]. The dataflow computing concept's initial ideas can now be applied and produce effective implementations with the advancement of the dataflow hardware (FPGA) and the programming language to support it.

Maxeler Technologies embrace this paradigm to accelerate HPC applications on their FPGA-based dataflow engines, using an in-house developed tool chain called MaxCompiler. Development using hardware description languages such as VHDL or Verilog required a high level of expertise and suffered from a low level of abstraction. This may diminish the ability to perform high-level optimizations at the system level. Another ap-

proach is to use High-Level Synthesis (HLS) that automates the generation of hardware circuits. The downside of using HLS is the inability to describe or understand the mapping from algorithm to hardware and the inability to perform low-level optimizations. Additionally, HLS uses a high-level design process to automatically generate hardware structures, limiting kernel acceleration, which suffers from input/output bandwidth restrictions [9]. MaxCompiler aims the best of both worlds, the high-level abstraction of HLS and the low-level control of hardware description languages. Every code line has a direct relation to the generated hardware and this gives the designer the ability of tight control by using an automated process.

The architecture of a Maxeler dataflow processing system includes dataflow engines (DFEs) bundled together with local memories working together with a CPU host, as depicted in Figure 2.12. The host CPU includes a software layer called MaxelerOS that allows the DFE to communicate with the software program usually written in C. The CPU uses the Simple Live CPU (SLiC) interface to activate the kernels and set the data transfers between the CPU and DFE through PCIe interconnect. Additionally, it is easier to rearrange the data on CPU to improve the data streaming (e.g., to have more linear access) on the DFE. Besides the data streaming between DFE and CPU, single values can be sent at runtime called scalar inputs. Each DFE has two types of memory; the Fast Memory (FMem) and the Large Memory (LMem). FMem is an on-chip memory that uses BRAMs with a capacity of several megabytes and access bandwidth of terabytes/second. LMem the on-board DFE memory implemented by large capacity DRAM chips able to store many gigabytes of data off-chip with access bandwidth of gigabytes/second. The bandwidth and flexibility of FMem is a key reason why DFEs can achieve such high performance on complex applications. Applications can effectively exploit the full FMem capacity because both memory access and computation are laid out in space so data can always be held in memory, close to computation, compared to the classical CPU architectures that only the lowest level of cache is close to computational units. Finally, the manager orchestrates the data movement between kernels (in case of multiple kernels) and to or from LMem.

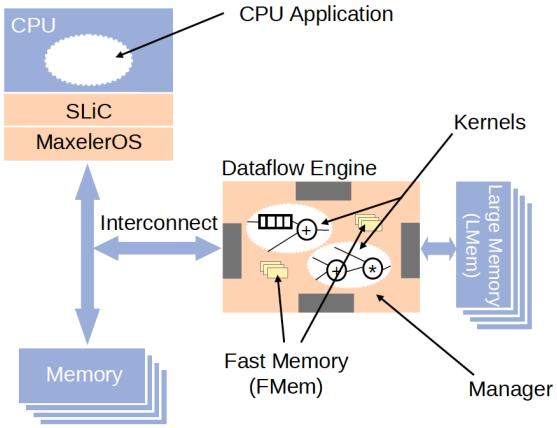


Figure 2.12: Dataflow engine architecture [8]

The kernel uses the abstraction of a logical tick; in every tick, one item is fully processed,

and it is possible to get data from previous or future ticks using stream offsets. While not everything can be executed simultaneously, the compiler automatically schedules the dataflow graph and inserts a First-In, First-Out (FIFO) buffers and registers to balance data transport. Each node/operation has a latency (e.g., floating-point adder has a latency of 12 ticks), and to achieve the required behavior, some streams need to be delayed. In Figure 2.13 the graph of the function $out = (a + b + c) + (a + b)$ is illustrated. Figure 2.13 (a), illustrates the unscheduled version of the dataflow graph, and in Figure 2.13 (b), the scheduled graph. Depending on the addition operation's latency, the depth of the FIFO buffer will be adjusted, and the compiler makes sure that the data will arrive at the correct tick.

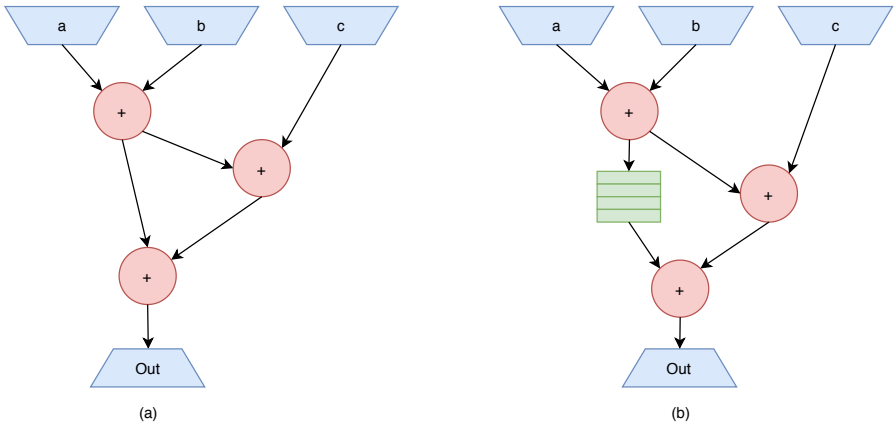


Figure 2.13: (a) Unscheduled dataflow graph (b) Scheduled dataflow graph

Data types

Compared to humans, a computer cannot distinguish the difference between "123" and "abc". A data type is a classification that indicates what type of value a variable has and what type of mathematical, relational, or logical operation can be applied to produce an error-free program. For example, the two most common data types are strings that classifies text and integers that classifies numbers. Deep down to the hardware level, those data types represent their content using zeros and ones in a specific way. The benefit of using hardware to accelerate HPC applications is that we can represent numbers in various ways and optimized it down to the bit level, which may prove crucial in many applications. The data types that are supported by the tools are the following:

- **Raw bits:** A datatype used to represent a binary word with a user-defined length;
- **Boolean:** A data type used for all Boolean operations with numeric values 1 and 0 that represent true or false, respectively;
- **Unsigned and Signed integer:** This data type represents only integer numbers signed or unsigned, as shown in Figure 2.14 (a) and (b). The Maxeler tools support both 32-bit and 64-bit integers;
- **Fixed-point:** As shown in figure 2.14 (c), fixed-point consists of two parts, the integer and the fractional part. The size of each part is variable and can be changed per need by changing its offset. For example, if we have 16-bits, we can represent

a 16-bit integer (offset = 0), or it can take the form of a fully fractional number (offset = -16), or it can have 8-bits for the integer and 8-bits for the fractional parts. Negative numbers can be represented using 2's complement representation; otherwise, the numbers can be unsigned;

- **Floating-point:** The floating-point data type represents a wider range of values compared to a fixed-point of the same size. As shown in figure 2.14 (d), a floating-point data type includes the fractional part, the exponent part, and the sign part. The tools supports between 4 and 16 bits for the exponent part, and the mantissa size depends on the exponent size. Mantissa bits can vary between 5 and 64-bits, and the sign part is 1-bit. This means that single and double-precision floating-point data types are supported by Maxeler's tools [53].

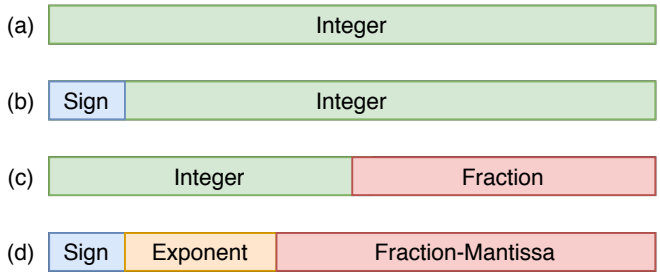


Figure 2.14: Datatypes supported by the tools(a) Unsigned Integer (b) Signed Integer (c) Fixed-point (d) Floating-Point

2.7 MaxJ

Maxeler Technologie's tools, MaxCompiler and MaxIDE (Integrated development environment) make use of an extended version of Java programming language called MaxJ. MaxJ adds operator overloading semantics to the Java language. Using MaxJ, a programmer describes the procedure of generating the dataflow graph instead of describing the graph itself.

2.8 MAX5 Dataflow Engine

Maxcompiler can target several FPGA based platforms, the target platform used in this work is the MAX5C. MAX5C utilizes Xilinx's UltraSCALE$^+$ VU9P FPGA [10]. This architecture consists of three dies (SLRs) and three DIMMs of memory (LMem). The resource, memory bandwidth, pci bandwidth and storage capacity are shown in Tables 2.1, 2.2 and 2.3.

LUTs	FFs	BRAMs	URAMs	DSPs
1,182,240	2,364,480	4,320	960	6,840

Table 2.1: MAX5C Resources [10]

PCIe	DRAM
3 GB/s	47.65 GB/s

Table 2.2: MAX5C Bandwidth

BRAM	URAM	DRAM
9.49 MB	33.75 MB	46 GB

Table 2.3: MAX5C Storage capacity

2.9 Summary

In this chapter information regarding parts of the human brain, the HH model and how is extended to support the inferior olivary nucleus model are given. Additionally the FPGAs, and Maxeler's dataflow computing paradigm are discussed. Finally, information regarding Maxeler's Technologies DFEs and the tools are briefly explained.

3 Related Work

This chapter introduce the related work that this thesis is based on. This work depends upon the previous work done by Rene Miedema during his Master Thesis entitled "flexHH: A flexible hardware library for Hodgkin-Huxley-based neural simulations" and section 3.1 will introduce it [38].

3.1 The flexHH library

The flexHH library origins from BrainFrame [54], a High-Performance Computing (HPC) framework in order to accelerate neuro-scientific experiments using multiple acceleration technologies such as Intel Xeon-Phi, Nvidia GPU and Maxeler DFEs. However, to make the framework more practical and useful to the neuroscientific community, neuroscientists must develop their custom models within BrainFrame using general libraries. FlexHH library is offering high performance and flexible library to simulate HH-based neural simulations on an FPGA-based platform.

To implement generalized kernels on the DFE, the HH and IO model's equations are generalized using a set of parameters. Otherwise, if no generalized equations are used each time a new set of equations are utilized for the simulation, synthesis for the given kernel will be needed, which is time-consuming. The following subsections describe the generalized equations starting from the HH-model, and then each extension is described that leads to the IO model. All the equations are based on the work done in [38].

3.1.1 HH model

In the equations of the HH-model the derivative of the voltage ($\frac{\mathrm{d}V}{\mathrm{d}t}$) is the summation of different currents divided by the capacitance of the membrane as shown in Equation 3.1. Equation 3.1 does not change between simulations and it can be used directly. The first current, called $I_{app(t)}$ is the external applied current and only pulse functions are supported as shown in Equation 3.2. When the time is between start time (t_{start}) and end time (t_{end}) it returns the set amplitude (A). The next current, is the $I_{channels}$ and is the summation of the currents flowing through all the ion channels as shown in Equation 3.3. $M_{gates[i]}$ is the number of gates per channel, y_i is the gate activation variable, p_i is an integer gate-dependent exponent , V is the cell's voltage and $V_{channel}$ is each ion channel voltage. The equations that calculates the derivatives used for the gate activation variables that are described in 2.6 - 2.8 have the same form as shown in Equation 3.4. The Equations 2.9 - 2.14 are represented as three equations as show in 3.5. In this equation, x_1, x_2, and x_3 are floating-point values that represents the variables of the equations, f_{type} is an integer value to select a function and V is the membrane

voltage. Finally, I_{leak} is the leakage current and is represented using Equation 3.6, g_{leak} corresponds to the conductance of each channel and V_{leak} is the leakage voltage.

$$\frac{\mathrm{d}V}{\mathrm{d}t} = \frac{I_{app(t)} - I_{channels} - I_{leak}}{C_M} \tag{3.1}$$

$$I_{app}(t) = \left\{ \begin{array}{l} A, \text{ if } t_{start} \leq t < t_{end} \\ 0, \text{ if otherwise} \end{array} \right\} \tag{3.2}$$

$$I_{channels} = g_{channel} \cdot \prod_{i=0}^{M_{gates[i]}-1} y_i^{p_i} \cdot (V - V_{channel}) \tag{3.3}$$

$$\frac{\mathrm{d}y_i}{\mathrm{d}t} = \alpha_i \cdot (1 - y_i) - \beta_i \cdot y_i \tag{3.4}$$

$$f(V, x_1, x_2, x_3, f_{type}) = \left\{ \begin{array}{ll} \frac{x_1 \cdot (x_2 - V)}{e^{(x_2 - V) \cdot x_3} - 1} & \text{if } f_{type} = 0 \\ x_1 \cdot e^{(x_2 - V) \cdot x_3} & \text{if } f_{type} = 1 \\ \frac{1}{e^{(x_2 - V) \cdot x_3} + 1} & \text{if } f_{type} = 2 \end{array} \right\} \tag{3.5}$$

$$I_{leak} = g_{leak} \cdot (V - V_{leak}) \tag{3.6}$$

Table 3.1: Parameters of the HH-model filled into Equation (4.4).

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

3.1.2 Custom ION gates

Thus far we discussed the HH-model, the first extension required by the IO-model, is the support of custom defined ion gates. This requires a number of additional equations to the standard gates as shown in Equations 3.7 - 3.11. In these equations s_d, q_d, l_s and n_s are gate variables, Ca_d^{2+} is the calcium ion concentration and $I_{cah,d}$ is the high-threshold calcium current in the dendrite. In comparison to the standard gates found in the HH-model these equations contain multiple exponent functions and it also include a min function. These equations are generalized and are shown in Equation 3.12. Supporting these equations results in having more complex functions and nine instead of three floating point values are used (xs).

$$\frac{\mathrm{d}s_d}{\mathrm{d}t} = min(0.00002 \cdot Ca_d^{2+} \cdot 0.01) \cdot (1 - s_d) - 0.015 \cdot s_d \tag{3.7}$$

$$\frac{\mathrm{d}q_d}{\mathrm{d}t} = \frac{\frac{1}{1+e^{\frac{V_{dend}+80}{4}}} - q_d}{\frac{1}{e^{-0.086 \cdot V_{dend} - 14.6} + e^{0.070 \cdot V_{dend} - 1.87}}} \tag{3.8}$$

$$\frac{\mathrm{d}Ca_d^{2+}}{\mathrm{d}t} = -3 \cdot I_{cah,d} - 0.075 \cdot Ca_d^{2+} \tag{3.9}$$

$$\frac{\mathrm{d}l_s}{\mathrm{d}t} = \frac{\frac{1}{1+e^{-\frac{V_{soma}+85.5}{-8.5}}} - l_s}{\frac{20 \cdot e^{\frac{V_{soma}+160}{30}}}{1+e^{\frac{V_{soma}+84}{7.3}+35}}} \tag{3.10}$$

$$\frac{\mathrm{d}n_s}{\mathrm{d}t} = \frac{\frac{1}{1+e^{-\frac{V_{soma}+3}{10}}} - n_s}{5 + 47 \cdot e^{\frac{-(50-V_{soma})}{900}}} \tag{3.11}$$

$$fCustom(f_{type}, V, xs) = \left\{ \begin{array}{ll} \frac{x_5 \cdot (xs[1]-V)}{x_0 e^{(x_1-V)\cdot x_2 + x_3}} + x_8 & \text{if } f_{type} = 0 \\ \frac{x_8}{x_0 e^{x_2(x_1-V)+x_3} + x_3 + x_4 e^{(x_5 \cdot (x_6 - v))} + x_7} & \text{if } f_{type} = 0 \\ \frac{x_0 \cdot e^{((x_1-V)x_2)+x_3}}{x_4 e^{((x_6-V)\cdot x_5)+x_7}} + x_8 & \text{if } f_{type} = 2 \\ min(x_0 V, x_1) & \text{if } f_{type} = 3 \end{array} \right\} \tag{3.12}$$

3.1.3 Multiple cell compartments

The next extension added is the support of multiple compartments in a neural cell. When having two or more compartments, a current is exchanged between two adjacent compartments and is calculated using Equation 3.13. In this equation, g_{int} corresponds to the internal conductance of the cell, $p_{i,j}$ is the surface ratio between the two compartments (between compartment i and j), and $V_i - V_j$ is the voltage difference between the adjacent compartments. When this model extension is supported, the current I_{mc} is added to the sum of currents for calculating dV/dt for each compartment, as shown in Equation 3.14.

$$I_{mc} = \frac{g_{int}}{p_{i,j}} \cdot (V_i - V_j) \tag{3.13}$$

$$\frac{\mathrm{d}V}{\mathrm{d}t} = \frac{I_{app} - I_{channels} - I_{mc} - I_{leak}}{C_M} \tag{3.14}$$

3.1.4 Gap junctions

Finally, the last extension is the support of gap junctions, which are intercellular connections. The intercellular current for each cell is calculated using Equation 3.15, which is repeated for all the cells in the network. c_0, c_1, and c_2 are constants, wi, j is a constant weight that denotes the connection strength between cells i and j and $V_{i,j}$ is the voltage between cells. When this model extension is supported, the gap junction current is added to the sum of currents to calculate dV/dt, as shown in Equation 3.16.

$$I_{gap} = \sum_{j=0}^{N_{cells}-1} (w_{i,j}(c_0 \cdot e^{(c_1 \cdot V_{i,j}^2)} + c_2) \cdot V_{i,j}) \tag{3.15}$$

$$\frac{\mathrm{d}V}{\mathrm{d}t} = \frac{I_{app} - I_{channels} - I_{gap} - I_{leak}}{C_M} \tag{3.16}$$

3.1.5 IO model

Having mentioned all the extensions added to the HH-model required by the IO model, the modeling hierarchy is shown in Figure 3.1. Figure 3.1 (a) illustrates the most abstract view of the model, a network of seven neural cells. Each cell includes three compartments, the dendrites, soma, and axon, as shown in Figure 3.1 (b), and current flows internally between each adjacent compartment as shown in yellow. The gap junction connections between cells are shown in blue color, and here we considered a fully connected network. Finally, Figure 3.1 (c) illustrates the ion channels for each compartment.

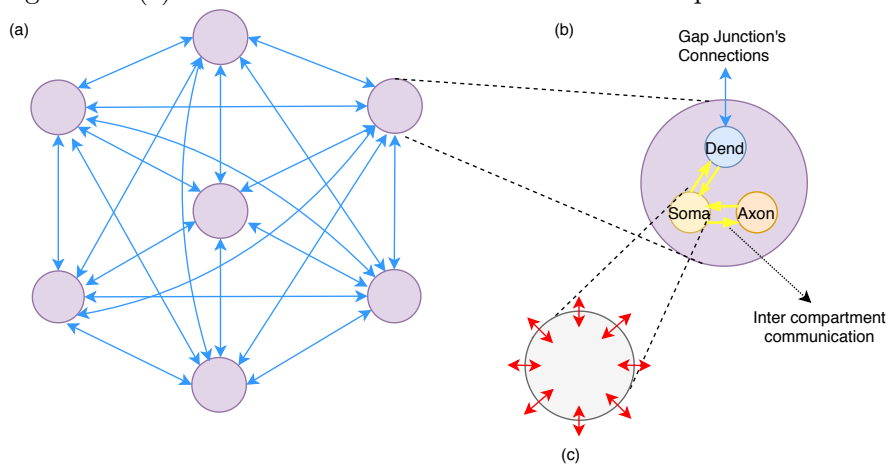


Figure 3.1: (a) Schematic overview of neural network of 7 cells. (b) A single neuron cell. (c) Single compartment showing its ion channel gates(red arrows)

4 Design Process

This chapter describes all the steps performed during the design of the improved flexHH hardware library. We will follow the steps of the Maxeler design methodology described in [25, 55], this process was developed to support as much as possible the programmer in developing HPC applications on Maxeler's dataflow engines. Moreover, all the different aspects of analysis, architecture, and implementation are integrated into this design methodology. Additionally, using this methodology helps us perform a fast design space exploration before the actual implementation. This will prevent spending time on architecture that does not fit or will not provide adequate speed up of our application. Maxeler's Technologies design methodology consists of:

1. Accurate analysis of the targeted application;
2. Design of a representative software model;
3. Performance modeling of architecture candidates;
4. Rigorous development of an application-specific architecture based on the results of the above parts.

4.1 flexHH library Analysis

The first task is to analyze the flexHH library to determine the computation-intensive parts and understand how to improve the application. In this work, we will optimize and improve the performance of an already implemented work done in [38], and a C-based model is already provided. The provided C model is implemented by Rene Miedema on behalf of Neurasmus B.V. and ICCS Athens as part of the Brainframe theme of the Erasmus Brain Project [56]. In this work, the C programming language is used since it fits our requirements, and most of the tools being used in this analysis support them. Using a software model provides many benefits; it provides an insight into the code. It is easier to profile, and it can assist in the verification of the output. Moreover, it can also be used as a debugging reference for the FPGA implementation.

The IO model of the flexHH library is chosen for this analysis since the other cases are a subset of this. The model is profiled to focus on which parts of the model predominate the execution time. It is important to remark that profiling is done to provide an initial assessment of which parts of the model need greater attention or have room for improvement. During the implementation, these parts will be possible candidates for optimization to speed up our application. Various tools are used to extract pertinent information, and for all the tools, the numbers that are illustrated were obtained on

an Intel Core i7 4790 (Quad-Core 3.6GHz), 8GB RAM using Deepin 15.9 (Debian) Operating System. A set of realistic data-sets should be used to extract the information from these tools to grasp a real-world use case as accurately as possible.

4.1.1 Static and Dynamic Code Analysis

This section introduces the static and dynamic code analysis. Static code analysis is emphasized in the source code. It will mainly help us understand our application and the data sizes that need to be transferred between accelerator and CPU. On the other hand, dynamic code analysis uses profiling programs that will help us understand our application's hotspots.

In all the simulation runs on the following tools, the gates, and the compartments parameters are set constant, and we simulated using two network sizes of 96 and 7,680 cells. The number of compartments and gates is set to 3 and 13 per cell, respectively. Gap junctions are enabled, and the custom gates functions are used. The simulated steps are set to 10,000. These network sizes are chosen to provide us with adequate information and compare a small and a larger network of neural cells.

Linux GNU GCC Profiling Tool (gprof) [57] is a performance analysis tool for Unix applications. To gather profiling information at run-time, gprof uses a sampling process; the sampling process is statistical, meaning that the profile data are not an exact but rather statistical approximation. Aside from that, gprof is a fast and easy to use profiler that provides us with fast results. Three simple steps are needed to use gprof; first, re-compile having enabled profiling (add -pg as an argument), execute the program and finally run gprof to analyze the profile data file (gmon.out).

Both Tables 4.1 and 4.2 illustrates the profiling results of the C model simulation using gprof. In the first table, we can see that iGapCell and fGap functions related to the gap junction computations are ranked first and fourth and takes 34.68% of the CPU time. fCustom, fExp, copyXs, and min functions are all related to the gate computations and takes up to 37.1% of total CPU time. The rest of the functions are used in the compartment's computations are taking less than 5% of the total CPU time. It is interesting that in Table 4.2, the model behavior completely changes. Having a larger network leads to an enormous increase in the functions related to gap junction computations and takes up to 94.63%of the total CPU time ranked first and second in Table 4.2. The rest of the functions takes 5.37%, which is negligible compared to the gap junction computations. This highlights the need for profiling since, as shown here, the programmer will focus on optimizing a single part of the application. The calls to each function, the cumulative time, and individual time per function are also mentioned for reference.

Table 4.1: Profiling results for C model with network size of 96 cells

Function	***Running time (%)***	***Cumulative time (s)***	***Individual time (s)***	***Calls***
iGapCell	*25.81*	*0.32*	*0.32*	*960,000*
HHGAP_CPU	*25*	*0.63*	*0.31*	*1*
fCustom	*21.78*	*0.90*	*0.27*	*24,960,000*
fGap	*8.87*	*1.01*	*0.11*	*92,160,000*
fExp	*7.26*	*1.10*	*0.09*	*49,920,000*
copyXs	*6.45*	*1.18*	*0.08*	*12,480,000*
calcIApp	*2.42*	*1.21*	*0.03*	*288,000*
min	*1.61*	*1.23*	*0.02*	*24,960,000*
getVnext	*0.81*	*1.24*	*0.01*	*2,880,000*
getPnext	*0*	*1.24*	*0.00*	*2,880,000*
getVprev	*0*	*1.24*	*0.00*	*2,880,000*

Table 4.2: Profiling results for C model with network size of 7,680 cells

Function	***Running time (%)***	***Cumulative time (s)***	***Individual time (s)***	***Calls***
iGapCell	*62.58*	*8,185.03*	*8,185.03*	*76,800,000*
fGap	*32.05*	*12,380.97*	*4,195.94*	*589,824,000,000*
sumArray32	*2.79*	*12,746.23*	*365.26*	*2*
HHGAP_CPU	*1.04*	*12,882.38*	*136.15*	*1*
fCustom	*0.74*	*1,2979.25*	*96.87*	*1,996,800,000*
copyXs	*0.32*	*13,021.14*	*41.89*	*998,400,000*
fExp	*0.30*	*13,060.41*	*39.27*	*3,993,600,000*
‘min	*0.08*	*13,070.88*	*10.47*	*1,996,800,000*
calcIApp	*0.03*	*13,074.8*	*3.92*	*230,400,000*
getVprev	*0.03*	*13,077.41*	*2.61*	*230,400,000*
getVnext	*0.02*	*13,080.02*	*2.61*	*230,400,000*
getPnext	*0.02*	*13080.49*	*0.47*	*230,400,000*

Valgrind [58] is another tool that allows us to profile our application in more detail that records the call history among functions in a run as a call-graph. Callgrind tool is part of the Valgrind framework, which profiles an application by transforming it into an intermediate language executed in a virtual processor emulated by Valgrind. The downside of using this tool is that it has a substantial run-time overhead; however, it has excellent precision.

The output file that includes the profiling information of the analyzed application produced by Callgrind is displayed using KCachegrind, as shown in Figures 4.1 and 4.2. Figure 4.1 shows the function ranking based on the total time spent on each function. Self-time refers to the time that each function takes, excluding the called functions. The number of called times of a function is also mentioned. Figure 4.2 shows the Callee Map of the iGapCell function where the surface represents its weight to the program.

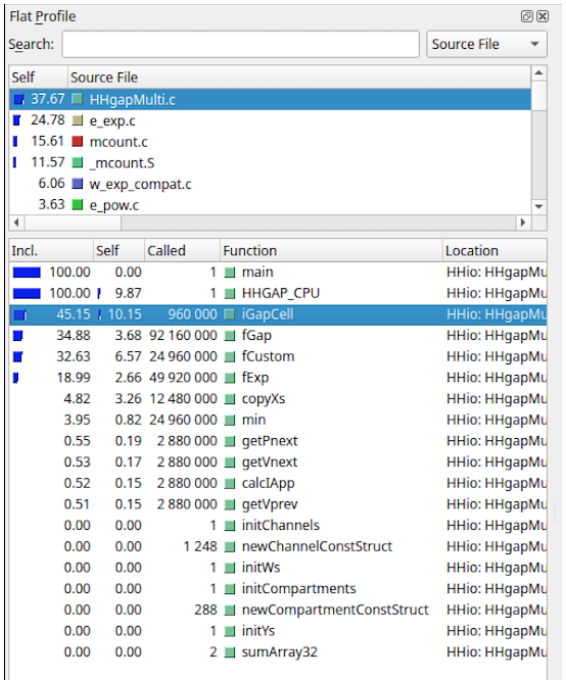


Figure 4.1: Valgrind Output: Function Ranking

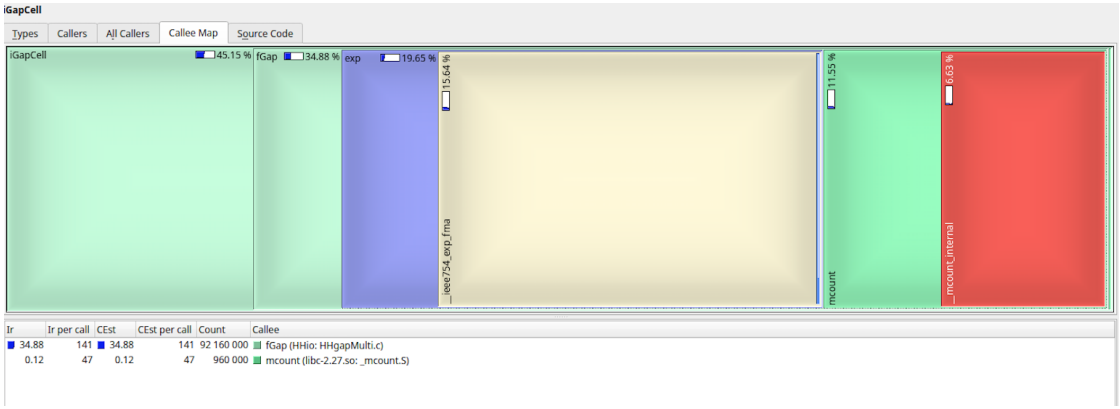


Figure 4.2: Valgrind Output: Callee Map

Valgrind's Callgrind output is converted using gprof2dot tool [59] to a call graph as illustrated in Figures 4.3 and 4.4. The graph shows the function hierarchy, and each block indicates the function name, execution time percentage including and excluding (in parenthesis) the called functions, and the number of times each function is being called. The application had a huge call graph, although functions that are below a certain threshold (0.50%) are pruned to focus on the meaningful ones. Using these two call graphs makes a comparison between the two cases easier. In the first Figure 4.3, where 96 neurons are being simulated, 45.15% of the total execution time is spent on iGapCell function. Simulating a network size of 7,680 cells shows that iGapCell dominates the execution time with 98.55%, as shown in Figure 4.4. Additionally, we can observe a heavy usage of floating-point exponential operations labeled as _ieee754_exp_fma, which takes the 23.66% in the 96-cell network and 46.74% in the 7,680-cell network of total

execution time. Finding a way to optimize or reduce this operation's usage will reduce the run-time of the application.

Intel® VTune™ Amplifier [60] software provides advanced sampling and profiling techniques that quickly analyze a given application focusing on optimizing performance on modern processors. Figure 4.5 shows the function ranking based on the CPU time, and VTune can also denote in which line of the application the execution time is spent, as shown in Figure 4.6. In Figure 4.6 most of the time is spent in fGap function. To conclude, Table 4.3 illustrates the function ranking, and again the gap junction computations take a substantial amount of the CPU time. It is also stated that the function __GI__exp dominates the simulation of 7,680 cells with 66% of total CPU time.

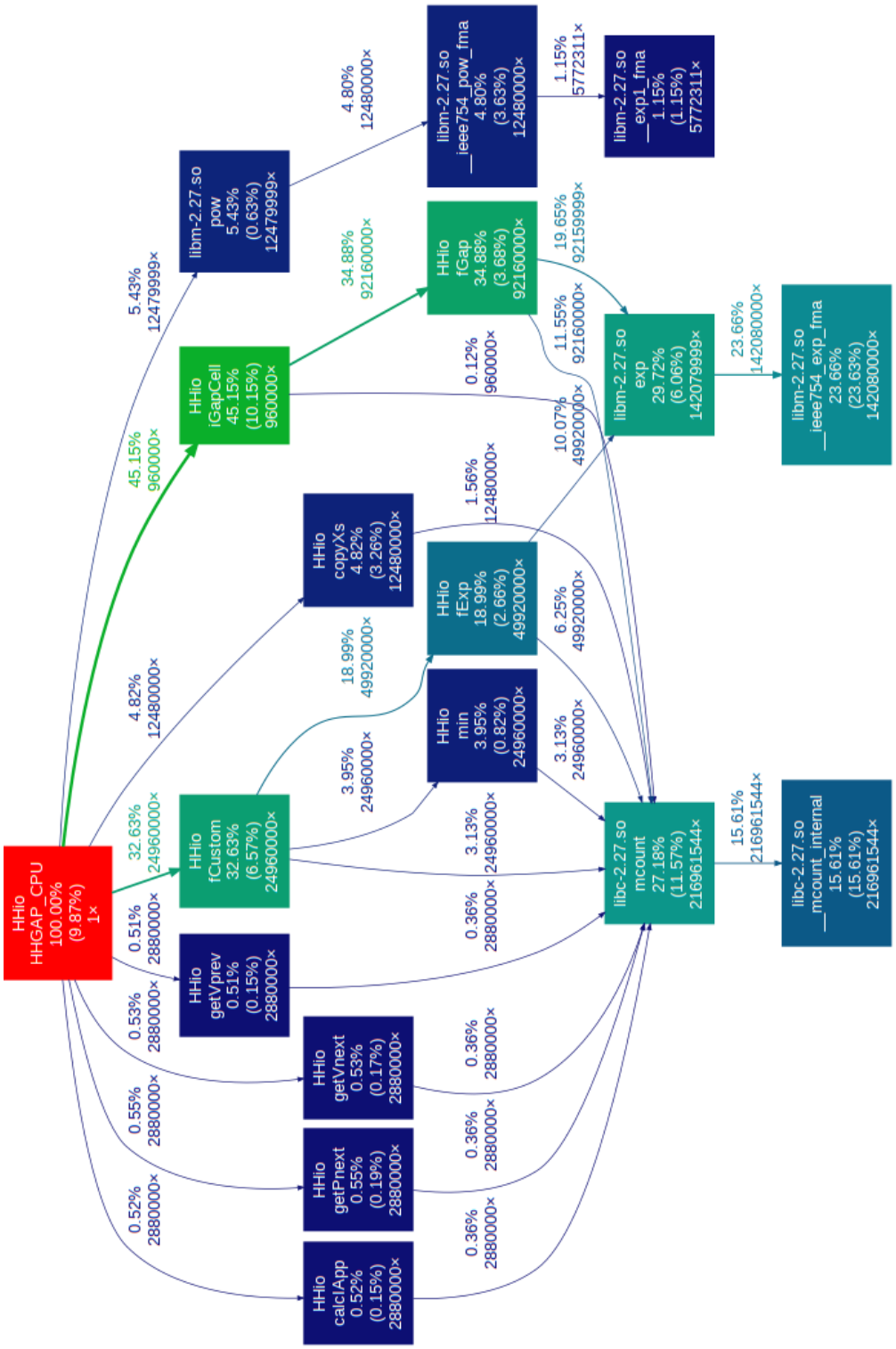


Figure 4.3: 96-cell simulation call-graph

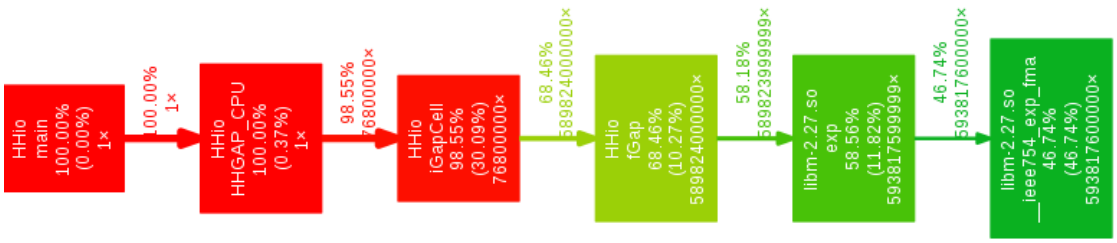
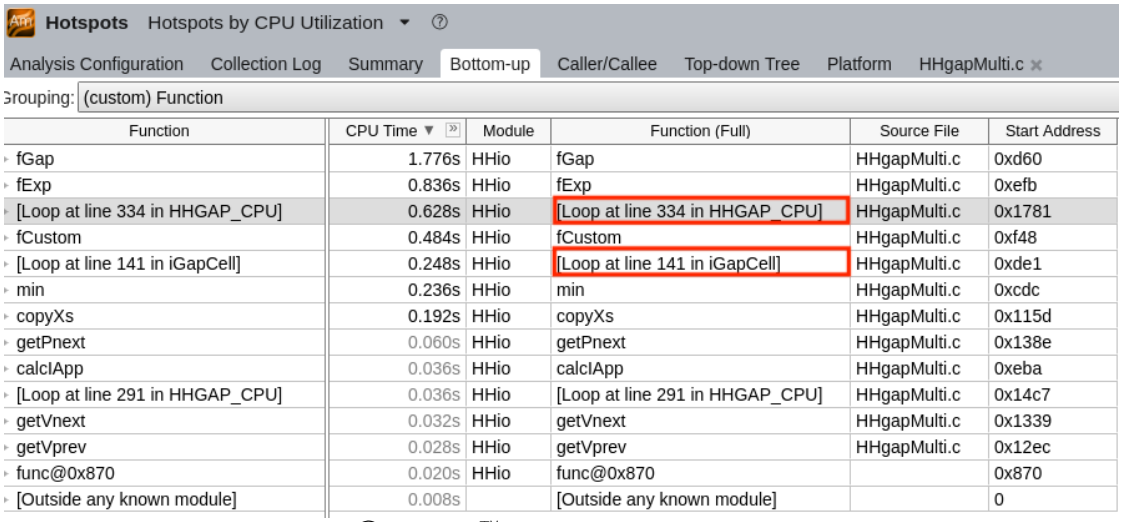


Figure 4.4: 7,680-cell simulation call-graph



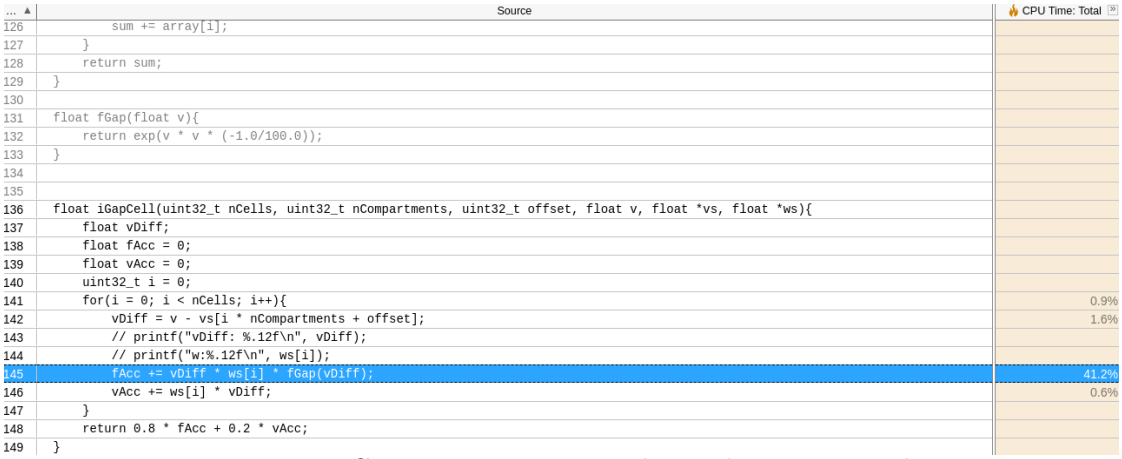
Hotspots Hotspots by CPU Utilization

Analysis Configuration | Collection Log | Summary | Bottom-up | Caller/Callee | Top-down Tree | Platform | HHgapMulti.c

Grouping: (custom) Function

Function	CPU Time	Module	Function (Full)	Source File	Start Address
fGap	1.776s	HHio	fGap	HHgapMulti.c	0xd60
fExp	0.836s	HHio	fExp	HHgapMulti.c	0xefb
[Loop at line 334 in HHGAP_CPU]	0.628s	HHio	[Loop at line 334 in HHGAP_CPU]	HHgapMulti.c	0x1781
fCustom	0.484s	HHio	fCustom	HHgapMulti.c	0xf48
[Loop at line 141 in iGapCell]	0.248s	HHio	[Loop at line 141 in iGapCell]	HHgapMulti.c	0xde1
min	0.236s	HHio	min	HHgapMulti.c	0xcdc
copyXs	0.192s	HHio	copyXs	HHgapMulti.c	0x115d
getPnext	0.060s	HHio	getPnext	HHgapMulti.c	0x138e
calcIApp	0.036s	HHio	calcIApp	HHgapMulti.c	0xeba
[Loop at line 291 in HHGAP_CPU]	0.036s	HHio	[Loop at line 291 in HHGAP_CPU]	HHgapMulti.c	0x14c7
getVnext	0.032s	HHio	getVnext	HHgapMulti.c	0x1339
getVprev	0.028s	HHio	getVprev	HHgapMulti.c	0x12ec
func@0x870	0.020s	HHio	func@0x870		0x870
[Outside any known module]	0.008s		[Outside any known module]		0

Figure 4.5: Intel® VTune™ function ranking based on CPU time



	Source	CPU Time: Total
126	sum += array[i];	
127	}	
128	return sum;	
129	}	
130		
131	float fGap(float v){	
132	return exp(v * v * (-1.0/100.0));	
133	}	
134		
135		
136	float iGapCell(uint32_t nCells, uint32_t nCompartments, uint32_t offset, float v, float *vs, float *ws){	
137	float vDiff;	
138	float fAcc = 0;	
139	float vAcc = 0;	
140	uint32_t i = 0;	
141	for(i = 0; i < nCells; i++){	0.9%
142	vDiff = v - vs[i * nCompartments + offset];	1.6%
143	// printf("vDiff: %.12f\n", vDiff);	
144	// printf("w:%.12f\n", ws[i]);	
145	fAcc += vDiff * ws[i] * fGap(vDiff);	41.2%
146	vAcc += ws[i] * vDiff;	0.6%
147	}	
148	return 0.8 * fAcc + 0.2 * vAcc;	
149	}	

Figure 4.6: CPU time percentage denoted in source code

Table 4.3: Function Ranking based on CPU time

Function Ranking (96 cells)	***CPU time(s)***	***Function Ranking (7680 cells)***	***CPU time(s)***
fGap	*1.77*	*__GI__exp*	*10,572.01*
fExp	*0.83*	*iGapCell*	*2,532.53*
HHGAP_CPU	*0.62*	*fGap*	*2,485.58*
fCustom	*0.48*	*fCustom*	*83.84*
iGapCell	*0.24*	*HHGAP_CPU*	*48.95*

Using these profilers pinpointed, which is the most computationally intensive part of our model, which is going to benefit most from hardware acceleration. Besides computation complexity, it is crucial to consider the data movements between functions to find the optimal partitioning between CPU and FPGA. The bandwidth between CPU and FPGA is often limited, leading to a bottleneck in an application. Hence, it is useful to move parts of the application in the FPGA that may have short execution time to reduce the number of data transfers.

First of all, it is evident that the gap junction functions (iGapCell and fGap) needs to be accelerated on hardware since are the most computationally intensive part of our application. Furthermore, to determine where the rest of the functional blocks will be placed, the data transfers and each block's execution time needs to be calculated. The application is separated into two functions, as shown in Figure 4.7, Function 1 includes the gap junction computations, and Function 2 includes the rest of the computations. The partitioning options of our application between CPU and FPGA are shown in Figure 4.7. The execution time per function is taken from gprof. The first option is to place everything on the CPU, which has a total execution time of 13,080.49 seconds. The second option is to accelerate Function 1 on DFE with a total execution time of 3,654.76 seconds. The data transfers from the LMem are also considered by dividing the data needed for 10,000 steps with the memory's bandwidth. In the third option, both functions are accelerated on DFE with a total execution time of 3,003,75 seconds. Function 2 has an execution time of 4.99 seconds when ported into DFE compared to 660.25 seconds when is placed on CPU. We can notice a vast difference in execution time between the three options, and it is clear that the third option is the most promising. The execution times for the DFE are approximations and are calculated using the number of iterations of each loop and then is divided by an estimated frequency (200 MHz in our case). In Function 1, the execution time is calculated using $\frac{N_{cells}^2 \cdot N_{steps}}{f}$ and for Function 2 the execution time is calculated using $\frac{N_{gates} \cdot N_{cells} \cdot N_{steps}}{f}$. The data transferred between each function is taken from the C model and is divided with the available interface's data transfer speed (e.g., PCIe Gen2 x8). The CPU execution time can vary if, for example, a workstation is used, which has better performance; this will lead to a decreased execution time. Moreover, the DFE execution time is considered without any optimization, such as hardware loop unrolling, which will be discussed later, or, for example, functions that may be computed in parallel. The partitioning may change once the performance is modeled, as it will be described in section 4.3 and is highly recommended to revisit and reconsider this partitioning.

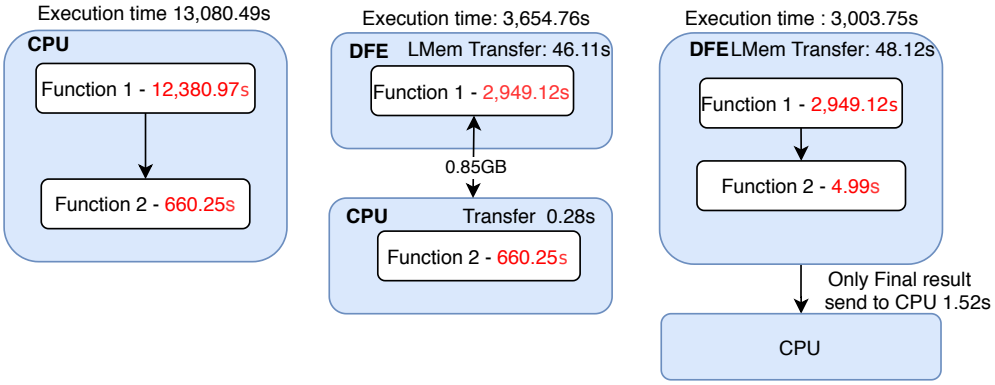


Figure 4.7: Partitioning options of the application and the impact on execution time

4.1.2 LoopFlow graphs

In this subsection, loop flow graphs are introduced, which help us get an insight into the volume of operations executed, their type, and the amount of data needed per loop. Loopflow graphs are chosen for this task since it is hard to observe, analyze, and understand data movements in an application. Using a loopflow graph will help us visualize and understand the previous analysis results and make the proper decisions before moving to implementation. Loopflow graphs focus on the loops in an application and how they interact with each other and are deemed helpful since most of the execution time is spent on loops.

Figure 4.8 illustrates the loop flow graph for the fully extended model of our application. This report will be centered on the most complex model (IO) to make explanations and analysis more manageable, but everything explained, holds for the rest of the models since are subsets of this model. A rectangle represents a loop, and the number of nested loops is annotated on top of each rectangle. The data transfers are illustrated as directed arrows, and its width denotes the amount of data that needs to be transferred. An element mentioned in each directed arrow is the amount of data transferred in bytes. The orange arrow denotes the inputs to the loops. The network size is set to 7,680 cells, the gate number is set 13, and the number of compartments is set to 3. Inside each loop's rectangle, the number and the type of operations per step are annotated. The number of operations is calculated by taking the maximum number of operations in each loop using the C model. Two of the inputs, the membrane voltage, and the gate activation variables have a double-sided arrow. In each new step, the old values are read, updated, and written back to memory. One can easily spot that the most computation-intensive loop is the gap junction loop. It has two orders of magnitude more computations than the gate loop and three orders of magnitude more computation than the compartment loop and it also requires the most data transfers. With this, the gap junction loop will undoubtedly be moved to DFE. It also makes sense to place the rest of the loops on DFE as it is also mentioned in the previous section. The membrane voltages are being used by all the loops in our application and are continuously updated. If the rest of the loops are placed on the CPU, synchronization will be needed from both sides, leading to a drop-in performance. Additionally, the gap junction will have to wait from the CPU side

to update the membrane voltage values and then continue with the next computations, impacting performance. In our case CPU will be responsible to initialize the parameter structures, rearrange input/output data and write data to output files. Finally, all loops are executed for a finite number of steps, which means that the number of operations and data needs to be multiplied by the number of steps.

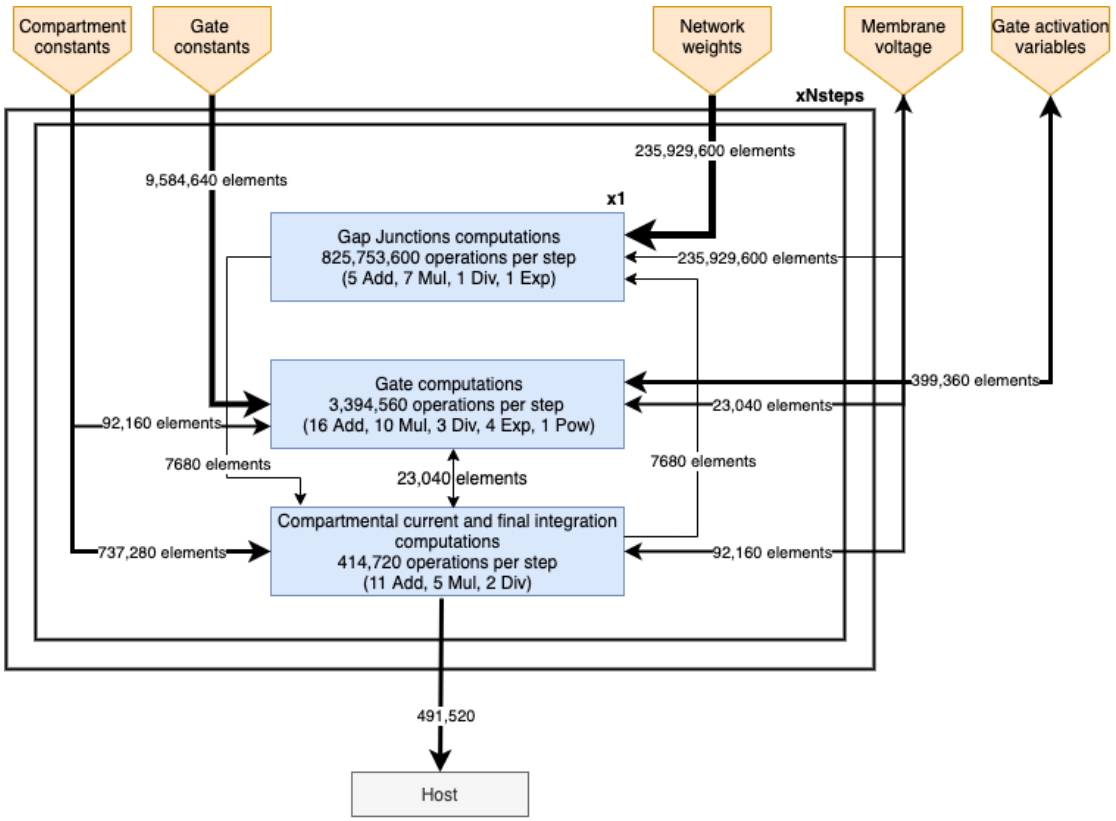


Figure 4.8: HHio Loopflow graph. Boxes represent loops and their internal operation types and count. In the top of each box the number of iterations is denoted. Arrows show data movements, and the number of elements (bytes) transferred for each step

4.1.3 Operation analysis

To understand the magnitude of each loop's executed operations, an operation analysis was conducted. The Figure 4.9 illustrates the number of operation per loop and Figure 4.10 the percentage of each operation per type. The impact of the gap junction loop is clearly shown in Figure 4.9, with the rest of the loops contribution being so minute that is barely visible on the graph. Compartment and gate loop use more additions and subtractions operations compared to gap junctions that multiplication is the most used type of operation. The reason for showing the percentages of each type of operation is clearly for comparison purposes. This will help later during the performance modeling to make decisions about the area since each operation has a different resource utilization. Additionally, knowing this is helpful to understand further which part is the most computationally intensive regarding the number of operations and the type and which part

worth to be optimized. Obviously, in our case, we already found out that gap junctions are the most computationally intensive part.

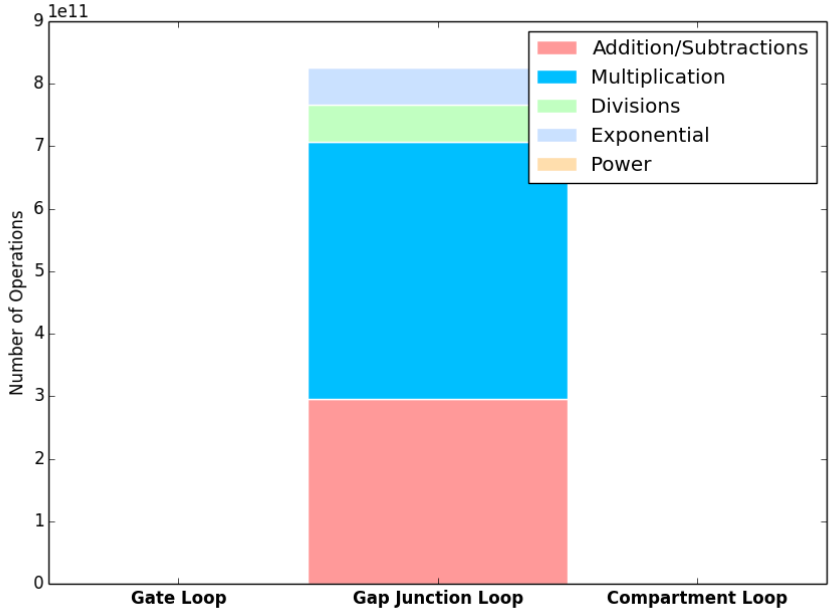


Figure 4.9: Number of operations per loop separated per operation type

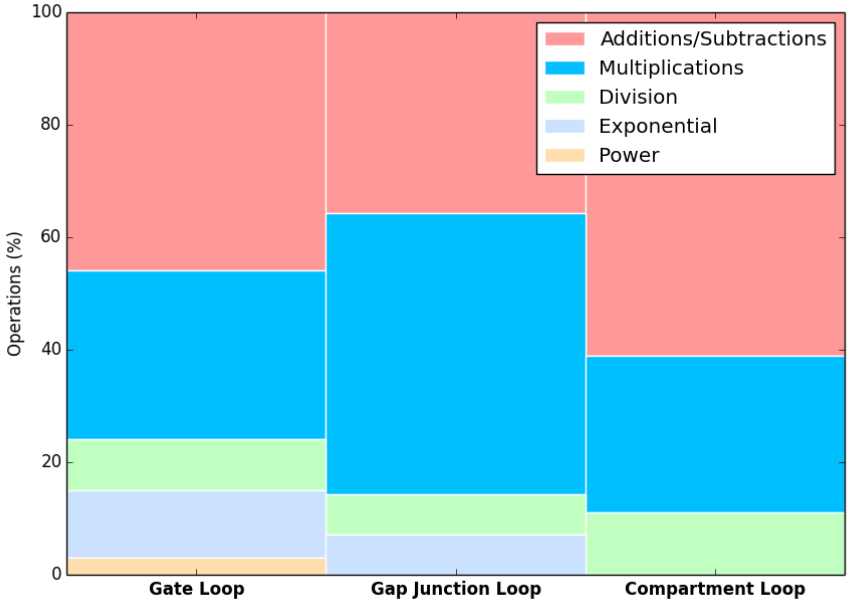


Figure 4.10: Percentage of each operation per loop

4.2 Software model

The software model is a simplified software implementation of the parts of the applications that are going to be ported onto DFE. In our case, the whole model will be ported onto the DFE. This means that a new C model is not needed. If this was not the case, the parts that will be ported should be extracted from the application and should be changed to represent the intended DFE implementation accurately. The parts that are going to be ported on DFE should be implemented as a standalone model. All the data shared with the other parts should be extracted and thoroughly validated with the original data to assist later during the implementation phase.

4.2.1 Numerical Analysis

One of the most essential and most challenging parts in implementing an application for DFEs is the numerical analysis. Instead of using floating-point arithmetic, a custom fixed-point arithmetic is sought that meet the demands of our application. This step is often neglected since it is time-consuming or believed that it would not improve the application. This analysis will provide us with information regarding the conversion from floating-point to fixed-point arithmetic. Three significant benefits that are achieved by converting an application from floating-point to fixed-point are; **(1)** the reduction of power consumption, **(2)** the reduction of hardware resources used, and **(3)** savings in memory bandwidth [61]. In this work, we are focused on minimizing resources and saving memory bandwidth. Minimizing resource utilization can lead to improved performance, considering that more resources will be available for computing capabilities. Minimizing memory bandwidth will help in improving the performance if the given application is memory-bound, and this will be further analyzed in the following sections. Additionally, the amount of memory needed to store the parameters will be reduced. This is yet another trade-off, as shown in Figure 4.11 between floating-point and fixed-point arithmetic. On the left side, floating-point arithmetic offers more precision, dynamic range, and less development time and on the other side, fixed-point offers less area utilization, less power consumption, and data compression.

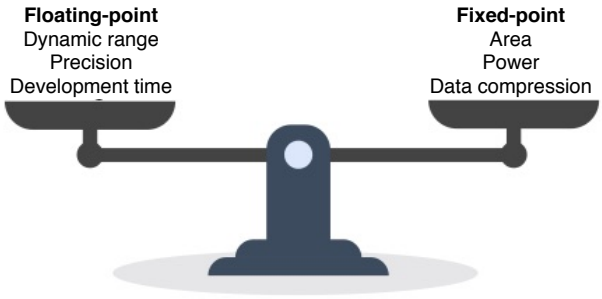


Figure 4.11: Trade-off between fixed-point and floating-point arithmetic

To find if a fixed-point implementation benefits worth over precision and dynamic range, further analysis of our software model is needed. This is achieved by using a set of tools/libraries provided by Maxeler Technologies that collects data during an application's

execution. The challenge in this step is to find the ideal fixed-point size, if it exists, that prevents overflow and underflow and accuracy close to the floating-point. Overflow in computer arithmetic means the calculation values exceed the maximum representable value, and underflow means that the results are smaller than what can be represented by the data type and are rounded to zero. The rest of this section will describe this process, which includes three stages mentioned below:

1. Value profiling;
2. Fixed-Point Simulation;
3. Error Calculation between floating-point and fixed-point.

Value Profiling

This analysis starts with the value profiling of our software model. For this step, a library can be used that automatically records exponent ranges for an appointed number of variables, and in this thesis, this library is provided by Maxeler Technologies. The most crucial variables are chosen to be traceable such as outputs and variables that hold intermediate results. Figures 4.12 - 4.17 illustrates the chosen variables exponent range. On the x-axis, their binary exponent range is shown; on the y-axis, the data snapshots per step, and each rectangle's color represent the percentage of the elements in a bucket. The statistical information regarding the value profiling for all the variables is taken per simulation step; otherwise, (e.g. per value update), a huge amount of data is created and is hard to analyze.

Intercellular current: Figure 4.12 illustrates the values of the intercellular current. During the first 4000 thousand steps, the values are zero since no external current is added to the model; this results in no differentiation in each neural cell's voltages; thus, the intercellular current is zero. As we can see, the values for the exponent ranges between 6 to -13 exponents values and most of the values are concentrated between -5 and -10 values.

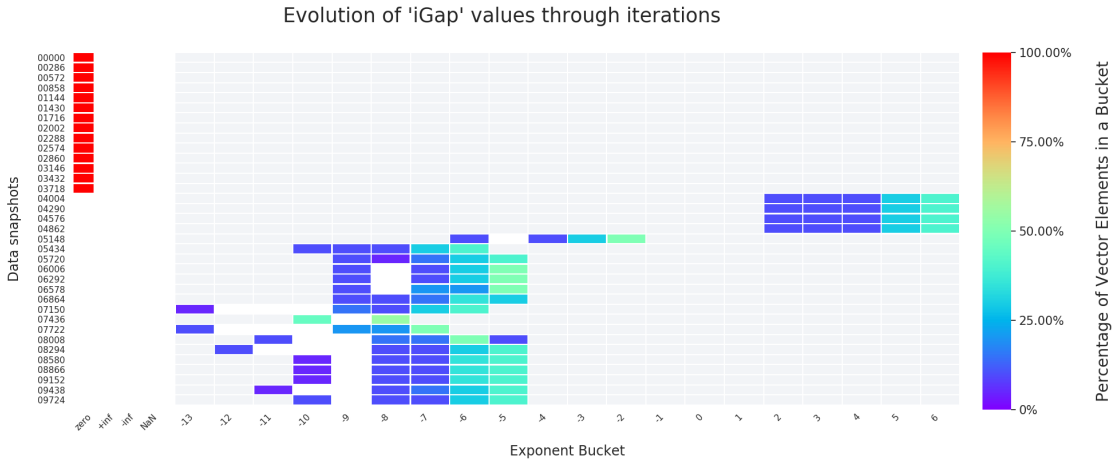


Figure 4.12: Intercellular current exponent range in a 10,000 step simulation

Channel current: Figure 4.13 shows the exponent range for each channel's current. Most of the binary exponent values are concentrated to the right side, ranging between 9 to -13 binary. On the other hand, we see a percentage of elements of about 10% ranging from -19 to -49 exponent.

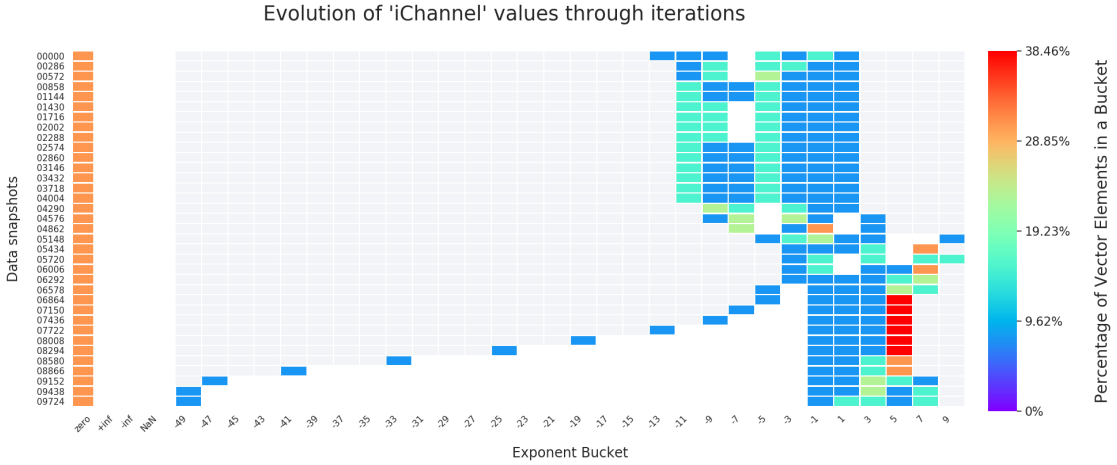


Figure 4.13: Channel current exponent range in a 10,000 step simulation

Summation of all channels current: This variable holds the sum of the currents flowing through all ion channels per compartment. In Figure 4.14, we can notice an interesting outcome, the binary exponent varies between 10 and -6 exponents. This means that the high binary exponent noticed from the separated channel currents is rounded to a number represented with less number of bits. This also means that we may lose accuracy during this summation depending on the allocated number of bits for fixed-point.

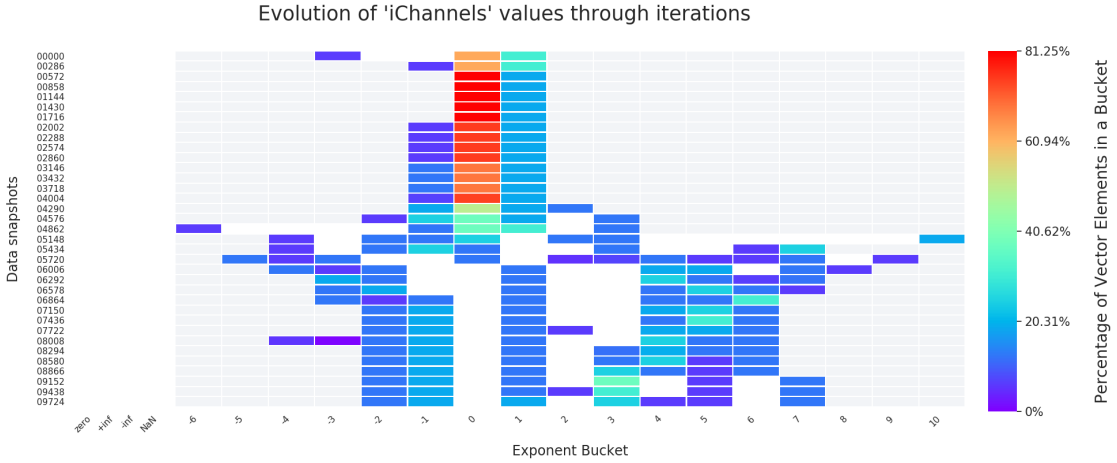


Figure 4.14: The sum of all channel currents exponent range in a 10,000 step simulation

Gate activation variables: Figure 4.15 shows the data snapshots for gate-activation variables. As in the channel's current, we can notice the same trend in binary exponents, a percentage of about 10% of elements reach a huge binary exponent of -53.

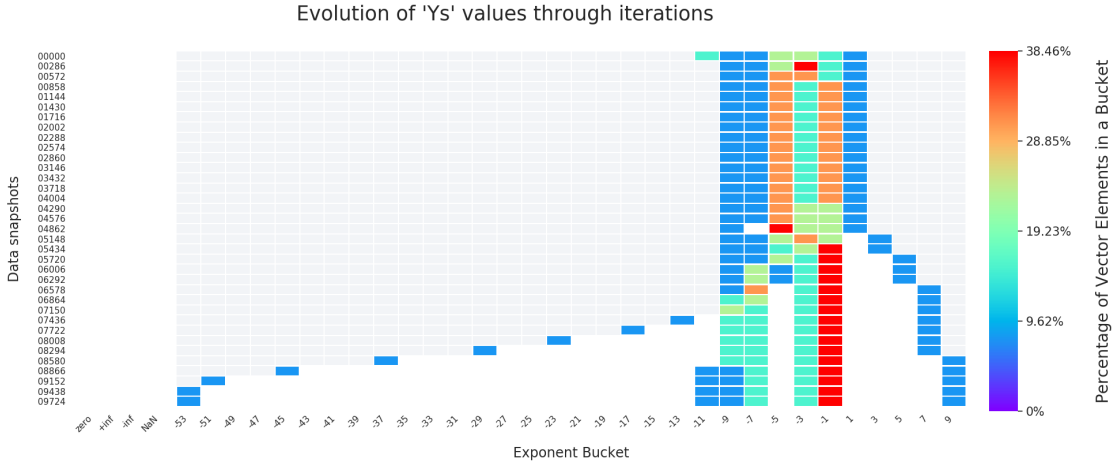


Figure 4.15: Gate activation variables exponent range in a 10,000 step simulation

Compartmental current: The compartmental current's binary exponents values are ranging between 5 to -7 but most of the values are concentrated between 5 to -3 binary exponent values as depicted in Figure 4.16.

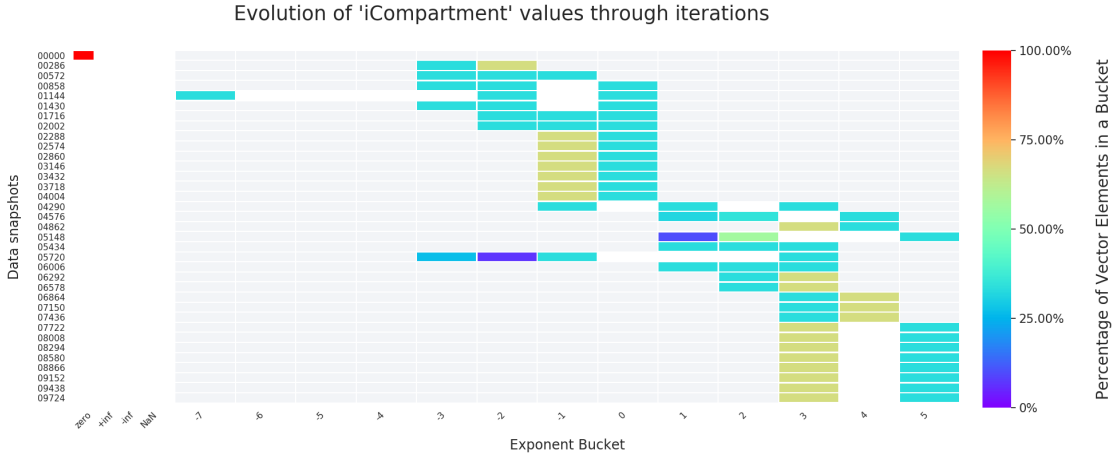


Figure 4.16: Compartment current exponent range in a 10,000 step simulation

Compartment voltages: The compartmental voltages consists of the voltages of axon, soma and dendrites compartments. Their binary exponent values are illustrated in Figure 4.17 and ranges between 6 and -2 and we notice a concentration of the exponent between 2 and 6.

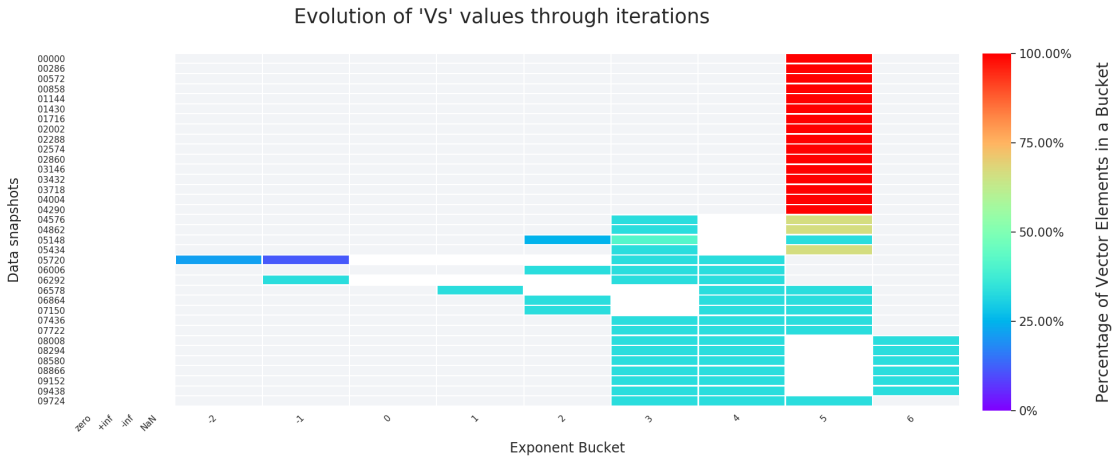


Figure 4.17: Compartment voltage values in a 10,000 step simulation

The output of the value profiling is all concentrated in Figure 4.18 for the variables mentioned above. The black lines enclose a possible scenario to use as a starting point in the fixed-point simulation. The values from the channel current and gate activation variables are not enclosed since both required a huge amount of bits that will negate the benefits of using fixed-point. Additionally, the percentage of the elements in those two variables is less than 10% and we will find out their impact during the next step. As a starting point in the fixed-point simulation, 10-bits will be used for the integer part and 13-bits for the fractional part.

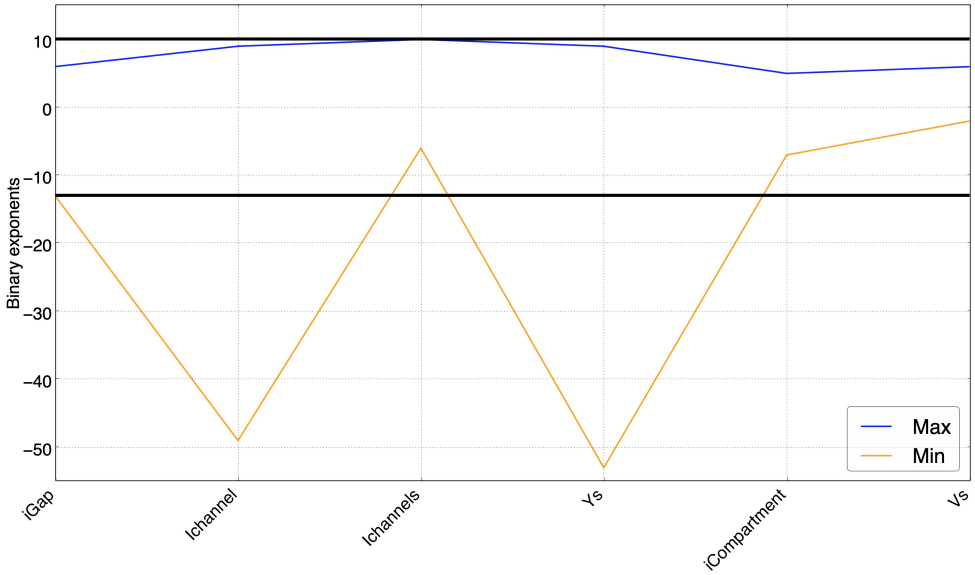


Figure 4.18: All profiled variables showing their decimal and fraction binary exponent

Observing the results above, three possible cases may be implemented regarding fixed-point, and the choice will be affected by the results from the fixed-point simulation:

1. **Change the entire application's operators to fixed-point datatype:**
   The first case is to change the entire application to the fixed-point data type.

Accuracy will be traded off for performance, and extra resources will be available to optimize our application (e.g., a higher unrolling factor for loops).

2. **Change only gap junctions and compartment computations**
Through the analysis, we can see that the gap junctions and compartment calculations can fully utilize fixed-point arithmetic since their exponent values are within an ideal range for fixed-point arithmetic. This will result in casting values from floating-point to fixed-point, and this cost needs to be considered.

Fixed-Point Simulation

In this section, the fixed-point simulation is performed considering the results from the value profiling done previously. The loss in accuracy is quantified, and the error is calculated to examine if the results are satisfying. For this analysis, Erasmus MC provided a Matlab file that describes our model with a set of realistic parameters to test fixed-point simulation. All the data are extracted from Matlab in different files, including the compartment constants, the gate constants, and the gap junction's connectivity matrix. The next step is to run the given C model in both floating-point and fixed-point with the extracted data sets, analyze the results, and quantify the error. For the fixed-point simulation, a library provided by Maxeler Technologies is used, and to use this library, the C-model is updated to C++ programming language. This part can be done using any other fixed-point library or tools; for example, Matlab has a tool that can help with fixed-point analysis as mentioned in [62]

The given Matlab file had five different variations regarding the interconnection matrix. The first one has the sparsest, and the last one the denser interconnection matrix, the rest of the parameters were the same. Using multiple data sets will show how our application reacts and see if using a fixed-point will produce results that are closed to the floating-point application. The densest connectivity matrix's simulation results will only be shown and discussed not to clutter this report with many graphs. The same analysis pattern is followed for the rest of the cases, and the results were the same.

For all the simulation conducted a network size of 100 neurons is simulated for 3,000 steps. This number of neurons was chosen because the simulation had high memory capacity demands. At some point, the system was out of memory, depending on the fixed-point data type size chosen.

The first choice was to use 10-bits as the decimal part and 13-bits for the fractional part taken from the value profiling section, but this did not produce satisfying results. It is important to mention that the gap junction weights data size is set to 16-bits, 1-bits for the decimal part, and 15-bits for the fractional part in all the cases. Figures 4.19 - 4.23 illustrates the error percentage for each fixed-point case and is calculated using Equations 4.1 and 4.2, and the average error is calculated using Equation 4.3. The absolute error in Equation 4.1 measures the difference between the measured values and the actual values, in our case the fixed-point values and the floating-point values are used respectively. The relative error in Equation 4.2 express as a fraction how large is the absolute error compared to the actual value and when multiplied by 100 you get the percentage of the error. The relative error is computed for each neuron in each step to

help us understand the magnitude of the error and its pattern. Many combinations of fixed-point sizes are included here to compare the difference in error. Seeing the graphs does not lead to an obvious choice since the average error is not large enough, but we can see that there are error spikes for certain cells that appear in all the combinations, which point us to investigate them. Figure 4.19 and 4.20 illustrates the 10/17 and 12/15 fixed-point cases with an average error of 0.2096% and 0.2672% respectively. Moreover, an error spike reaching an error of $2.2 \cdot 10^5$ % can be seen in both figures. This error leads to a completely different behavior for a certain neurons.

$$Error_{absolute} = Measured\ Value - Actual\ Value \tag{4.1}$$

$$Error_{relative} = \left| \frac{Error_{absolute}}{Actual\ Value} \right| \tag{4.2}$$

$$Error_{average} = \frac{\sum_{i=0}^{Values} Error_{relative}}{Values} \tag{4.3}$$

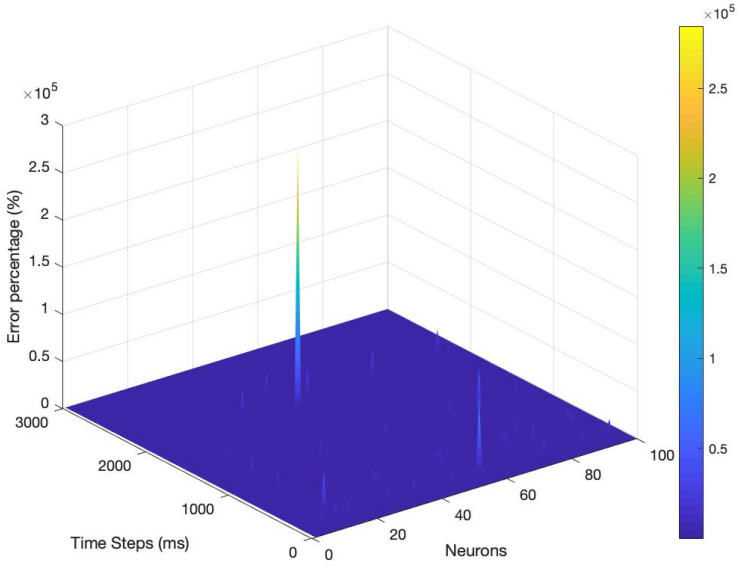


Figure 4.19: Error in a network of 100 neurons simulated for 3,000 steps (dt=0.01) using fixed-point datatype with 10 decimal and 17 fractional bits for axon compartment

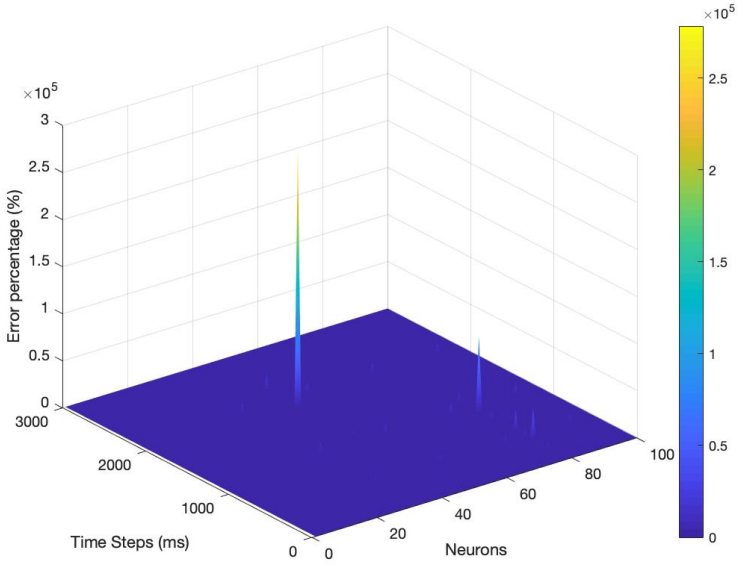


Figure 4.20: Error in a network of 100 neurons and 3,000 steps (dt=0.01) using fixed-point datatype with 12 decimal and 15 fractional bits for axon compartment

In the following three figures we can notice that, by increasing the decimal part to 13-bits, it results in two orders of magnitude less average error (from 0.2096% to 0.0089%). Having more than 13-bits does not improve the average error, and the error spikes still exist, as shown in Figure 4.21. When the fractional part is increased to 15-bits, it results in a smaller error spike, from $2x10^5$ to $1.3x10^5$, as shown in Figure 4.23.

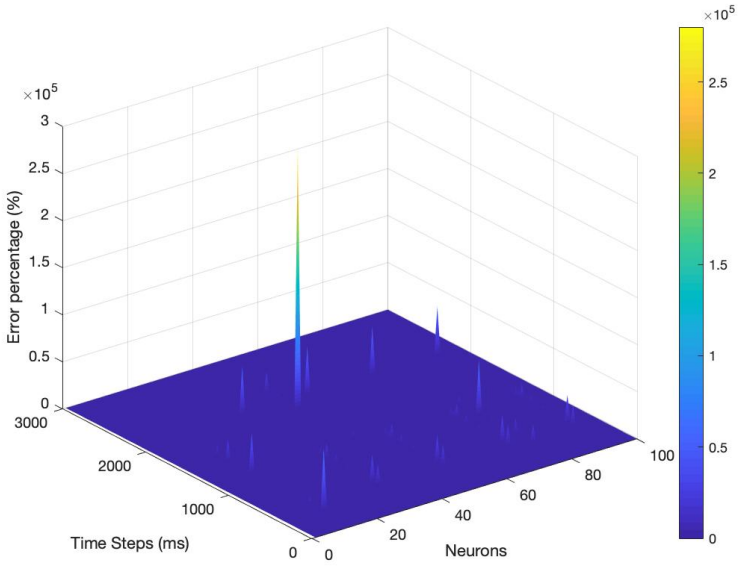


Figure 4.21: Error in a network of 100 neurons simulated for 3,000 steps (dt=0.01) using fixed-point datatype with 14 decimal and 13 fractional bits for axon compartment

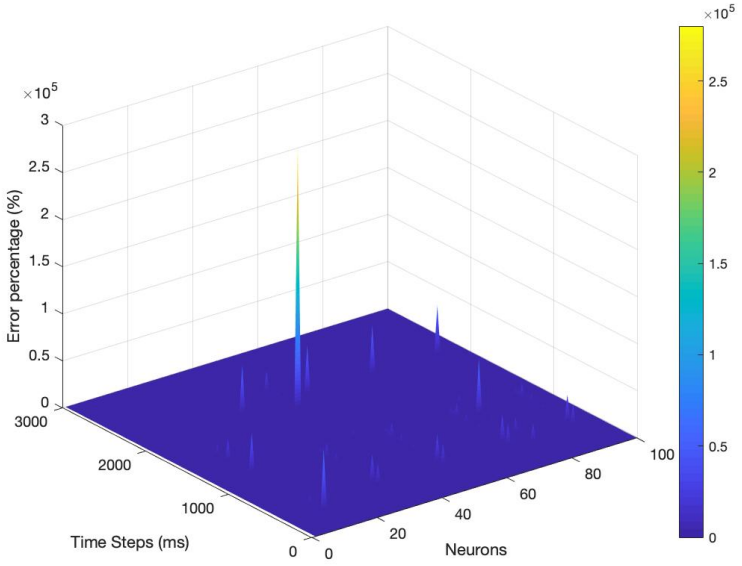


Figure 4.22: Error for a network with 100 neurons and simulated for 3,000 steps (dt=0.01) using fixed-point datatype with 13 decimal and 14 fractional bits for axon compartment

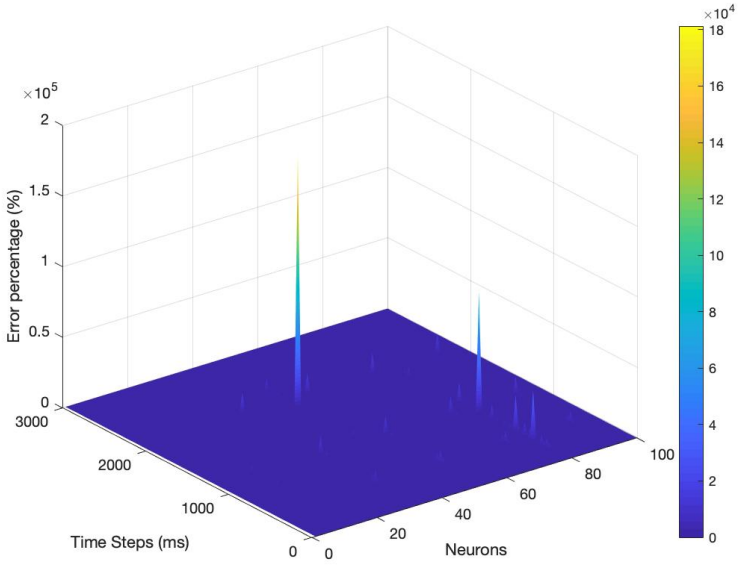


Figure 4.23: Error in a network of 100 neurons simulated for 3,000 steps (dt=0.01) using fixed-point datatype with 13 decimal and 15 fractional bits for axon compartment

The huge error spikes that are found in all the previous graphs are further investigated to determine the degree to which it affects the output. Two of the neurons that had a huge error spike can be seen in Figure 4.24 and Figure 4.25. The blue line represents the fixed-point and the red line represents the floating-point neuron's axonic voltage

output. As shown the options 10/17 (Decimal/Fractional), 13/14, and 14/13 have a flat output compared to floating-point. For neuron 45 we noticed that the 12/15 case has a phase difference where the 13/15 have a behavior close to reference. For both 12/15 and 13/15 options for neuron 78, we notice a phase difference, but the behavior is close to the reference, and Erasmus MC deemed it an acceptable output. Increasing the decimal part does not improve the output but increasing the fractional part does improve it as shown in the last subplot with size 13/19. The choice of 28-bits considering the average error, the spike errors, and its smaller number of bits seems the best choice. Important to mention that by checking the results with Erasmus MC, they pointed out the behavior is the most important thing to consider. Having an error that does not affect the output behavior can be considered as a valid output.

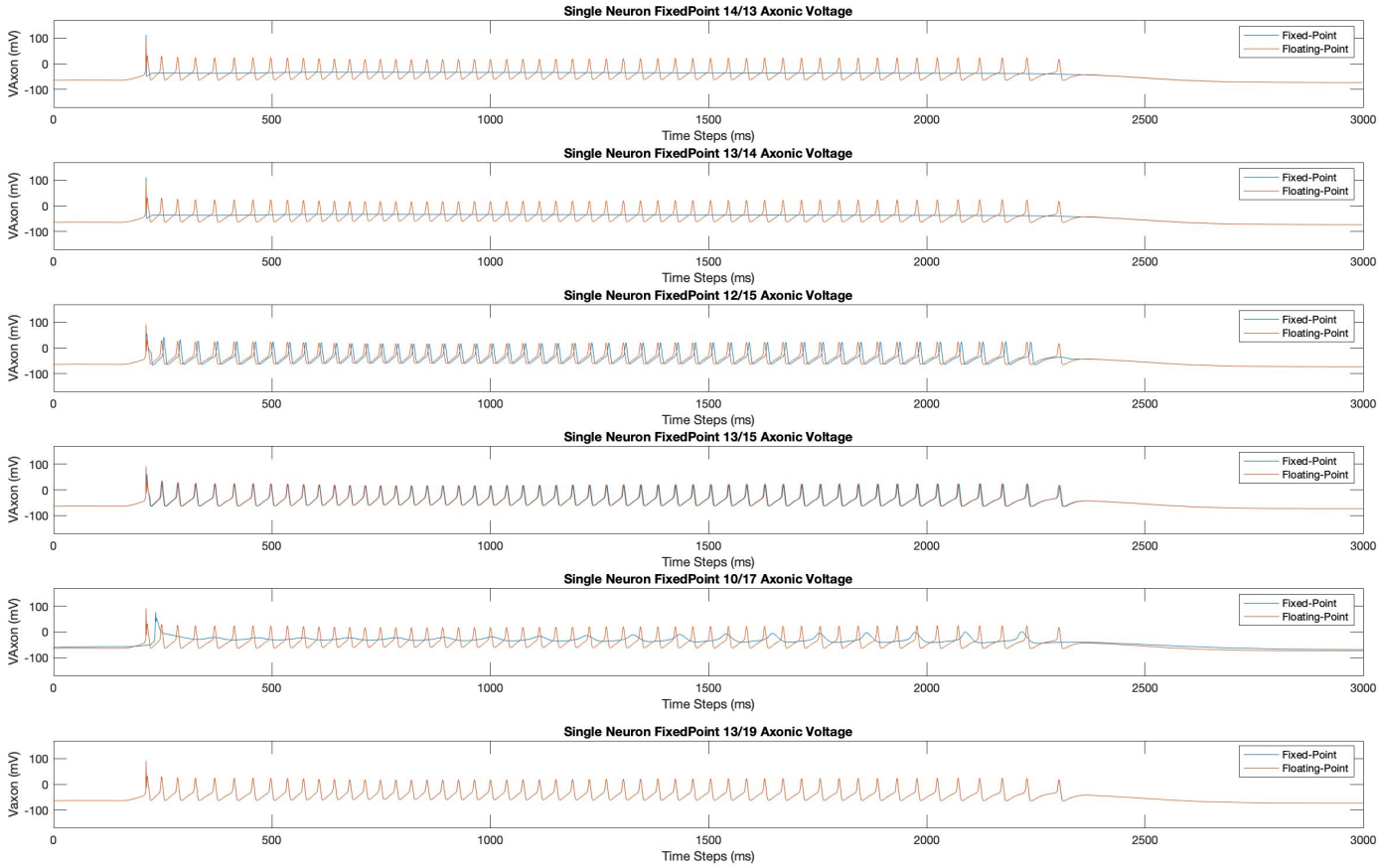


Figure 4.24: Output comparison between fixed-point and floating-point for Neuron 45

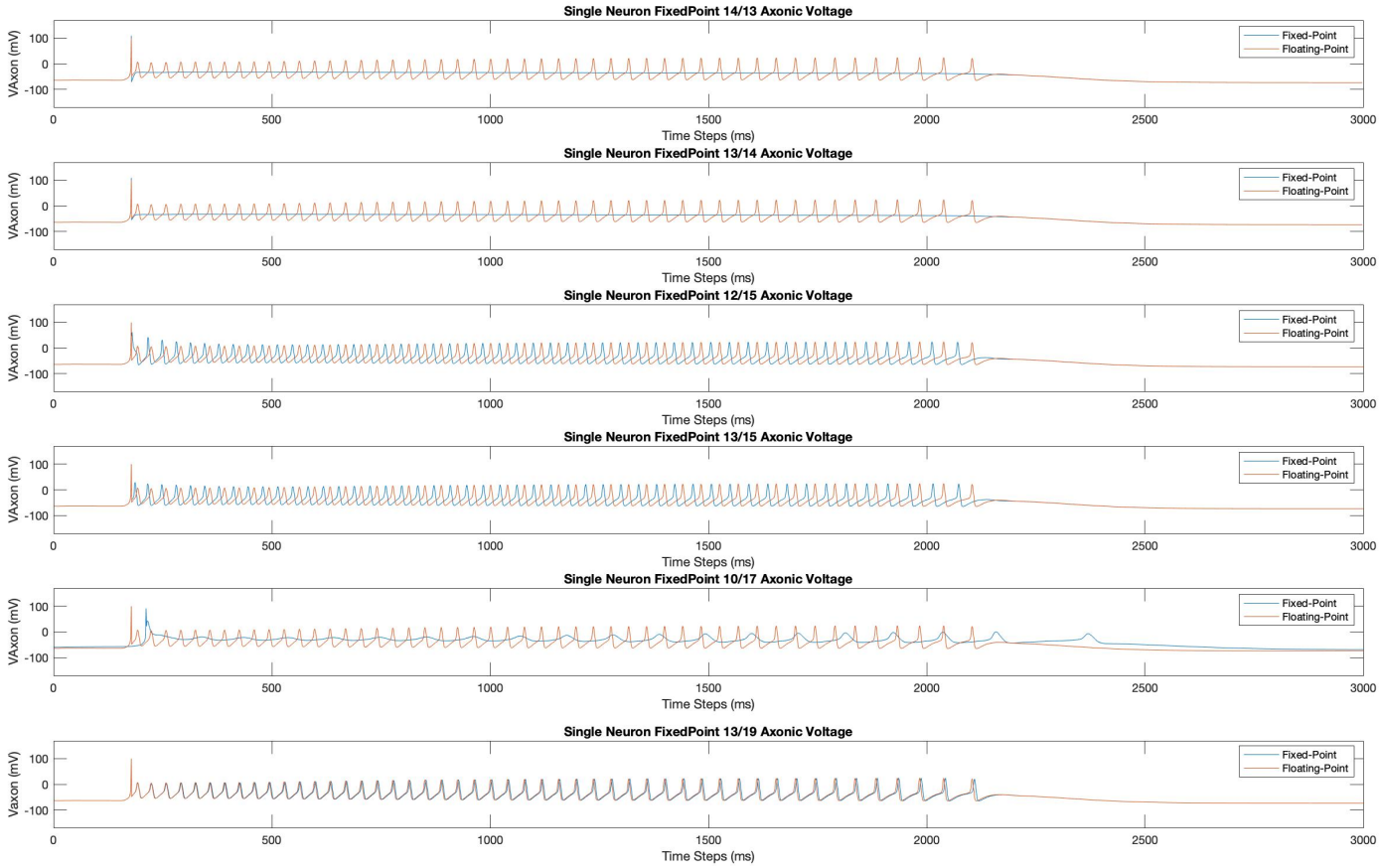


Figure 4.25: Output comparison between fixed-point and floating-point for Neuron 78

Besides, the axonic compartment, the somatic and dendritic compartments are also analyzed. The somatic and the dendritic compartments had better behavior and smaller error compared to the axonic compartment with the chosen fixed-point datatype. Figures 4.26 and 4.27 illustrates the error graph for somatic and dendritic compartments. The same analysis is executed, and the spikes are also analyzed separately to see if these errors have a huge impact on the output. These two compartments followed a different error pattern with an average error of 0.0037% for the somatic compartment and 0.0568% for the dendritic compartment. Finally, the output for both compartments is investigated and dendritic compartment output was close to the reference. On the other hand, the somatic compartment had a neuron where its output had a phase difference as shown in Figure 4.28.

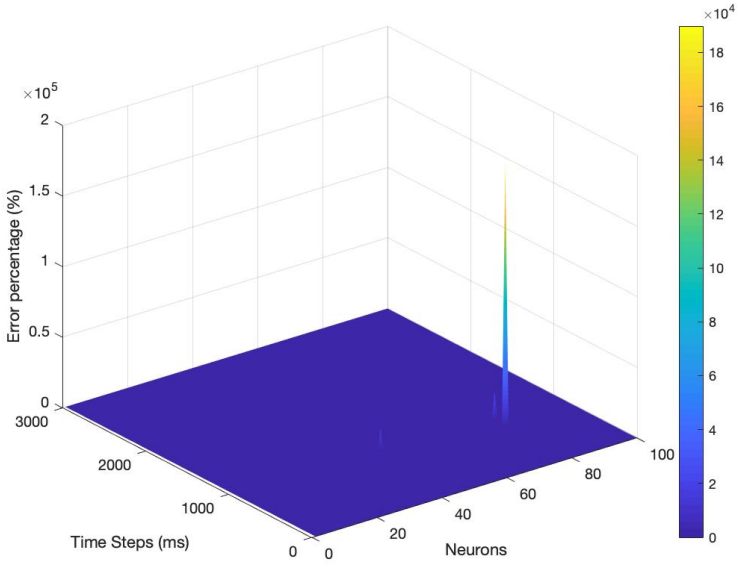


Figure 4.26: Error in a network of 100 neurons simulated for 3,000 steps (dt=0.01) using fixed-point datatype with 13 decimal and 15 fractional bits for soma compartment

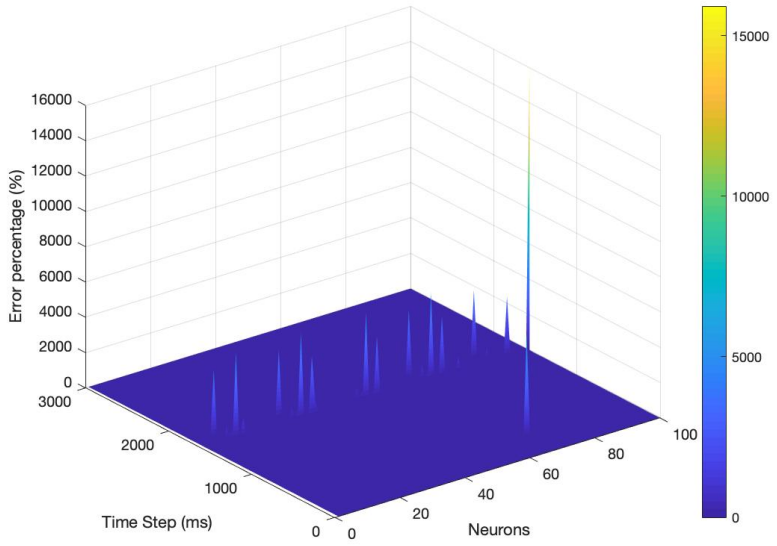


Figure 4.27: Error in a simulation of 100 neurons simulated for 3,000 steps (dt=0.01) using fixed-point datatype with 13 decimal and 15 fractional bits for dendrite compartment

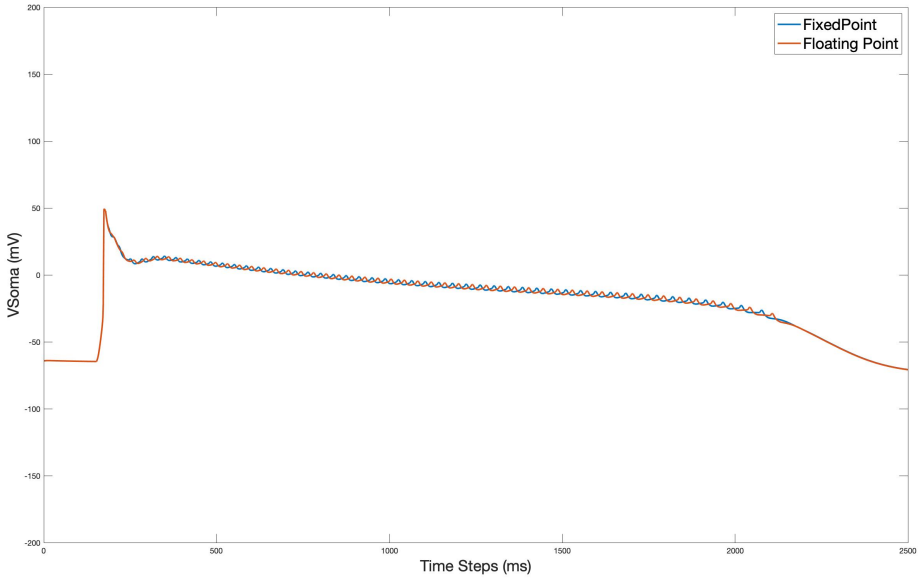


Figure 4.28: Somatic voltage output for fixed-Point(13/15) and floating-point for neuron 78

Finally, the error of the gate activation variables is analyzed. In this part, it was difficult to concentrate all the gate activation variables for all neurons in one graph. Hence, the error for just two neurons is illustrated. The output for each neuron gate activation variables can be seen in Figures 4.29 and 4.30. The x-axis represents the timestep, the

y-axis represents each neuron gate activation variable's output, and the z-axis represents the error percentage. The first figure illustrates a small error on the output using the chosen fixed-point data type with an average error of 0.0011% and a peak error of 0.08%. The second figure includes some high peaks that reach up to 22% of error while the average error is 0.1645%. These peaks appeared only in the last gate activation variables. This error may be produced in an intermediate calculation because the fractional bits are not enough to represent the intermediate results. Losing accuracy is expected, so we decided to continue with this fixed-point datatype as it seems that the results produced are better than what was expected. The error is only concentrated in two sections in the graph, and the average error is low. Using the results of this analysis, it is decided that all the model will be changed to a fixed-point. Implementing the other hardware options will cost in complexity and area since numerous castings will take place between different fixed-point sizes, or extra logic will be needed. These results will be considered again in the area usage prediction of the performance model section to take the final decision when we also considered the area utilization of fixed-point.

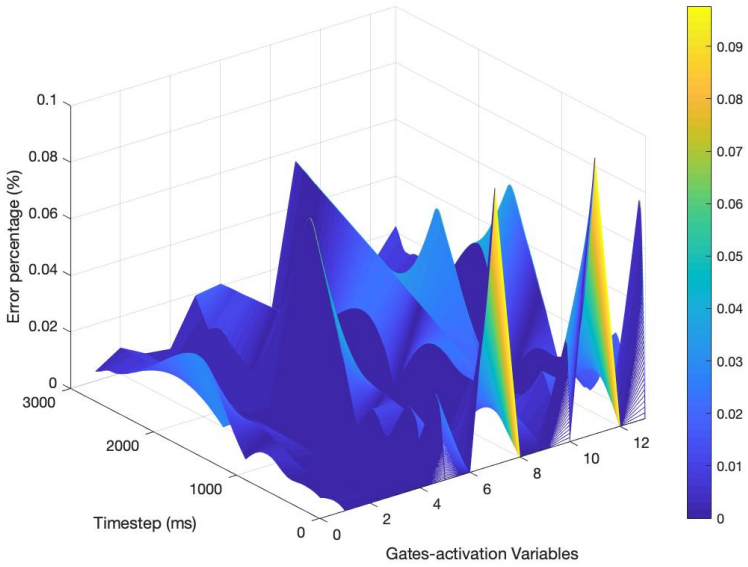


Figure 4.29: Error percentage for all the gate activation variables for neuron 0 using 13 decimal and 15 fractional bits

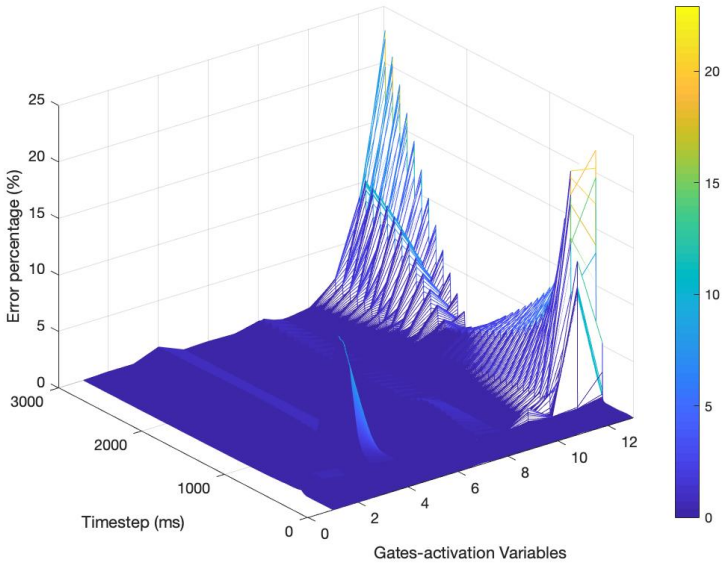


Figure 4.30: Error percentage for all the gate activation variables for neuron 50 using 13 decimal and 15 fractional bits

4.3 System Architecture

Using all the information from the previous section a system architecture is created, as shown in 4.34, which is used to estimate bandwidths (LMem and PCI) and compute time. Additionally, a detailed hardware system architecture for our application can also be created, which is shown in the following chapter since each modification is highlighted and explained. Firstly, which data needs to be streamed from LMem and PCIe needs to be determined. In our example, the neuron and the gate structures are all streamed from LMem since their size increases when you increase the number of neurons and as is also mentioned in [38], LMem provides a more stable kernel performance. Additionally, they required a lot of space to be stored on-chip. The same holds for the gap junction weights. It is left to decide which data needs to be stored on-chip and be kept close to the computations. You can find this by analyzing the software implementation and find which variables are often used. In this case, the cell/compartment voltage potentials, and the gate activation variables are often used and are continuously updated throughout the execution. This indeed is verified once the area estimation is done since the memory space needed may be larger than what is supported. Next, you need to consider the application's output and see if you need to send it back to the host through PCIe directly or save it in LMem if needs to be reused. For this application, the outputs are the cell/compartment voltage potentials and the gate activation variables which are stored in FMem; this means we don't need to save it in LMem, and it can be directly streamed to the host. To summarize, to create system architecture, you need to think about the data transfers to and from LMem, what is stored in on-chip memory. More details will be given in the following two chapters.

4.4 Performance Model

In this section, everything until now will be considered to create an accurate performance model. The performance model is used for rapid design space exploration without running the design's place and route. It is called rapid since it cancels out options that are not feasible (e.g., optimization that requires more hardware resources that are already available). It also guides the architectural design decisions taken and evaluates the final implementation. This means that the performance model can assist in achieving the best of our application by efficiently utilizing the resources and bandwidth of the selected platform. Finally, the execution time of a problem based on a defined size is estimated, providing a fast first estimation of the speed-up. Performance modeling is an iterative process to produce a refined version of the architecture; however, only the final results are presented here. On top of that, it promotes the concept of thinking before implementing, which is often neglected and leads to spending much unnecessary effort and time.

The reason behind the ability to estimate performance and area, is that DFEs only includes predictable building blocks; thus, it is possible to estimate the performance and area for a given architecture accurately enough. In other possible HPC platforms such as are CPUs and GPUs, the level of predicting performance on a given application is limited because they also include complicated mechanisms like caches and branch predictors. The performance model is created using a spreadsheet (e.g., Microsoft Excel), and all the information is recorded there.

The total time it takes to process a given workload on a DFE (T_{total}) can be calculated, as shown in equation 4.4. T_{init} is the time needed to initialize the DFE, for example, to set up memory, cast values from floating-point to fixed-point, or to fill the computational pipelines. It can be neglected if the workload is sufficiently large, and the execution time dominates the initialization time. The initialization time cannot be predicted since it depends on the used platform, but Maxeler mentions that a SLiC action overhead is between 1 and 100ms. In this work, T_{init} is going to be ignored, since this time is inevitably needed, and it cannot be calculated. T_{exec} is the time that the actual execution takes and consists of the time it takes to perform the computations (T_{comp}, section 4.3.2), the time it takes to transfer data between host and FPGA (T_{comm} section 4.4.3), and the time it takes to transfer the data between FPGA and the on-board memory (T_{mem}, section 4.3.4). Execution time is calculated using 4.5.

$$T_{tot} = T_{init} + T_{exec} \tag{4.4}$$

$$T_{exec} = max(T_{comp}, T_{mem}) + T_{comm} \tag{4.5}$$

4.4.1 Area Usage Prediction

Predicting area usage is crucial to determine the timing requirements. We first need to discover the degree of parallelism that our application can achieve, based on our available

resources. To do that, we need to understand how FPGA is using its hardware, which is explained below:

1. **Arithmetic Operations**
   The area used by the arithmetic operations can be calculated using the software model. We first count the number of arithmetic operations separated by type. Subtractions and additions can be counted as one since the same hardware is used for both operations. Next, the number of resources used by each operation type of the targeted FPGA device is found by conducting microbenchmarks or by using FPGA's vendor documentation [63, 64, 65, 66]. To conduct a microbenchmark, you place and route simple designs to extract each operation's resource utilization. Using MaxCompiler simplifies the whole process since it provides source-code resource annotation per line of code, which helps to get the exact number of resources used by each operation type. A microbenchmark for both floating-point and fixed-point is executed, and the results are shown in Table 4.4. Finally, the overall area usage can be estimated as the area cost of a single operation multiplied by the number of operations per type.

Table 4.4: Resource utilization per operation taken from Microbenchmark

	Floating-Point(32-bits)				*Fixed-Point(28-bits)*				*Fixed-point(27-bits)*			
Operation	*LUTs*	*FFs*	*BRAMs*	*DSPs*	*LUTs*	*FFs*	*BRAMs*	*DSPs*	LUTs	FFs	BRAMs	DSPs
Add/Sub	*200*	*316*	-	*2*	*32*	*33*	-	-	27	28	-	-
Mul	*117*	*174*	-	*2*	*4*	*99*	-	*4*	5	45	-	2
Div	*799*	*1387*	-	-	*1143*	*2357*	-	-	826	1716	-	-
Exp	*438*	*941*	*1*	*9*	*526*	*1368*	*1*	*10*	412	949	1	10
Cast	-	-	-	-	*330*	*437*	-	-	330	437	-	-

2. **IP modules**
   IP modules are considered any peripherals such as PCIe, memory interfaces, controllers, and others. To find the resource requirements, microbenchmarks can be used. Maxeler Technologies recommend to assume a slightly higher memory and logic used to lengthen the safety margins for scheduling and other needed control logic that is hard to predict. The number used in this report for the PCI and the memory controllers are found in [25].

3. **On-Chip Memory**
   On-chip memory is used for three different purposes. Operations and IP modules, but these are already included in the above two categories. Additionally, on-chip memory is used to buffer data and for FIFOs that schedules the kernel dataflow graph. The on-chip memory for data buffering can be calculated using microbenchmarks, but it may be easier and faster using the vendor's documentation [47]. For example, the device used in this work supports many arrangements for memory, which can be determined based on our application's requirements. The number of write/read ports requirements and the port width need to be determined since it dictates how the memory will be initialized. For example, when floating-point values are used, we need to store 32-bits of data, and if one read and one write port are required, this will lead to each BRAM be initialized as two independent 18Kb BRAM width size 512x36bits. When each BRAM's depth is known, the

estimated number of the total memory used can be calculated using Equation 4.6. The required depth is the number of parameters that we need to store, the required width is the size of each parameter, and the required ports are the number of ports necessary, all divided by what is supported by the hardware. Predicting the memory used for FIFOs that schedules the kernel dataflow graph cannot be done accurately. Although, since the application will be implemented using multiple kernels we included the FIFO buffers placed between kernels data streams. the default depth is 512 but this can be further increased to accommodate the application needs.

$$BRAM_{req} = \left\lceil \frac{depth_{req}}{depth_{hw}} \right\rceil \cdot \left\lceil \frac{width_{req}}{width_{hw}} \right\rceil \cdot \left\lceil \frac{ports_{req}}{ports_{hw}} \right\rceil \tag{4.6}$$

$$BRAM_{FIFO} = betweenKernelstreams \cdot \frac{FIFO_{depth}}{depth_{hw}} \tag{4.7}$$

4. **Loops**
   Dealing with loops is crucial in hardware. The loops implemented in software can be fully unrolled or partially unrolled on hardware. Unrolling on hardware means that the number of operations needs to be multiplied by the unroll factor, as shown in Figure 4.31. This means that the hardware utilization is also multiplied by the unroll factor, as are the input data streams. Fully unrolled loops may not be feasible due to the area or bandwidth limitations; this means that loops can be partially unrolled. Multiple cycles are needed to compute the final result of a partially unrolled loop. For example, a loop with N^2 iterations and an unroll factor of 24 it needs $\frac{N^2}{24}$ iterations to compute the final result.

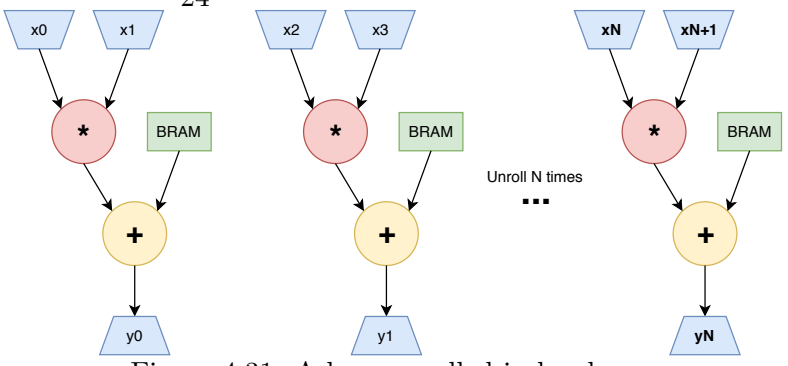


Figure 4.31: A loop unrolled in hardware

5. **Conditional Statements**
   In contrast to conditional statements on CPUs, on hardware, it is necessary to implement logic that can deal with all the possible branches of these statements. The final result is selected between the alternatives branches using a multiplexer. This means that the resource requirements for an application are all the possible branches of the conditional statement and the cost of a multiplexer.

Figures 4.32 and 4.33 illustrates the estimated resource utilization for both floating-point and fixed-point. The illustrated resource utilizations shown are for the HHio model, with the gap junctions computations being unrolled in total 192 times for floating-point and 180 times for fixed-point. We notice that DSP utilization is higher for fixed-point compared to floating-point. It was expected since our application requires higher accuracy than what is "optimal". The usage of 27-bits fixed-point datatype was going to provide reduction in hardware usage, as shown in Table 4.4. It requires 2 DSPs per multiplication, as floating-point compared to 28-bit fixed-point datatype, which requires 4 DSPs. On the other hand, fixed-point arithmetic does not use DSPs for additions/subtractions compared to the 2 DSPs used by floating-point operations. Since the gap junction loop is the one being unrolled, based on the operation analysis in section 4.1.3, we see that gap junctions include more multiplications than additions/subtractions and thus the increased DSP usage.

On the other hand, we can see that floating-point utilizes 17.79% more LUTs and 7.56% more Flip-Flops but 1.39% less BRAMs. The resource utilization is kept close to 80% since, in the following steps, estimated frequencies are based on designs with less than 80% resource utilization (as proposed by Maxeler). The benefit of decreasing resource utilization is not evident here, but this is the reason this analysis is executed. Each application varies, and many applications will benefit more from using fixed-point arithmetic. In this work, fixed-point arithmetic will be investigated since there is also the benefit of data compression, which will save memory bandwidth and capacity and it will be discussed later. This is an iterative step in our process, and with more results and insight for the application and the tools, the architecture is continuously updated until the final result is reached.

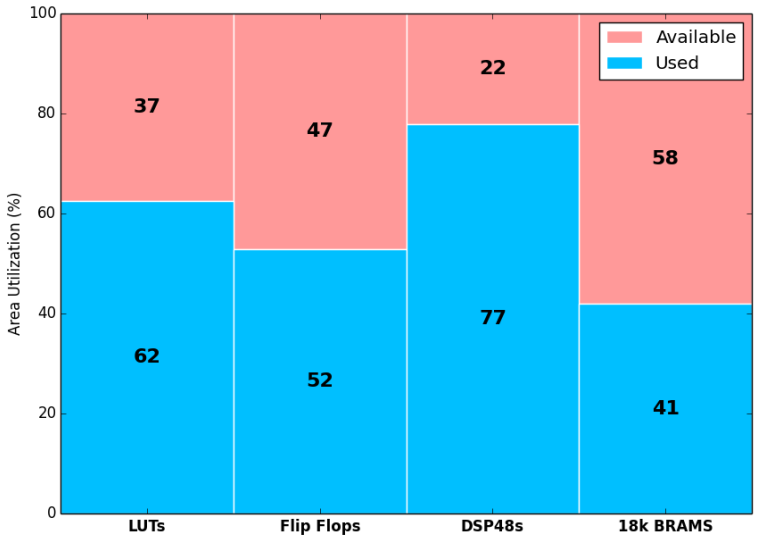


Figure 4.32: Estimated Resource Utilization using floating-point operators

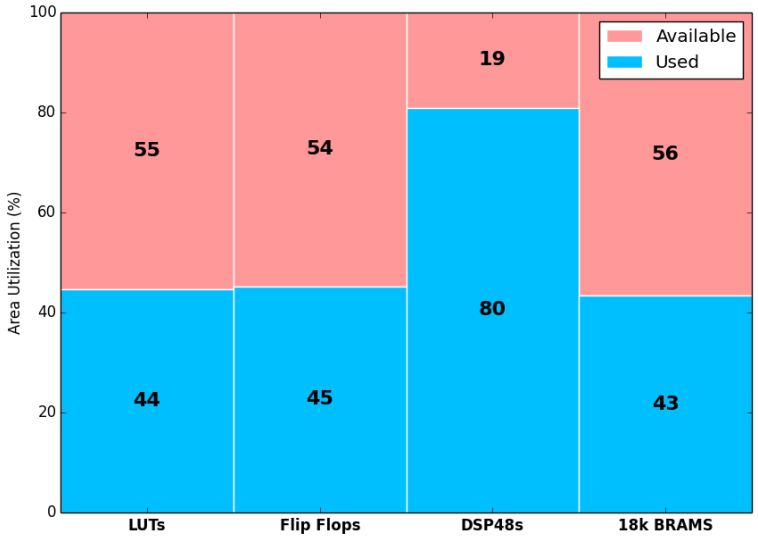


Figure 4.33: Estimated Resource Utilization using fixed-point operators

We first need to define an initial architecture to determine what to model in the performance model. As mentioned in Chapter 2, the Xilinx Ultrascale architecture utilizes three individual dies called SLRs, and inter-connectivity between these dies is limited. When the hardware is unrolled more times, it spreads the design in multiple SLRs. Consequently, long routing paths between dies result in tools not finding a way to route the design. As such, it is often a good idea to treat each SLR as it would be a separate FPGA. Considering the estimated resource utilization, it is obvious that our application needs to be split into multiple kernels since it will consume more than 30% of the total FPGA fabric (roughly every die utilizes 33% of the total resources). As shown in Figure 4.34, the most promising solution is to separate the gap junction loop into two different kernels. This makes sense since it is the most computation-intensive part of our application; thus, it will require more resources when it is unrolled. By dividing the design into more kernels gives space to the tools to find an efficient placement of the kernels across the different SLRs. The first gap junction kernel will calculate the first half cells intercellular current and the second kernel, the last half cells intercellular current. The main kernel will include the gate and the compartment loops. Finally, the integration part will use the data received from each gap junction kernel, and it will calculate the final result. FMem will be used to store intermediate results, as is the intercellular current, which needs a number of iterations to be calculated. The membrane voltages and the gate activation variables are updated in each step are stored in FMem to be close to the computations.

Figure 4.34: HHio system architecture

4.4.2 Compute Performance Prediction

After estimating the area usage, the compute performance can be predicted without considering the bandwidth limitations. The performance is calculated by dividing the total data processed by the product of frequency and the data processed per cycle. The data processed per cycle is the unroll factor of each gap junction kernel, as shown in Equation 4.8. It is safe here to neglect the main kernel computation time since all kernels compute in parallel, and gap junction's kernels will dominate the computation time. The frequency cannot be predicted, although Maxeler Technologies mentions that it can be safely estimated to be between 200-350 MHz. This is estimated based on experiments using artificial designs to fill up the chip up to 80%. In our application, the data needed to be computed per step is the total number of iterations for the gap junction kernel. The frequency is set to 200MHz, and the unroll factor is taken from the previous section for each data type version per kernel (96 for floating-point and 80 for fixed-point). N_{cells} and N_{steps} represents the network size, which is set to 7,680 cells, and the number of steps, which is set to 1,000. The total time needed is **1.53** seconds for floating-point and **1.84** seconds for fixed-point.

$$T_{compGJ} = \frac{n_{tot}}{f * n_{perCycle}} * N_{steps} = \frac{\frac{N_{cells}}{2} \cdot N_{cells}}{f * unrollfactor} * N_{steps} \tag{4.8}$$

4.4.3 I/O Bandwidth Usage

The bandwidth between the host and the accelerator (FPGA) depends on the physical interconnect that exists in each platform. Usually, for acceleration platforms and in our case, PCIe interconnect is used. PCIe uses bidirectional links; this means that the full bandwidth can be used in both read and write directions simultaneously. Our platform supports up to 4GB/s, which can be found using the vendor's manual. This number only describes the theoretical maximum performance; realistically, one can roughly achieve 80% [9]. To estimate the communication time between host and accelerator, the maximum data sizes need to be estimated streamed in either direction, choose the largest stream, and divide it by the physical interconnect bandwidth (PCIe) as is also depicted in Equation 4.9. The accelerator's input data are the initialization parameters that are required only during the first step and can be neglected. On the other hand, the output data includes the voltages for each compartment and the gate activation variables per cell for all steps; this is what dictates the communication time. The calculated parameters sizes used are calculated having a network of 7,680 cells, 3 compartments for each cell, 13 gates, and 1,000 steps, and the communication time is calculated to be **0.14** seconds. The fixed-point results will be cast to floating-point since the host does not support fixed-point arithmetic, which means that the output stream's total size is not changed. When the models support single compartments or fewer gates, this means that the communication will decrease, but to not clutter the report with many numbers, the worst case is shown.

$$T_{comm} = \frac{max(Data_{in}, Data_{out})}{Bandwidth \cdot efficiency} = \frac{N_{cells} * N_{steps} * (N_{compartments} + N_{gates}) \cdot size}{Bandwidth \cdot efficiency} \tag{4.9}$$

4.4.4 On-Board Memory Behaviour

The on-board memory used in our FPGA platform is DDR memory-based, the bandwidth for this memory is unidirectional which results in both directions (read/write) sharing the bandwidth. With this in mind, the time spend on memory transfers can be calculated using Equation 4.10.

$$T_{mem} = \frac{S_{in} + S_{out}}{BW_{DDR} \cdot DDR_{efficiency}} \tag{4.10}$$

No data is written back to LMem since is decided to use the PCIe interface to transfer data to the CPU host at the end of the execution. This means that S_{out} can be neglected, and S_{in} is described by Equations 4.11 and 4.12 for each type of kernel. $Size_{compartmentStruct}$ and $Size_{GateStruct}$ represents the size of each structure as shown in Table 4.5 and $Size_{weight}$ the size of each gap junction weight, all in bits. $N_{compartments}$ and N_{Gates} represents the total number of compartments and gates. Table 4.5 illustrates the sizes of all the supported models in both floating-point and fixed-point. The cell structure represents the single compartment cells and the gate structures supporting

custom gates or not are also shown. This table is constructed using the software model and custom fixed-point sizes for integer parameters are added based on the needs of the application.

$$S_{MainKernel} = (Size_{compartmentStruct} \cdot N_{compartments} + Size_{GateStruct} \cdot N_{Gates}) \cdot N_{steps} \tag{4.11}$$

$$S_{GapJunctionKernel} = (\frac{N_{cells}}{2} \cdot N_{cells} \cdot Size_{weight}) \cdot N_{steps} \tag{4.12}$$

The BW_{DDR} represents the DDR memory bandwidth, which is 15.8GB/s per DIMM. Finally, the parameter $DDR_{efficiency}$ represents the proportion of theoretical DDR bandwidth achievable for a given data-set. To achieve good memory performance, it is crucial to make any memory access as linear as possible, which is the case in our application. Additionally, $DDR_{efficiency}$ is important to understand since it makes a huge difference in DDR performance, and it is explained using an example. Our FPGA platform includes three DIMMs of DDR memory. If all DIMMs are used together using a single memory controller, a burst of data of 1,536-bits is possible. Each DIMM can have a burst of 512 bits if used alone (different memory controllers). This means that if we need to read a 3,072-bits size of data, two bursts are needed to access them. To achieve good memory performance is recommended to have a burst of large enough data, as shown in Figure 4.35. In many cases, it may worth using separate memory DIMMs to achieve a higher number of bursts; for example, data of 4,096-bits can achieve an efficiency of 75% when a single DIMM is used compared to 35% when all DIMMs are used.

Additionally, the split of the memory controller needs to be considered if the design spreads in all the SLRs to avoid SLR crossings because it will affect timing. This means each kernel will have its own memory controller utilizing one DIMM of memory. We can now revisit the unroll factor considering bandwidth limitations for two cases, having different memory controllers per kernel and the case utilizing a single controller for all the kernels. Using the Equations 4.13 - 4.16 we can calculate the bandwidth needed when one memory controller is employed. The bandwidth of both compartment and gate constants is found by using a ratio of the times the data are needed from the kernel divided by the total amount of ticks. The bandwidth of each gap junction kernel is the total number of gap junction weights multiplied by the unroll factor and the frequency. Then using the Equations 4.17 - 4.20 we can calculate the bandwidth when three memory controllers are employed. The bandwidth for both gap junction kernels is as in the first case, on the other hand, the bandwidth needed to stream compartment and gate constants is calculated using a ratio of the total number of compartments/gates divided by the total number of gates. In the second case, bandwidth will be underutilized once the main kernel finishes its computation and it will wait for the gap junction kernel to finish to move to the new step.

Single Controller:

$$BW_{compConstants} = Size_{compartmentStruct} \cdot \frac{N_{cells} \cdot N_{compsPerCell} \cdot uf}{N_{cells} \cdot \frac{N_{cells}}{2}} \cdot f \tag{4.13}$$

$$BW_{gateConstants} = Size_{GateStruct} \cdot \frac{N_{cells} \cdot N_{gatesPerCell} \cdot uf}{N_{cells} \cdot \frac{N_{cells}}{2}} \cdot f \tag{4.14}$$

$$BW_{GapJunctionWeights} = Size_{weight} \cdot f \cdot uf \tag{4.15}$$

$$BW_{Total} = BW_{compConstants} + BW_{gateConstants} + BW_{GapJunctionWeights} \tag{4.16}$$

Three memory controllers:

$$BW_{compConstants} = Size_{compartmentStruct} \cdot \frac{N_{cells} \cdot N_{compsPerCell}}{N_{cells} \cdot N_{gatesPerCell}} \cdot f \tag{4.17}$$

$$BW_{gateConstants} = Size_{GateStruct} \cdot f \tag{4.18}$$

$$BW_{MainKernel} = BW_{compConstants} + BW_{gateConstants} \tag{4.19}$$

$$BW_{GapJunctionWeights} = Size_{weight} \cdot uf \cdot f \tag{4.20}$$

All the information from estimated area utilization to estimated timings and bandwidth is concentrated in Table 4.6. The application is bound by the LMem bandwidth; the application reaches the memory-bound for an unroll factor of 16 and 32 per kernel, for floating-point and fixed-point respectively. The same holds for both memory controller options (Option 1 and Option 2 in the table). The unroll factors are also used to calculate the number of bursts and memory efficiency based on Figure 4.35. The bandwidth for option 2 is separated into three columns. The first column shows the bandwidth required from the first kernel (main kernel) and the rest two columns show the bandwidth for the two gap junction kernels. Both versions of the model (floating-point and fixed-point) are going to be implemented since we need to compare them but additionally a third implementation will be also implemented. The third option is to use floating-point operations to keep the accuracy and the lowest DSP usage and stream the gap junction weights using fixed-point datatype to achieve a higher unroll factor before it becomes memory-bound.

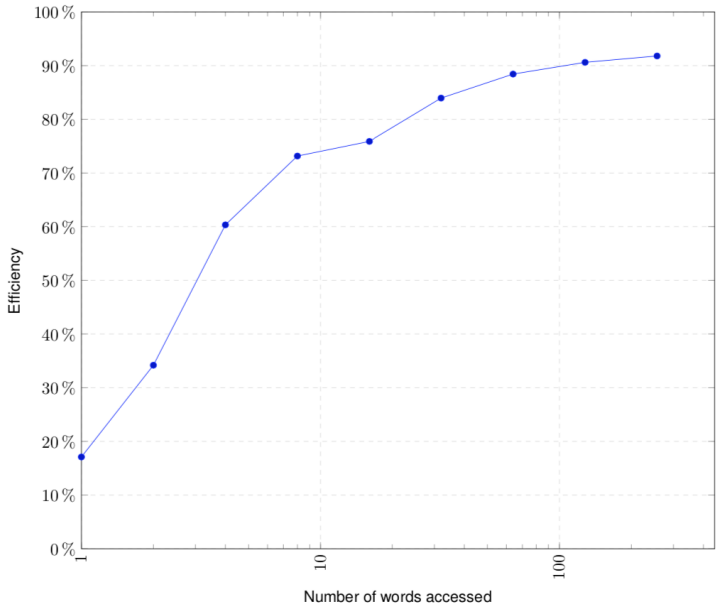


Figure 4.35: DDR4 memory efficiency as measured on an Alveo U250 [9]

Table 4.5: Floating-point and Fixed-point data-structures sizes

	Floating - Point			*Fixed-Point*				
Structrure / Datatype size	*Integer*	*Floating-Point*	*Size*	*4-bits*	*16-bit*	*24-bits*	*28-bits*	*Size*
Gate Structure	*3*	*9*	*384*	*4*	-	-	*8*	*240*
Gate Structure (Custom)	*3*	*21*	*768*	*3*	-	-	*21*	*600*
Cell Structure	*2*	*4*	*192*	-	-	*2*	*4*	*160*
Compartment Structure	*3*	*5*	*256*	*1*	-	*2*	*4*	*164*
Gap Junctions weights	-	*1*	*32*	-	*1*	-	-	*16*

4.5 Summary

In this chapter, the design process was applied. It started with the HHio model analysis to understand it nature and the execution hotspots requiring more attention. Following using the software model, we conducted a numerical analysis to search for possible fixed-point arithmetic sizes with respect to accuracy that will benefit our application. Finally, a performance model is created that helped us to estimate the performance based on the resource utilization. The execution time and the time spent to transfer data from memory are also estimated. Finally, it is found the HHio model becomes memory-bound, and the maximum unroll factor reached is calculated.

Table 4.6: Resources utilization, memory bandwidths and time spent on computation, communication for varying unroll factor

	Unroll factor	*LUTs(%)*	*FFs(%)*	*BRAMs(%)*	*DSP(%)*	*Tcomp(s)*	*Tcomm(s)*	*Tmem(s) Option 1*	*Tmem(s) Option 2*	*BW(GB/s) Option 1*	*BW(GB/s) Option2*		
Floating-Point	*16*	*16.21*	*13.22*	*38.25*	*14.74*	*9.21*	*0.14*	*21.97*	*38.60*	*24.87*	*19.25*	*11.92*	*11.92*
	32	*25.47*	*21.14*	*38.99*	*38.99*	*4.61*	*0.14*	*13.81*	*19.85*	*49.78*	*19.25*	*23.84*	*23.84*
	64	*49.99*	*36.97*	*40.47*	*52.64*	*2.30*	*0.14*	*9.29*	*11.58*	*99.52*	*19.25*	*47.68*	*47.68*
	96	*62.51*	*52.8*	*41.95*	*77.9*	*1.54*	*0.14*	*8.33*	*10.68*	*151.37*	*19.25*	*71.52*	*71.52*
	128	*81.03*	*68.64*	*43.43*	*103.16*	*1.15*	*0.14*	*7.43*	*9.26*	*207.39*	*19.25*	*95.36*	*95.36*
Fixed-Point	*16*	*14.01*	*12.68*	*38.25*	*16.01*	*9.21*	*0.14*	*12.91*	*23.15*	*12.71*	*14.84*	*5.96*	*5.96*
	32	*20.65*	*19.71*	*38.99*	*30.05*	*4.61*	*0.14*	*10.55*	*19.27*	*25.44*	*14.84*	*11.92*	*11.92*
	64	*33.91*	*33.78*	*40.47*	*58.12*	*2.30*	*0.14*	*7.74*	*9.91*	*50.88*	*14.84*	*23.84*	*23.84*
	96	*47.17*	*47.84*	*41.95*	*86.19*	*1.54*	*0.14*	*5.66*	*7.71*	*77.94*	*14.84*	*35.76*	*35.76*
	128	*60.45*	*61.91*	*43.43*	*114.26*	*1.15*	*0.14*	*4.64*	*5.78*	*108.2*	*14.84*	*47.68*	*47.68*

5 Implementation

This chapter begins information regarding flexHH library implementation and its limitations regarding performance. Following that, the new DFE implementation and how the improvements are conducted are discussed in the next section. Next, details regarding the fixed-point arithmetic implementation and how it is incorporated are given. Finally, the final implementation for each model, the unroll factor, and the frequency achieved are also shown.

In total, the flexHH library includes five different models, as shown in Table 5.1, and each model supports additional features. As is already mentioned in Chapter 2, the models simulated can be as simple as a network of HH cells or complex as a network with IO cells (HHio).

Table 5.1: Supported features per implemented kernel in the flexHH library

	***Custom ion* gates**	***Multiple cell compartments***	***Gap junctions***
HH	✗	✗	✗
HH+custom	✓	✗	✗
HH+custom+multi	✓	✓	✗
HH+gap	✗	✗	✓
HH+custom+multi+gap (HHio)	✓	✓	✓

Figure 5.1 illustrates the general architecture for the HH, HH+custom and HH+multi+custom kernels. A general schematic is shown for these three models since the differences between them are not excessive. The difference between HH+custom+multi kernel and HH+custom is that the first one supports multiple compartments where the second does not. This means that the compartmental current calculation is not needed, and all gates belong to a single compartment/cell. The difference between HH+custom and HH kernels is that the first one supports custom ion gates where the second does not. As shown in Figure 5.1, the cell parameters that include the gate and cell constants are stored in LMem before the kernels start the execution. During the simulation, the output is also stored in LMem. At the end of each simulation, the DFE's output is read by the CPU host and printed to a file. Each kernel includes four distinct parts; the calculation of the compartmental current (calcIcompartment), which only exists for the HH+custom+multi model, the calculation of the gate variables (calcCateVariables), the calculation of the channel current (calcIchannels), and finally, the externally applied current (iAppFunc). All the above parts provide a current used to calculate the membrane voltage, as shown in Equations 3.1 and 3.14. To finish the calculation for each cell, it needs $\frac{gates}{unrollfactor}$ ticks. In the case of the

HH+custom+multi model, the compartmental current is calculated in parallel. To proceed to the next compartment/cell computation, the gates per compartment/cell must be calculated first. The output of all models comprises the compartment/cell voltage and the gate activation variables. Gate current calculations are unrolled in hardware, and the maximum performance is achieved with an unrolling factor of 4 for HH kernel and 2 for the other two kernels as mentioned in [38]. Having a higher unroll factor will not lead to a performance benefit since the models becomes the memory bound.

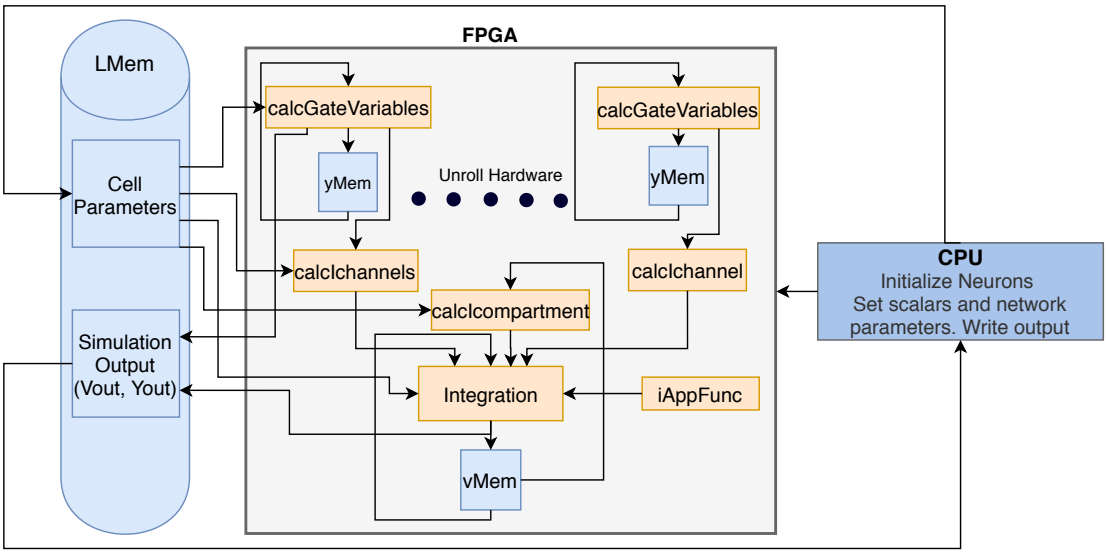


Figure 5.1: System Architecture of the Previous work for HH+custom+multi models

Figure 5.2 is created to illustrate the IO and the HH+gap kernel that supports the gap junction extension. HH+gap and IO kernels are identical to HH and HH+custom+multi kernels respectively with added the extension of the gap junctions. As illustrated in Figure 5.2, the architecture has a lot in common with the previous versions, with the addition of the gap junctions extension, there are some substantial changes. First of all, extra parameters are stored in LMem, representing the weights of the connections between cells/compartments. Moreover, the gap junction's calculation block is added, which calculates the current between one cell/compartment and the other cells/compartment in the network.

In the flexHH library, the HHio model has connections with other cells through gap junctions through the dendritic compartment. For the dendritic compartment, all the currents besides the gap junction current are summed and stored in IRestMem on-chip memory. Once the gap junction computations are finished, each dendritic compartment's respective value is read from IRestMem memory, updated with the gap junction current, and finally, each dendritic compartment's voltage is calculated and then stored in FMem(vMem) and LMem. FMem is used for storage since the membrane voltages are constantly used and fast access is needed. LMem is used for storage because, at the end of the simulation, the CPU host will read all these values. The compartments without the gap junctions are calculated and stored directly in FMem and LMem while the compartments with the gap junction connections are updated and stored at the end

of each step. The HH+gap kernel supports only a single compartment cell; thus, all cells connect to other cells through the gap junction connections. This simplifies things since all values stored in IRestMem need to be updated once gap junction computations are finished. In both cases, gate computations are not unrolled because now gap junctions are unrolled, and bandwidth is needed for that computation block.

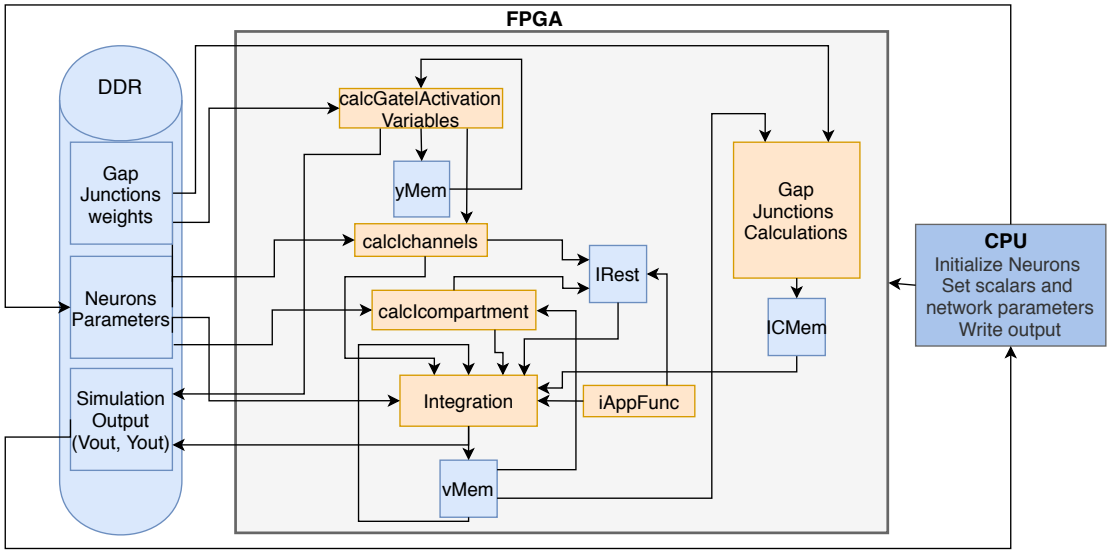


Figure 5.2: System Architecture of the flexHH library of the HHio model

Figure 5.3 illustrates an example how the gap junction computation block functions for both HH+gap and the IO models. In this example, as a test case, a network of 36 cells and an unroll factor of 6 is used. Recall that due to bandwidth we cannot fully unrolled this loop and partial unrolling is used. The number of computations executed in parallel depends on the unroll factor, and the number of iterations needed to calculate the intercellular current for a cell is $N_{cells}/unrollfactor$. This means to fully calculate the current for a cell, 6 iterations are needed. In the first 36 ticks the computations between all cells (0<i<35) and the first 6 (0<j<5) cells of the network are executed and then this is repeated till all the intercellular currents are calculated. In this example, we can see that instead of having a computation time of $N_{cells} \cdot N_{cells}$ ticks, we have $N_{cells} \cdot \frac{N_{cells}}{uf}$ ticks. The result produced by the gap junction block on each tick is updated in FMem called ICMem using a tree-adder to add the partial summation. During the final iteration, the stored value is read, updated, and sent to the integration part. The network weights between cells are provided from LMem in each tick and the data stream is unrolled as many times as the gap junction computations.

All model's source code is converted to the newest platform's version, called MAX5. This change mostly affected all the flexHH library models manager, and many deprecated commands regarding LMem needed to be updated. Once a working version of the HHio model was obtained, Place and Route for the new platforms were executed to detect its limits. The HHio model of the flexHH library achieved an unroll factor of 24 with a frequency of 180 MHz. Unrolling more results in tools not finding a successful placement

for the design. Table 5.2 illustrates the resource utilization of the HHio model and as we can clearly see, our FPGA is underutilized.

Considering the resource utilization of Table 5.2 the assumptions in the previous chapter holds. The most obvious limiting factor for not achieving a higher unroll factor is that the design goes beyond a single die, and the tools have difficulty in the place and route of the design. When three different dies are used in one platform, it leads to many limitations when implementing a large complex design. Pipelining everything manually and place everything efficiently cannot happen when a complex design is implemented. This means that extensive signal propagation delays across multiple dies will be present, which does not help meet timing. When most of the SLRs connections are occupied, and the longer routing paths between SLRs impact the maximum frequency or results in tools not be able to route the design properly and meet our design goals. This scenario is considered in the design process of the previous chapter, and we can move on to the modifications of the previous implementation.

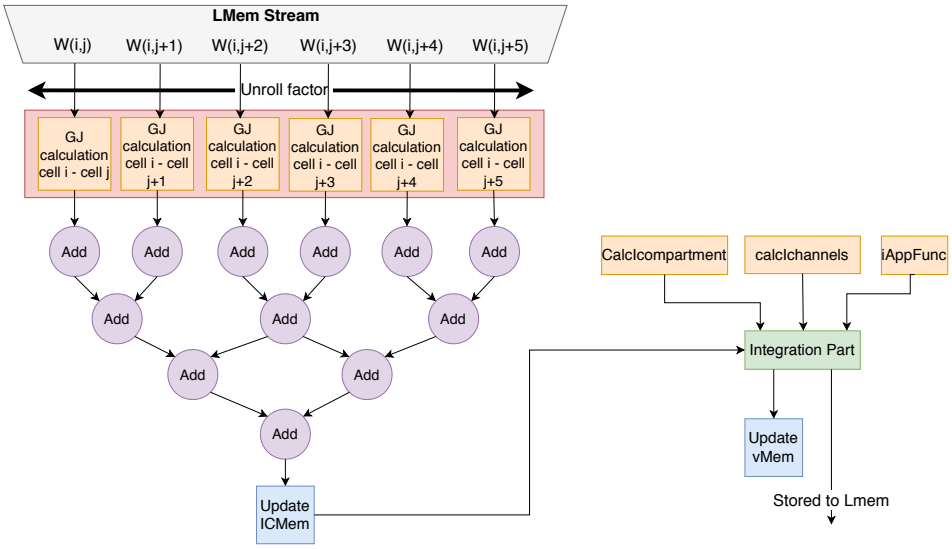


Figure 5.3: Gap junctions computation block

Table 5.2: Resource Usage for the HHio model of flexHH with an unroll factor of 24

Resource Type	***Used***	***Available***	***Percentage***
LUTs	*193,669*	*1,182,240*	*16.38*
Primary FFs	*318,469*	*2,364,480*	*13.46*
DSP Blocks	*1,388*	*6,840*	*20.29*
Block memory (BRAM18)	*1,875*	*4,320*	*43.40*
Block memory (URAM)	*234*	*960*	*24.37*

5.1 DFE Implementation and Improvements

In this section, the modifications done to the flexHH library are discussed. In this work, we are mainly focused on the models that include the gap junction extension since they have much room for performance improvement. The architecture for the models that

do not include the gap junction extension is not changed. Nevertheless, all models are updated to the latest platform and extended with a fixed-point version, and any modification done is mentioned.

5.1.1 From Custom Interface to Dynamic SLiC Interface

The first modification is the change of how the output is read from the host. Instead of having the data saved in LMem, they are directly sent using PCIe to the CPU host. This small change will free up LMem space and bandwidth since no transfer of the output to LMem between steps will be needed. This could not be achieved using the custom interfaces for memory reading and writing, and all the models are changed to use the dynamic SLiC interface. Dynamic SLiC interface is easier to program; it results in a neater manager and is more flexible. Previously, everything was calculated in the manager, such as LMem addresses for each set of parameters and the scalars inputs. With this change, everything is calculated in the host, and only the vital information is sent to the manager. Using this interface is substantially different from using custom interfaces, where the user had to initialize read and write interfaces for LMem. The dynamic SLiC interface allows more control over the engine allocation (DFE) and action manipulation using a set of actions.

The listings 5.1 and 5.2 illustrates an example of the dynamic SLiC interface actions. Actions are separated into two sets, initialization, and execution; this helps in debugging when an error occurs and in more structured code. In both sets of actions, the maxfile and engine ID needs to be set. The star placed in the parenthesis means the hostname is taken from the configuration variable "default_engine_resource"; more details can be found in Maxeler's SLiC API. The first set of actions shown in listing 5.1 sets the LMem streams content from the host using PCIe. The streams and the kernels that are not needed at the moment needs to be ignored. It is important to set the memory addresses correctly to avoid spending huge time debugging. Finally, the command to run the engine with the defined actions is used, and when it is finished, the set of actions needs to be deallocated.

```
1   max_file_t *maxfile = HHio2K_init();
2   max_engine_t *engine=max_load(maxfile,"*");
3   max_actions_t *act=max_actions_init(maxfile,NULL);
4
5   max_queue_input(act,"setChannelConst_CPU",channelConstants,
    sizeChannelConstants);
6   max_lmem_linear(act,"setupChannelConst",0,sizeChannelConstants);
7
8   max_ignore_lmem(act,"nChannels_Lmem");
9   max_ignore_kernel(act,"HHioKernel");
10
11  max_run(engine,act);
12  max_actions_free(act);
```

Listing 5.1: Dynamic SLiC interface for the initialization actions

The second set of actions shown in 5.2 sets the PCIe inputs, outputs, the number of ticks per kernel, the scalar inputs, and the LMem streams to the kernel. There are many

commands to connect the steams from LMem to the kernel, depending on what needs to be accomplished. In this case, the command **max_lmem_linear_advanced** is used to read the data from LMem since we access the data linearly. This command supports wrapping to the beginning of the array when the end of the data array is reached. The arguments are as follows; the name of the action, name of the stream, start address, the size of the array, the array's total size to be read, and the offset; all sizes are in bytes. The total size of the array, is the array's total size for all the steps of the simulation. So in each new step, this command wraps the given LMem stream at the beginning of the array, hence the offset's zero value.

```
1     max_actions_t *act1 = max_actions_init(maxfile,NULL);
2
3     max_set_ticks(act1, "HHioKernel",(nSteps * nTicksGapPerStep)+bufferSize
      );
4     max_set_uint64t(act1,"HHioKernel","nSteps",nSteps);
5
6     max_queue_input(act1,"setVin_CPUk1",vIn,sizeVIn);
7
8     max_lmem_linear_advanced(act1,"nChannels_Lmem",addressNChannels,
      sizeNChannels,nSteps * sizeNChannels,0);
9
10    max_ignore_lmem(act1,"setupNChannels");
11    max_queue_output(act1,"vOut",vOut,sizeVOut);
12
13    max_run(engine,act1);
14    max_actions_free(act1);
15    max_unload(engine);
```

Listing 5.2: Dynamic SLiC interface the execution actions

5.1.2 Multiple kernels application

Recall that, in the previous implementation, the gap junctions could only be unrolled up to 24 times when implemented in a single kernel. The idea is to separate the kernel into another two kernels to achieve a higher unroll factor and utilize our platform's other two SLRs. Recall that the previous chapter pointed out that we should implement the gap junction block into different kernels since when unrolled, the area utilization is increased. Dividing the design into more kernels will decrease the high routing paths and give the tools the chance to find an efficient placement and routing of our application. The current FPGA platform has three dies and gap junction computations are split into two kernels. This will result in three kernels, two kernels for the gap junction computations, and one for the rest of the computations. Figure 5.4 illustrates a detailed system architecture of the HHio model. Each Gap junction kernel will compute half of the cell's intercellular current while the main kernel will compute the gate current and the compartmental current. All the changes will be elaborated in the following sections.

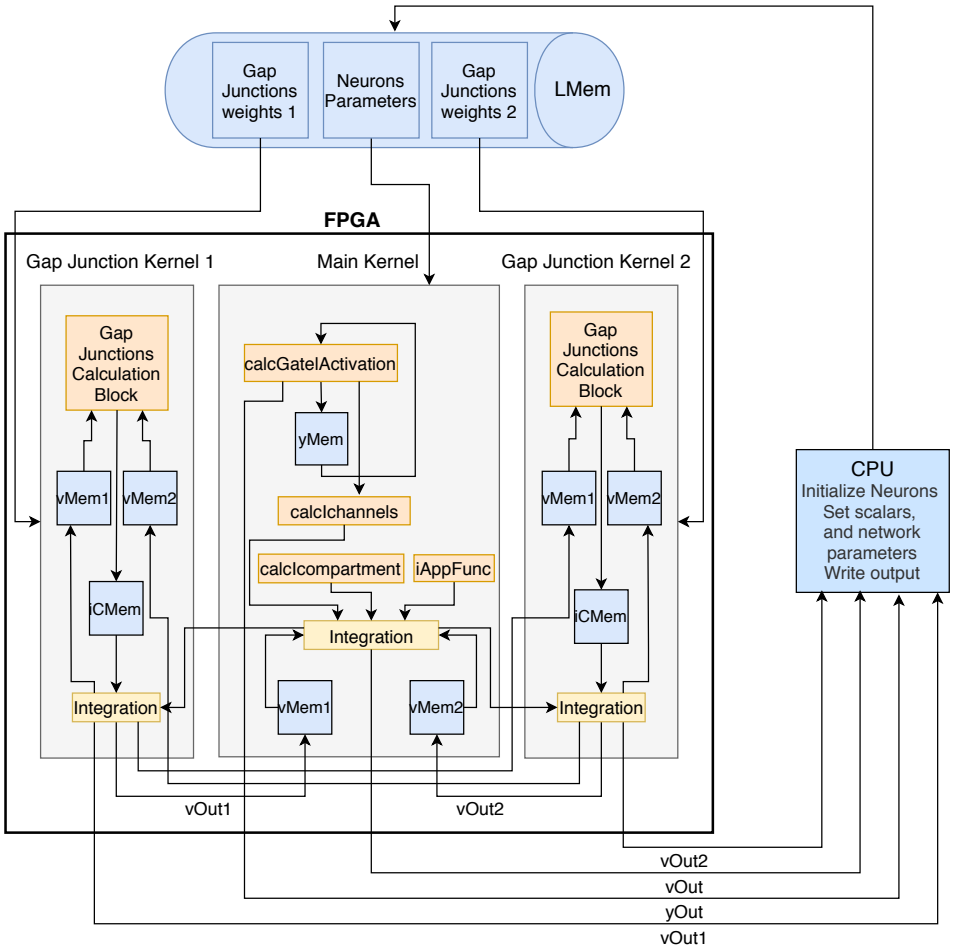


Figure 5.4: New system architecture of the HHio model

This, at first, seemed to be an easy task, but it proved to be the toughest one since multiple kernels are globally asynchronous. This means that a control logic to 'synchronize' all kernels are needed to achieve the expected behavior. At first, the integration part was placed inside the main kernel to limit data transfer between kernels. When the main kernel finish computing it waits for the gap junction kernels to transfer the intercellular current values to finish the update of the dendritic compartments/cell voltages. This means that the gap junction kernels need to be stalled till the main kernel produces the updated voltages since are also needed by these two kernels. We want the gap junction kernels to never stop computing and not have this stalling like execution.

To relax this strict dependency and optimize the application, the update of the voltage (integration part) is moved inside each gap junction kernel, as shown in Figure 5.4. This means that the updated voltages are now streamed out to the CPU host from each gap junction kernel and not the main kernel. This change triggered a change in the CPU code; the data needs to be rearranged in the correct order before being written into the output file since the output is coming from three kernels. Additionally, each gap junction kernel needs to transfer the updated values of the voltages (vOut) to the other kernels.

The gap junction kernels do not stall waiting for the main kernel to update the voltages. Instead, the main kernel waits for an update from the gap junction kernels which doesn't affect the computation time. The update between the gap junction kernels will not affect the computation (e.g. data not being available). Each gap junction kernel is set to start computing with the already stored data in each kernel and this gives time to transfer the values between them. To have a correct operation, the ticks for each gap junction kernel needs to be larger than the main kernel, as shown in Equation 5.1.

$$\frac{N_{cell}}{2} \cdot \frac{N_{cell}}{uf} > N_{totalChannels} \tag{5.1}$$

This decision entailed many changes to the design that needed to be considered before moving to the implementation. Table 5.3 shows the FMem usage from the previous version and what is changed for the current version when the application is split into three kernels. A brief explanation is given for each FMem instance and its usage.

- **iRestMem** memory stores the result of the summation of the currents I_{mc}, I_{gates}, I_{leak} and I_{app}.
- **gapAddressMem** memory stores the addresses for each compartment that connect to other cells through gap junctions.
- **vMem** memory stores the membrane voltages for each compartment.
- **ICMem** memory stores the intermediate results of each inter-cellular current for each compartment or cell (single compartment).

The FMem usage is increased by 28.93%, which is calculated using the current max compartment size set for the kernel, which is 24576, as shown in Table 5.3. Additionally, in Table 5.3, each FMem's memory subscript denotes the kernel in which the FMem is instantiated (GJ = Gap Junction, MK = Main Kernel). Using the performance model, this means the need for an extra 50 BRAMs, which is not a prohibitive number of BRAMs. To calculate the compartment/cell voltage derivative inside gap junction kernels, the elastance and the summation of the rest of the currents are needed. These values are sent to both gap junction kernels from the main kernel. ICMem memory is halved in each gap junction kernel since each kernel computes half of the network's cells. Additionally, extra memory is needed to store the membrane voltages in each gap junction kernel. As shown in Figure 5.4, the vMem used to store the derivative voltages for all the kernels needs to be split in half. This is a result of the restrictions on writing into memories by the tools:

- If you write to a memory address in a kernel tick, you can not call read with the same address in the same tick. Attempting to do so will return undefined data.
- You are limited to either a maximum of 2 calls to write and no calls to read on memory or 1 call to write and any number of calls to read.

The second restriction applies in our case. Once the gap junction kernels send their output to the main kernel, the data are written to the same memory, and the main

kernel needs to read the data from that memory. This leads to instantiating two write ports (1 per kernel) and one read port, which is not supported by the tools. The solution to this was to split the memory into two equal parts, and each gap junction kernel will write to the respective memory. This resulted in extra control logic added to the main kernel to write and read to/from the correct memory depending on the compartment/cell number (the first memory stores the first half cells data and the second the last). The membrane voltage memory in each gap junction kernel is also split for the same reason.

Table 5.3: FMem sizes for the HHio model for flexHH library and the new implementation

flexHH Implementation		*New Implementation*	
FMem	*Size (bytes)*	*FMem*	*Size (bytes)*
iRestMem	$N_{compsMax} \cdot Size_{var}$	$vMem1_{GJ1,GJ2,MK}$	$\frac{N_{compsMax}}{2} \cdot Size_{var}$
ICMem	$N_{compsMax} \cdot Size_{var}$	$vMem2_{GJ1,GJ2,MK}$	$\frac{N_{compsMax}}{2} \cdot Size_{var}$
gapAddressMem	$N_{compsMax} \cdot \log_2 N_{compsMax}$	$gapAddressMem_{MK}$	$N_{compsMax} \cdot \log_2(N_{compsMax})$
vMem	$N_{compsMax} \cdot Size_{var}$	$ICMem_{GJ1,GJ2}$	$\frac{N_{compsMax}}{2} \cdot Size_{var}$
Total:	*339717*	*Total:*	*438021*

The most difficult part during these modifications was to synchronize when each kernel is accepting valid data from the other kernels. Control inputs are needed in each kernel since not all the transmitted data are valid. Additionally, if the output is control (to transmit only valid data) this means no data will be available to the other end, and the kernels will stall. The first solution was to use non-blocking inputs shown in Listing 5.3. This command works the same as the normal input command; however, it generates dummy values if no data are available to the kernels. Additionally, each input includes a valid signal to distinguish the valid data from the dummy data. This approach seemed logical at first but in the end it didn't work when the application was tested on FPGAs. Non-blocking inputs are often used for networking applications that kernels do not run for a precise number of ticks which this do not applied in our case.

```
1 NonBlockingInput<DFEVar> Temp = io.nonBlockingInput("temp1",fpType);
```

Listing 5.3: Nonblocking Input

To solve this a better understanding of how kernels are implemented at the manager level was needed. Between kernels, FIFO buffers are placed and each kernel pulls data when the control signal is set to true. If data are not available a kernel will stall till data becomes available again. Contrarily, if there is no space available in the FIFO buffer, to push values, the kernel will stall again. In this stage, you need to be wary of deadlocks that exist if insufficient data are being produced or consumed. With this information, a communication diagram is created to illustrate how kernels interact with each other, LMem, and the CPU host. As shown in Figure 5.5 during the first N_{cells} ticks the gap junctions kernels initialize the FMem with values sent from CPU and the main kernel start computing. During the fist $N_{cells} \cdot N_{channels}$ ticks the main kernel pushes the (I, S) values into the FIFO buffers which correspond to the elastance and the summation of the currents (excluding intercellular current) and then the kernel stalls.

These values are then needed after the $\frac{N_{cells}}{2} \cdot (\frac{N_{cells}}{uf} - 1)$th tick so these values are push into the FIFO before they actually needed and stored there till are needed. After the $\frac{N_{cells}}{2} \cdot (\frac{N_{cells}}{uf} - 1)$th tick both gap junction kernels start producing output values using the store (I,S) values. The main kernel is stalled on input coming from both these kernels and when values become available on these buffers the kernel restarts its computations. These inputs are forced to be scheduled together at the same position in the pipeline to arrive in sync. Additionally, the manager FIFOs buffers depth is increased to $\frac{N_{cells}}{2}$ to hold all the transferred values till are needed.

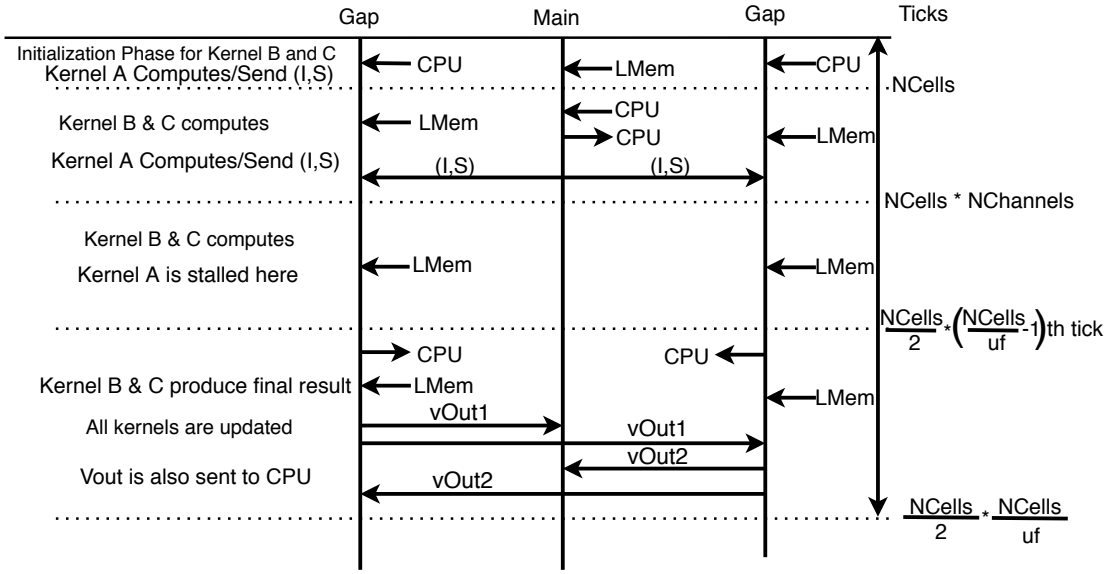


Figure 5.5: Communication Diagram

The membrane voltages streamed from each gap junction kernel to the main kernel are first streamed, stored, and then the main kernel is moved to the next step. Figure 5.7 illustrates the control logic implemented inside the main kernel to store the membrane voltages produced either from the main kernel or a gap junction kernel. When the main kernel resumes its execution and is not the first step the data are valid, the respective counter (GJ1 and GJ2) is increased by one till it reached the maximum number of $\frac{N_{cells}}{2}$. Multiplexers are used to decide the enable signal, the data, and the addresses to vMem. The input to gapAddressMem is the counter output, which translates each dendrite compartment to the respective address in vMem, as shown in figure 5.6.

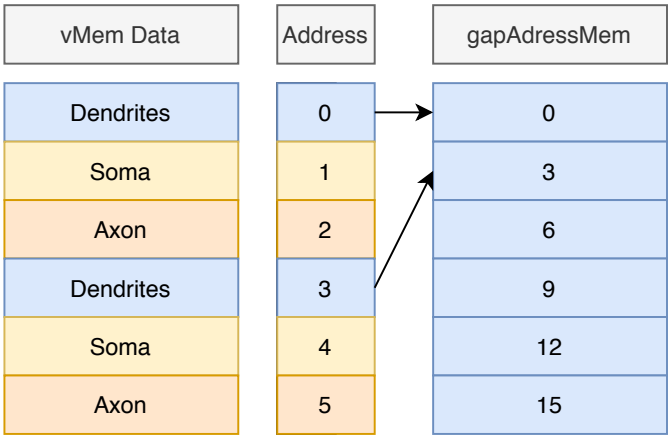


Figure 5.6: Membrane voltage and gapAddressMem structure data

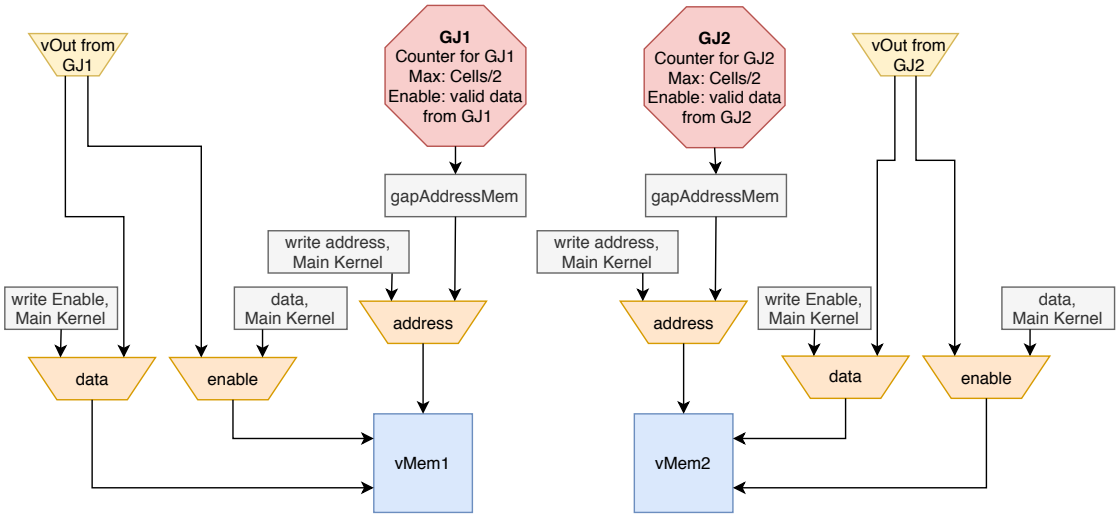


Figure 5.7: Main Kernel: The control logic used to store the membrane voltage in the Main Kernel

5.1.3 FMem Optimization

Moreover, in the flexHH library for both the HHio and HHgap models, the membrane voltage for the other compartment/cell needs to be read to calculate their voltage difference during the intercellular current computations. An example is illustrated in Figure 5.8, vOwn is the voltage of the respective compartment/cell, and vOther is the voltage of the other compartment/cell that needs to be read. In this small example, we have a network of 6 cells and an unroll factor of three. This means that three FMem ports are needed per gap junction kernel in 5.8 (a). Unrolling this loop leads to an increase of the read ports, and this means increasing the use of BRAMs and possibly other resources (FFs and LUTs). To optimize this, instead of having multiple read ports, we read a vector with the size of the unroll factor. The FMem structure is changed from $\frac{nCellMax}{2} \cdot Size$ to $\frac{nCellMax}{unrollfactor \cdot 2} \cdot unrollfactor \cdot Size$ as shown in 5.8 (b), the Size

parameter refers to the size of each gap junction weight in bits. The structure of the FMem needs to be changed to read unroll factor elements from memory per tick. An additional serial to parallel shift register was needed for this optimization. The update values of the membrane voltages are serially produced in each tick, and to be stored into the new FMem structure, they need to be stored as a vector. The shift register waits for unroll factor membrane voltages to shift into the register and then are stored in FMem. This optimization will decrease the BRAM usage by a lot considering Equation 4.3 considering that it scales with the number of ports.

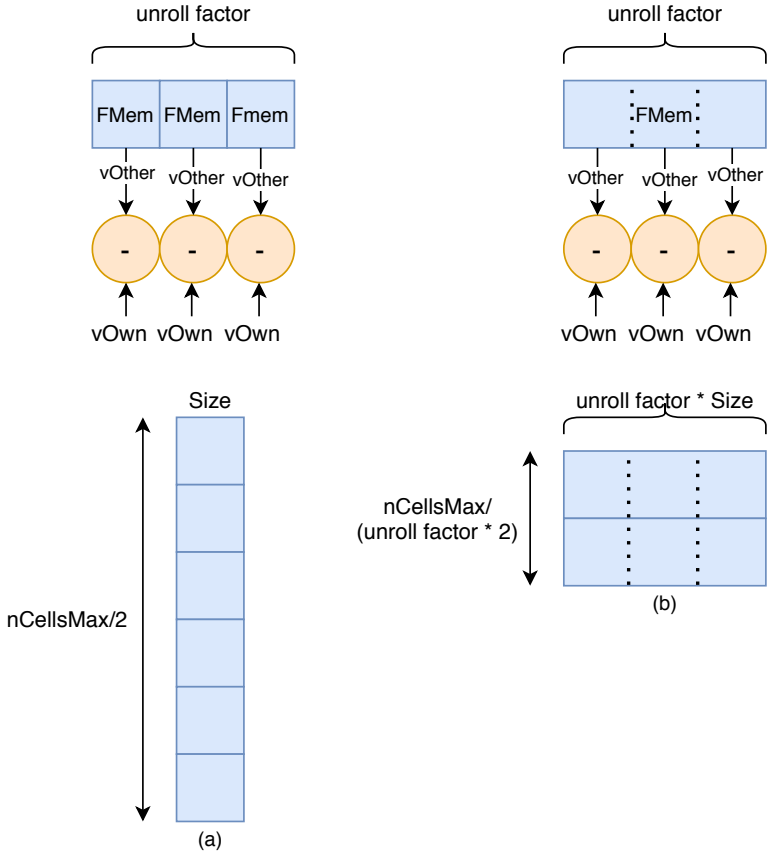


Figure 5.8: FMem optimization: (a) FMem with multiple ports (b) Read FMem values as a vector

5.1.4 Loop Length

Since all kernels are pipelined, the programmer needs to be wary of each kernel's pipeline loop length. The loop length in both HHio and HHgap models may be larger than the number of cells. This has a result of the production of erroneous results if the number of cells simulated is smaller than the loop length. This occurs because data exit the pipeline later on time, and membrane voltages are not updated correctly. This means that the next step's computations will use older data since they are not updated in time and produce wrong results. To avoid this error, a limitation should be set, the number of cells needs to be at least $\frac{N_{cells}}{2} > LoopLength$ since the number of cells is divided

into two kernels. The application has a loop length of 189 for floating-point and 105 for fixed-point for each gap junction kernel, which means that a minimum number of 378 or 210 cells are needed. Furthermore, this problem also exists during the storage of membrane voltages in the main kernel streamed by the gap junction kernels. The main kernel waits for loop length ticks after the first value is streamed into the main kernel. This does not affect the computation time since the main kernel finishes its operation earlier than the gap junction kernels.

5.1.5 Kernel's operation

To recap, each kernel's operation is explained below:

- **Main Kernel:** Calculates the gate activation variables, each channel's current, and the membrane voltage for the compartment that does not connect with gap junctions to other cells. The summation of all currents besides the intercellular current is sent to each gap junction kernel (the first $\frac{N_{cell}}{2}$ is streamed to GJ1 and the last $\frac{N_{cell}}{2}$ is sent to GJ2).

- **Gap Junction Kernel 1:** Calculates the intercellular current for the first $\frac{N_{cell}}{2}$ cells, added to the current sent from the main kernel, and the membrane voltages of cells/compartments are updated. The updated membrane voltages are then streamed to both Gap Junction Kernel 2 and the Main Kernel.

- **Gap Junction Kernel 2:** Calculates the intercellular current for the last $\frac{N_{cell}}{2}$ cells, added to the current sent from the main kernel, and the membrane voltages of cells/compartments are updated. The membrane voltages are then streamed to both Gap Junction Kernel 1 and the Main Kernel.

The above modification change the computation time from $N_{cell} \cdot \frac{N_{cell}}{unrollfactor}$ to $\frac{N_{cell}}{2} * \frac{N_{cell}}{unrollfactor}$ per kernel.

5.1.6 Fixed-Point Arithmetic

To incorporate fixed-point arithmetic in an application can be done by providing the preferred data type for the operation and MaxCompiler handles the rest. The tricky part is how to provide these fixed-point numbers to the kernel since the host code, which is written in C, does not support fixed-point arithmetic. One approach is to stream all the floating-point values and cast it inside the kernel to fixed-point. This will not improve bandwidth utilization, and it will result in area and performance overhead since all computations are going to require a casting function. Another approach is to use a third-party fixed-point arithmetic library, which is limited only to support the standard sizes of 16 and 32 bits, which is not what is needed. A third option is to create an extra kernel that accepts all the streams before are streamed to LMem and cast them into the required fixed-point size. The third option has the benefits of casting the values to any

fixed-point size, incurs less area, and performance overhead since they are done once at the beginning of each simulation. The number of castings depends on the number of parameters. Additionally, it compresses the data, which helps in saving bandwidth and storage in LMem.

The task of creating a new kernel to cast the parameters to fixed-point was made easier to implement since our models were converted from a custom manager interface to a dynamic SLiC interface. This helped control each stream of data to each kernel without any problem and without instantiating complicated custom interfaces. The following tables illustrate the parameters included in each structure and the sizes of both floating-point and fixed-point. Tables 5.4 and 5.5 illustrates the structures for both HH and HH+gap models, the floating-point version requires a size of 384-bits for the gate structure and 192-bits for the cell structure. The fixed-point structures require 256-bits for the gate structure and 160-bits for the cell structure. This conversion leads to a compression of 33% for the gate structure and 16% compression for the cell structure, an important reduction when a huge network is going to be employed for both bandwidth and LMem memory. The values for each variable are chosen based on the previous analysis to achieve the desired accuracy. Furthermore, many of the variables such as aFType, bFtype, and p only require 4-bits given the application's current functionality.

Table 5.4: Gate structure variables for the HH and HH+gap models

	Floating-point	***Fixed-point***
Variable	***Type***	***Type***
aFType	*32-bit unsigned integer*	*4-bit unsigned fixed-point (4/0)*
aXs	*3 32-bit single-precision-floating-point*	*3 28-bit signed fixed-point (13/15)*
bFType	*32-bit unsigned integer*	*4-bit unsigned fixed-point (4/0)*
bXs	*3 32-bit single-precision-floating-point*	*3 28-bit signed fixed-point (13/15)*
p	*32-bit single-precision-floating-point*	*4-bit unsigned fixed-point (4/0)*
g	*32-bit single-precision-floating-point*	*28-bit signed fixed-point (13/15)*
vChannel	*32-bit single-precision-floating-point*	*28-bit signed fixed-point (13/15)*
yInit	*32-bit single-precision-floating-point*	*20-bit signed fixed-point (5/15)*

Table 5.5: Cell structures variables for the HH and HHgap models

	Floating-point	***Fixed-point***
Variable	***Type***	***Type***
iAppStart	*32-bit unsigned integer*	*24-bit unsigned fixed-point (24/0)*
iAppEnd	*32-bit unsigned integer*	*24-bit unsigned fixed-point (24/0)*
iAppAmplitude	*32-bit single-precision-floating-point*	*28-bit signed fixed-point (13/15)*
S	*32-bit single-precision-floating-point*	*28-bit signed fixed-point (13/15)*
vLeak	*32-bit single-precision-floating-point*	*28-bit signed fixed-point (13/15)*
gLeak	*32-bit single-precision-floating-point*	*28-bit signed fixed-point (13/15)*

Tables 5.6 and 5.7 illustrates the structures for HH+custom, HH+custom+multi and HHio models. The floating-point implementation needs 768-bits for the gate structure and 256-bits for the compartment structure. The fixed-point structure requires 572-bits for the gate structure and 192-bits for compartment/cell structure. Utilizing fixed-

point achieves compression of 25.5% for the gate structure and 25% compression for compartment/cell structure.

Table 5.6: Gate structure variables for the HH+custom, HH+custom+multi and HHio

	Floating-point	***Fixed-point***
Variable	***Type***	***Type***
aFType	*32-bit unsigned integer*	*4-bit unsigned fixed-point (4/0)*
aXs	*9 32-bit single-precision-floating-point*	*9 28-bit signed fixed-point (13/15)*
bFType	*32-bit unsigned integer*	*4-bit unsigned fixed-point (4/0)*
bXs	*9 32-bit single-precision-floating-point*	*9 28-bit signed fixed-point (13/15)*
p	*32-bit single-precision-floating-point*	*4-bit unsigned fixed-point (4/0)*
g	*32-bit single-precision-floating-point*	*28-bit signed fixed-point (13/15)*
vChannel	*32-bit single-precision-floating-point*	*28-bit signed fixed-point (13/15)*
yInit	*32-bit single-precision-floating-point*	-

Table 5.7: Compartment structure variables for the HH+custom, HH+custom+multi and HHio models

	Floating-point	***Fixed-point***
Variable	***Type***	***Type***
iAppStart	*32-bit unsigned integer*	*24-bit unsigned fixed-point (24/0)*
iAppEnd	*32-bit unsigned integer*	*24-bit unsigned fixed-point (24/0)*
iAppAmplitude	*32-bit single-precision-floating-point*	*28-bit signed fixed-point (13/15)*
S	*32-bit single-precision-floating-point*	*28-bit signed fixed-point (13/15)*
vLeak	*32-bit single-precision-floating-point*	*28-bit signed fixed-point (13/15)*
gLeak	*32-bit single-precision-floating-point*	*28-bit signed fixed-point (13/15)*
nCa	*32-bit unsigned integer*	*4-bit unsigned fixed-point (4/0)*
vInit or gp	*32-bit single-precision-floating-point*	*28-bit signed fixed-point (13/15)*

5.1.7 Placement & Route

Placement & route took considerable time in this work, and a successful build could not be achieved due to timing errors, and information regarding this will follow. To solve these timing-related errors, the produced timing reports proved useful. Most of the errors were setup errors, and they were related to the memory controller. This is because when a single memory controller is used for all the kernels placed in different SLRs, there will be SLR crossings that affect timing. A solution to this is to break the memory controller into multiple controllers. Memory controllers are asynchronous and thus did not provide any problem in the case that gap junctions are supported as each kernel is programmed to be fully asynchronous. On the other hand, in the models without the gap junctions, the unrolled streams related to the gates could not be broken to multiple controllers since we need data to come synchronously. This left us with the choice of splitting the streams between memory controllers, as mentioned below.

A memory controller for each kernel is used for the models with the gap junction extension and are described below:

- **Memory Controller 1:** Includes all the input streams connected to the main kernel and one DIMM of memory is used for this controller.
- **Memory Controller 2:** Includes intercellular connection weights for Gap Junction Kernel 1 and one DIMM of memory is used for this controller.
- **Memory Controller 3:** Includes the input stream that transfers the inter-cellular connection weights for the gap junction kernels one DIMM of memory is used for this controller.

Two memory controllers for the models without the gap junction extension are instantiated and are described below:

- **Memory Controller 1:** Includes the gates parameter streams and two DIMMs are used with this controller.
- **Memory Controller 2:** Includes the cell constants and any other required streams and one DIMM is used for this controller.

The change from a single memory controller to multiple led to a smaller burst size (less DIMMs per controller) and different total data width since the data structures are split across different memory controllers. This means that now data are not multiple to the burst size, and this produces an error. Additionally, in the case of the fixed-point version of our application, padding now is not an option to solve the problem (e.g., padding the gate structure from 578-bits to 1024-bits to be multiple to burst will occur loss of bandwidth and memory). By using **setAllowNonMultipleTransitions(true)** the compiler will automatically insert an aspect change. This aspect change resolves this issue by finding the least common multiple between the data width and the physical port and build a FIFO of that size. For example, in a structure with 572-bits and a single memory controller, which has a burst of 512-bits, the least common multiple is 146432-bits. This has as a disadvantage, the high hardware usage of these wide FIFOs and if not, enough data are written to that large FIFO, the application will stall. To mitigate this problem, AspectChangeIO (ACIO) from the MaxPower library that is provided by Maxeler is used. ACIO can pack or unpack arbitrary kernel types into or from frames of higher width, and data are automatically split across multiple frames and padded. More information can be found in MaxPower's library API. Again padding will be used here but since it may break the data in multiple frames this may result in less padding.

Each kernel needs to exchange values between the other two kernels, and SLR crossings will be present. It is recommended to add extra registers to those links to relax timing constraints. At least two register levels are needed per SLR crossing; this means that each signal is registered before and after each crossing. In our application, in each input and output that connects to other kernels, two registers are added using the command shown in Listing 5.4. Moreover, the memory controller commands and data are pipelined to help meet timing closure. The set of commands are shown in Listing 5.5.

```
1  DFELink setupWs2ToLMem = LmemInterfaceW2.addStreamToLMem("setupWs2",
   LMemCommandGroup.MemoryAccessPattern.LINEAR_1D);
2  setupWs2ToLMem.setAdditionalRegisters(2);
```

Listing 5.4: Command to add registers in a link before and after SLR crossing

```
1  LMemConfig lmemconfig = makeLMemConfig();
2  lmemconfig.setMcpCommandPipeliningDepth(5);
3  lmemconfig.setMcpDataPipeliningDepth(5);
```

Listing 5.5: Addition of more pipelining stages in LMem

The Xilinx Vivado design suite implementation process includes logical and physical transformations of the design called implementation strategies. An implementation strategy is a defined approach for resolving the design's synthesis or implementation challenges targeting a specific design goal (power optimization, timing optimization, area optimization). Using the previous unsuccessful placement & route runs the most promising implementation strategy was **PERFORMANCE_BALANCE_SLLS**, that tries different variations of placements for SSI devices with more aggressive crossings of SLR boundaries. The chosen implementation strategies are based on which ones had the smaller timing score (when it fails due to timing, it mentions a timing score) and strategies with the same directives. Multiple implementation strategies can be used in a single run and are executed in different threads, as shown in Listing 5.6.

```
1 buildConfig.setImplementationStrategies(ImplementationStrategy.
    PERFORMANCE_NET_DELAY_HIGH, ImplementationStrategy.
    PERFORMANCE_BALANCE_SLLS, ImplementationStrategy.
    PERFORMANCE_SPREAD_SLLS);
```

Listing 5.6: Implementation strategies commands

All these changes successfully helped in having a successful placement & route of the models. To find the model's maximum operating frequency, the frequency of 200MHz used in the performance model is used as a starting frequency. Intervals of 15MHz are used to find if the frequency can be pushed further.

5.1.8 Implementation Results

Before our methodology is evaluated in the next chapter, general comments about the implementation are discussed. An unroll factor of 172 for the fixed-point version and 192 for the floating-point is achieved. A frequency passed the 200MHz could not be achieved. Vectorizing the FMem memory in the gap junction kernels led to tremendous savings in hardware resources. The floating-point version could not achieve an unroll factor larger than 128 because LUTs and Flip-Flops hardware resources were depleted. This FMem optimization achieved a reduction of 33.87% in LUTs, 17.67% in Flip-Flops, 39,47% in BRAMs, and 43.75% in URAMs. This helped the floating-point version to reach a higher unroll factor and decrease the place and route process time due to this high hardware resource usage. Many important lessons are learned from the implementation section;

firstly, analyzing before reaching this point is crucial since many decisions made during in the design process help in overcoming these implementation challenges. Moreover, the tools help in discovering what is going wrong and where. The timing report helped pinpoint where the problems are and solve them.

5.2 Summary

In this chapter, the steps taken to optimize the previous implementation of the flexHH library with focus on the the models that supports the gap junctions are explained. Detailed information and the reason behind each change is discussed, and crucial information regarding the needed commands are also stated. In the implementation phase, the performance model is kept as the guide and each neglected modification is checked with the performance model and then is implemented. Having a detailed performance model with many estimated scenarios, as is the case with the memory controllers, helps make decisions that are already analyzed. Finally, during each change, the output is verified using the C model, and this helped to reduce the time used in debugging.

6 Evaluation

In this chapter, the new implementation of both HHio and HH+gap is validated using the C model in section 6.1. In section 6.2, the performance model's evaluation is presented only for the HHio model since both share the same findings. This section includes the hardware resource usage, and the execution time evaluation. The error evaluation between floating-point and fixed-point implementations is presented in section 6.3.

6.1 DFE validation

Many changes are done in the previous implementation regarding the models that supports the gap junctions extension, which are split into multiple kernels. To make sure that everything works as expected, the output of the DFE implementation is validated using the C model. Having a C model helped in evaluating the output of the DFE for larger networks and simulation steps since checking that on MaxCompiler simulation is not feasible. Figure 6.1 illustrates the error of the compartmental voltages between CPU and the DFE for a network of 480 cells simulated for 30,000 steps ($dt = 0.01$) for the HHio model. The x-axis represents the compartments of the network, the y-axis the time step, and the color corresponds to the error. Using this colormap graph helps us to observe the behavior of all the network and pinpoint outstanding error. The parameters used to validate all the models in this chapter are shown in Appendix A and are taken from [38]. The average error is 0.16% and as shown in Figure 6.2 the output of the axon voltage of a single neuron is not influenced by the error.

The error of the HH+gap model is illustrated in Figure 6.3 having an average error of 0.94%. Between $0.5 - 1 \cdot 10^4$ steps, we notice a bit higher error for all the neurons but as shown in Figure 6.2 the output of a single neuron is not influenced. In both models, this error must be related to rounding errors.

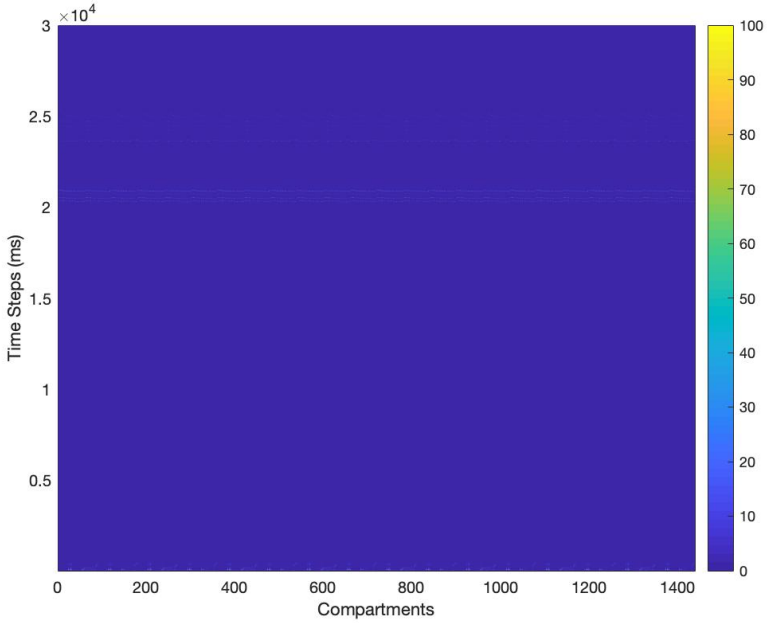


Figure 6.1: Error between floating-point and C model for all the compartments of the HHio model

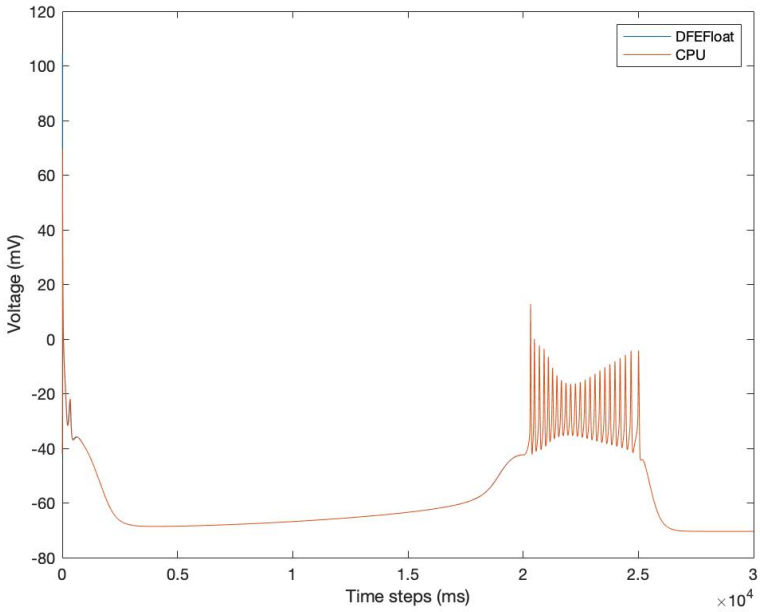


Figure 6.2: Output of the voltage of a single compartment (axon) of both DFE and C code of Neuron 8 of the HHio model

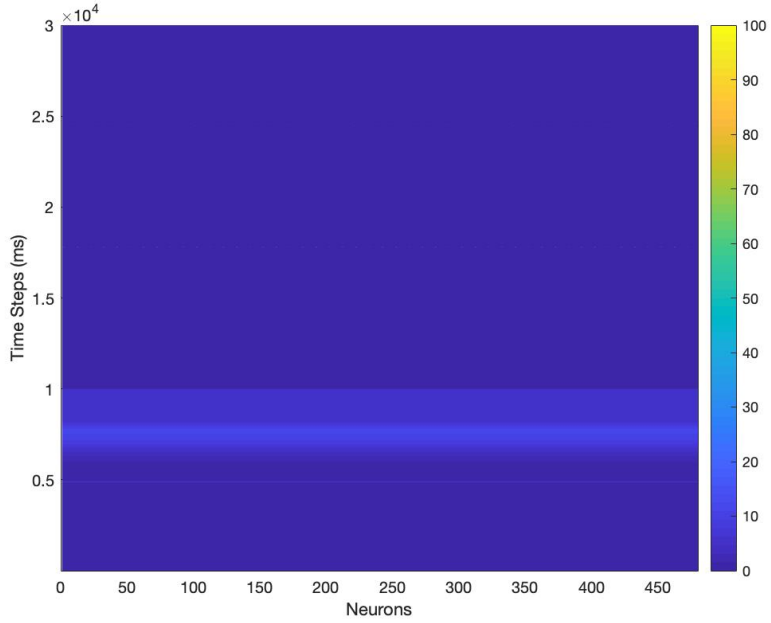


Figure 6.3: Error between floating-point and C model of the HH+gap model

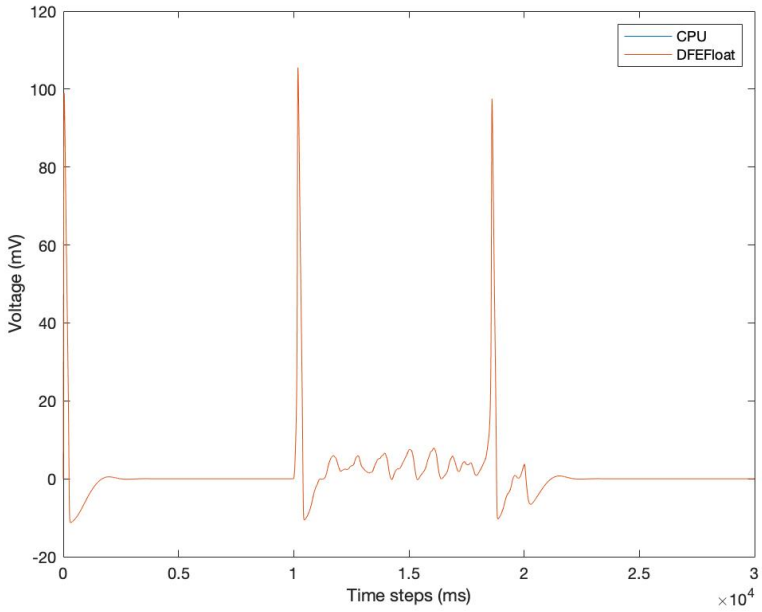


Figure 6.4: Output of the voltage of a single neuron of both DFE and C code of neuron 0 of the HH+gap model

6.2 Performance model Evaluation

6.2.1 Hardware resource usage

In this section, the estimated hardware usage is compared to the actual hardware usage using the final results from the place and route process for the HHio model.

Tables 6.1 and 6.2 illustrates the actual hardware resource usage and the estimated hardware resource usage for both floating-point and fixed-point for the maximum achieved unroll factors. As shown, the predicted resource usage is close to the actual one considering that we took the worst-case scenario during the area estimation. The error between estimations and the actual hardware usage is less than 10% for the floating-point and less than 20% for the fixed-point version of the application. The extra resources used are for the memory controllers, the pipeline registers, manager address state machines, and aspect ratio registers. We only included the memory controllers in our predictions, but it seems the outcome is the one expected, a close approximation of the actual resource usage. Additionally, it is important to consider additional BRAMs required by the manager and this is taken from previous builds. The same outcome is also seen in the fixed-point version of the application. The purpose is to avoid an infeasible solution or an application that exceeds the FPGA's platform available resources. Additionally, it crucial to know the limiting resource type of your application usually being the on-chip memory or the DSPs.

Table 6.1: Final and estimated hardware resource usage results of HHio model using floating-point arithmetic

	Resource Usage			
Resource	***Used***	***Estimated***	***Available***	***Deviation(%)***
LUTs	*707,036*	*738,499*	*1,182,240*	*4.44*
Primary FFs	*1,142,763*	*1,247,726*	*2,364,480*	*9.18*
DSP Blocks	*5,004*	*5,320*	*6,840*	*6.31*
Block memory (BRAM18)	*1,690*	*1,812*	*4,320*	*7.21*
Block memory (URAM)	*198*	-	*960*	-

Table 6.2: Final and estimated hardware resource usage results of HHio model using fixed-point arithmetic

	Resource Usage			
Resource	***Used***	***Estimated***	***Available***	***Deviation(%)***
LUTs	*504,588*	*508,615*	*1,182,240*	*0.79*
Primary FFs	*872,602*	*1,027,215*	*2,364,480*	*17.71*
DSP Blocks	*5,408*	*5,295*	*6,840*	*-2.08*
Block memory (BRAM18)	*1,574*	*1,792*	*4,320*	*13.85*
Block memory (URAM)	*161*	-	*960*	-

6.2.2 Performance

The performance of the HHio model is depicted in Figure 6.5. The y-axis represents the execution time on DFE in seconds and the x-axis the different unroll factors. The total execution time is better than what is estimated for both versions. This is most likely due to the DDR_{eff} included in the performance model, it seems that the memory efficiency

is not accurately captured in smaller unroll factor. This is because the size of data streamed are not multiple to the burst size and a more pessimistic efficiency factor is used. Additionally, the execution time without the efficiency factor is also illustrated. As we can see the floating-point version execution time after an unroll factor of 16 (per kernel) does not see any improvement and this is due to the fact that our application becomes memory bound. On the other hand, we notice that fixed-point become memory bound after an unroll factor of 32 (per kernel). This outcome is as predicted in the performance model. A mixed version is also implemented that utilizes fixed-point arithmetic to stream the gap junction weights and it has the same performance as fixed-point. Finally, the execution time when the memory bound is reached is 30% and 43% higher for fixed-point and floating-point compare to the execution time excluding the DDR_{eff}. This means that the actual execution time lies between the estimated values and the execution time without including the DDR_{eff} factor. This helps in creating margins of the actual execution time. The performance for the HH+gap model is identical to the HHio model except that the cells and gate structures streamed to it have a smaller size, but since the performance is limited by the gap junction it does not affect the execution time.

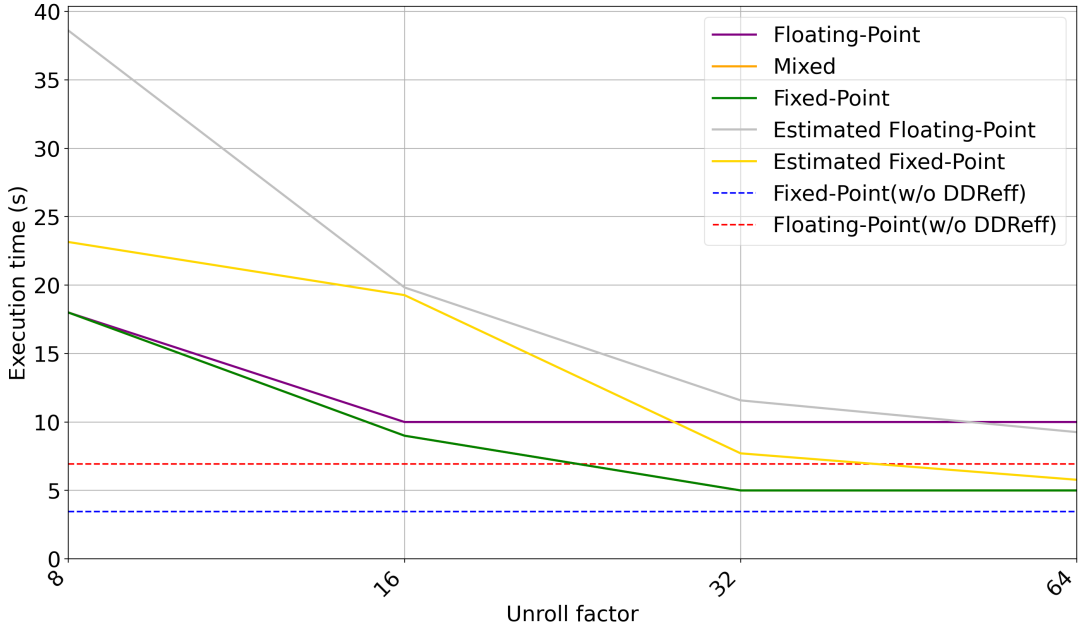


Figure 6.5: Execution time for both floating-point and fixed-point of the HHio model

6.2.3 Performance on models without gap junctions connections

All models becomes memory bound as is also discussed in [38], in this work due to time restrictions all models are place and route setting the unroll factor to four. The fixed-point version of all the models experienced a performance speed up of 1.2x compared to floating-point. This is related to the bandwidth savings provided by fixed-point arithmetic. Loss of performance exists in all models that the gate computations are unrolled since in order to meet timing, two memory controllers are used. Using, a single controller for the cell/compartment structures will mean a loss in bandwidth which could be used

for the gate structures streams (unrolled streams). How can this be improved is further explained in the future work section.

6.3 Floating-point and Fixed-point Error Evaluation

To quantify the percentage of the error and the average error between floating-point and fixed-point on DFE the Equations 4.1, 4.2 and 4.3 from Chapter 4 are used. The simulation parameters used for each model are mentioned in Appendix A and are taken from [38]. In this section, all the models are evaluated regarding the error between floating-point and fixed-point.

6.3.1 HH

In Figure 6.6 the error for the HH model is illustrated. As shown there are two distinct areas where the percentage error is over 100% while there are areas where the error is below 10%. The average error is 608%, and this huge average error is related to the huge percentage error of these two yellow regions shown in Figure 6.6. By observing the output of a single neuron as shown in Figure 6.7 the error is related to values that are close to zero. The behavior of both outputs is close but an error exists before and after the external current is applied to the model ($1 \cdot 10^4 - 2 \cdot 10^4$ steps). By analyzing the membrane voltage potential values it seems that in floating-point, in these yellow regions the membrane voltages potential derivative values are really small numbers. This results, in fixed-point not be able to represent the voltage derivative values and the membrane voltage potentials do not change over the time steps and hence the error. On the other hand, considering just the output behavior, fixed-point captures the behavior of the model.

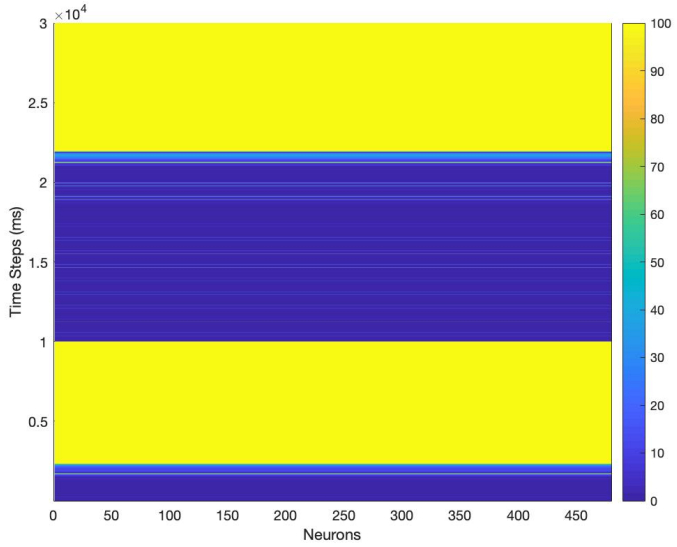


Figure 6.6: Error between floating-point and fixed-point version of HH model

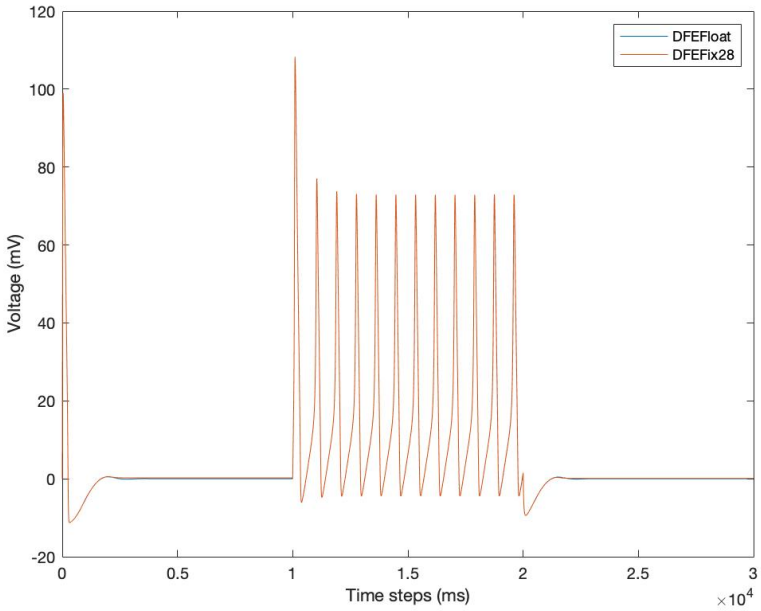


Figure 6.7: Voltage output of the axon compartment of neuron 0

6.3.2 HH+gap

The same behavior as the HH model is also seen in the HH+gap model as shown in Figure 6.9 with an average error of 914%. Again this error is related to the yellow regions before and after the injected current as in the previous model. Extending the HH model with gap junction extension does not lead in changing the output behavior as shown in Figure 6.8. Finally, considering just the output behavior, fixed-point captures the behavior of the the floating-point.

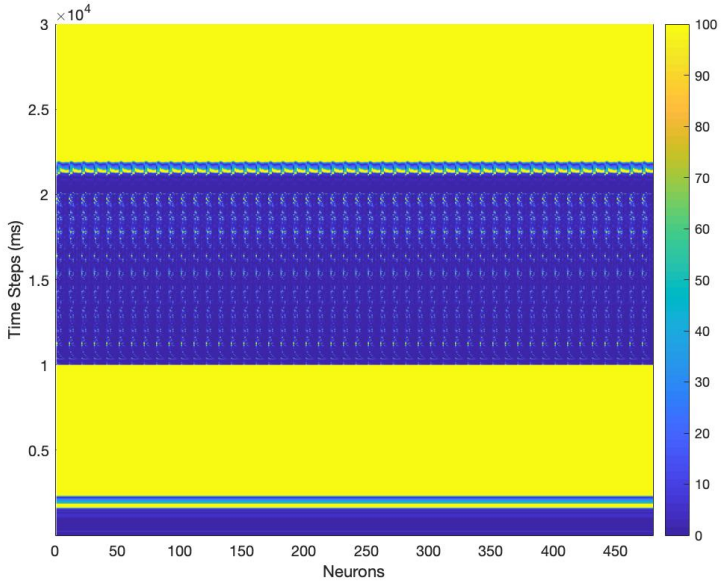


Figure 6.8: Error between floating-point and fixed-point version for all the neurons of HH+gap model

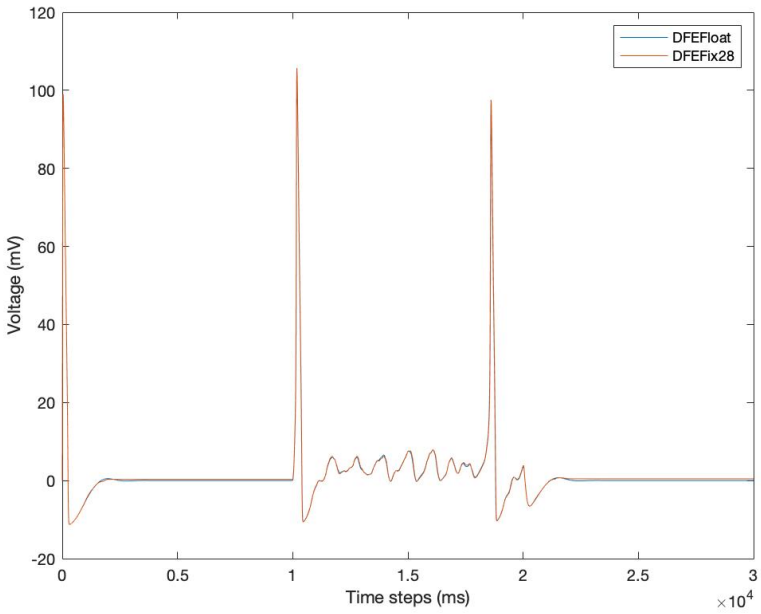


Figure 6.9: Voltage output of the axon compartment for neuron 0

6.3.3 HH+custom

The percentage error of the model that supports the custom gates extension is depicted in Figure 6.10 with an average error of 38.06%. In Figure 6.11 we can observe this error is related to a phase difference between both data types and a peak from fixed-point

once the external current is applied (at $1 \cdot 10^4$th step). Besides the error in phase, we see that fixed-point behavior is close to the floating-point.

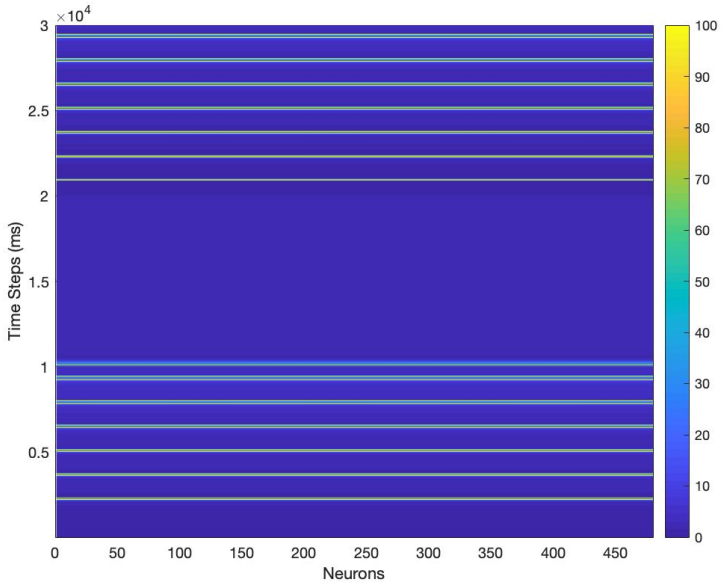


Figure 6.10: Error between floating-point and fixed-point version of HH+custom model

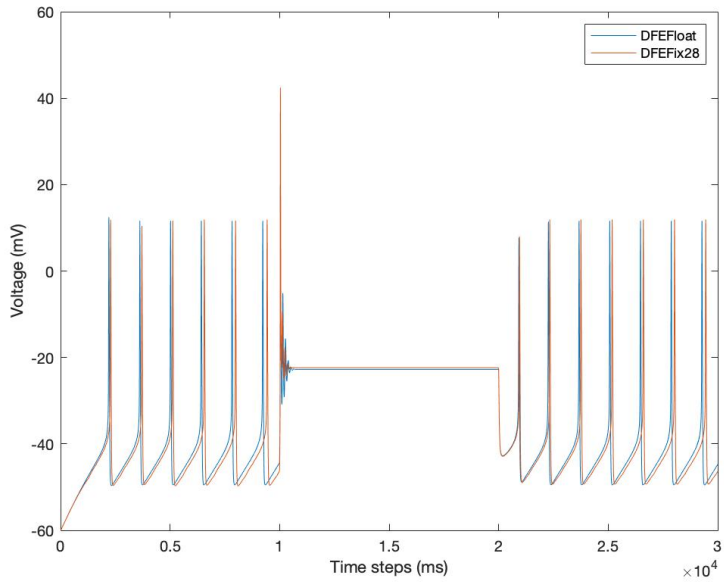


Figure 6.11: Axonic voltage for both floating-point and fixed-point of neuron 0

6.3.4 HH+custom+multi

The HH+custom+multi has a different output behavior compared to the previous three models. As we can see in Figure 6.12 the error is concentrated to when an injected current is supplied to the model (at $2 \cdot 10^4$ step). The fixed-point output of a single

neuron shown in Figure 6.13 has different outcome compared to floating-point. This will be further discussed in the following subsection since a huge error is also observed in the HHio model which supports multiple compartments.

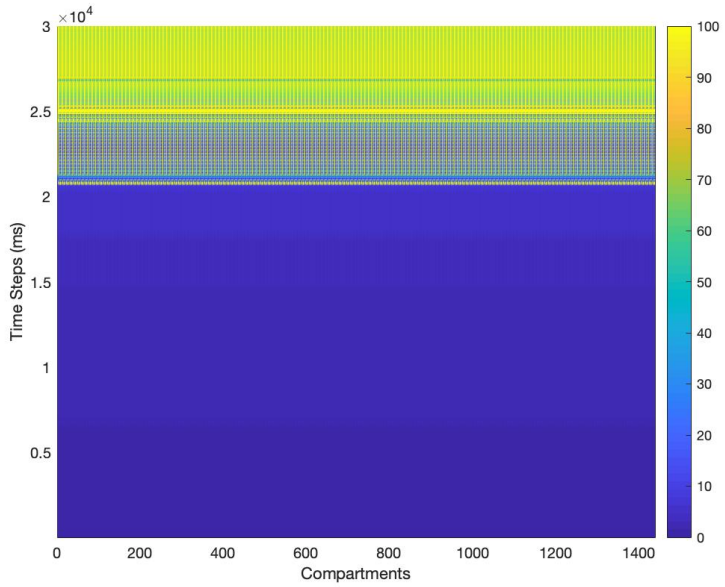


Figure 6.12: Error between floating-point and fixed-point of HH+custom+multi model

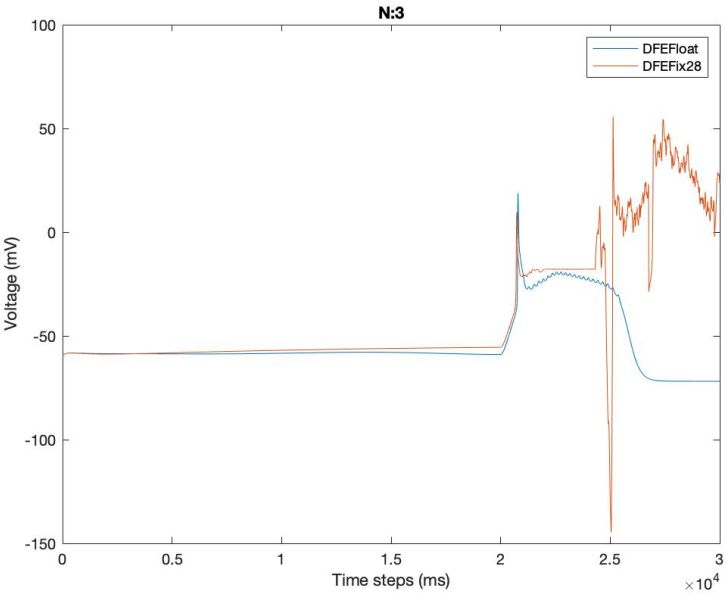


Figure 6.13: Axonic voltage for both floating-point and fixed-point of neuron 0

6.3.5 HHio

Figure 6.14 illustrates the error for all the compartmental voltages. We can noticed that the error between $2 - 2.5 \cdot 10^4$ steps peaks is over 100%. This is when an external

current is injected in our model and in Figure 6.17 that illustrates the output of a single compartment, the difference in the output is obvious. Most likely this huge error may be due to the less precision offered by fixed-point arithmetic. To elaborate, Figure 6.15 shows the first dendritic activation variable for the dendritic compartment. As we can see after the 20,000th step the fixed-point output has a flat result compared to the floating-point. By analyzing the output file, floating-point had small increments between steps for all the gate activation variables while in fixed-point the output stayed constant. These small increments are represented as zero values in fixed-point but this needs to be furthest investigated. Additionally, recall that in the fixed-point analysis we have found that the gate activation variables had an increased demand for fractional bits but the voltage output did not produce this much error. This is a result of not simulating a large network of neurons for a longer time for the error to be observable. Finally, considering the previous models, supporting multiple compartments also influence the output behavior since this is not observed in the first three models.

The average error between floating-point and the option where fixed-point arithmetic is used only for the gap junction weights is shown in Figure 6.16. The average error is 0.0189% and it was expected since besides the gap junction weights everything else is streamed and computed using floating-point arithmetic. This option provides the best of both worlds; not losing the accuracy from using fixed-point arithmetic and the advantage of using fixed-point arithmetic in parts of the application to save bandwidth.

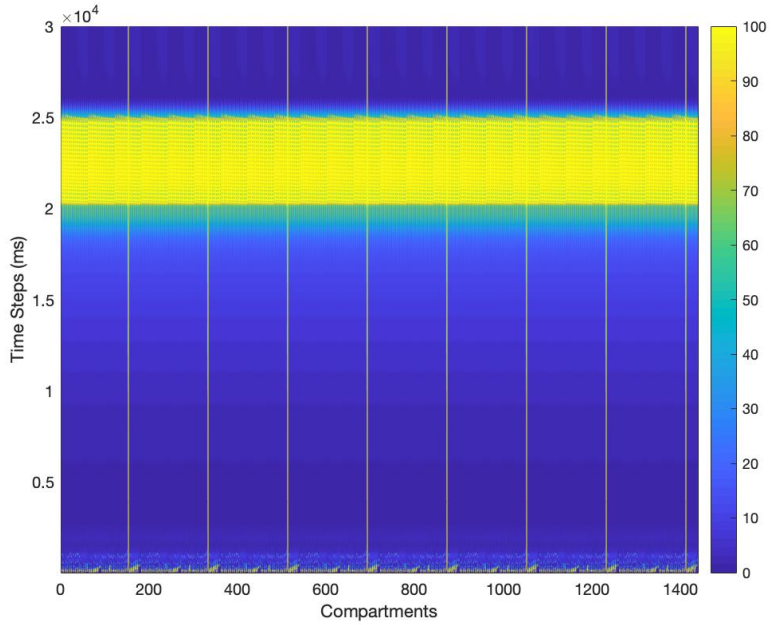


Figure 6.14: Error between floating-point and fixed-point of HHio model

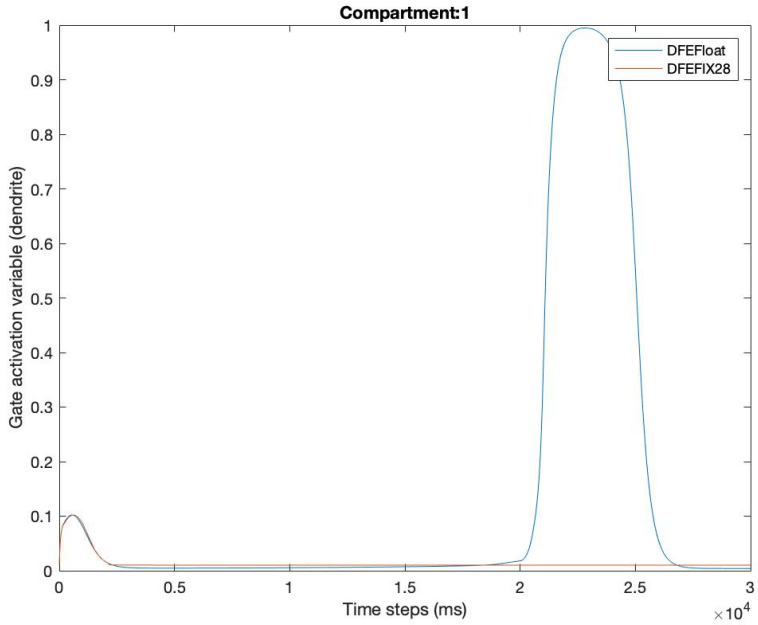


Figure 6.15: Error between floating-point and fixed-point of the first dendritic gate activation variable

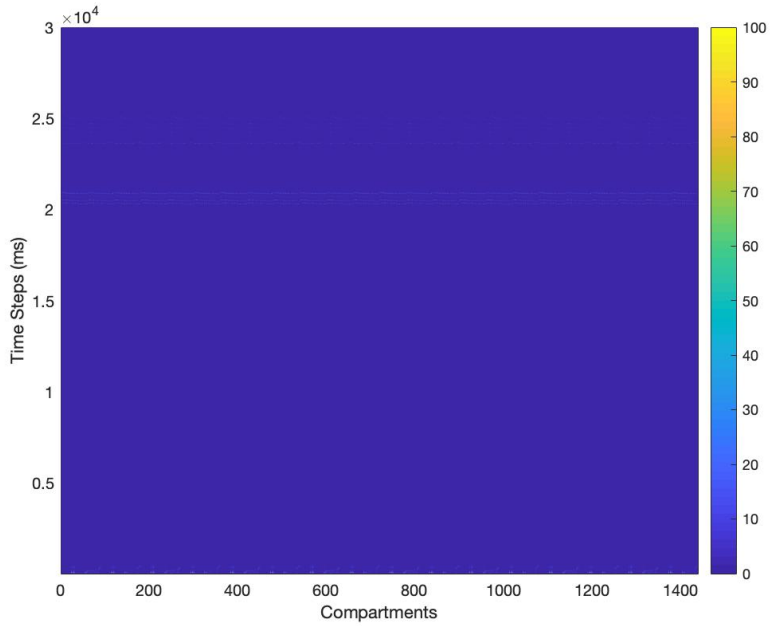


Figure 6.16: Error between floating-point and the mixed version of HHio model

The output for floating-point (DFEFloat), fixed-point (DFEFIX28) and the option that gap junction weights are streamed in fixed-point (Mixed) are depicted on Figure 6.17. Floating-point and the mixed option have the same behavior while the huge error that exists in fixed-point is clearly visible. As mentioned earlier, this needs to be further

investigated to find if this is indeed related to the less precision provided by fixed-point.

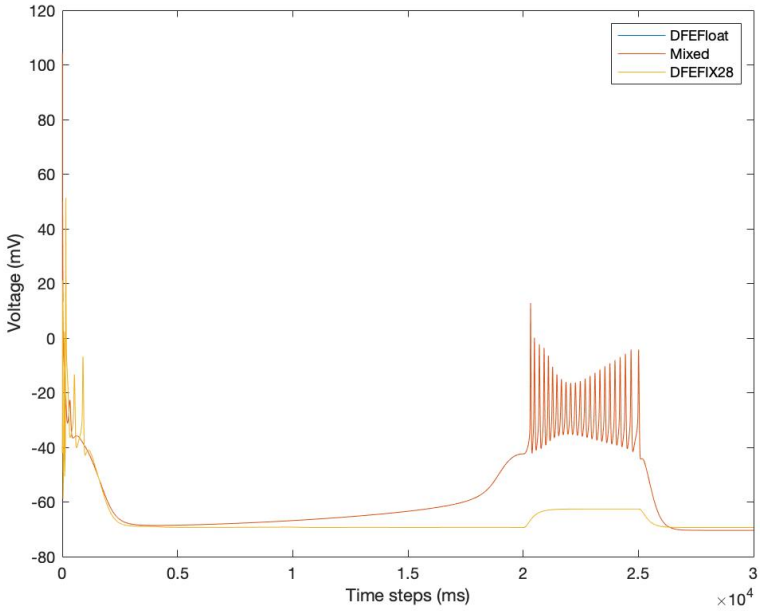


Figure 6.17: Voltage output of the axon compartment for neuron 0

7 Conclusions

7.1 Discussion

In this thesis, we used Maxeler's Technologies design process to improved the performance of the flexHH library on Maxeler's Data-Flow Engines (DFE).

Using Maxeler's technologies desing process, an accurate analysis of the flexHH library is conducted with focus on the most complex model. This analysis included the profiling of the flexHH library, operation analysis, and loop-flow graphs to determine the computation requirements and data transfers. Following that, a fixed-point analysis was conducted to determine if there is a possible candidate that fits our application's accuracy requirements. This information helped us create an accurate performance model based on the hardware resources estimation and bandwidth limitations. Finally, using all this information, the final implementation was created.

7.2 Research Questions

Can fixed-point arithmetic provide substantial improvement on performance for flexHH library without losing substantial accuracy compared to floating-point?

The models that do not support the gap junctions extension (HH, HHcu, HHcumu) achieved a 1.2x speedup by employing fixed-point arithmetic. The downside is that the model that support multiple compartments (HHcumu) experience a huge error compared to floating-point. Both models that supports the gap junction extension (HH+gap, HHio) achieved a 2x speedup, using the fixed-point arithmetic. HH+gap has a behavior close to floating-point while the HHio model output did not match the expected output behavior of floating-point. Finally, by employing fixed-point arithmetic just on the gap junction weights for the HHio model introduced 2x speedup without losing accuracy compared to floating-point. This mean that fixed-point analysis is helpful and needs to be considered especially if performance is of an essence and the application is memory bound.

Can the models of the flexHH library be implemented in such a way to be scalable using multiple DFEs

Indeed this work laid the foundations to speed up neuron modeling further by scaling up the models using multiple DFEs. As things stand now in technology, this application will not improve further by not losing much accuracy. The gap junction weights can be pushed down to 8-bits to achieve an unroll factor of 128 in total but this depends clearly on the accuracy needed. A solution to this is to scale up the application using more

DFEs and this is now possible using this multiple kernel optimized implementation of both HH+gap and HHio models. This is further explained in the following section.

7.3 Future Work

In this section, recommendations are given to improve the application further.

7.3.1 Multiple DFEs

In a Maxeler data-flow supercomputing system, multiple data-flow engines can be connected via a high-bandwidth MaxRing interconnect. The MaxRing interconnect allows applications to scale linearly with multiple DFEs. This is now possible because the gap junction kernel is extracted from the main application and divided into two kernels. Using multiple DFEs will boost performance since our application is memory bound, and each DFE will have its own LMem. The concept idea is shown in Figure 7.1. On the first DFE, the main kernel will be instantiated alongside two gap junction kernels as in the current application. For the rest of the DFEs, a gap junction kernel will be instantiated per LMem. The blue arrows represent streams from CPU to LMem that set the parameters and the red arrows represent the streams from LMem to each DFE. Orange arrows are the initialization parameters, streamed only during the first step of the simulation and the black arrows are the data transfers between kernels.

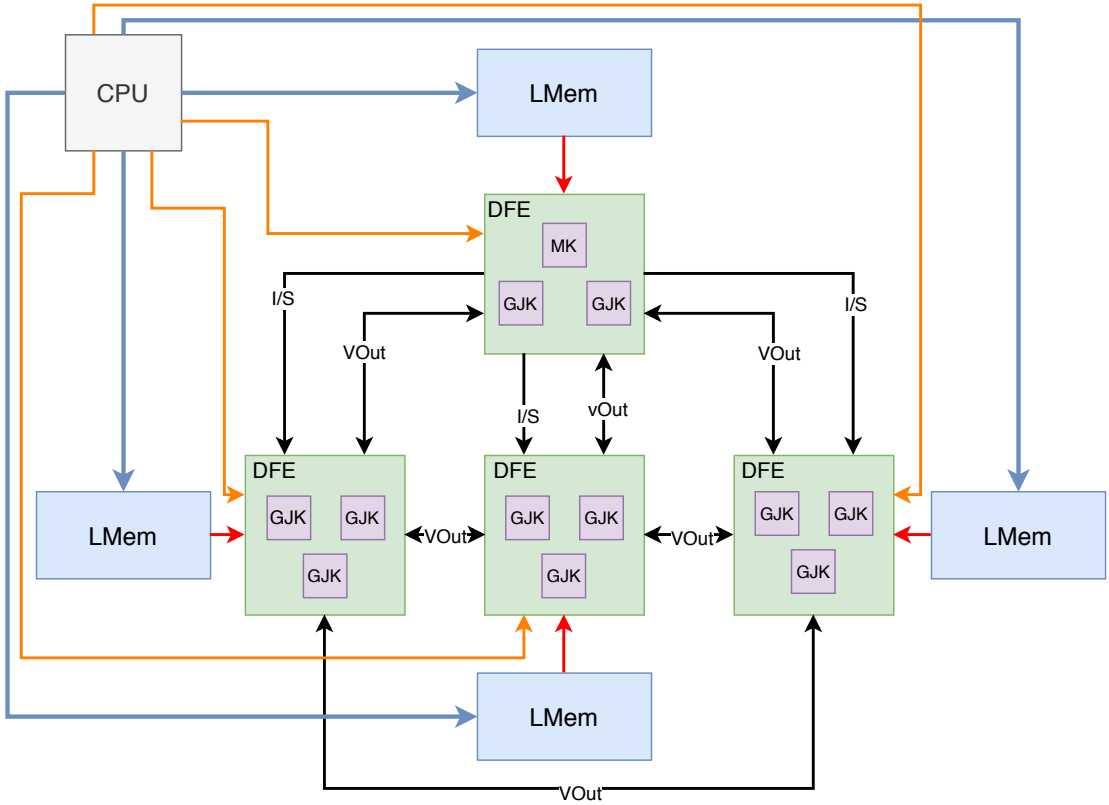


Figure 7.1: Multiple DFEs application

The main kernel will finish its computations, and then it will stall till the membrane voltages from the other DFEs are pushed into the manager FIFOs. For example, if an unroll factor of 32 per kernel is achieved, this means that in total, using a four DFE set up, we can achieve a total unroll factor of 352. A large network of neural cells can be simulated that divides the unroll factor per kernel. For example, let us consider a network of 61,440 neuron cells, and using the Equation 7.1, we can calculate the ticks needed per kernel. In this formula, N_{DFE} represents the number of DFEs, and N_{GJ} represents the number of instantiated gap junctions kernel per DFE. Each gap junction kernel will need 4,915,200 ticks to compute the intercellular current. The main kernel during the fist 798,720 ticks will compute and send the rest of the currents (I) with the elastance (S) to the gap junction kernels. Each gap junction kernel will need these values during the final iteration of the gap junction loop (after the 4,907,520th tick).

$$Ticks_{GJ} = \frac{N_{cells} \cdot N_{cells}}{unrollfactor \cdot N_{DFE} \cdot N_{GJ}} \tag{7.1}$$

$$Ticks_{MK} = N_{cells} \cdot N_{gates} \tag{7.2}$$

The data transfers between each gap junction kernel need to be considered; if the membrane voltages are not received before moving to the new step of computations, we will have an erroneous result. Figure 7.2, illustrates an example solution to mitigate this problem. A network size of 8 cells, an unroll factor of 4, and 2 DFEs are used for simplicity (we consider a kernel per DFE). Each rectangle denotes the computation between cell i and cell j. The blue rectangles represent the computations that the membrane voltages needed are produced inside the kernel and in orange, the membrane voltages that need to be streamed from the second DFE. During the final iteration, the membrane voltages from each gap junction kernel are produced and stream out to other kernels. To make sure the correctness of the application, we can transpose the computations in Y-direction; the second DFE can start computing using the membrane voltages produced by it, as shown in 7.2 to give plenty of time to receive the values from other kernels.

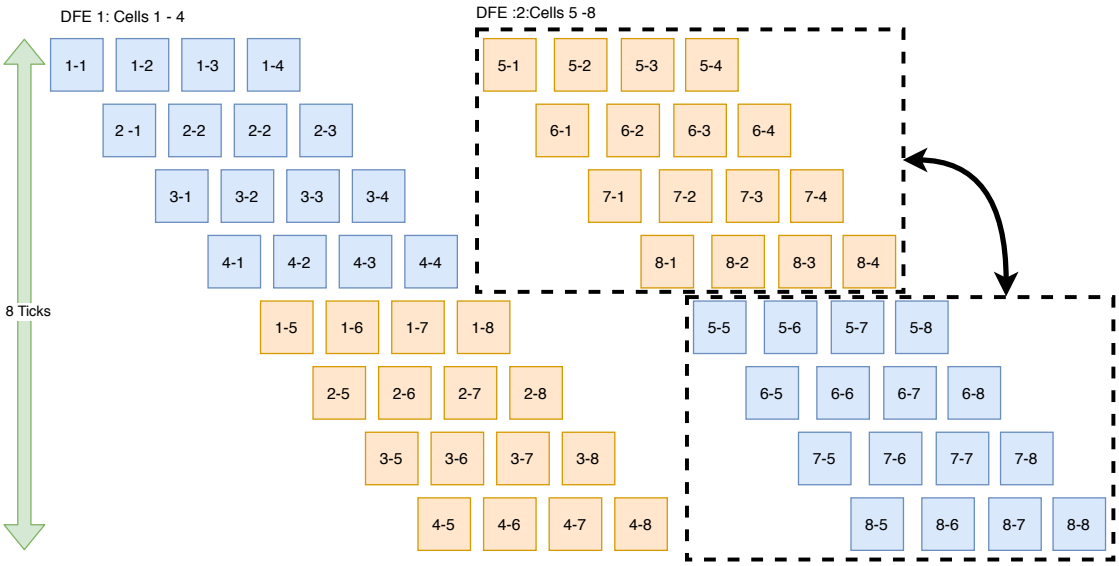


Figure 7.2: Multiple DFEs Application : Y-transposition in gap junction kernel

7.3.2 Design for portability

MaxCompiler can target different FPGA platforms and the spectrum of the supported platforms is constantly increasing. It is often required to targeted multiple platforms at the same time either being Alveo, MAX5C, or Amazon F1 instances. Moreover, as new FPGAs and DFEs become available, it will be of interest to support these newer and probably faster and more capable devices. This is interesting to include in this work since Erasmus MC has both MAX5C and Alveo cards and additionally launching the application on Amazon is desired. When porting a DFE based application from one platform to another you need to consider again the performance scalability. This means that we need to consider the difference between platforms and achieve performance close to what is possible on the new platform. For example, MAX5C includes 3 DIMMs of memory while Alveo (U250) includes 4 DIMMs of memory, this means that an extra Gap junction kernel can be instantiated to utilize the extra memory. Many changes need to take place and be considered to adjust the implementation to be dynamically based on the platform. What enables this, is the multiple kernel implementation done in this work since the current FPGA platforms and possibly future ones will be developed using multiple dies technology.

7.3.3 Models that do not include gap junctions

The models that do not support gap junctions are not optimized in this work besides using fixed-point arithmetic. Recall that these models are also memory bound and inefficient utilization of bandwidth due to the split of memory controller exists. These models do not require a vast amount of resources so instantiating multiple kernels instances is possible. One optimization which is going to help to efficiently utilize memory bandwidth is to instantiate one kernel per LMem DIMM and execute more neurons in parallel. This can be done since neurons do not interact with each other through gap

junctions. This will result in each kernel utilizing memory bandwidth efficiently and not be wasted on the cell/compartment parameters streams and increase the number of neurons computed in parallel.

Bibliography

[1] S. M. W. Dale Purves, *Neuroscience 3rd Edition*. Sinauer Associates Inc, 2004.

[2] W. M. K. Wulfram Gerstner, *Spiking Neuron Models*. Cambridge University Press, 2002.

[3] K. S. Eric Widmaier, Hershel Raff, *Vander's Human Physiology: The Mechanisms of Body Function*. McGraw-Hill, 2001.

[4] R. R. Llinás, "The olivo-cerebellar system : a key to understanding the functional significance of intrinsic oscillatory brain properties," *Frontiers in Neural Circuits*, vol. 7, no. January, pp. 1–13, 2014.

[5] R. N. Wulfram Gerstner, Werner M. Kistler and L. Paninski. (2014) Neuronal dynamics. Accessed 7/07/2020. [Online]. Available: https://neuronaldynamics.epfl.ch/online/Ch2.S2.html.

[6] B. Zeidman. (2017) All about fpgas. Accessed 10/07/2020. [Online]. Available: https://towardsdatascience.com/introduction-to-fpga-and-its-architecture-20a62c14421c.

[7] Xilinx. (2012) Large fpga methodology guide including stacked silicon interconnect (ssi) technology. Accessed 10/04/2020. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/ug872_largefpga.pdf.

[8] M. Technologies, "Multiscale dataflow programming-tutorials," Maxeler Technologies, Tech. Rep., 2018.

[9] ——, "Maxware dataflow acceleration process and optimisation book," Maxeler Technologies, Tech. Rep., 2018.

[10] Xilinx. (2020) Xilinx virtex ultrascale+. Accessed 9/07/2020. [Online]. Available: https://www.xilinx.com/support/documentation/selection-guides/ultrascale-plus-fpga-product-selection-guide.pdf#VUSP.

[11] Intel. (2020) Over 50 years of moore's law. Accessed 26/06/2020. [Online]. Available: https://www.intel.com/content/www/us/en/silicon-innovations/moores-law-technology.html.

[12] D. Rotman. (2020) We're not prepared for the end of moore's law. Accessed 26/06/2020. [Online]. Available: https://www.technologyreview.com/2020/02/24/905789/were-not-prepared-for-the-end-of-moores-law/.

[13] T. Simonite. (2016) Moore's law is dead. now what? Accessed 26/06/2020. [Online]. Available: https://www.technologyreview.com/2016/05/13/245938/moores-law-is-dead-now-what/.

[14] ——. (2016) Intel puts the brakes on moore's law. Accessed 26/06/2020.[Online].Available: https://www.technologyreview.com/2016/03/23/8768/intel-puts-the-brakes-on-moores-law/.

[15] Altera, "Accelerating high-performance computing with fpgas," 2007, accessed 26/06/2020. [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01029.pdf.

[16] D. Eaton. (2018) Turning big data challenges into opportunities with fpga-accelerated computing. Accessed 26/06/2020. [Online]. Available: https://www.datacenterdynamics.com/en/opinions/turning-big-data-challenges-opportunities-fpga-accelerated-computing/.

[17] D. Proch. (2019) Using fpgas to survive the death of moore's law, part 1. Accessed 26/06/2020. [Online]. Available: https://www.networkcomputing.com/data-centers/using-fpgas-survive-death-moore%25E2%2580%2599s-law-part-1.

[18] A. Hock-Koon. (2020) Hardware acceleration ii: how we used an fpga to accelerate gbdts. Accessed 26/06/2020. [Online]. Available: https://developers.amadeus.com/blog/hardware-acceleration-fgpa-to-accelerate-gbdt.

[19] A. Kingsley-Hughes. (2018) Intel fpgas picked up by dell emc and fujitsu. Accessed 26/06/2020. [Online]. Available: https://www.zdnet.com/article/intel-fpgas-picked-up-by-dell-emc-and-fujitsu/.

[20] Amazon. (2020) Amazon ec2 f1 instances. Accessed 26/06/2020. [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/.

[21] Alibaba. (2020) Compute optimized instance families with fpgas. Accessed 26/06/2020. [Online]. Available: https://www.alibabacloud.com/help/doc-detail/108504.htm.

[22] Xilinx. (2017) Baidu deploys xilinx fpgas in new public cloud acceleration services. Accessed 26/06/2020. [Online]. Available: https://www.xilinx.com/news/press/2017/baidu-deploys-xilinx-fpgas-in-new-public-cloud-acceleration-services.html.

[23] Microsoft. (2020) What are field-programmable gate arrays (fpga) and how to deploy. Accessed 26/06/2020. [Online]. Available: https://docs.microsoft.com/en-us/azure/machine-learning/how-to-deploy-fpga-web-service.

[24] 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: Association for Computing Machinery, 2014, p. 89–98. [Online]. Available: https://doi.org/10.1145/2554688.2554790

[25] N. Voss, P. Ziegenhein, L. Vermond, J. Hoozemans, O. Mencer, U. Oelfke, W. Luk, and G. Gaydadjiev, "Towards real time radiotherapy simulation," in *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, vol. 2160-052X, 2019, pp. 173–180.

[26] C. Mind. (2000) Structure of the brain. Accessed 20/11/2019. [Online]. Available: http://controlmind.info/human-brain/structure-of-the-brain.

[27] BrainFacts.org. (2020) The brain. Accessed 16/06/2020. [Online]. Available: https://www.brainfacts.org/3d-brain#intro=false&focus=Brain&zoom=false.

[28] E. Britannica. (2020) Midbrain. Accessed 3/07/2020. [Online]. Available: https://www.britannica.com/science/midbrain.

[29] ——. (2020) Pons. Accessed 3/07/2020. [Online]. Available: https://www.britannica.com/science/pons-anatomy.

[30] ——. (2020) Medulla oblongata. Accessed 3/07/2020. [Online]. Available: https://www.britannica.com/science/medulla-oblongata.

[31] SpinalCord.com. (2020) Parietal lobe: Function, location and structure. Accessed 16/06/2020. [Online]. Available: https://www.spinalcord.com/parietal-lobe.

[32] G. Smaragdos, G. Chatzikonstantis, R. Kukreja, H. Sidiropoulos, D. Rodopoulos, I. Sourdis, Z. Al-Ars, C. Kachris, D. Soudris, C. I. De Zeeuw, and C. Strydis, "Brainframe: A node-level heterogeneous accelerator platform for neuron simulations," *Journal of Neural Engineering*, vol. 14, no. 6, 2017.

[33] 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.

[34] T. Barclay. (2020) Cerebellar peduncle. Accessed 7/07/2020. [Online]. Available: https://www.innerbody.com/image_nerv02/nerv62-new.html#continued.

[35] J. R. De Gruijl, P. Bazzigaluppi, M. T. G. de Jeu, and C. I. De Zeeuw, "Climbing fiber burst size and olivary sub-threshold oscillations in a network setting," *PLOS Computational Biology*, vol. 8, no. 12, pp. 1–10, 12 2012. [Online]. Available: https://doi.org/10.1371/journal.pcbi.1002814

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

[37] E. M. Izhikevich, "Which model to use for cortical spiking neurons?" *IEEE Transactions on Neural Networks*, vol. 15, no. 5, pp. 1063–1070, 2004.

[38] R. Miedema. (2014) Flexhh: A flexible hardware library for hodgkin-huxley-based neural simulations. Accessed 9/07/2020. [Online]. Available: https://repository.tudelft.nl/islandora/object/uuid:84792c80-c0b8-489c-80f1-4436ed961844?collection=education.

[39] N. Schweighofer, K. Doya, and M. Kawato, "Electrophysiological properties of in-

ferior olive neurons: A compartmental model," *Journal of neurophysiology*, vol. 82, pp. 804–17, 09 1999.

[40] N. Schweighofer, E. Lang, and M. Kawato, "Role of the olivo-cerebellar complex in motor learning and control," *Frontiers in neural circuits*, vol. 7, p. 94, 05 2013.

[41] Xilinx. (2020) What is an fpga? Accessed 11/05/2020. [Online]. Available: https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html.

[42] F. Fallahlalehzari. (2020) Fpga vs gpu for machine learning applications: Which one is better? Accessed 9/07/2020. [Online]. Available: https://www.aldec.com/en/company/blog/167--fpgas-vs-gpus-for-machine-learning-applications-which-one-is-better.

[43] Cadence. (2020) What is an fpga? Accessed 11/05/2020. [Online]. Available: https://resources.pcb.cadence.com/blog/2019-fpga-vs-asic-differences-and-choosing-best-for-your-business.

[44] S. H.L. (2017) Purpose and internal functionality of fpga look-up tables. Accessed 10/07/2020. [Online]. Available: https://www.allaboutcircuits.com/technical-articles/purpose-and-internal-functionality-of-fpga-look-up-tables/.

[45] Xilinx. (2017) Ultrascale architecture configurable logic block. Accessed 29/05/2020. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug574-ultrascale-clb.pdf.

[46] S. Churiwala, *Designing with Xilinx ® FPGAs using Vivado.* Springer, 2017.

[47] Xilinx. (2014) Ultrascale architecture memory resource. Accessed 29/05/2020. [Online]. Available: http://xilinx.eetrend.com/files-eetrend-xilinx/download/201408/7556-13668-ug573-ultrascale-memory-resources.pdf.

[48] K. Saban. (2012) Xilinx stacked silicon interconnect technology delivers breakthrough fpga capacity, bandwidth, and power efficiency. Accessed 10/04/2020. [Online]. Available: https://www.xilinx.com/support/documentation/white_papers/wp380_Stacked_Silicon_Interconnect_Technology.pdf.

[49] V. Milutinovic, J. Salom, N. Trifunovic, and R. Giorgi, *Guide to DataFlow Supercomputing.* Springer, 2015.

[50] J. B. Dennis and D. P. Misunas, "A preliminary architecture for a basic data-flow processor," in *Proceedings of the 2nd Annual Symposium on Computer Architecture*, ser. ISCA '75. New York, NY, USA: Association for Computing Machinery, 1974, p. 126–132. [Online]. Available: https://doi.org/10.1145/642089.642111

[51] T. Agerwala and Arvind, "Data flow systems: Guest editors' introduction," *Computer*, vol. 15, pp. 10–13, 1982.

[52] N. Trifunovic, V. Milutinovicl, J. Salom, and A. Kos, "Paradigm shift in big data supercomputing: Dataflow vs controlflow," 2015.

[53] M. Technologies, "Maxcompiler kernel numerics tutorial," Maxeler Technologies, Tech. Rep., 2018.

[54] N. C. Lab. (2020) Brain simulation library on hpc platforms and brain experimentation support. Accessed 17/06/2020. [Online]. Available: https:https://neurocomputinglab.com/research-themes/brainframe/.

[55] N. Voss, "Methodology for complex dataflow applications," Ph.D. dissertation, Imperial College London, South Kensington, London SW7 2AZ, United Kingdom, 11 2020, an optional note.

[56] Erasmus. (2020) The erasmus brain project. Accessed 14/03/2019. [Online]. Available: http://erasmusbrainproject.com/.

[57] R. S. Jay Fenlason. (1998) Gnu gprof. Accessed 24/02/2019. [Online]. Available: https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html.

[58] V. Developers. (2000) Valgrind. Accessed 27/02/2019. [Online]. Available: http://valgrind.org/.

[59] J. Fonseca. (2019) gprof2dot". Accessed 7/03/2019. [Online]. Available: https://github.com/jrfonseca/gprof2dot.

[60] Intel. (2020) Intel® vtune™ amplifier. Accessed 21/03/2019. [Online]. Available: https://software.intel.com/en-us/vtune.

[61] A. Finnerty and H. Ratigner. (2017) Reduce power and cost by converting from floating point to fixed point introduction : Xilinx data type support.

[62] MathWorks. (2020) Get started with fixed-point designer? Accessed 15/04/2020. [Online]. Available: https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html.

[63] Xilinx. (2020) Performance and resource utilization for adder/subtracter v12.0. Accessed 14/03/2019. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/ru/c-addsub.html.

[64] ——. (2020) Performance and resource utilization for divider generator v5.1. Accessed 14/03/2019. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/ru/div-gen.html.

[65] ——. (2020) Performance and resource utilization for floating-point v7.1. Accessed 14/03/2019. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/ru/floating-point.html.

[66] ——. (2020) Performance and resource utilization for multiply adder v3.0. Accessed 14/03/2019. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/ru/xbip-multadd.html.

Appendices

Simulation Parameters

A.1 HH

$$V = 0 \tag{A.1}$$

$$m = 0.5 \tag{A.2}$$

$$h = 0.5 \tag{A.3}$$

$$n = 0.5 \tag{A.4}$$

$$I_{app}(t) = \left\{ \begin{array}{l} 50, \text{ if } (t > 10000) \leq (t < 20000) \\ 0, \text{ if otherwise} \end{array} \right\} \tag{A.5}$$

where t is the number of steps

A.2 HH+gap

$$V = 0 \tag{A.6}$$

$$m = 0.5 \tag{A.7}$$

$$h = 0.5 \tag{A.8}$$

$$n = 0.5 \tag{A.9}$$

$$I_{app}(t) = \left\{ \begin{array}{l} 20 \cdot (i\%10), \text{ if } (t > 10000) \leq (t < 20000) \\ 0, \text{ if otherwise} \end{array} \right\} \tag{A.10}$$

$$w_{i,j} = 0.003 \tag{A.11}$$

where t is the number of steps
i,j are the indexes of the cells

A.3 HH+custom

$$V = -60 \tag{A.12}$$

$$h = 0.9 \tag{A.13}$$

$$x = 0.2369847 \tag{A.14}$$

$$I_{app}(t) = \left\{ \begin{array}{l} 50, \text{ if } (t > 10000) \leq (t < 20000) \\ 0, \text{ if otherwise} \end{array} \right\} \tag{A.15}$$

where t is the number of steps

A.4 HH+custom+multi

$$V_{dend} = -60 \tag{A.16}$$

$$r_d = 0.0112788 \tag{A.17}$$

$$s_d = 0.0049291 \tag{A.18}$$

$$q_d = 0.0337836 \tag{A.19}$$

$$CA2Plus = 3.7152 \tag{A.20}$$

$$V_{soma} = -60 \tag{A.21}$$

$$k_s = 0.7423159 \tag{A.22}$$

$$l_s = 0.0321349 \tag{A.23}$$

$$h_s = 0.3596066 \tag{A.24}$$

$$n_s = 0.2369847 \tag{A.25}$$

$$x_s = 0.1 \tag{A.26}$$

$$V_{axon} = -60 \tag{A.27}$$

$$h_a = 0.9 \tag{A.28}$$

$$x_a = 0.2369847 \tag{A.29}$$

$$I_{app}(t) = \left\{ \begin{array}{l} 6, \text{ if } (t > 20000) \leq (t < 25000) \\ 0, \text{ if otherwise} \end{array} \right\} \tag{A.30}$$

where t is the number of steps

A.5 HH+custom+multi+gap (HHio)

$$V_{dend} = -4 \cdot (i\%20) \tag{A.31}$$

$$r_d = 0.0112788 \tag{A.32}$$

$$s_d = 0.0049291 \tag{A.33}$$

$$q_d = 0.0337836 \tag{A.34}$$

$$CA2Plus = 3.7152 \tag{A.35}$$

$$V_{soma} = -2 \cdot (i\%30) \tag{A.36}$$

$$k_s = 0.7423159 \tag{A.37}$$

$$l_s = 0.0321349 \tag{A.38}$$

$$h_s = 0.3596066 \tag{A.39}$$

$$n_s = 0.2369847 \tag{A.40}$$

$$x_s = 0.1 \tag{A.41}$$

$$V_{axon} = -6 \cdot (i\%10) \tag{A.42}$$

$$h_a = 0.9 \tag{A.43}$$

$$x_a = 0.2369847 \tag{A.44}$$

$$I_{app}(t) = \left\{ \begin{array}{l} i\%20, \text{ if } (t > 20000) \leq (t < 25000) \\ 0, \text{ if otherwise} \end{array} \right\} \tag{A.45}$$

$$w_{i,j} = 0.01 \tag{A.46}$$

where t is the number of steps
i,j are the indexes of the cells