CHALMERS

Brain modelling with reconfigurable acceleration

Master of Science Thesis in the program Embedded Electronic System Design

Reza Zakernejad
Roozbeh Soleimanifard

Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, July 2014

Brain modelling with reconfigurable acceleration

REZA ZAKERNEJAD
ROOZBEH SOLEIMANIFARD

Examiner: Ioannis Sourdis

Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden July 2014

Abstract

In this thesis we port a brain cell model into the Maxeler machine and improve performance while using the advantage of re-configurability of this machine. Hodgkin-and-Huxley model is used in this work to simulate the behavior of certain type of neurons called Inferior Olive [1]. Currently general-purpose machines are able to simulate a few tens of brain cells at (brain) real-time performance; the goal of this thesis is to improve performance by an order of magnitude. That is to simulate hundreds of brain cells in real-time which, for human brain, limits the simulation time for each step to 50 μSec.

The Maxeler machine uses a powerful general purpose processor in parallel with one or more FPGA cards, together with a powerful compiler and dataflow programming technique.

Simulating the behavior of a network of cells has three main challenges. First of all it is computationally intensive due to the accuracy of the model and thus complexity regarding hardware implementation (on FPGA). Secondly, there are data dependencies between different simulation steps, which require special attention in our dataflow computing model. During the thesis work different solutions have been developed to reach the fastest way to transfer intermediate results from one simulation step to the next. The last challenge comes from the fact that, it is desired for this simulation to support all-to-all communication between the simulated network cells. Thus each cell in a network has to have the potential to connect with any other cell.

By using a Maxeler machine as the platform, implementing computationally intensive parts of the model on FPGA and taking the advantage of fully pipelined execution and the parallelism, the performance has been improved by more than x3046, x253 and x4 comparing to MATLAB, C code and other FPGA implementations respectively.

Acknowledgements

It is our pleasure to show our gratitude toward everyone who supported us during this thesis work.

Firstly, we really appreciate the help and great suggestion of our supervisor, Ioannis Sourdis for his intelligent ideas. During this thesis work he supported us by answering our questions moderately and criticizing our ideas gently. All discussions we had with Ioannis been significantly interesting and advantageous.

Secondly we are thankful to Catalin Ciobanu (Chalmers University of Technology) who was a great help to us in working with Maxeler machine. Thanks for giving us your valuable time.
We are thankful to Yixin Liu and Oscar Siby for reading the drafts of our work and their valuable comments. Your comments truly helped us to improve this thesis.

We would like to express our appreciation to Christos Strydis and Giorgos Smaragdos (Erasmus Brain Project) for giving us the brain model application and Diego Oriato (Maxeler Technologies) for helping us through this work. We believe that your help and comments enhanced our improvement our work significantly.

Last but not least:

A sincere gratitude to my girl friend Nazila Mokri. My appreciations for her patience and endurance while I invited Roozbeh to our place every now and then to work on the thesis together. Thanks for understanding. (Reza)

I want to express my great appreciations to my dear friend Nazila Mokri for her patience and endurance AND also I have to say that without her delicious foods this work would be impossible. (Roozbeh)

Reza Zakernejad
Roozbeh Soleimanifard
Gothenburg, Sweden, Jan, 2014

Table of Contents

Chapter1: Introduction

Neuroscientists try to analyze and understand the behavior of human brain. This is a very challenging task, requiring substantial research effort to build accurate brain models. Besides developing realistic brain models, another major challenge is supporting the large amount of computations needed to simulate a large network of brain cells, using these models.

Besides to neuroscience, in other fields such as robotics and artificial intelligence, it is useful to simulate the behavior of biological neurons. Using more efficient neuron models would improve the efficiency of such systems. Last but not least, brain neuron models can lead to new computer architectures, which can be used in designing more powerful processors (brain inspired computing).

Brain cell models have been implemented in hardware both in general-purpose processors and FPGAs. General-purpose processors are slow and they are not able to simulate a large number of brain cells in real-time. FPGAs have a better potential for this task due to their hardware speed and parallelism. Currently, related approaches can simulate a network of 96 brain cells successfully in real-time [1]. In this thesis the main goal is to increase the performance of brain cell simulations as well as the number of brain cells while keeping latency within the real-time boundaries.

This chapter describes the problem addressed in this thesis and the thesis objectives.

1.1 problem statement

In this thesis we study and use C implementation of an accurate Inferior Olive brain cell model, developed by researchers in Erasmus Medical Center and Erasmus Brain Project, Rotterdam. This brain cell mode simulates the behavior of certain type of a human brain cells and will be described in section 2.1.

A modeled brain cell should be executed in brain real-time. This timing constraint comes from the fact that to evaluate the accuracy of the model, it should respond in real-time to be comparable with a real brain cell. This constraint is 50 µSec according to the Erasmus Medical Center neuroscientists. This 50 µSec represents one simulation step of a brain cell. The output of a cell (updated state) has to be used as its input in the next simulation step.

In the brain, cells are connected with each other and are forming a network. A network of identical brain cells needs up to 100% connectivity, but in this thesis and other related works (see section 2.2) this connectivity may be reduced for various reasons described in Chapter 3.

Currently, general-purpose machines are able to simulate a few tens of brain cells at real-time performance. The aim of this thesis is to simulate a network of modeled brain cells, with as many cells as possible with 100% connectivity, while remaining in the brain real-time constraint. We pursue our goal using reconfigurable hardware. In this approach Maxeler MAX3424A machine has been chosen which is a high performance computer system composed of a high-end general-purpose processor and an FPGA board with Virtex-6 XC6VSX475T FPGA [2]. This machine will be described in section 2.3.

1.2 Thesis objectives

The overall goal of this thesis work is to improve the performance of a specific brain cell network model using reconfigurable acceleration. We aim at increasing the number of simulated cells from a few tens (that current solutions support) to a few hundreds. In other words, the purpose of this thesis work is to improve the performance of Inferior Olive brain cell modeling compared to previous general-purpose processors or FPGA implementations. To reach this goal we chose hardware reconfigurable acceleration, which can be easily reprogrammed to change the models as well as it can improve performance. Using a dedicated compiler simplifies the task. This compiler allows the user to program the machine with a restricted JAVA instead of complex VHDL programming.

Maxeler MAX3424A system enables one to have hardware acceleration with the benefit of its FPGA card. Therefore the whole C code or at least parts of it (i.e. computationally intensive parts) has to be ported to this system using Maxeler compiler (known as MaxCompiler). The compiler will be described in section 1.3. In order to use the MaxCompiler one has to follow a dataflow model to be able to use FPGA card and have hardware acceleration.

To improve performance, some modifications on both software and hardware parts are needed. The description of our modifications performed in this thesis work can be found in sections 3.3 and 5.3.

To show the performance benefits of our solution, there will be comparison with related works.

The final task of this thesis is to provide simulation flexibility of brain network model, which enables one to easily change the simulation parameters, such as number of simulation steps and input parameters of the cell, while preserving the benefits of reconfigurable acceleration in a Maxeler machine. There will be a more detailed description on flexibility or re-configurability task in Chapter 3.

1.4 Thesis outline

This thesis is organized as follows. The next chapter explains the Inferior Olive neuron cell and how cells are connected in a network. It also describes the Maxeler reconfigurable machine and dataflow programming. Chapter 3 shows different solutions experienced through this thesis and their advantages and disadvantages. The last design in Chapter 3 has the best results and is evaluated in Chapter 4. Chapter 4 also compares this approach with previous related works in brain modeling. Chapter 5 draws conclusion and discusses directions of future work.

Chapter 2: Background

In this chapter a brain model for the Inferior Olive is described in detail for the reader to be familiar with how a brain cell operates and acts in a network of cells [4]. Then, related work is discussed briefly. Also a more detailed description of the Maxeler machine is presented at the end of the chapter.

2.1 The Inferior Olive

The chosen brain cell model of this thesis work simulates the behavior of the Inferior Olive (IO) cells. We provide some background information about the IO cells, their structure and interconnect, as well as information about the IO model used.

2.1.1 The IO Neuron Cell

An IO brain cell or an IO neuron, illustrated in Figure 2.1, is comprised of three parts, what neuroscientists call compartments namely, the Dendrite, the Soma and the Axon compartment.

Dendrite acts as the input of neuron. It provides connections with other cells in a network of cells through electromagnetic and chemical stimulations, generates its own output as a level of voltage and delivers it to the Soma which is the main body of an IO cell. As can be seen it has a tree-like structure to be able to absorb more signals and coming influences from other cells, Dendrites are covered by Synapses [3].

Soma gathers the signals, which come from Dendrites, and passes them to the output part of the cell. It contains the nucleus, which holds the DNA and chromosomes. Soma provides the needed energy for the cell but does not have an active role in transfer of the signals.

Axon is the output of the cell; if the signals coming from the Soma exceed the threshold limit of the Axon's terminal it generates a signal, which will be passed to another neuron or a muscle cell.

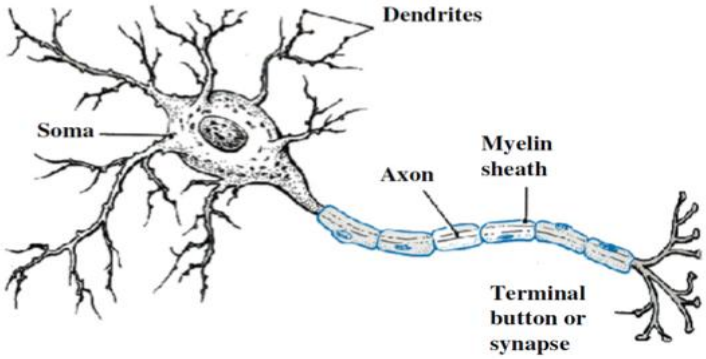

Figure 2.1: Basic illustration of a biological neuron [4].

2.1.2 Inferior Olive Brain Model

In this section, we introduce the brain model used. It has been written in C code and utilizes dynamic memory allocation to send and receive variables to and from the computational parts. The model simulates a network of cells for multiple (e.g. hundred thousands) simulation steps.

- One cell computation

Figure 2.2 shows the structure of the IO model used, its internal connections, compartments and its inputs and outputs. There are three blocks, which have been named as CompDend, CompSoma and CompAxon. These blocks represent different mathematical equations which compute the new state of the cell. Each cell is represented by 20 parameters, which are categorized in the three compartments. Each compartment computes some of these parameters. Parameters represent potassium, calcium and sodium levels that affect the behavior of a brain cell.

In reality a brain cell is connected to adjacent cells through its inputs and outputs, and at the same time it receives external signals from outside world and generates an output, which is used to study the behavior of the cell.

A brain cell receives the effect of the level of the potassium, calcium and sodium as a level of electric signals, uses them to generate new levels of the elements, which represent the next state of the cell.

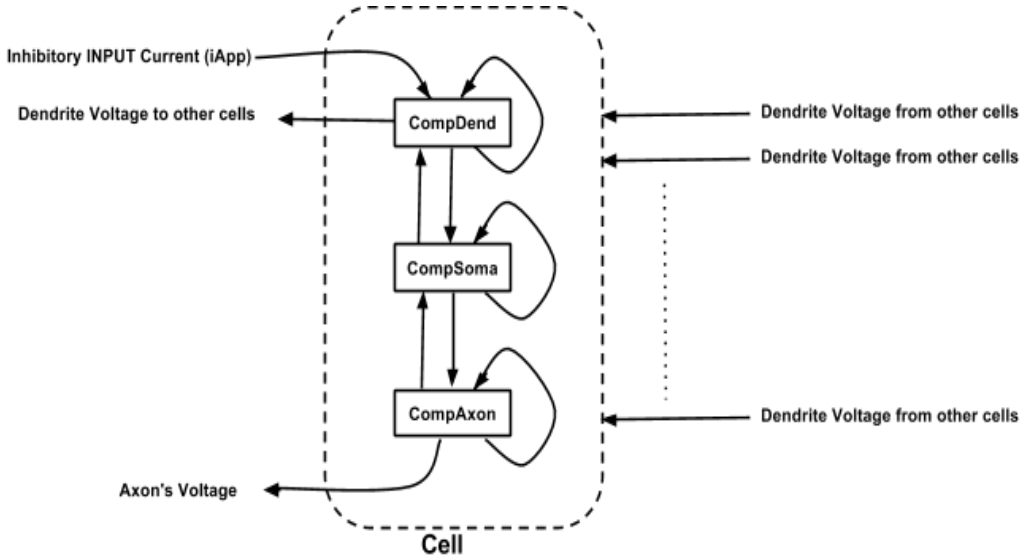

Figure 2.2: IO Model by Jornt de Gruijl (NIN, Amsterdam), based on the two-compartment model (dendrite-soma) [5].

The connection between adjacent cells is done though their Dendrite voltages. Figure 2.2 shows the block diagram of the IO brain model. It illustrates that each cell receives Dendrite voltage from all other cells in the network, and generates one dendrite voltage, which is an input to its neighbors.

Inhibitory INPUT Current (iApp) is an external signal. It represents the effect of the outside world to the brain cell like when you touch something hot or a coarse substance etc. Axonal voltage is the output of a cell which represents the behavior of the neuron, neuroscientists observe this signal to study the behavior of the model to see if it is an acceptable model of the real brain cell or not.

To study the behavior of a brain cell we need to simulate it for a time interval of at least a few seconds. In reality each cell changes its state and generates new parameters in less than 50 µSec. Each state is fed by some of the parameters from the previous state; shown in Figure 2.2 by the curved lines connected to each block.

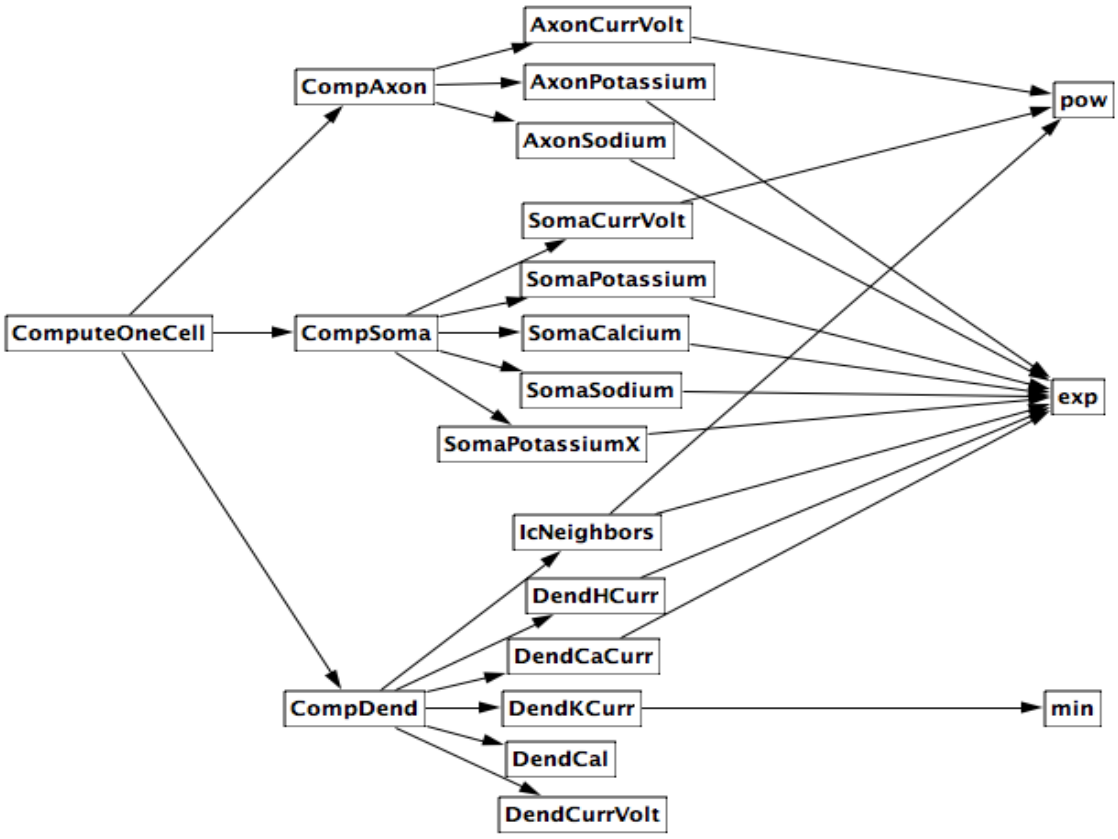

Figure 2.3: Three functions have sub-functions and mathematical functions.

To compute new parameters for a cell, the model requires computing complex formulas that use multiple complex arithmetic operations such as exponents, multiplications and divisions. The large number of arithmetic operations and the type of variables (floating point) make the model computationally intensive. Figure 2.3 shows which arithmetic function such as, power (pow), exponent (exp) and minimum (min), is used by which brain cell model function. Basic arithmetic functions are not included in Figure 2.3.

- ## Level of Parallelism

From Figure 2.4, one can see how functions and sub-functions can be executed in parallel. DendCurrVolt, SomaCurrVolt and AxonCurrVolt also need values, which are computed by other

functions in their sections, i.e. CompDend, CompSoma and CompAxon (those inputs are not included in Figure 2.4), but at a simulation step each section is completely independent from others therefore they can be executed in parallel with each other.

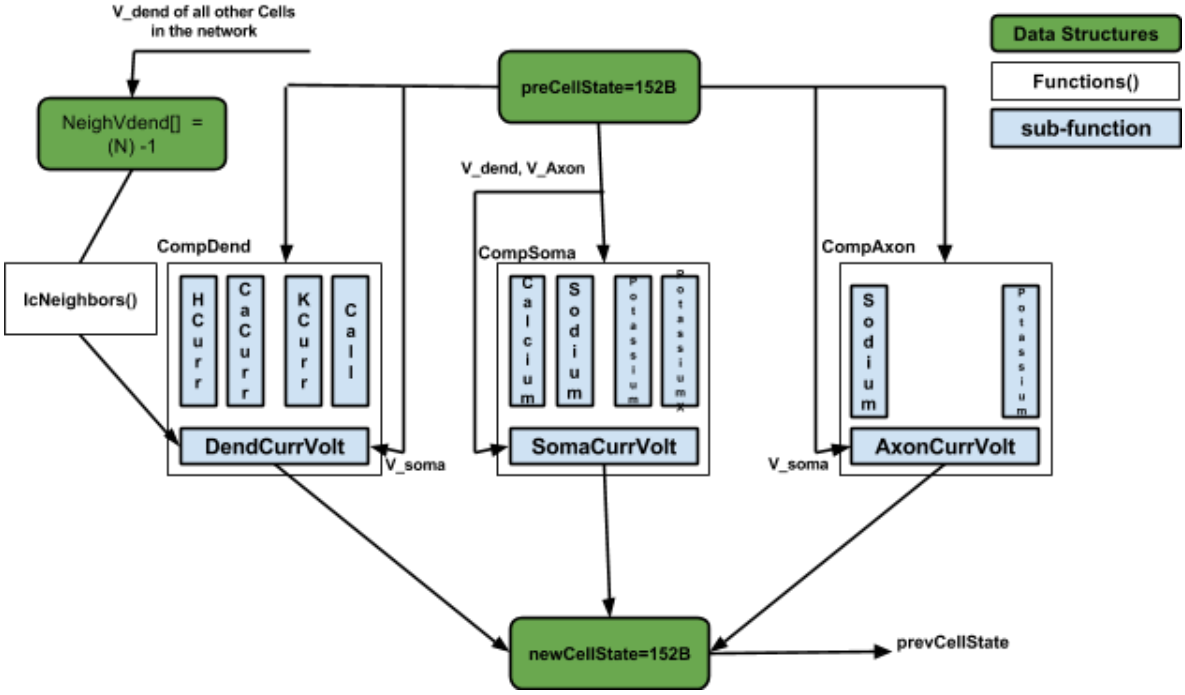

Figure 2.4: How functions and sub functions and data structures are related to each other.

Inputs V_dend, V_soma and V_axon that feed sub functions, are results of the previous simulation step and therefore do not create dependencies within a simulation step. Consequently, each function and its sub functions can be executed in parallel to other functions, it depends on the available hardware resources whether a designer decides to execute them in parallel or not.

A Network of Cells

In the case of full connectivity all cells in a network are connected to every other, therefore each cell is affected by all other cells. The influence that each cell receives from all other cells in a network is denoted as IC value. The IC value is computed by IcNeighbors function, which receives the Dendrite voltage of all other cells. IC value has to be computed separately for each cell. In the original IO model, each cell is connected to only eight adjacent cells.

Each cell receives iApp which is an externally evoked input current in addition to 19 other feedback parameters. For the execution of a network containing N number of IO cells, the initial state is provided to cells externally before starting execution. Then subsequently their state is reevaluated using (besides external inputs) as input their current state and are saved in a memory location which is called newCellState.

- ### Simulation Step

To observe and study the behavior of a network of brain cells, it needs to be executed in a specific time interval. One simulation step is the time in which all new parameters for all cells in a network are computed.

In reality a network of brain cells changes its state (generates all new parameters) every 50 µSec. This means that the simulations have to compute each cell or a network of cells in 50 µSec to be within real-time. As an example, if it is required to simulate a network of brain cells for 120,000 simulation steps, a time interval of 6 seconds is required. There is a data dependency between consecutive simulation steps, as the cell state of one step provides input to the next step.

- ### Conclusion

The chosen brain cell model application has data dependencies between simulation steps and is computationally intensive. Since each cell is connected to all other cells in a network, communication between cells can be a bottleneck for the simulation performance.

2.2 Related works in Brain Modeling

There have been many attempts in the past to simulate brain cells behavior. There are a couple of categories of models used to model the behavior of brain cells. Spiking Neural Network (SNN) and Synapses Learning are two well-known such approaches. The first one seems more convenient to be implemented and is more widely used. In this thesis a conductance model, represented by Hodgkin and Huxley in 1952 has been used [6]. The conductance model describes the behavior of neuronal compartments of spiking neurons. This model is complex enough and dynamic to be able to simulate accurately cells. Conductance models are known as the most accurate for simulating real life neuron behavior.

Below we present some related works on accelerating brain simulation. We have to mention that all of the below mentioned related works used either Integrated-and-Fire (IaF) or Leaky-IaF, which are not as accurate as conductance brain cell model.

Ghani et al. implemented a neuron model on a Virtex2 FPGA [7]. The model simulates brain cells with a connectivity of 10 connections per cell and utilized 120 LUTs per cell. Cassidy et al. implemented a network of 32 cells on a Spartan3 FPGA [8]. Their model is 3125 times faster than real time. It is estimated that the maximum number of cells in a network can be increased to 64 cells.

Schrauwen et al. proposed a design to implement a neural network using serial execution of arithmetic computations [9]. The network is comprised of 56 interconnected cells. Their model simulates the network 2930 times faster than the real time on a Spartan3 FPGA. In a Virtex4 FPGA the number of cells in the network increased to 1400 cells and the speed increased to 5860 times faster than real time. With the same architecture as Schrauwer, Glackin et al. in 2005 presented an implementation of a network of 168 cells on a Virtex 2 FPGA [10]. With

simplest connectivity they implemented a network of 4200 cells at 12500 times faster than the real time.

Shayani et al. implemented a network of 161 cells with 16 synapses [11]. They used a Virtex 5 FPGA and their design runs 4210 times faster than real time. It is claimed that the design can implement 10000 cells in real time. Horacio Rostro-Gonzalez et al. presented a full connectivity network of 100 cells [12]. They implemented the models on a GPU and a Spartan 3 FPGA for comparison purposes. The GPU runs 3.3 times faster than real time and Spartan3 FPGA simulates 4.6 times faster than real time.

Finally, Cassidy and Andreou introduced an implementation of neural simulation and creation of prostheses and neuromorphic system on a Spartan3 FPGA that runs 5000 times faster than the real time, the design contains a network of 32 cells, which are not connected to each other [13].

2.3 Maxeler reconfigurable machine

In this section we describe the Maxeler machine in detail. The Maxeler system architecture is described and subsequently its dataflow programming model is explained. Then, a programming example is given.

2.3.1 Maxeler system architecture

The Maxeler system consists of a host CPU, one or more FPGA boards and high speed memories. CPU and FPGA(s) are connected through PCIe [14]. In case of existence of more than one FPGA boards, they are connected to each other via MaxRing [2]. Figure 2.5 illustrates the Maxeler architecture without a MaxRing.

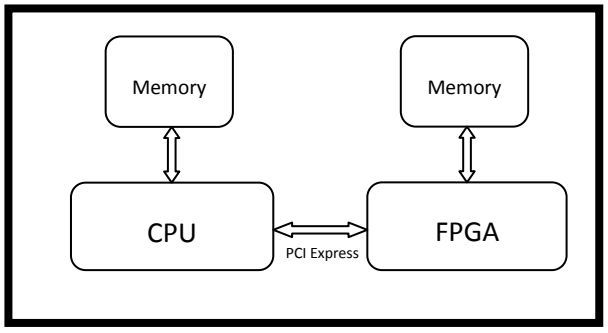

Figure 2.5 Maxeler architecture.

The computationally intensive parts of the brain cell model at hand can be implemented in the FPGA, we call these parts kernels. Each kernel has its own inputs and outputs and may be connected to other kernels. Connections between one kernel to other kernels and memories are managed through another module called manager. One important point for achieving high performance in Maxeler machine is that it separates the computational part(s) as kernel(s) while communicational task between kernels, memories and the host CPU is done through the manager. This enables the system to have a deep pipeline for computational hardware part(s). Also the performance can be further improved since each kernel can be executed in parallel with others [15].

- Kernel

Kernels are responsible for computations inside the FPGA in a Maxeler machine. MaxCompiler includes a kernel compiler, which compiles the code written in JAVA, into a hardware FPGA configuration. The kernel compiler itself considers a JAVA software library of Maxeler JAVA syntaxes. Maxeler technologies has developed custom syntaxes to be used in kernel JAVA codes, to make it easier to code the dataflow hardware parts.

Kernels are one-way streaming computational cores without any feedback. It is worth mentioning that one can support kernels (with feedbacks) by using memories such as FPGA's BRAM or external DRAM or by using arrays. This type of kernels has been used in this thesis and is discussed later in Chapter 3. In case a kernel with feedback needs to be supported, for example a for-loop, the compiler fully unrolls the loop. This creates multiple copies of the body of the loop. These identical hardware copies operate in parallel. In case of data dependency between different iterations of the loop, the compiler uses FPGA's BRAMs. Unfortunately, this is supported only for simple loops. One can find more information regarding loops in kernels in MaxCompiler Loop tutorial [16].

Below blocks are used to make a kernel graph. The blocks are called nodes [15]. A complete graph is illustrated in Figure 2.7.


Computational nodes, which can perform basic arithmetic and logic operations.


Value nodes. These values can either be set by host application or pre-defined constants.


Stream offsets. These nodes enable one to have access to previous or later data in a data stream according to position of current data in same data stream.


Multiplexer nodes for making decisions.


Counter nodes. Such a counter helps one to have access to a specific position in a data stream.


Input and output ports for input and output data streams.

It is possible to have one or more kernels implemented on FPGA(s) in a Maxeler system. Each kernel has inputs and outputs data streams, which are connected to the host code, other kernels or different types of memories. PCIe data bus provides fetching/sending data from/to the FPGA with the speed of 2GB/s between CPU and FPGA. The manager is a JAVA code that schedules and synchronizes every connection inside the hardware configuration. The MaxCompiler compiles the manager JAVA code into the corresponding hardware. In addition the manager configures design settings such as operating frequency.

Host code is a part of the application that can be written in C/C++ or FORTRAN and runs on the host CPU. With host code one can send/get data streams and constant values to/from kernels

via the PCIe data bus. Other than setting the data streams, the host code plays another role in a Maxeler system; that is to load the `.max` file to FPGA(s). This file contains hardware configurations and is the result of compiling and building the manager and kernel(s) JAVA files. One can start the execution of kernel(s) by calling a kernel via the host application. If there is more than one kernel and one of them is called to be executed by host, all of them will start their execution. This is because all kernels have been implemented on the FPGA as a Data Flow Engine (DFE) and basically the host calls the DFE. A designer should note that data transfer over PCIe may take longer than the desired timing constraints of the design. Therefore it can be said that it is not a good idea to have lots of data transfers back and forth between host code and FPGA(s), since it may affect performance dramatically.

Each Maxeler system consists of one or more FPGA(s). One can implement the computationally intensive parts of the application in reconfigurable hardware. Each FPGA is known as a DFE, which can be connected to other DFEs (if the Maxeler system consists of more than one FPGA board) through a high bandwidth MaxRing bus through manager.

Maxeler provides a tool to simulate a design in their system. This simulator leads to a much faster development time and also enables the designer to view the execution of a kernel by using predefined debug syntaxes inside kernel JAVA code [2].

Finally, the MaxCompiler provides predefined syntaxes for optimizations, which can be applied in both the kernel and the manager JAVA codes [17]. One can perform pipelining and placement optimizations of the kernel hardware configurations inside kernel JAVA code. These optimizations are such as setting build configuration (as the same as “effort” in building a hardware configuration in Xilinx tools), stream, memory clock frequencies inside manager JAVA code and so on. For more details we refer the reader to MaxCompiler optimization cheat sheet [17].

2.3.2 Dataflow programming

In control-flow programming a program code converted into a list of different instructions to be fetched sequentially one by one from memories (instruction caches) then move to the processor and be executed.

In dataflow programming, data flow into a chain of operational units in the form of streams. Operational units perform simple arithmetic/logic operations from memory until the whole computation is completed. In dataflow computing one is able to expand the number of operational units so the same processes can be done simultaneously. To make it more clear one can imagine dataflow programming as a factory production line in which each worker (operational unit/DFE) does a specific job. Data flows into those workers like a stream and each one does its job on the data. Each worker can handle a piece of the whole production job and many of those small pieces of works can be done at once. Figure 2.6 illustrates the principle of dataflow programming [2].

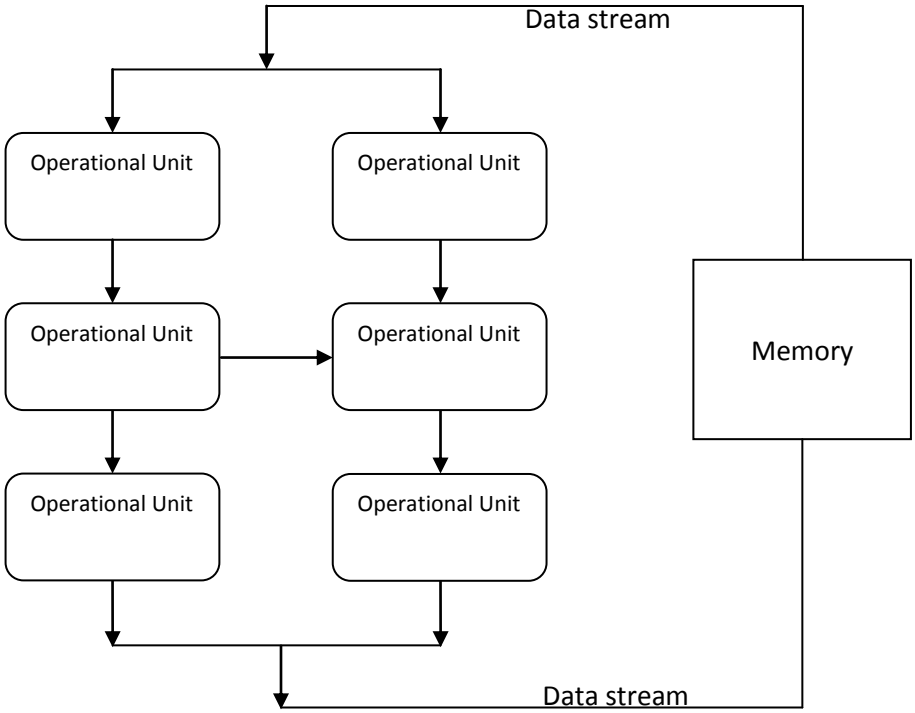

Figure 2.6 Dataflow programming.

2.3.3 Simple example

To make all above descriptions more clear, a very simple example will be reviewed in this chapter. The example has been taken from Maxeler Compiler tutorial [2]. We tried to add everything described before to this example such as debugging, optimizations and so on, so one can be aware of how to use the above mentioned issues inside JAVA code. This simple example further defines how dataflow programming works.

Consider that we want to implement the following function in a dataflow program.

$$y_i = \begin{cases} \dfrac{x_i + x_{i+1}}{2} & if\ i = 0 \\ \dfrac{x_i + x_{i-1}}{2} & if\ i = N - 1 \\ \dfrac{x_i + x_{i+1} + x_{i-1}}{3} & otherwise \end{cases}$$

This can be done easily in a C code as below:

```
Void SimpleExampleCPU (int size, float dataIn, float expected) {
       expected[0] = (dataIn[0] + dataIn[1]) / 2;
       for (inti = 1; i < size – 1; i ++) {
              expected[i] = (dataIn[i –1] + dataIn[i] + dataIn[i + 1]) / 3;
       }
       expected[size – 1] = (dataIn[size – 2] + dat-aIn[size – 1]) / 2;
}
```

Now let us to review different steps needed for creating a dataflow kernel. At the beginning an input stream should be defined. In this case an input of C float type (8-bit exponent and 24-bit mantissa) values can be declared as the following.

```
DFEVar x = io.input(”x” , dfeFloat(8, 24));
```

DFEVar is a MaxCompiler data type and has a DFE run-time value, while Java data types are compile-time values. One should consider that these two different data types cannot be converted to each other. DFEVar can be considered as a wire that points to an edge in the dataflow graph. Different data types can be assigned to DFEVar like Boolean, integer etc. For example in Java a Boolean can be assigned to a variable by writing the syntax below:

```
bool x = 1;
```

While in MaxCompiler it shall be written as follows:

```
DFEVar x = constant.var(dfeBool(), 1));
```

DEFVars are more flexible than Java data types and can change frequently, depending on the type of operation being performed.

Also a scalar input should be defined for declaring the upper boundary condition ($i < size - 1$) while the lower boundary condition is ($i = 0$). This scalar inputs value is `size` with the type of 32bit unsigned integer.

```
DFEVar x = io.scalarinput(”size” , dfeUInt(32)) ;
```

In Maxeler world, each cycle of kernel execution is called a tick. At each tick one datum from the input data stream enters the kernel. In this example counting ticks helps to detect that the boundary conditions are fulfilled or not. The MaxCompiler provides a counter which counts the number of kernel ticks (cycles). There is more information about counters available in MaxCompiler tutorial [2].

```
DFEVar count = control.count.simpleCounter(32, size);
```

We can define logical flags for the lower bound and upper bound according to boundary conditions and if we are below upper bound and above lower bound it means that we are within the boundaries.

```
DFEVar aboveLowerBound = count > 0;
DFEVar belowUpperBound = count < size - 1;
DFEVar withinBounds = aboveLowerBound & belowUpperBound;
```

The designer needs to have access to the current data (x_i), previous data (x_{i-1}) and next data (x_{i+1}) in the data stream. One can have access to the current data by just calling `x`, but the designer should use stream offsets to have access to the previous and next data in the data stream.

```
DFEVar prevOriginal = stream.offset(x, -1);
DFEVar nextOriginl = stream.offset(x, 1);
```

If we are not within the boundaries then we have to put zero as the values of previous data and next data. This means we have to create a multiplexer, which selects between zero and previous/next data. This can be done with a conditional syntax in a Maxeler system.

```
DFEVar prev = aboveLowerBound ? prevOriginal : 0 ;
DFEVar prev = belowUpperBound ? nextOriginal : 0 ;
```

Also we should have another multiplexer for selecting the correct value for the denominator of the function since, it is 2 just when we are on boundaries but in other cases it is 3.

```
DFEVar divisor = withinBounds ? constant.var(dfeFloat(8, 24),3): 2 ;
```

Until now we declared the input stream and the previous and the next data as different DFE variables. Now we have to write a code regarding the computational part in JAVA. Sometimes it is needed to not make some parts of the code fully pipelined to overcome latency problems. In such a case one can put the computational parts between two different predefined optimization codes. This makes it possible to define how deeply we want the compiler to pipeline the part of code (which is in between of the optimization syntaxes) [17]. Since this is a very simple example using predefined optimization codes may not make any difference in the final performance but we mentioned it here just as an example. One may track the intermediate results. In that case a debug line of code can be used.

```
optimization.pushPipeliningFactor(0.5);
DFEVar sum = prev + x + next;
debug.printf("\ sum=%d\n", sum );
DFEVar result = sum / 3;
optimization.popPipeliningFactor(0.5) ;
```

Now we can define the output of the kernel.

```
io.input(”y” , result, dfeFloat(8, 24)) ;
```

The kernel graph of such a design can be found in Figure 2.7.

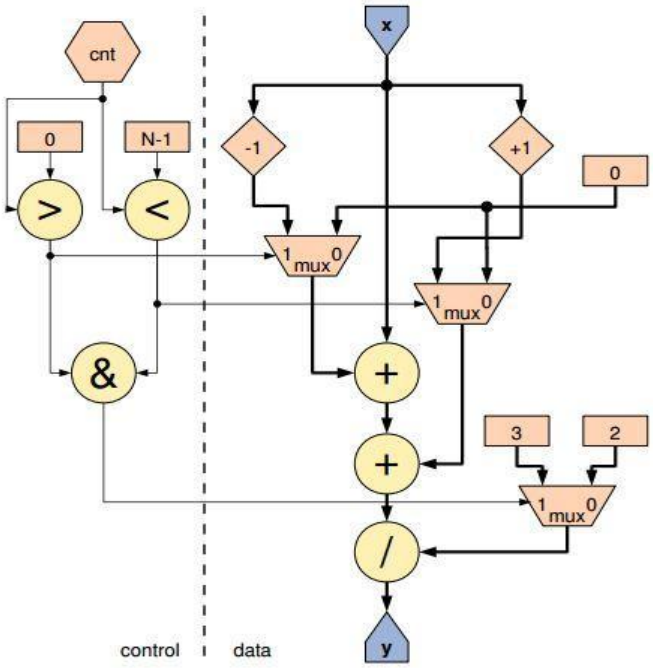

Figure 2.7 Simple example kernel graph [15].

2.3.4 Benefits of Maxeler over general-purpose machines

Maxeler machine provides high performance computing solutions using reconfigurable hardware acceleration. The architecture of such a machine was explained above. Using dataflow programming (versus control-flow programming in general purpose machines) with exploiting of different levels of parallelism and optimizations make Maxeler machine one of

the first choices for high performance computing. Maxeler machines has large amount of high bandwidth memory which make the machine suitable for computationally intensive applications. MaxCompiler provides data encoding, data compression and analytical optimization which is also beneficial in speeding up the computations by lowering size of data transfer. Since the application used in this thesis is computationally intensive, the analytical optimization of MaxCompiler is a benefit of using this machine for us.

2.4 Conclusion

The Inferior Olive brain model is chosen to be implemented in this thesis due to its accuracy. This accuracy comes with an expense of complexity. In addition of the application complexity, another challenge is the feedback of the application. This feedback means a data dependency between consecutive simulation steps.

In General Purpose Machines, such simulation with the same brain model is performed successfully with a few tens of cells. The goal of this thesis is to accelerate the simulation and simulate few hundreds of brain cells. In this approach the Maxeler machine is chosen due to its high performance and the reconfigurable hardware acceleration potential.

Chapter 3: Brain modeling in Maxeler reconfigurable machine

In this chapter, different designs, which have been implemented on the Maxeler machine, is discussed. These designs have been used to accelerate the IO brain model and are evaluated by their advantages and disadvantages.

3.1 Alternative design options

In all our designs we use the same inferior olive cell model implemented on the FPGA as a neuron hardware kernel. This kernel is executed N times to simulate a complete network of N cells. A complete simulation is done by consecutive execution of the kernel i.e. N times by the number of simulation steps. To simulate a network, time multiplexing is used. The operating frequency of the FPGA is much higher than the brain model timing constraint per simulation step. It shows that the kernel can be executed several times within the 50 µSec, and therefore simulate larger networks of cells.

It needs to be mentioned that the simulation of one cell needs more than one cycle. The number of pipeline stages is an important factor to determine the number of cycles needed to simulate a cell.

- Bottlenecks

One bottleneck of simulating IO brain model is the communication between different brain cells in a network. It comes from the demand of full connectivity in the network of brain cells. To compute the IC value (defined under “A network of cells” in section 2.1.2) for one cell in the simulation step *t*, Dendrite voltages of all other cells from simulation step *t-1* are needed. This causes demanding connectivity between cells.

Second bottleneck is the data dependency between consecutive simulation steps. In each simulation step a cell needs to be fed with the parameters generated in previous step. In other words intermediate results have to be passed from one step to the next one.

Third bottleneck is the FPGA area consumption due to the heavy computations of the IO brain cell model.

In this thesis different designs have been tried out to find the most efficient way to increase the performance and decrease resource utilization.

By taking advantage of time-multiplexing we overcame the intensive connectivity problem. Once a cell is computed its Dendrite voltage is kept into a block of memory and after computing all cells is completed, there is a memory containing all Dendrite voltages. In doing so, to compute IC value for one cell all Dendrite voltages are available in the block of memory without needing to connect all cells to each other.

Next problem with IC computation is that it requires a significant amount of resources. Listing 3.1 shows the code for computing IC value:

```
For (i = 0 ; i< N ; i++){
            V = prevV_dend - neighVdend[i];
            f = 0.8 * exp(-1*pow(V, 2)/100) + 0.2;
            IC = IC + (CONDUCTANCE * f * V);}
```

Listing 3.1: IC computations.

As can be seen from listing 3.1, for a network of N cells, a significant amount of computation is repeated for generating the IC value of only one cell.

Last but not least is, the large amount of computations needed to compute parameters (the state) of one cell. One can observe that the brain model utilizes complex arithmetic operations. It makes the application very resource demanding, as can be seen in the coming sections one cell without IC computations, consumes about half of the FPGA resources.

Therefore, we can conclude that exchanging the intermediate results between consecutive simulation steps is a major bottleneck for brain cell modeling. In this thesis, this has been addressed in four different ways:

1- Exchange data between host and DFE through PCIe for each simulation step (DFE fed by the Host) ,
2- Using DRAM on FPGA board (Using DRAM) ,
3- Using FPGA's BRAM (Using on-chip BRAM),
   And finally …
4- Taking the advantage of all pipeline stages, using data stream offset (Full Pipeline).

3.1.1 DFE fed by the Host

In this approach, the host sends all initial values for a complete network to the kernel, kernel computes all new parameters for cells and once all cells are completed, the kernel sends back all new parameters to the host.

The host computes the IC values for all cells and sends a complete set of parameters containing IC values back to the kernel as inputs for the next simulation step. Figure 3.1 illustrates the block diagram of this solution.

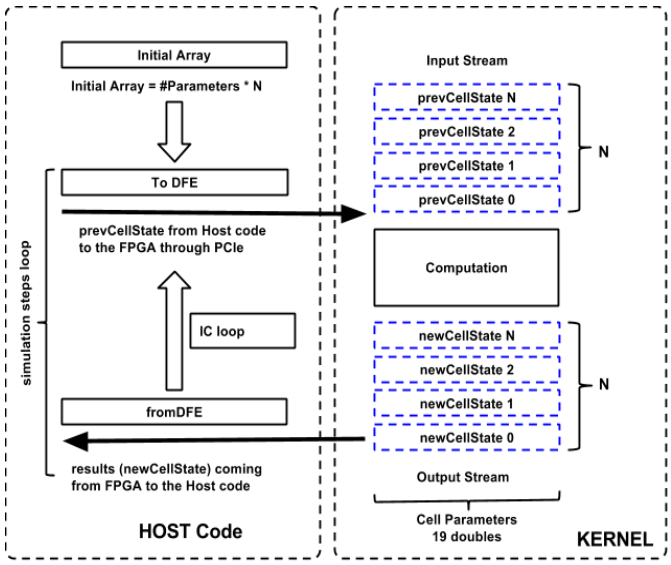

Figure 3.1: Exchange and IC computation in Host.

- Summary

Figure 3.2 and Table 3.1 illustrate the time measurement for transfer data through PCIe to the DFE. They illustrate execution time for transmitting parameters for a network of N cells without any computation. It includes time for set-up, send and receive data between the kernel and the host and tear-down the FPGA. In this design, the kernel needs to be called by the host code at each simulation step. The time for calling the kernel and transfer data for a network of 8 cells is 262.8 µSec while the time constraints for the brain cells simulation is 50 µSec. Consequently, this solution cannot fulfill the time restriction for a network of brain cells.

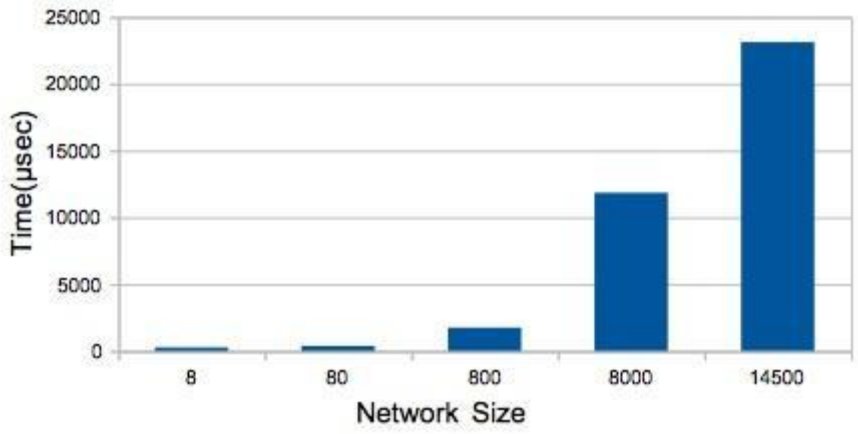

Figure 3.2: Data transfer time measurement for PCIe to FPGA and vice versa.

N	8	80	800	8000	14500
T(µSec)	262.8	376	1743.8	11836.6	23106.4

Table 3.1: Data transfer through PCIe.

3.1.2 Using DRAM

In this approach a DRAM is available in the FPGA board is used. It is utilized to store intermediate results and send them back to the DFE in the next simulation step. The DRAM (also called LMem, Large Memory), is used as a continuous memory in the system and is read or written in burst mode. Figure 3.3 illustrates the DRAM solution. In this part, firstly a very short introduction to the LMem is provided, then the solution based on LMem and its bottlenecks are discussed.

- LMem

MaxCards have a large external DRAM on board, it can be used to store large amount of data and provide input to the DFE. The DRAM can be read or written directly by the host without requiring special code in the manager or kernel. It is convenient for loading initial data into the DRAM before the kernel and the manager start running. The same DRAM contents, which have been written or read by host, can be read or written by the DFE as well, this is done using the same API (Application Programming Interface) for the host code and the DFE.

```
Kname_Lmem_writeLMem(size, 0 *size, inA);
Kname_Lmem_writeLMem(size, 1 *size, inA_1);
…
Kname_Lmem_readLMem(size, 0 * size, oA);
Kname_Lmem_readLMem(size, 1 * size, oA_1);
```

Listing 3.2: Writing and reading data into and from LMem from host code.

Listing 3.2 shows the commands to read and write data from and to the DRAM. `size` is the size of data, to be read or written, next argument into the parenthesis is the start address of reading or writing data and the last argument is the port name. Same port name and same start address are used in the kernel. Manager connects the host and the kernel by defining ports, which are the same as in the host and the kernel.

There is a memory controller in the DFE component of the MaxelerOS that generates a command-based interface to the DRAM. There is also a command queue and a data buffer, which are used to read and write data from or to the DRAM. The memory controller reads commands and contents of the data buffer, then writes the data to the appropriate location (address) in memory or reads data from the appropriate location in the memory and then writes them into a data buffer. This can be seen in Listing 3.3.

```
LMemCommandStream.makeKernelOutput("AcmdStream",
                        wordCount === 0 &burstCount<totalBursts,
                        totalBursts * offsetA + burstCount,
                        constant.var(dfeUInt(8), 1),
                        constant.var(dfeUInt(1), 0),
                        constant.var(dfeUInt(1), 0),
                        constant.var(false));

LMemCommandStream.makeKernelOutput("oAcmdStream",
                        wordCount === 0 &burstCount<totalBursts,
                        totalBursts * offsetoA + burstCount,
                        constant.var(dfeUInt(8), 1),
                        constant.var(dfeUInt(1), 0),
                        constant.var(dfeUInt(1), 0),
                        outEn);
```

Listing 3.3: Command stream structure for DRAM.

The FPGA and the DRAM are connected to each other through input and output streams, each input or output stream is assigned to one command stream by the manager. It is shown in listing 3.4A and 3.4B.

```
DFEVector<DFEVar>inA = io.input("inA", Type, enableIO);
io.output("oA", oA, Type, enableIO);
```

Listing 3.4A: input and output connect FPGA to the DRAM.

```
DFELinkinA = addStreamFromOnCardMemory("inA",k.getOutput("AcmdStream"));
DFELinkoA = addStreamToOnCardMemory("oA",k.getOutput("oAcmdStream"));
```

Listing 3.4B: manager assigns input and output stream of DRAM to appropriate command streams.

Each portion of DRAM has its own command stream and ports. The MaxCompiler supports only up to 16 streams or 16 blocks of DRAMs at once. Once data is completely processed by the kernel, command stream generates an interrupt to the host, which indicates the data are ready to be read by the host.

LMem is also able to operate at a frequency higher than the FPGA, for the Max3424A it is about 400 MHz. There is also a bus of 384 bits that connects the FPGA to the DRAM, the combination of the wide bus and the DRAM frequency provides a bandwidth of 38.4 GB/s which simply is computed as BW = 400 MHz * 384 bits = 153.6 Gb/s. If the FPGA operates at 100 MHz and the DRAM operates at 400 MHz, 1536 bits can be transferred each cycle between the DRAM and the FPGA. The other option would be 768 bits/cycle if the FPGA operates at 200 MHz.

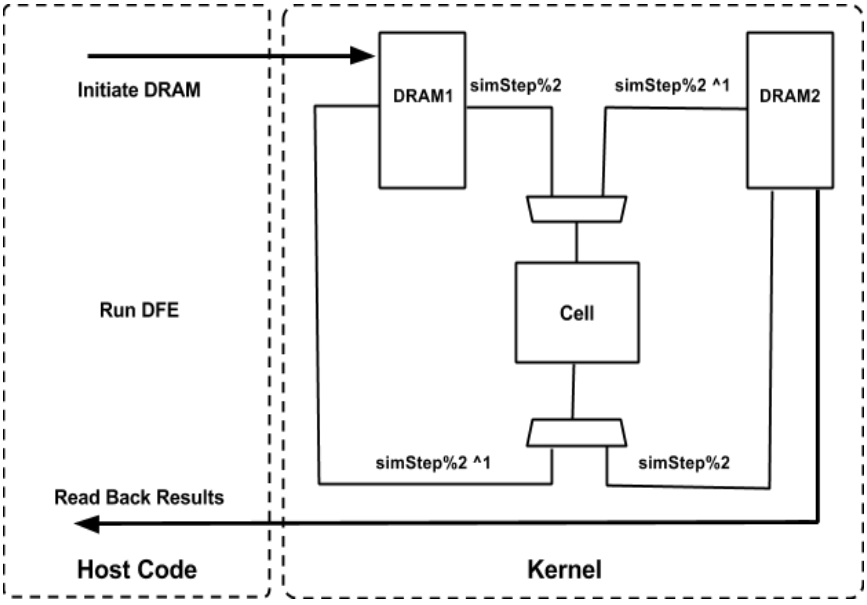

Figure 3.3: Exchange using LMem. Each simulation step one DRAM provides input another keeps the results their rolls are changed next simulation step.

To transfer more than one word per cycle, it is needed to define ports between the FPGA and the DRAM as arrays or vectors. In this case an interrupt signal and `enableIO` have to be carefully generated. This is shown in Listing 3.5.

```
DFEVectorType<DFEVar> vectorType = new DFEVectorType<DFEVar>(Type, PIPES);
DFEVector<DFEVar> inA = io.input("inA", vectorType, enableIO);
```

Listing 3.5: generating interrupt signal and enableIO.

`PIPES` are the number of data, which is expected to be transferred each cycle. Since the DRAM and the FPGA are connected through a 384-bit data bus, each 384 bit of data transfer would take one cycle.

In this approach initial values for a network of cells have been written into the LMem directly by the host code through PCIe, before the kernel and the manager start executing. Two segments of LMem with the size of 1920 words are defined as DRAM1 and DRAM2. Initial

values for a network of 80 cells are written into the DRAM1. Figure 3.4 shows the structure of a LMem segment. After completing initialization, the kernel runs and fetches data from DRAM1; new parameters will be written into the DRAM2. These results will be fetched again as the input for the next simulation step.

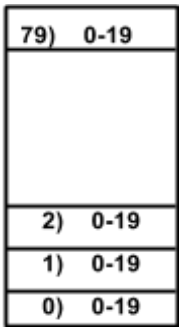

Figure 3.4: The structure of LMem segments to keep the intermediate results for 80 cells.

A counter has been defined to count the simulation steps. It has also been used to define which DRAM block provides the input for the kernel and which one keeps the new parameters. The counter is defined as the following.

```
simStep = chain.addCounter(steps,1);
```

The counter is called `simStep`, it is incremented by one after N cells are computed. It has a maximum value of “steps”, which defines the number of simulation steps.

- ## Bottleneck

The MaxCompiler is not responsible for the delays that are generated because of the pipelining stages. According to the Maxeler loop tutorial with regard to the result, the input in each simulation step is not entered to the kernel in the row as it should be [16]. In a kernel which its input is the output of the previous step, the output will not be ready at the appropriate time that is supposed to be applied as the input in the next step. To make this clear consider Figure 3.5.

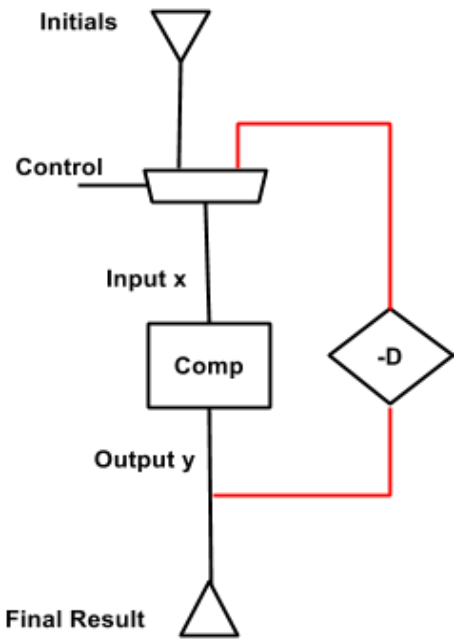

Figure 3.5: problem with the delay of pipelining stages.

The corresponding result to the input x1 (i.e. y1), which is supposed to be applied to the system as input x2 will not be ready at the exact time that it is expected by the system, but after a delay of D. The delay D comes from the fact that the kernel needs D unit of time to compute the output y1. The MaxCompiler avoids compiling such a design, because the results are not correct. To make the program compile-able a number of delays have to be added to the design in order to break the loop, as it is illustrated in the Figure 3.5. The delays are provided by adding offsets to the feedback path (red line).

In data flow programming the number of input streams and the number of cycles have to be exactly the same for the design to be executable. Adding D number of offsets (delays) makes the design compile-able but to have an executable design the kernel needs to be executed for D more cycles.

DRAM reads and writes a block of data, to have access to a certain part of it we need to read a block of data and then extract the desired one. This makes DRAMs less suitable for our design.

Additionally DRAM is slow in comparison to on-chip BRAM. The maximum speed of data transfer between the DRAM and the FPGA is 38.4 GB/s while the speed of data transfer from BRAM is about 20 TB/s. It leads us to attempt a design that uses on-chip BRAM instead of DRAM.

Summary

As illustrated in Figure 3.6, the execution time of this design is short enough to be a good candidate design for simulating the brain cell modeling. It has to be mentioned that this design is tested for a small application (only a memcopy with no computations). Adding the brain modeling computations to the design, would make the kernel more complicated and adding more pipeline stages. In addition, it would lead to incorrect results due to the fact that MaxCompiler does not manage delays which come from pipelining. This problem has been resolved in the Full Pipeline design described later in this chapter.

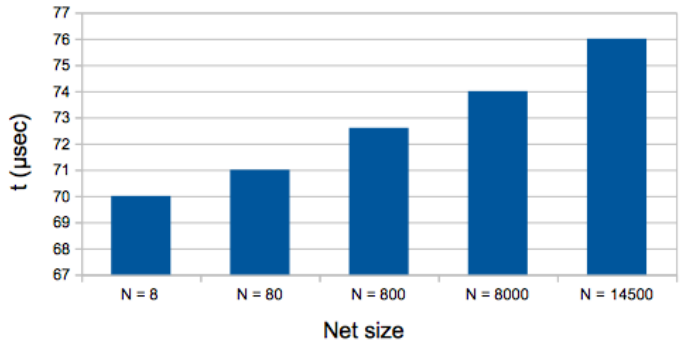

Figure 3.6: The results for DRAM memcopy test Time is in micro seconds.

In this every simple memcopy 24 floating-point numbers for each cell are written into DRAM by the host. The Kernel reads floating-point numbers and copies them into another segment of the DDR. For a network of 800 cells (just data communication and no cell computations) the time for set-up, run and tear-down is about 72 µSec which is higher than our 50 µSec timing constraint.

3.1.3 Using on-chip BRAM

In this design, BRAM embedded in the FPGA, is utilized to store intermediate results and feed them back in the DFE at each simulation step. BRAM (also denoted as FMem, Fast Memory) is an on-chip static RAM which can hold several MBs of data. Virtex-6 (XC6VSX475T) FPGA that has been used in MAX3424A, has 1064 or 2128 dual-port Block RAM of 36 Kbits or 18 Kbits respectively [18]. Each block RAM has two completely independent ports that share nothing but the stored data.

When using FMem, there is no need to move intermediate data in and out of the chip. Using FMem will increase the performance of the application by keeping data transfers inside the DFE.

- FMem

The kernel needs to be fed with active streams to stay alive. As all inputs and output to FMem are streams it can be utilized as a source of stream to feed the kernel.
There are three steps to utilize a block of FMem i.e. allocating, writing and reading, it is shown in listing 3.6. To allocate FMem a basic declaration is used, it takes two parameters, the type of the data which is stored into the FMem and the number of items i.e. the depth of the memory.

```
Memory<DFEVar> cellP0 = mem.alloc(dfeFloat(8,24), N);
cellP0.write(inputAddress, data, enable);
DFEVar dataP0 = cellP0.read(outputAddress);
```

Listing 3.6: Fast memory allocation.

The `mem.alloc` function returns a memory object that can be used to access the FMem. A Java generic DFEVar parameterizes the memory and indicates the type of stream that is saved in the FMem.
First line in listing 3.6 shows an allocated memory with a depth of N and the type of floating point. It is called cellP0 and can be read and written independently.

To write into a block of FMem there should be an address of DFEVar, a data of proper type and an enable of DFEBool.

To read from a block of FMem there should be an address and a DFEVar to store data. In the third line, the data, which has been stored in the `outputAddress` of the cellP0, is restored into the `dataP0`, which is a variable of DFEVar. Reading and writing one parameter into or from FMem happens in one cycle.

Figure 3.7 illustrates the structure of a set of BRAMs to keep parameters of a network of N cells. As can be seen all parameters that have the same character are kept into one BRAM, for example dendrite voltage of cell number 0 which is the first parameter of a cell is stored in the first address of `cellP0` and the dendrite voltage of the cell number 1 is stored in the second address of `cellP0`. This structure enables us to extract easily all dendrite voltages to compute the IC.

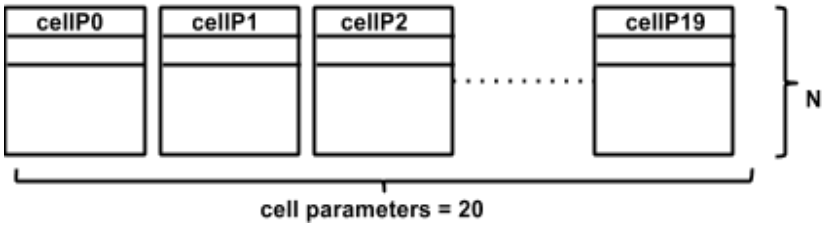

Figure 3.7: BRAM structure to store intermediate results.

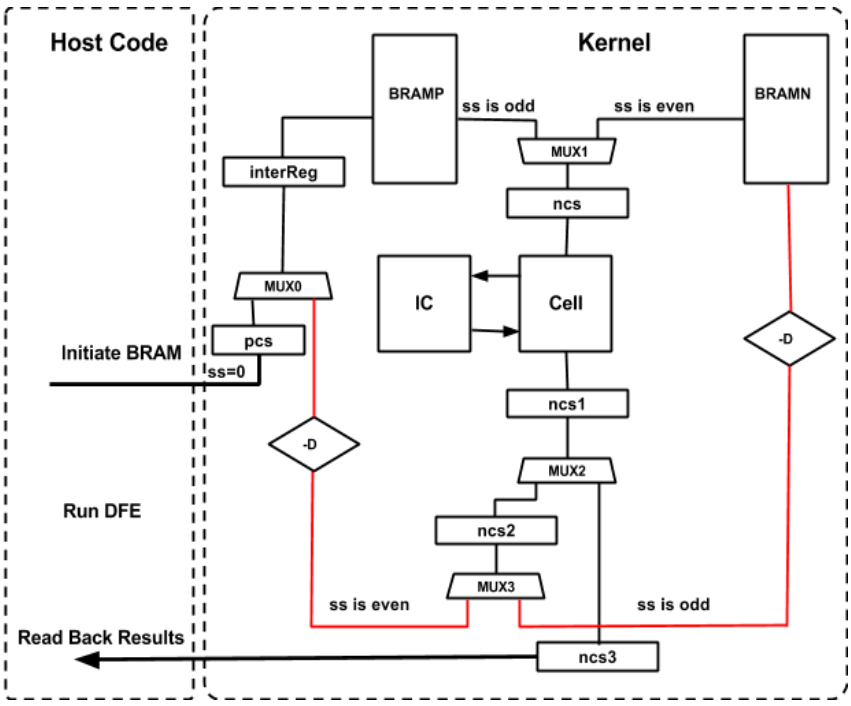

Figure 3.8: Illustrates BRAM solution in details.

Figure 3.8 illustrates the block diagram of our BRAM design, in which 20 blocks of FMem have been used as input blocks i.e. BRAMP and 20 as output blocks i.e. BRAMN.

In this design all computations are performed in the DFE and the process of the simulation of a network is mainly controlled by a simulation step counter i.e. `ss` or so called `simStep`.

```
DFEVar simStep = chain.addCounter(steps, 1);
```

BRAMP receives initial parameters for a network of cells at simulation step zero, `MUX0` which is a ternary if statement, selects input stream which comes from the host to initiate the BRAMP in the first simulation step .

The input stream of the kernel is an array (`pcs`), which is initialized in the beginning of the computations.

```
interReg = simStep> 0 ? ncs2 : pcs ;
```

A ternary if statement chooses the first argument after question mark if the condition placed between "=" and "?" is true and chooses the second argument if the condition is false.

In the first simulation step (`simStep = 0`) initial parameters coming from PCIe for a complete network are written into the BRAMP without being read by the kernel yet. In the next simulation step when the `simStep` is odd (`simStep = 1`) initial values are written into the input register which is called `ncs`. The design sends `ncs` contents to the computational part and new parameters which have been computed by the computational part are stored into the output intermediate register `ncs1`. Finally the content of `ncs1` are written into the BRAMN. In each simulation step the computational part iterates N times to compute a complete network of N cells. In this way, all new parameters in the first step will be computed and the results are written into the BRAMN.

After computing a complete network in `simStep = 1`, `simStep` is incremented by one and it will become an even number, now the route for moving data is changed to: from BRAMN to BRAMP. To control the correct route for the streams, the simulation step counter controls `MUX0`, `MUX1`, `MUX2` and `MUX3`. `MUX2` steers parameters to the host after a complete set of simulation steps are done. The final parameters move to the host code from output register `ncs3` through an output port:

```
io.output("newCellState", ncs3, VectorType,finalstep);
```

The manager, like all other output streams that are connected to the host, connects the output `newCellState` to the host code; it is defined as a DFELink as below:

```
DFELink newCellState = addStreamToCPU("newCellState");
```

IC computation is actually a part of cell computation but to optimize the application and achieve better performance, two separated kernels are defined; they operate in parallel to compute IC value and the rest of the cell computations. A separate kernel, which is called Kernel IC (`kIc`), computes IC; it is defined in the manager and is connected to the `kCell` (Kernel Cell).

To move dendrite voltages from `kCell` to the `kIc` and send back IC value from `kIc` to the `kCell` there are two controlled ports connecting two kernels as is shown in the listing 3.7.

```
DFEVar icin = (simStep> 2);
DFEVar vdout = (simStep>0);

DFEVar IC = io.input("IC", dfeFloat(8,24), icin);
…
io.output("vDend",newCellState1[0], dfeFloat(8,24), vdout);
```

Listing 3.7: ports kIc to/from KCell.

In doing so, while the `kCell` is busy with cell computations, `kIc` is busy with gathering dendrite voltages and compute the IC values.

• Bottlenecks

The drawback of this design is that it does not take advantage of full pipelining. This design sends one cell for being computed and waits until the result is ready and then sends next cell. Since the application has a pipeline of 334 stages only for cell computation except IC, and due to the limited FPGA resources, there cannot be a network size bigger than 32 cells implemented on the FPGA. Under these circumstances, in this design the IO network is computed in groups of 32 cells which have the latency of 334 cycles. In this case we take advantage of less than 10 percent of the pipelining. This is depicted in Figure 3.9 which shows a kernel with four pipeline stages [16].

• Summary

In this approach BRAM is used to store intermediate results to send them back as inputs to the beginning of the next simulation step. Although, BRAM is fast and provides high bandwidth, the design is not able to take advantage of pipelining. Additionally, due to the complex structure of multiplexers, arrays, ternary ifs and BRAMs the design is more difficult to debug. To overcome these issues the next solution is introduced.

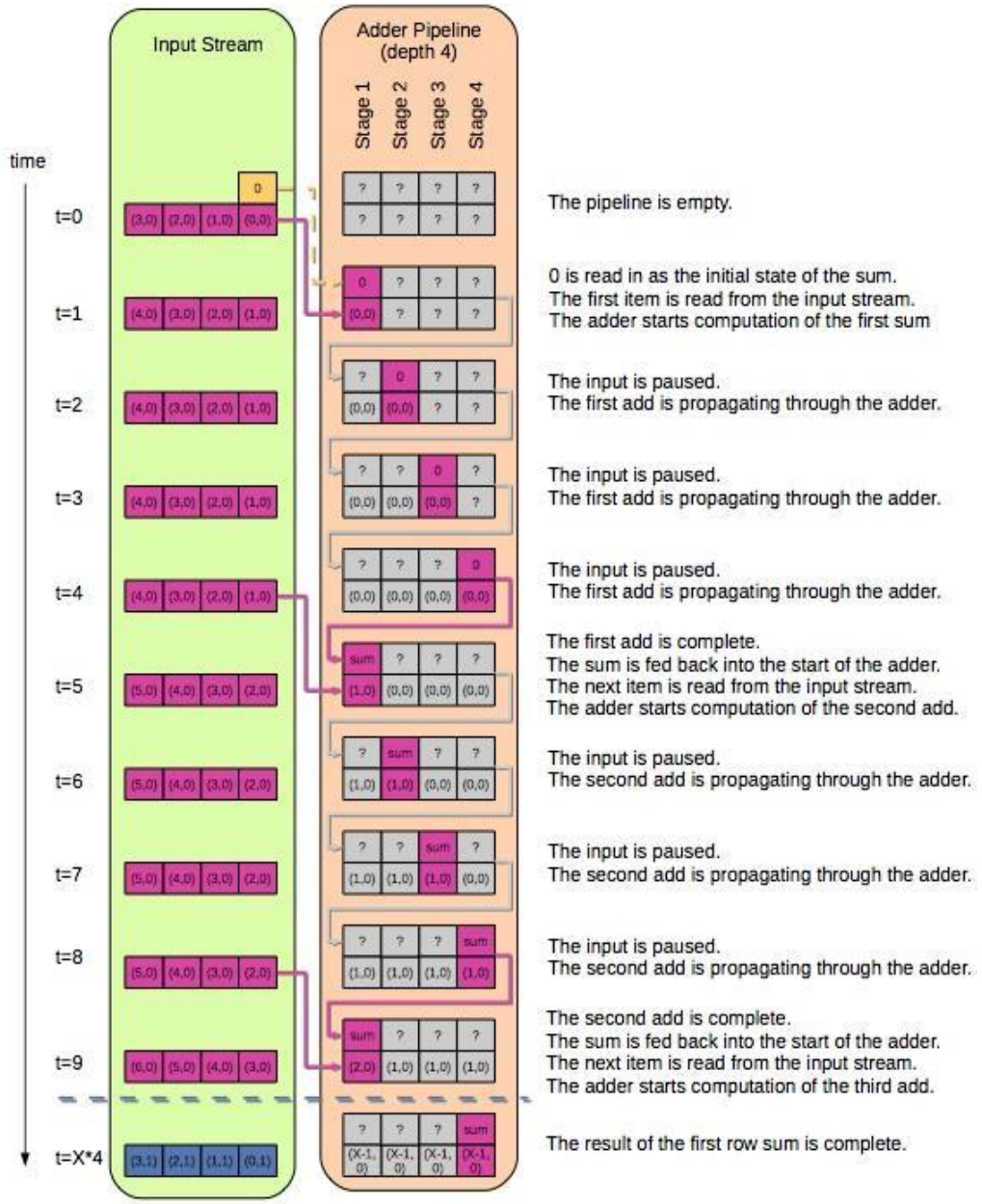

Figure 3.9: underutilized pipeline for our application there is #pipes stages [16].

3.1.4 Full Pipeline

To overcome the problem in the BRAM design (limitation of pipeline usage) we need to have a new design in which the DFE is able to fetch one cell each cycle instead of one cell each `PIPES` cycles (where `PIPES` is equal to the number of pipeline stages) like in the BRAM design. In this approach a network of cells are sequentially pushed to the DFE to be processed, if the network size is the same as the number of pipeline stages, then it takes full advantage of pipelining. But it should be kept in mind that more cells need more time to be processed. The real-time of 50 µSec is the restriction on applying more cells.

Cells in a network are independent from each other while cells in consecutive simulation steps are dependent to each other. If the network size is large enough, while the first cell is being

simulated, the pipeline can be fed by other cells, in this way pipeline stages can be fed consecutively every cycle.

The problem we need to address is to increase the capacity of IC computation. Listing 3.8 illustrates the optimized IC code. In optimized IC any operation that is common for all iterations have been removed outside the loop [1]:

```
for(inti = 0 ; i< #cells ; i++)
{
oneSliceAdd[i] = constant.var(dfeUInt(9),i);
vdiAdd = icCond_t * (cellLength/unFac) * sliceCnt_t +
oneSliceAdd[i].cast(ramAddType);
V = vdp - vdi;
f = V * KernelMath.exp(-0.001 * V * V);
Facc = f + Facc;
Vacc = V + Vacc;}
IC =CONDUCTANCE * (.8 * Facc + .2 * Vacc);
```

Listing 3.8: Modified IC computation.

The amount of FPGA resources needed to implement the IC computation is lower than in previous designs. Non-modified IC computation consumes 44% of FPGA resources for a loop with 32 iterations while after the optimizations described in Listing 3.8 the IC computation utilizes 34% for a loop of 40 iterations. By comparing Listing 3.8 and Listing 3.1, it can be observed that the modified IC needs fewer computations.

- ## Unrolling IC computation

Reducing the required area from 44% to 34% for the IC computation is not enough to cover the required number of cells for a network of hundreds of cells. The solution is to only partly unroll the IC computation. By unrolling we mean that to repeat in space a smaller for-loop for several times to complete the for-loop in listing 3.8. For example if a for-loop of 40 iterations is implemented, for a network of 400 cells, the for-loop in listing 3.8 has to be unrolled for 10 times, each time computes 40 cells and passes `Vacc` and `Facc` to the next 40.

The implemented partly unrolled IC utilizes an array to keep intermediate results of `Vacc` and `Facc` after each iteration, add all elements of the array to each other to find the final `Vacc` and `Facc`. In the end, IC value is computed by the last line of Listing 3.8.

Designs with partly unrolled IC are able to simulate a few hundreds of cells and the usage of pipeline is increased. For example for a network of 600 cells there are 800 pipeline stages, consequently, the usage of pipeline is 6/8 compared to 32/334 in the BRAM design.

# of Cells	80	160	320	400	440	480	520	560	600	640	680	720	800	1120
Unrolling Factor	2	4	8	10	11	12	13	14	15	16	17	18	20	28
# of Pipeline Stages	644	668	716	740	752	764	776	788	800	812	824	836	860	982
# of iterations of the IC loop	2	4	8	10	11	12	13	14	15	16	17	18	20	28
# of required cycles per simulation step	884	1468	3596	5140	6032	7004	8058	9190	10400	11692	13064	14516	17660	33462

Table 3.2: The network size and related unrolling factor and pipeline stages for a for-loop of 40.

Table 3.2 shows the dimension of the network and the number of IC-loop iterations needed when partly unrolled (that is unrolling 40 IC-loop iterations). Required cycles are the number of cycles for each dimension that are needed to simulate one complete network. These numbers are described later on, in the Implementation section.

The optimized number of for-loop iteration we reached is 40, less than 40 decreases resource utilization, and more than 40 decreases the operating frequency. For example for the same number of cells (720 cells) a for-loop of 48 iterations can operate at 180 MHz while one with 40 runs at 205 MHz. Having a for-loop of 40 iterations, about 88% of the resources are used while 48 iterations for-loop, utilizes 98% of resources.

- ## Implementation

In this approach there is only one kernel for all needed computations. This means that the cell computations are performed in sequence with the IC computations. Execution time is split into two parts, in the first part, all parameters except IC values are computed and in the second part IC values are computed. In the first part all Dendrite voltages are provided and are kept into a BRAM, called `vdRam`. In the second part `vdRam` is read and IC values are computed, each IC value is written into a BRAM called `icRam`. In the next iteration all parameters are read from the previous iteration using stream-offset and IC values are read from corresponded `icRam` address.

In this design there is a port from the host to the kernel to send the number of steps. Adding this port enables us to run the simulation for different numbers of simulation steps without rebuilding the design. Since the number of cells and the number of unrolling factor requires modifications of the design and affects the hardware, it is not possible to apply these variables from the host; they need to be hard coded and by changing any of them requires to rebuild the design.

Figure 3.10 illustrates the design, which has been implemented to maximize the benefit of pipelining.

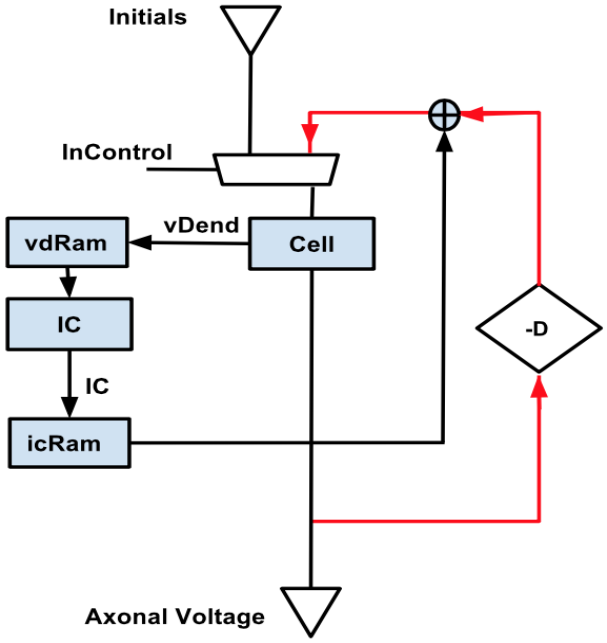

Figure 3.10: Full Pipeline schematics.

To control the process there are three enable signals. The first one is `vdwrEn` by which writing the Dendrite voltages (`vd`) in the appropriate addresses of `vdRam` is controlled. According to the compiler the first `vd` is generated after a number of cycles equal to the number of pipeline stages (*# of pipes*), thus after *# of pipes* cycles `vdwrEn` is enabled and stays active for the number of cycles equal to the number of simulated cells to all `vd`s are being written into `vdRam`. From this point onwards, all `vd`s are written and the second part of kernel starts processing. After all `vd`s are written, the second enable is activated, `vdreEn`, it shows that previously written `vd`s, have to be read to IC phase. In this part in each number of cycles equal to the unrolling factor (`unFac`), one IC value is generated and is written into the `icRam`. After (`unFac * network size`) cycles, all IC values are generated and the second part of the kernel is performed.

The third enable signal, `cellCond`, shows that IC values and the rest of parameters need to be fetched to compute cells' parameters. The diagram in Figure 3.11 illustrates the process and how enable signals control different inputs and outputs:

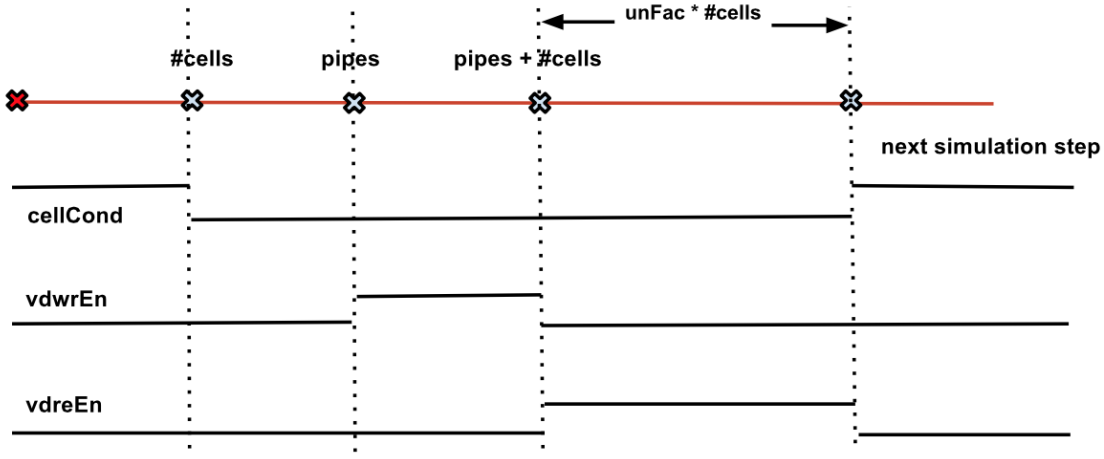

Figure 3.11: enables and process for one simulation step, red line shows the number of cycles.

There are four address generators to compute the write and read addresses to and from `icRam` and `vdram`.

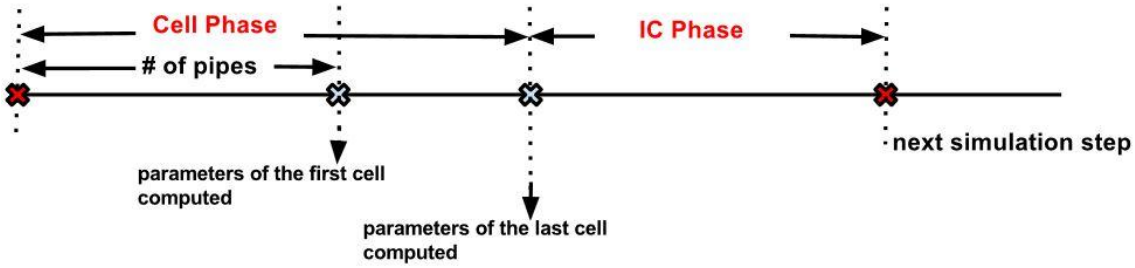

Figure 3.12: computing phases of simulating a network of IO brain cells.

As shown in Figure 3.12, needed parameters for a simulation step are provided from the previous step, IC value is generated during IC phase (the phase where the IC value is computed) and the others are computed during cell phase (the phase where the cell parameters are computed which is equal to the *# of pipes* cycles + network size). Cell parameters are transferred to the next simulation step using stream offset and IC values go to the next simulation step by reading IC values from the `icRam`. Generated parameters in the cell phase are kept in "new cell state" array, which is a DFEVar, and then the contents of "previous cell state", is replaced by the contents of "new cell state" to provide input stream for the next step.

Figure 3.13, illustrates the process of fully-used pipeline stages for a simple adder with four stages of pipeline. IO brain model has pipeline stages according to the Table 3.2.

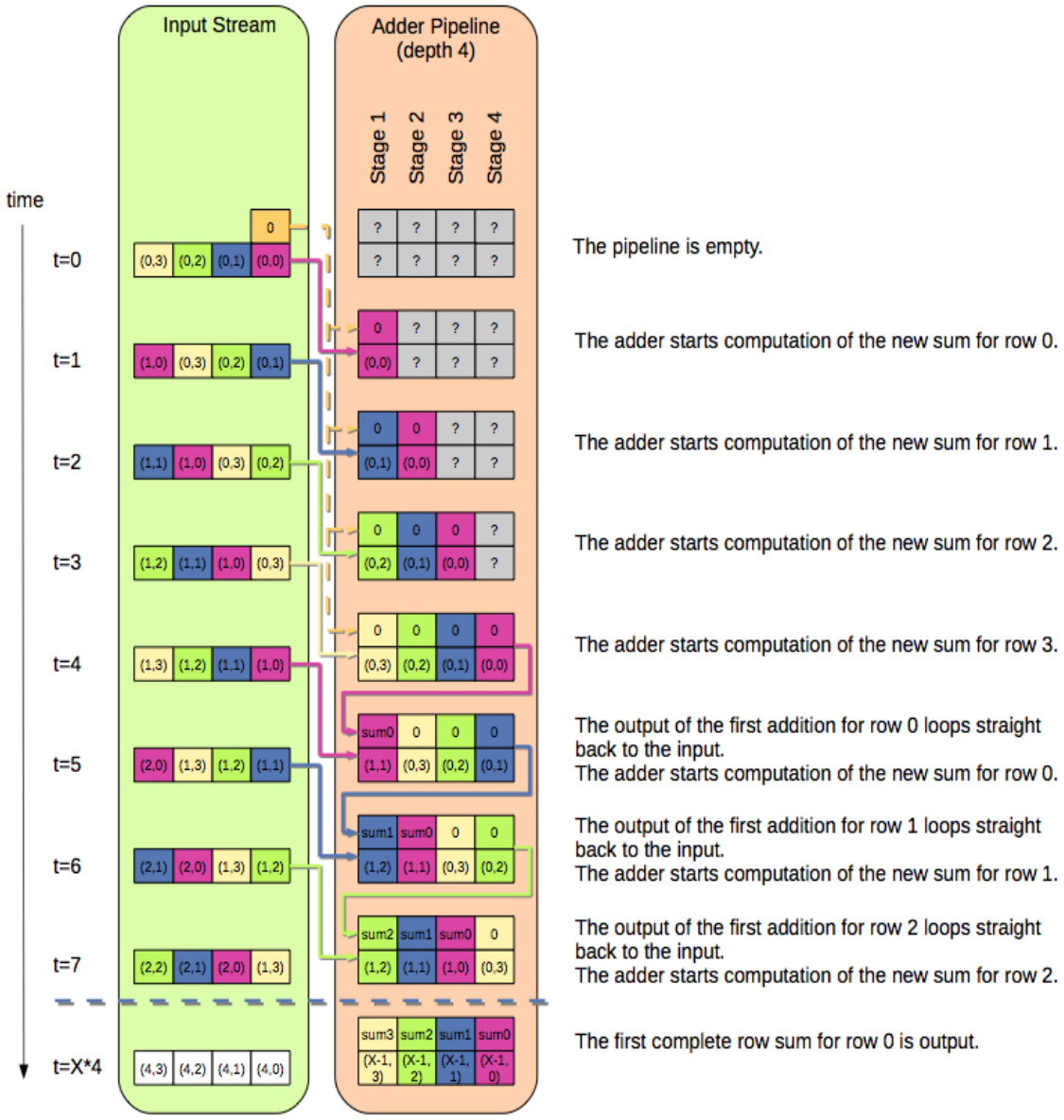

Figure 3.13: the process for a simple adder (cell kernel implementation has 334 stages) [16].

3.2 Conclusion

The goal of the thesis is to accelerate the IO brain cell model with the demand of full connectivity between all cells in a simulated network. In this chapter we described four different designs, which have been implemented on the Maxeler machine. Each design has its own advantages and disadvantages. The last design that uses stream offset in a sequential way is the best one and as can be seen from the results (Chapter 4) meets the timing constraints for a network of 400 cells. It does not transfer intermediate results through PCIe or DRAM, which are much slower than the internal BRAMs.

In addition, using BRAM is significantly simpler and more flexible than DRAM. Using PCIe has a cost of set-up and tear-down time of the FPGA for each simulation step and that is about 25 milliseconds. In addition, data transfer through PCIe, which is about 2GB/s, is much slower than the BRAM connection, which is about 20TB/s.

The key point in the our last design is that all data transfers and computations take place within the same kernel using very fast data transfer techniques provided by FPGA.

Chapter 4: Evaluation

This chapter presents the evaluation of our last design, described in Chapter 3, implemented on a Maxeler MAX3424A machine. We further make a comparison between our approach and design that uses Vivado HLS. Finally, this chapter provides a qualitative comparison between the brain cell simulation output produced by our Maxeler implementation, the output of the Vivado design as well as the output of a software implementation (C code and MATLAB) running on a general purpose computer.

4.1 Parameters

There are four design parameters for the brain models.

A. `netSize`: The number of cells in a simulated network (known as dimension).

B. `unFac`: The unrolling factor of the IC for-loop. `unFac` 40 in a for-loop of 400 (Network Size 400) means that 40 iterations are unrolled and need to be executed 10 times to complete to 400 iterations.

C. `pipes`: the number of cycles that the kernel needs to execute the computations of one cell.

D. `cellParameters`: the number of parameters which represent the behavior of a brain cell.

4.2 Results

The accelerator achieves real-time execution for a network of 400 brain cells with 100% connectivity at an operating frequency of 110 MHz. Table 4.1 shows the results for the Full Pipeline design. It illustrates the execution times of the design for different network sizes, their operating frequency and unrolling factor. Second row shows the execution time for a complete simulation of 120,000 steps in seconds while third line contains the execution time of a single simulation step of the network in µSec. As can be seen, each network according to its size operates up to a specific frequency. The frequency drops slightly as the number of cells increases.

Our design can simulate a network of 400 cells for 120,000 steps in 5.71 seconds at 110 MHz. This translates to 47.5 µSec per simulation step.

N	80	160	320	400	440	480	520	560	600	640	680	720	800	1120
T/S(s)	.920	1.3	4.13	5.71	7.29	8.02	9.78	10.51	12.54	14.1	15.7	17.5	21.2	40.1
T/N(µs)	7.66	10.8	34.4	47.5	60.7	66.8	81.5	87.58	104.5	117.5	130	145	176	334
F(MHz)	120	140	130	110	105	105	100	105	100	100	100	100	100	100
unFac	2	4	8	10	11	12	13	14	15	16	17	18	20	28
Time-Score	0	0	0	0	0	0	0	0	0	0	0		0	0

Table 4.1: Execution time vs. operating frequency and unrolling factors.

Figure 4.1 shows the execution time for different network sizes in µSec. For each network size, several frequencies were tested to find the maximum one without timing errors. The most effort has been put on the sizes of about 400 cells. To have an estimation of open, close and sending initial parameters for a network of 400 cells, it has been executed for different simulation steps and the results are illustrated in Figure 4.2. According to the Table 4.2 for only one simulation step, the execution time is 32.2 mSec. The execution time of a network of brain model, without opening and closing the FPGA and data transfer, is the number of cycles, which

are needed for a complete execution, divided by the operating frequency. For a network of 400 cells it is 46.72 µSec (i.e. according to the Table 3.2, 5140/110 MHz). For a complete simulation of 120,000 steps the delay of opening and closing the device and the time for data transfer of each simulation step is 0.8 µSec (i.e. 47.5 – 46.7 = 0.8).

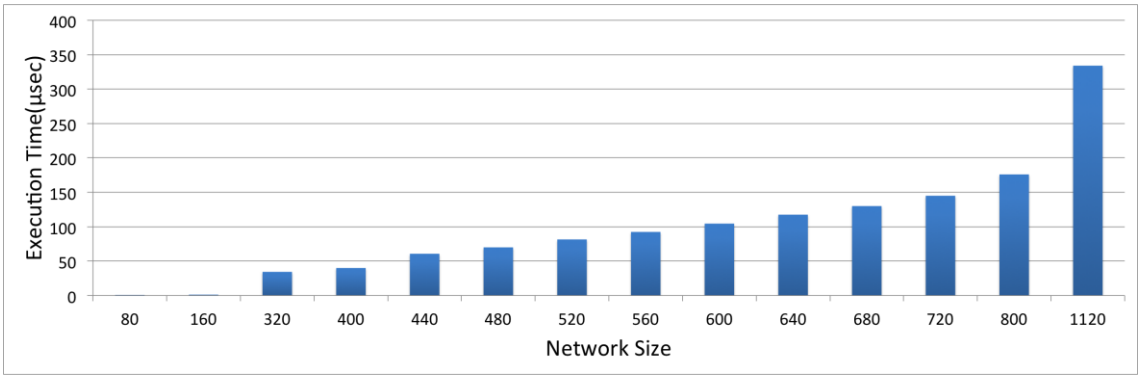

Figure 4.1: Execution times for fully pipelined solution.

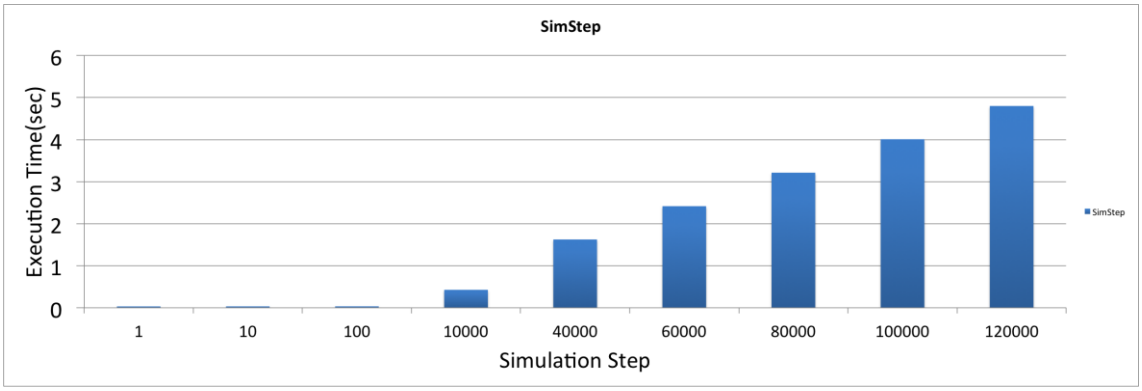

Figure 4.2: execution times for a network of 400 cells for different simulation steps.

Steps	1	10	100	10000	40000	60000	80000	100000	120000
Time (Second)	0.0322	0.03327	0.03432	0.4266	1.6243	2.4168	3.2137	4.0088	4.80

Table 4.2: Execution time for a network of 400 cells and different simulation steps.

4.3 Design space observations

The machine used to accelerate the IO model of brain cells is a MAX3424A of Maxeler Technologies with 24GB DRAM on its Virtex-6 (XC6VSX475T) FPGA board. According to Xilinx documentation, Virtex-6 XST is one of their largest FPGAs with fast IO, implemented by 40 nm technology [18].

The brain model requires a large number of floating point computations. Such computations need a large amount of FPGA resources which of course varies depending on the dimension of the network. In other words, larger networks need more resources. Table 4.3 shows final resource usage for a network of 400 cells.

Block memory (BRAM18)	475 / 2128 (22.32%)
Logic utilization	255135 / 297600 (85.73%)
LUTs	226764 / 297600 (76.20%)
Primary FFs	232751 / 297600 (78.21%)
Secondary FFs	61177 / 297600 (20.56%)
Multipliers (25x18)	1004 / 2016 (49.80%)
DSP blocks	1004 / 2016 (49.80%)

Table 4.3: final resource usage for a network of 400 cells.

Our last design, Full Pipeline, consumes 475 Block RAM each block storing 18 kb, for a network of 400 cells. Since there are only two blocks of BRAM, which are explicitly declared to keep Dendrite voltages and IC values, the rest of used BRAM blocks are used to transfer intermediate results between consecutive steps. `icRam` and `vdRam` are declared in the code as follows:

```
Memory<DFEVar>icRam = mem.alloc(dfeFloat(8,24), netSize);
Memory<DFEVar>vdRam = mem.alloc(dfeFloat(8,24), netSize);
```

4.4 Comparison

In this section a comparison is made between our solution implemented in the MAX3424A Maxeler machine and an FPGA design implemented with Vivado HLS. The main goal of both solutions is to improve the performance of brain cell model, both have developed the same model and both provide full connectivity in the network. There are also results of executing the model on a general purpose computer with a 4-core Xeon 2.66GHz and 20GB RAM [1]. The performance of our design is about x3041 better than MATLAB implementation, about x253 better than C code implementation and about x4 better than Vivado HLS implementation at a network size of 400 cells. It is possible to do this comparison in performance for any network size which both designs can support, such as 480. For this network size, our design is about x3655 better than MATLAB implementation, about x149 better than C code implementation and about x5 better than Vivado HLS implementation. Table 4.4 shows these speed-ups. Table 4.5 shows execution time for different network sizes implemented in our Full Pipeline design, C code implementation and the implementation uses Vivado HLS.

Design	**Improvement**
MATLAB implementation	3041.91
C Code implementation with single float	253.01
FPGA Accelerator using Vivado HLS	4.16

Table 4.4: Performance improvement of Maxeler *MAX3424A design comparing to MATLAB, C code and Vivado HLS implementations.*

N	80	96	160	192	288	320	384	400	440	480	520	560
Maxeler	7.66		10.88			34.4		47.5	60.7	70	81.5	92.41
Vivado		48.54		104.15	173.42		254.22			346.54		
PC (C code)		583.55		1903.7	4014.7		6863.2			10430.45		

Table 4.5: Execution time for different size on Maxeler, Vivado and GPP.

N	576	600	640	672	680	720	768	800	864	960	1056	1120
Maxeler		104.5	117.5		130	145		176				334
Vivado	45.38			565.74			692.62		821.2	970.94	1132.38	
C code	14841			19811			25705.6		32556.85	39786	47803	

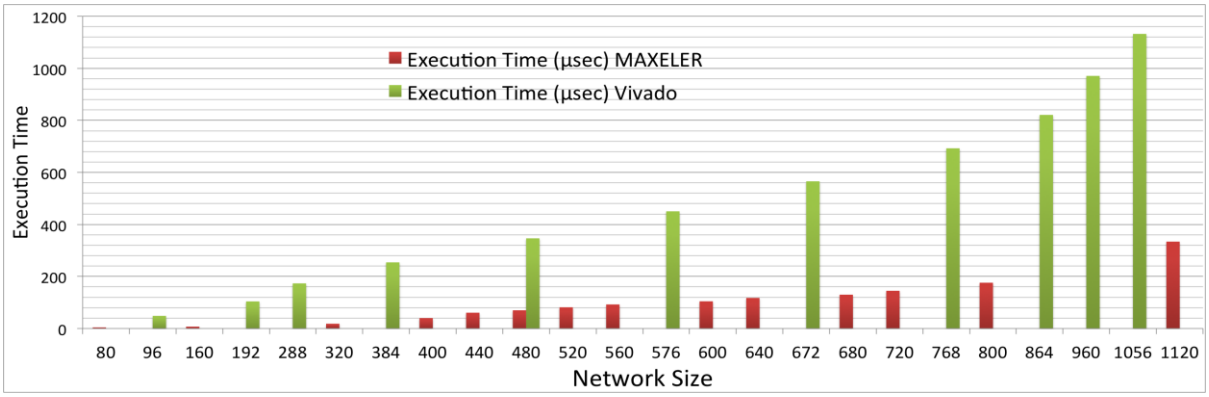

Figure4.3: Execution time Vivado vs. Maxeler machine.

Figure 4.3 shows execution time of our last design with columns in red compared to execution time of the solution uses HLS Vivado with columns in red.

Table 4.6 shows the break down to the processing throughput dedicated in different arithmetic and logical operations for a network of 400 cells.

Op type	Rest of computations for 1 cell	IC for 1 cell	Rest of computations for 400 cells	IC for 400 cells
<=	124	96	49600	38400
Negate	7	95	2800	38000
<	2	0	2	0
Truth-and	96	0	38400	0
==	192	0	76800	0
+	276	285	110400	114000
-	39	95	15600	38000
/	50	95	20000	38000
*	76	475	30400	190000
Total	862	1141	344800	456400

Table 4.6: Floating point operations for one cell and for a network of 400 cells.

The maximum throughput for a network of 400 cells operating at 110 MHz is 801200 FLOPS.

4.3 Result verification

To evaluate the correctness of the simulation results, extracted from our Maxeler implementation the outputs of our implementation have been compared against the reference C code output. Since all cells are initialized with the same parameters at the beginning of the simulation, all cells should have the same state in each simulation step. The equality of cells'

state is verified in the host code, if there is any difference, an Error Flag will be set to "1" and the position of the cell in the network is kept in an Error Address register.

Figure 4.4 and 4.5 show `vAxon` (Axonal voltage), which are generated using C code and Maxeler MAX3424A respectively. The test-bench simulates about 6 seconds of brain modeling, which is equal to 120,000 simulation steps (each step 50 µSec). After 20,000 simulation steps, an input signal (`iApp`) with a level of 6 mVolt is applied to all cells for a duration of 500 simulation steps. As can be seen in Figure 4.4 and Figure 4.5 the cells' state before returning back to their state, respond by producing a spike.

As can be seen both figures are quite similar, Figure 4.6 plots the absolute deviation. Maximum deviation between two results happens at the moment of the spikes due to the `iApp` changing. According to the Figure 4.6, maximum deviation is 0.14 mV. This occurs when `vAxon` reaches -31 mV that yields an error of 0.92%. This error does not affect the simulator functionality and proves that Maxeler results are acceptable [1].

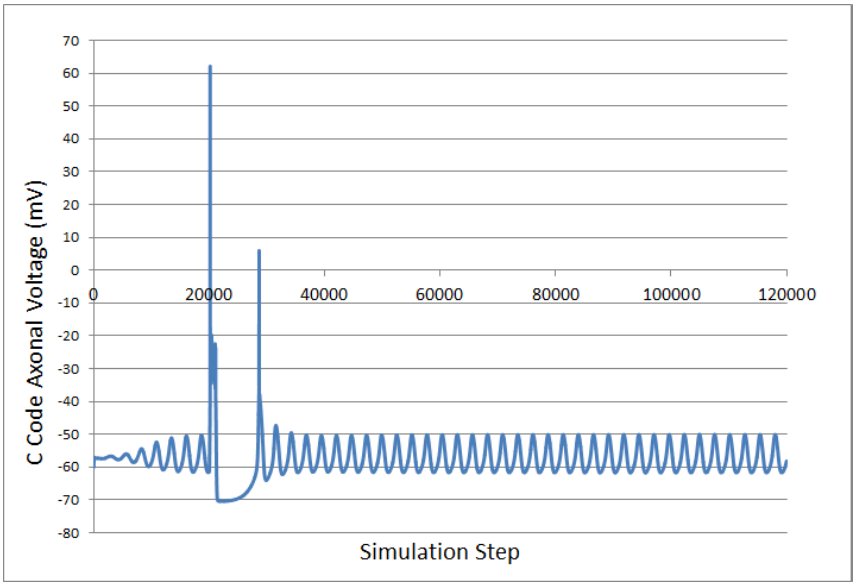

Figure 4.4: Axon Voltage simulated by C code.

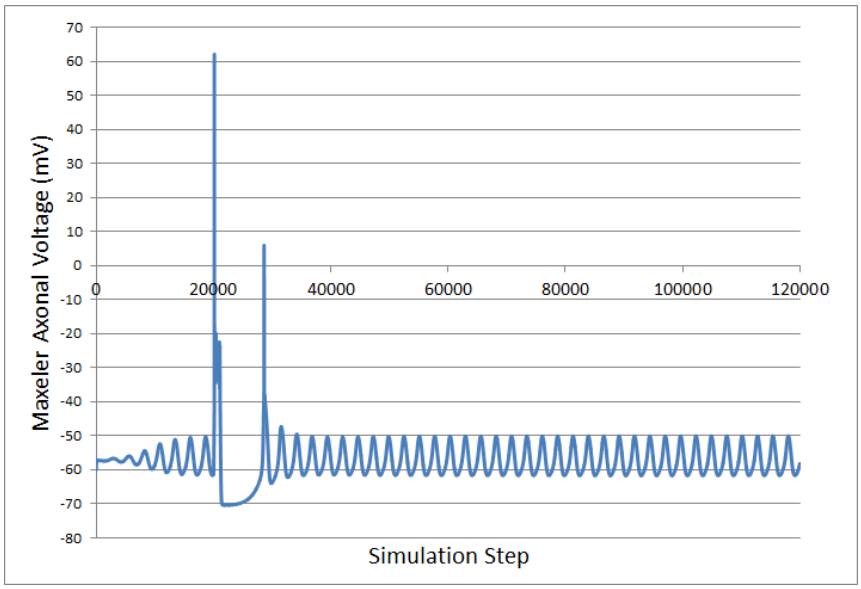

Figure 4.5: Axon Voltage simulated by Maxeler.

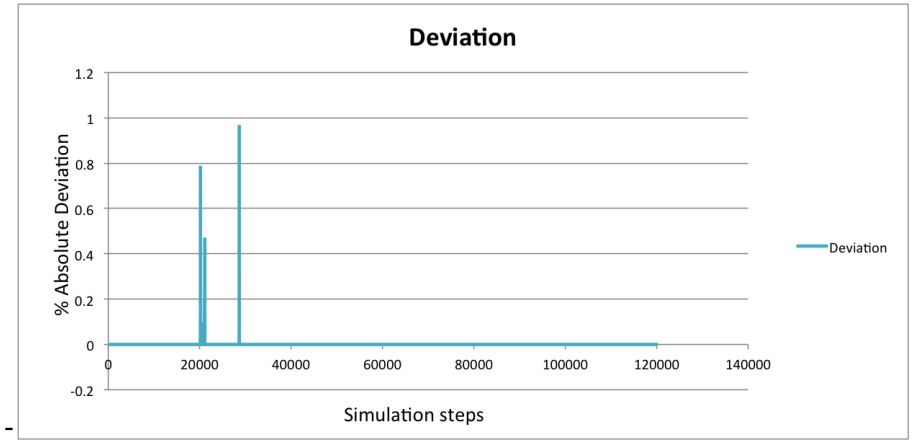

Figure 4.6: Absolute deviation (Msaxeler vs. C Code).

4.4 Conclusion

Our last design discussed in Chapter 3 achieves real-time execution for a network of 400 brain cells with 100% connectivity at an operating frequency of 110 MHz. It executes a complete simulation of 120,000 steps in 5.71 seconds which translates to 47.5 µSec per simulation step. Our design improved the performance of the MATLAB, C code and Vivado HLS simulations of the IO brain cell model, by about x3041, x253 and x4 respectively.

Correctness of the results generated by the Maxeler is evaluated by comparing them against the results of the C code implementation. Results from Maxeler have a maximum deviation of %0.92 from the results of C code that is less than the acceptable maximum deviation.

Chapter 5: Conclusion

The detailed conclusions and future work are discussed next. We first present the summary and thesis contributions and then suggest future directions.

5.1 Summary

The overall goal of this thesis was to improve the performance of a brain cell model using reconfigurable acceleration. This model was first developed in C by a group of researchers to model the behavior of the IO brain cells [1]. We accelerated the simulation of the IO cell using Maxeler machine.

First we analyzed the C code and understood the computations needed. To go through the application, we used some tools like Netbeans and GPROF to understand its arithmetic functions, the dependences between different computations, memory utilization, CPU usage, etc.

Since the application was planned to be executed on a Maxeler machine, we further studied the architecture of the machine and its tool chain (i.e. MaxCompiler). In doing so, we ported the application on the machine after recoding it in JAVA.

Various alternative designs were implemented on the machine were studied and evaluated to see which one suits best our timing constraints. More precisely, the following designs were developed: PCIe, DRAM, on-chip BRAM and Full Pipeline. Finally a fully pipelined "Full Pipeline" design was chosen as the best way to accelerate the application.

5.2 Thesis contributions

The targeted thesis goal which was to accelerate the simulation of the IO model has been achieved by using a Maxeler machine. We successfully improved the performance of simulation of the IO model by 4.16 times more than the Vivado HLS implementation. The architecture of the design allows two different parts of the application to execute sequentially in time (i.e. time division) while they are fully pipelined. The implementation is a reconfigurable design since one can change design parameters such as network size, cell input current and number of simulation steps to observe different results.

5.3 Future work

The design can be improved in two ways; the first one is to increase the performance by the two different kernels operating in parallel. By doing so, there is one kernel for IC and one kernel for the rest of the cell computations. The IC values are computed and subsequently passe to the Cell kernel. Cell kernel can receive one IC value through FPGA's BRAM and computes the parameters for one cell. Meanwhile the IC kernel can continue processing another IC value. The bottleneck of this approach could be that the IC kernel computations are more time consuming than Cell kernel, although, this can be overcome by making a better synchronization between the two kernels. Figure 5.1 illustrates this suggestion.

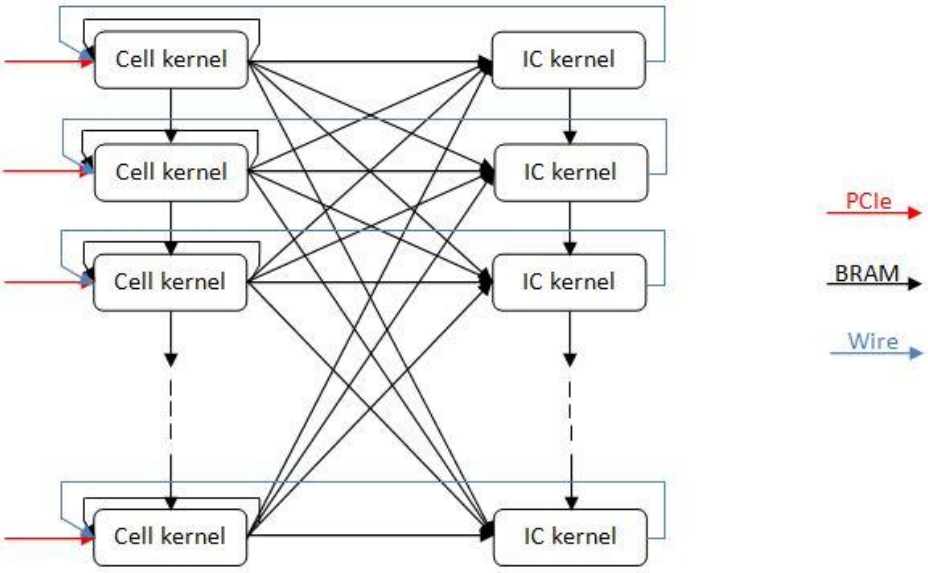

Figure 5.1: Parallel execution of Ic and Cell kernels architecture.

A second improvement could be to increase the number of cells by using a Maxeler machine with a larger number of FPGA devices. Maxeler machine allows connecting several MAX cards to each other through a special connection, which is called MAXRING. By adding more MAX cards to the design it is possible to simulate more cells. One should consider communication latency between different MAX cards since it will affect the overall performance gradually in case of full connectivity of cells.

Bibliography

[1] Georgios Smaragdos, Sebastian Isaza, Martijn Van Eijk, Ioannis Sourdis and Christos Strydis, "FPGA-based Biophysically-Meaningful Modeling of Olivocerebellar Neurons", 22nd ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), Monterey, California, February, 2014.

[2] Maxeler Technologies, “Multiscale Dataflow Programming, Version 2013.2.1”, June 11, 2013

[3] Chudler, Eric H, "Brain Facts and Figures", *Neuroscience for Kids*,June 20, 2009

[4] N. Carlson, “Foundations of Physiological Psychology”, Simon & Schuster, 1992

[5] Schweighofer N, Doya K, Kawato M. “Electrophysiological properties of inferior olive neurons: A compartmental model”. J Neurophysiol;82(2): 804-17,Aug, 1999

[6] A. L. Hodgkin and A. F. Huxley, “quantitative description of membrane cur- rent and application to conduction and excitation in nerve”, Journal Physi- ology 117 (1954), 500–544

[7] A. Ghani, T. M. McGinnity, L. P. Maguire, and J. Harkin, “AREA EFFICIENT ARCHITECTURE FOR LARGE SCALE IMPLEMENTA- TION OF BIOLOGICALLY PLAUSIBLE SPIKING NEURAL NET- WORKS ON RECONFIGURABLE HARDWARE”, International Conference on Field Programmable Logic and Applications (FPL ’06), August 2006, pp. 1–2

[8] A. Cassidy, S. Denham, P. Kanold, and A. G. Andreou, “FPGA Based Silicon Spiking Neural Array, IEEE Biomedical Circuits and Systems”, Conference, BIOCAS 2007, November 2007, pp. 75–78

[9] B. Schrauwen and J. Van Campenhout, “Parallel hardware implementation of a broad class of spiking neurons using serial arithmetic”, IN PROCEEDINGS OF ESANN’06, 2006, pp. 623–628

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

[11] H. Shayani, P. Bentley, and A. M. Tyrrell, “A Cellular Structure for Online Routing of Digital Spiking Neuron Axons and Dendrites on FPGAs”, ICES ’08 Proceedings of the 8th international conference on Evolvable Systems: From Biology to Hardware, 2008, pp. 273–284

[12] H. Rostro-Gonzalez, J. H. Barron-Zambrano, C. Torres-Huitzil, and B. Gi- rau, Low-cost hardware “Implementations for discrete-time spiking neural networks”, Cinquieme conference plenierefrancaise de Neurosciences Com- putationnelles, 2010.

[13] A. Cassidy and A. G. Andreou, “Dynamical Digital Silicon Neurons”, IEEE Biomedical Circuits and Systems Conference, BioCAS2008, November, 2008, pp. 289–292

[14] Solari, Edward; Congdon, Brad (2003), “Complete PCI Express Reference: Design Implications for Hardware and Software Developers”, Intel, ISBN 978-0-9717861-9-6, 1056 pp.

[15] Maxeler Technologies, “MaxCompilerWhite paper”, February 27, 2011

[16] Maxeler Technologies, “Acceleration Tutorial Loops and Pipelining”, Version 2013.2.1

[17] Maxeler Technologies, “Maxcompiler Optimization Cheat Sheet”, Version 2013.2.1

[18] XILINX, “Virtex-6 Family Overview”, January 19, 2012