FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Simulating Cardiac Dynamics using Maxeler DataFlow Super-computing

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Medizinische Informatik

eingereicht von

Lilly Maria Treml, BSc.
Matrikelnummer 01528458

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Dr. Ezio Bartocci
Mitwirkung: Dipl.-Ing. Haris Isakovic

Wien, 9. Oktober 2018

Lilly Maria Treml	Ezio Bartocci

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Simulating Cardiac Dynamics using Maxeler Dataflow Super-computing

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Medical Informatics

by

Lilly Maria Treml, BSc.
Registration Number 01528458

to the Faculty of Informatics

at the TU Wien

Advisor: Dr. Ezio Bartocci
Assistance: Dipl.-Ing. Haris Isakovic

Vienna, 9th October, 2018

Lilly Maria Treml

Ezio Bartocci

Erklärung zur Verfassung der Arbeit

Lilly Maria Treml, BSc.
Vienna

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 9. Oktober 2018

Lilly Maria Treml

Danksagung

An dieser Stelle möchte ich all jenen danken, die durch ihre fachliche und persönliche Unterstützung zum Gelingen dieser Diplomarbeit beigetragen haben. Beginnend bei meinen Betreuern Ezio Bartocci und Haris Isakovic die mir mit ihrem Wissen beistanden. Darüber hinaus möchte ich auch Ivan Milankovic von Maxeler Technologies meinen Dank aussprechen, der sich Zeit und Mühe genommen hat alles zu testen was ich ihm gesendet habe. Auch möchte ich mich bei meinen Freunden und meiner Familie bedanken, die mir jederzeit beigestanden haben und an mich geglaubt haben.

Acknowledgements

At this point I would like to thank all those who contributed to the success of this diploma thesis with their professional and personal support. Starting with my supervisors Ezio Bartocci and Haris Isakovic who helped me with their knowledge. In addition, I would also like to thank Ivan Milankovic of Maxeler Technologies, who has taken the time and effort to test everything I have sent him. Also, I would like to thank my friends and family, who were always by my side and believed in me.

Kurzfassung

In der heutigen Zeit sind Computer basierte Simulationen ein wichtiges Werkzeug zur Erforschung verschiedenster Phänomene im kardiologischen Bereich. Diese Simulationen reichen von der Untersuchung einzelner Ionen-Kanäle bis hin zur Prognostizierung von Herzarrhythmien. Aufgrund der Komplexität der zugrundeliegenden mathematischen Funktionen, brauchen diese Simulationen ein hohes Maß an Rechnerleistung.

In dem letzten Jahrzehnt wurde daher ein hoher Aufwand betrieben, Computer basierte Herzsimulationen zu beschleunigen, indem man parallele Algorithmen verwendet, die sowohl handelsübliche Multi-Core CPUs als auch Many-Core GPUs ausnutzen. Diese Algorithmen wurden hauptsächlich in imperativen Programmiersprachen entwickelt, welche die Kontrollstruktur des Programmes vorschreiben. In dieser Arbeit untersuche ich Hardware basierte Beschleuniger, bereitgestellt von Maxeler Dataflow Technology, zur Verringerung der Simulationszeit elektrischer Aktivität in einem Kabel aus Myozyten. Diese Technologie basiert auf dem Datenfluss-Paradigma, in welchem Datenverarbeitungselemente, genannt Pipelines, gleichzeitig arbeiten und so verbunden sind, dass die Ausgabe eines Elements die Eingabe für das Nächste ist. Zusätzlich vergleiche ich die Vor- und Nachteile des untersuchten Ansatzes mit aktuellen State-Of-The-Art Technologien, wie CPU und GPU, in Bezug auf Leistung und Ressourcenverbrauch.

Abstract

Nowadays, computer-based simulation is an important tool to study various phenomena in cardiac biology ranging from investigating the properties of single ion channels to predict the onset of cardiac arrhythmia.

Due to the complexity of calculations, these simulations are generally computationally expensive and resource intensive. In the last decade, there has been a great effort to accelerate computer-based cardiac simulation by implementing efficient parallel algorithms that leverage multi-cores CPU and many-cores GPUs currently available also in common personal computers. These algorithms are generally developed using imperative languages dictating the control flow of the program. In this work I am investigating the use of hardware-based accelerators provided by Maxeler Dataflow Technology to improve the simulation time of the electrical activity of a homogenous 1D cell cable and an isolated single cell. This technology relies on dataflow paradigm, in which the data processing elements, called pipelines, operate concurrently and are connected in a way that the output of one element is the input for the following/next one. In my work I am additionally comparing the advantages and disadvantages of the proposed approach with the current state-of-the-art based on CPU and GPU regarding performance and resource utilization.

Contents

CHAPTER 1

Introduction

1.1 Motivation and Problem Statement

To provide early diagnosis of cardiac arrhythmia computer simulation of the cardiac electrophysiology is used. Despite several approaches of implementations and development of different models, real-time simulation for large models is still not possible yet. Considering these facts, a new approach to come closer to real-time simulation of cells will be presented in this thesis.
The overall aim of this work is to analyze the advantages and disadvantages of the Maxeler Technology implementation in comparison to GPU and CPU State of the Art implementations. Further the specific technical attributes of Maxeler Technology are used to optimize the implementation and analyzed regarding possible trade-offs.
Additionally several formal models of cardiac cells, will be implemented using Maxeler Dataflow Computing technology and also be analyzed with regard to speed and resource efficiency.

1.2 Aim of the Work

The aim of this work is to gather information about the advantages and disadvantages of using Maxeler Technology to simulate cardiac cells. Therefore a given model of the cardiac electrophysiology, using Reaction-Diffusion-Equation to describe the electrical stimuli, will be re-implemented using Maxeler Dataflow.

This will allow a detailed analysis of the resource efficiency and probable speed optimization of the implemented simulation. Furthermore, to get statistically relevant results, the values will be compared to known values of GPU and CPU implementations. Thus, the following research questions will be answered:

- How to compute the solution of partial differential equations using Maxeler Technology and finite differences?
- What are the overall disadvantages and advantages of using GPU instead of CPU to simulate cardiac cells?
- What are the overall disadvantages and advantages of Maxeler Technology compared to GPU/CPU?
- Does the usage of Maxeler Technology show significant improvements in speed and resource efficiency compared to other technologies?
- Is there a trade-off between precision and efficiency when using Maxeler and if so, what is the trade-off?
- What are the differences in the models used?

Possible ideas for further research, especially regarding real-time simulation, will be suggested in this work. Additionally this thesis will provide the relevant background information regarding Maxeler Technology, cardiac arrhythmia and other implementations.

1.3 Methodological Approach

1. **Literature Review**
   Available research to provide relevant background information to back up this research approach.
2. **Maxeler Technology**
   The GPU-based implementation needs to be re-implemented using Maxeler Technology.
   To achieve this, there will be several steps of the implementation. First, the equation will be implemented using Maxeler. The next step is the simulation of an isolated single cell. Based on this simulation, we will implement a cable of cells.
   Further several models of isolated single cells will be implemented, which will be used for further analysis.
   This implementation will be run and tested on a Maxeler VM, which simulates a Maxeler machine. As the Maxeler VM only provides to test and evaluate the computation, we will use a real Maxeler for performance testing.
3. **Analysis**
   The analysis consists of several parts, first the difference between the techniques used for simulations (CPU, GPU, Maxeler).
   Another part will be the analysis of using different cardiac models using an isolated single cell to evaluate the accuracy using Maxeler Technology. We will focus here also on the feature to use non-standard precision and compare those to CPU

implementations using single and double precision. Furhter, we will analyse the performance and resource usage of an homogeneous 1D cell cable.

1.4 Structure of the Work

As the topic of this thesis includes medical and technical terms and procedures, the reader will be provided with the necessary preknowledge.
First, since the simulation of cardiac cells, primarily aims to provide a possibility for early diagnosis of cardiac arrhythmia, we provide an overview of the medical definition of cardiac arrhythmia. Subsequently, the simulation of cells will be explained gradually. Therefore, this work also provides an overview of cell-to-cell communication. Following the biological preliminaries, we introduce the reader to the theoretical and mathematical background.

Additionally, we introduce the reader to state-of-the-art technology. Furthermore, as we aim to provide a comparison of the different technologies, a detailed introduction to Maxeler Technologies concludes the background section.

After establishing all necessary preliminaries we present discuss the implementation using Maxeler Technologies. Furthermore, to ensure a clean implementation and full exploit of the Maxeler features, the implementation will be done gradually, starting with a single cell and building up to a 1D cable.

As mentioned before, we set the Maxeler implementation in comparison to known state-of-the-art implementations. The reader will therefore be introduced to the experimental set up and afterwards we present and discuss the results regarding performance and memory efficiency.

The work is then concluded with a brief overview of future work and related projects.

CHAPTER 2

Background

2.1 Cardiac Arrhythmia

In a healthy heart the rhythm of the myocardial electrical impulses are controlled by the sinus node. There are several diseases which can disturb those signals and produce abnormalities or perturbations in the heartbeat. Those deviations are known as cardiac arrhythmias. Due to the many underlying conditions, which can cause cardiac arrhythmias, treatment needs to be personalized accordingly. Besides medication to control the arrhythmias, the most common adjunctive therapy are implantable pacemakers and defibrillators. These devices inherently present a highly invasive procedure. Therefore their durability and confidence are basic requirements in the trial phases [19]. By simulating cardiac electrophysiology, those devices can be tested ex vivo for different types of arrhythmias. Furthermore simulating cardiac arrhythmias, can provide deeper insight of the underlying complex processes and diseases.

2.2 Simulation of Cardiac Electrophysiology

2.2.1 Overview of Cell-to-Cell Communication in Cardiac Cells

Generally communication between cells is carried out by electric impulses, due to changes in cellular ion concentrations. The most important ions in this process are potassium (K^+) and sodium (Na^+). Usually, the concentration of Na^+, Ca^{2+} and Cl^- are higher in extracellular fluids, which leads to a potential difference between inside and outside of the cell. This difference is referred to as membrane potential. Furthermore the stable state of the ion concentration is called *"ion homeostasis"* [7].

During resting state the membrane is permeable to K^+ which results in a resting potential between -90 and -80mV. In response to stimulation, the membrane potential can change rapidly due to active *ion channels*. To activate the ion channels a depolarization between

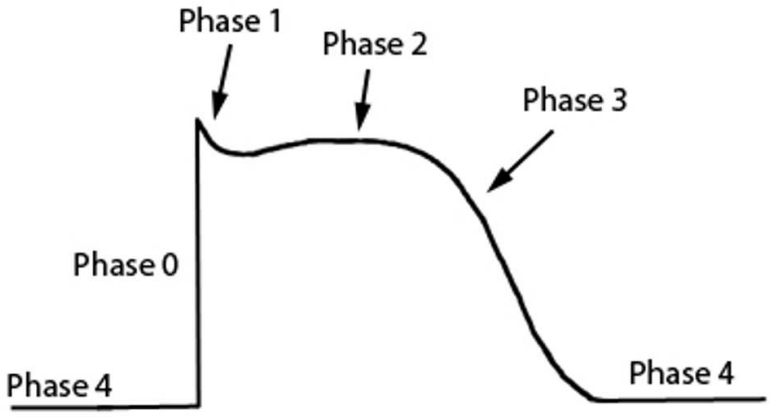

Figure 2.1: Action Potential of a Cardiac Cell [1]

-70 and -60mV is needed [7]. This activation triggers a feedback process, which is split into four phases, were the fourth phase represents the resting state, see Figure 2.1. The initial phase is characterized by a rapid upstroke, about 100mV for a millisecond. This upstroke is provoked by an external stimulus, which leads to the opening of Na^+ ion channels and therefore to the depolarization of the cell. In the first phase the cell repolarizes for a short time, as the K^+ start to open. Further, Ca^{2+} channels start to open, this results in the second phase. This phase is called *plateau phase* and is a distinctive feature of cardiac cells. The plateau is formed due to the stream of Ca^{2+} ions into the cell and K^+ ions outwards and lasts about 200ms [7]. During the plateau phase the Ca^{2+} channels slowly start to close and lead to the third phase. In this phase only the K^+ channels are open, which leads to the restoration of the resting membrane potential of -90mV. When the resting potential is met, the K^+ return to their initial closed state and the cell is back at its resting state and consequently in phase four [7].

2.2.2 Cardiac Models

In 1952 Hodgkin and Huxley [23], were the first to model the propagation of an action potential in nerve fibers, using a giant squid axiom. This breakthrough in describing the basic electrochemical processes in cells through mathematical models enabled more research of mathematical models for human tissue. One of the first methods adapting the model presented by [23] to fit cardiac cells was presented by D. Noble [30] in 1962 using Purkinje fibers.

Modern models to describe cell electrophysiology either adapt these models or simplify them, depending on the tissue simulated, and respectively electrical characteristics. This results in a broad range of complexity, as the models use more or less *state variables* to describe the processes. In this thesis, we mainly use the *Minimal Model* introduced by [9]. To compare the applicability and accuracy of Maxeler we further use the *Beeler-Reuter*

Model introduced by [6] as well as the *Karma Model* introduced by [25]. The differences between each model will be shortly elaborated in this chapter.

Beeler-Reuter Model

The Beeler-Reuter Model is a generic model and also the first ventricular model. It uses four ionic currents and eight state-variables to calculate them [6]. The according equation by [6] to calculate the voltage variable is presented in Equation 2.1, where C_m is the membrane capacitance.

$$\boxed{\frac{\partial u}{\partial t} = \frac{1}{C_m}(i_{k1} + i_{x1} + i_{Na} + i_s - i_{stim})} \tag{2.1}$$

The equations introduced by [6] for the four currents are shown in Equation 2.2. Where i_{k1} represents the time dependent outward potassium current, i_{x1} the time dependent activated outward current, i_{Na} the inward sodium current and finally i_s the slow inward current, which is mainly carried by calcium. The subtracted current i_{stim} represents an outside stimulus.

$$\boxed{\begin{aligned} i_{k1} &= \tfrac{p_1\{\exp[p_2(u+p_3)]-1\}}{\exp[p_4(u+p_5)]+\exp[p_6(u+p_5)]} + \tfrac{p_7(u+p_8)}{1-\exp[p_9(u+p_8)]} \\ i_{x1} &= \tfrac{x_1 p_{10}\{\exp[p_{11}(u+p_{12})]-1\}}{\exp[p_{13}u]} \\ i_{Na} &= p_{14}m^3hj + p_{15}(u-p_{16}) \\ i_s &= p_{17}df(u - p_{18} - p_{19}\ln(Ca)) \end{aligned}} \tag{2.2}$$

The model uses 63 different parameters [11] which are labeled p_i within the equations. Further, the calculation of the gate variables x_1, m, h, j, d and f according to [6] is presented in Equation 2.4. As stated by [6], these variables are calculated in the same manner, only differing in related parameters (C_i). Therefore they are generalised with the label y. Finally, Equation 2.3 shows the differential equation by [6] to calculate the intracellular Ca^{2+} concentration.

$$\boxed{\frac{\partial Ca}{\partial t} = p_{20}i_s + p_{21}(p_{22} - Ca)} \tag{2.3}$$

$$\boxed{\begin{aligned} &\tfrac{\partial y}{\partial t} = \alpha_y(1-y) - \beta_y y \\ &\text{where} \\ &\alpha_y, \beta_y = \tfrac{C_1\exp[C_2u]+C_3(u+C_4)}{\exp[C_5(u+C_4)]+C_6} \end{aligned}} \tag{2.4}$$

Karma Model

Derived from the model introduced by [30] this model eliminates two of the four state-variables and was developed to show spiral wave breakup due to alternans. The remaining variables are then used to describe the voltage within the cell and an according gate variable. The gate variable "v" represents the slow sodium inactivation and the slow potassium activation combined by a Heaviside function [25]. The according equations introduced by [25] are shown in Equation 2.5.

$$\begin{aligned} &\frac{\partial u}{\partial t} = D\nabla u - u + [\gamma - (\tfrac{v}{v*})^{xm}]h(u) \\ &\frac{\partial v}{\partial t} = \epsilon[\Theta(u-1) - v] \\ &\text{where} \\ &h(u) = ([1 - \tanh(u-3)]\tfrac{u}{2}) \text{ or } (u^2 - \delta u^3)/\alpha \end{aligned} \tag{2.5}$$

The variable parameters in the equation are then: xm which describes the sensitivity of the wave front, ϵ which relates the maximum ADP to the time scale of the upstroke and finally $v*$ which controls the APD pulse dynamics [25].

As this model only uses two state variables to describe the internal currents, the physiological properties which can be described are restricted. In contrast to the other two models used, this model is not considered a ventricular model but as Purkinje cell model, as it was derived from [30].

Minimal Model

This model is a simplified model to simulate different types of cardiac cells in humans and uses the minimal number of differential equations to do so. The number of differential equations needed, is reduced by using four state variables. These variables are then used to calculate an overall sum of the transmembrane currents in human myocytes. These are represented in three main categories: fast inward, slow inward and slow outward currents [9].

The equations according to [9] are shown in Equation 2.6. The state variable u represents the voltage of the cell, by summation of three currents [9]. Those currents represent the currents regarded for calculation in the cell. Where j_{fi} is the fast inward current, j_{so} the slow outward current and j_{si} the slow inward current [9]. The calculation for those currents according to [9] are shown in Equation 2.7.

$$\begin{aligned} &\partial_t u = \nabla(\widetilde{D}\nabla u) - (j_{fi} + j_{so} + j_{si}) \\ &\partial_t v = (1 - H(u-\theta_v))(v_\infty - v)/\tau_v^- - H(u-\theta_v)v/\tau_v^+ \\ &\partial_t w = (1 - H(u-\theta_w))(w_\infty - w)/\tau_w^- - H(u-\theta_w)w/\tau_w^+ \\ &\partial_t s = (1 - \tanh(k_s(u-u_s)))/2 - s)/\tau_s \end{aligned} \tag{2.6}$$

$$\boxed{\begin{aligned} J_{fi} &= -vH(u-\theta_v)(u-\theta_v)(u_u-u)/\tau_{fi} \\ J_{so} &= (u-u_o)(1-H(u-\theta_w))/\tau_o + H(u-\theta_w)/\tau_{so} \\ J_{si} &= H(u-\theta_w)ws/\tau_{si} \end{aligned}} \tag{2.7}$$

In every equation the H stands for a standard Heaviside function. The constants used can either be looked up in [9] or directly in the code, as some adaptations were made.

Despite the simplicity of the model, it is still accurate enough to generate pseudo EEG data and can be used to simulate important electro physical properties, like the action potential duration (APD) and the conduction velocity (CV). Further, it is possible to simulate large-scale tissue, with a combination of different types of cells [9].

However, the models has limitations. As it only considers a sum of the currents in myocytes, the simulation of intra cellular calcium waves is not possible, as well as the change in membrane potential of an isolated cell and particular ion channels [9]. Still, it can be used to simulate mutated tissue (like Brugada Syndrome) or effects of pharmalogical agents [9].

2.2.3 Further Dynamics

As shown in the previous chapter, there are many forces and variables in systems biology. Those forces and variables can be generalized by using basic principles of electrical circuits to describe changes of currents. One of those generalizations are *Reaction-Diffusion systems*. Those systems describe changes of concentration over space and time. More precisely the Diffusion Term is a partial differential equation, which describes the spatial distribution over time and the reaction term the concentration change inside the cell. The reaction terms according to each model, were presented in the previous section and can be used for a single cell implementation. In a cell cable of variable dimension, the diffusion term is used to calculate the propagation of the cell.

The following chapter provides necessary mathematical insights in the calculation of the diffusion term and alternative time stepping methods.

Diffusion Term

Basically when applying the diffusion equation, these two key assumptions need to be satisfied [24]:

1. We assume a linear relation between gradient and flux.
2. The diffusing substance must not be changed.

Those two assumptions are satisfied by the passive propagation of voltage in cells, as well as heat propagation [24]. According to citejackson2006molecular, by using these basic principles, the diffusion equation can be derived from Fick's law and expressed as shown

c	A
	b^T

Table 2.1: Representation of a Runge-Kutta method

in Equation 2.8. Where C is the concentration, D the proportionally constant (diffusion constant) and ∇^2 is the Laplacian operator from Equation 2.9 according to [24].

$$\boxed{\frac{\partial C}{\partial t} = D\nabla^2 C} \tag{2.8}$$

$$\boxed{(\delta^2/\delta x^2) + (\delta^2/\delta y^2) + (\delta^2/\delta z^2)} \tag{2.9}$$

Numerical Integration Methods

Many times differential equations are used to predict a value in the foreseeable future. To solve the time depended equations, numerical integration methods can be used, which can further influence the performance of the computation. In this work, we use the *Euler Method*. This method was published in Euler's three volume work from 1768 to 1770 [10]. The basic principle of this method is inspired by particle movements. This movement is within a timespan, so short that the velocity v_0 does not change. With a nearly constant v_0, the new position of the particle is approximately δt multiplied by v_0 [10]. Due to its simplicity, the Euler method is not accurate enough for higher order problems. One class of methods used for the solution of high order problems would be the Runge-Kutta methods. These implicit and explicit methods adapt the Euler method in means of performing several stages within a time step. When using explicit methods, the computed stages can be combined to a higher order approximation at the end of each time step. Whereas implicit methods require to solve the equations for every time step for all stages simultaneously [33]. As stated in [5] as well as [33] the choice of integration method can influence the performance. The representation according to [10] of these methods is generally a table, as shown in Table 2.1. Where c is a vector indicating the position within the steps, A represents the dependencies within a matrix and b^T represents the final result dependencies as derivatives [10].

2.3 State Of The Art

Next to models, the simulation of human cells requires high performance technology. Especially when a real-time simulation is needed. State of the art simulation include besides parallel and sequential CPU based, several parallel implementations based on

GPU for a variation of cell types. GPU implementations prove to be 5-40 times more efficient over parallel CPU implementations, depending on the algorithm used [32][5].

In this work we provide a hardware based solution to accelerate the simulation. Therefore, we also introduce the reader to similar technologies, in particular Field Programmable Gate Array (FPGA). For this reason we also discuss the state-of-the-art related to this technology.

In the following we first introduce the reader to the GPU and FPGA technology and then we discuss the related work concerning the use of this technologies in simulation of cardiac dynamics

2.3.1 GPU

Architecture

As mentioned before, CPUs are designed to run a high variety of programs, even purely serial ones. Therefore the architecture includes a complex logic and large caches for these operations. GPU architectures instead are designed to perform mostly arithmetic operations, with a small cache [12]. The implementation of simulations on GPUs are focused on *NVIDIA* GPUs, as these provide a computing platform to perform general computing on GPUs, called *Compute Unified Device Architecture (CUDA)*. Another option to implement GPU based applications, is to use *OpenCL (Open Computing Language)*. This programming language is a C-based open source project created by *AMD*, *IBM*, *Intel* and *NVIDIA*. In contrast to CUDA, this programming language is device and platform independent. Nevertheless, in this work we use CUDA for the GPU simulation and therefore focus on this language. We will therefore provide an overview of the programming model using CUDA in the next subsection.

Programmable GPUs are called *general purpose graphics processing unit (GPGPU)*. Using a Peripheral Component Interconnect Express (PCIe) bus, it is possible to execute data transfers between CPU and GPU. These buses provide, depending on the PCIe version a high bandwidth per lane, e.g. 500MB/s for PCIe 2.0, PCIe 3.0 provides twice as much [34]. Usually the GPU requires the most bandwidth of all peripherals in the system, therefore the design is laid out for a 16-lane PCIe slot. A general schematic of the PCIe hardware architecture, is shown in Figure 2.2 [34]. The actual architecture further depends on the system setup. The setup sketched in Figure 2.2 assumes a single discrete GPU, while other setups can include multiple GPU clusters or integrated GPUs.

CUDA Programming Model

Compute Unified Device Architecture (CUDA) is both, a general-purpose parallel-computing architecture and a programming model, which enables interactions between CPU and GPU. Generally when developing in CUDA, CPU is referred to as "*host*" while the GPU is referred to as "*device*". For obvious reason, CPU and GPU work most efficient with separate memories, as shown in Figure 2.2. These are usually only accessible

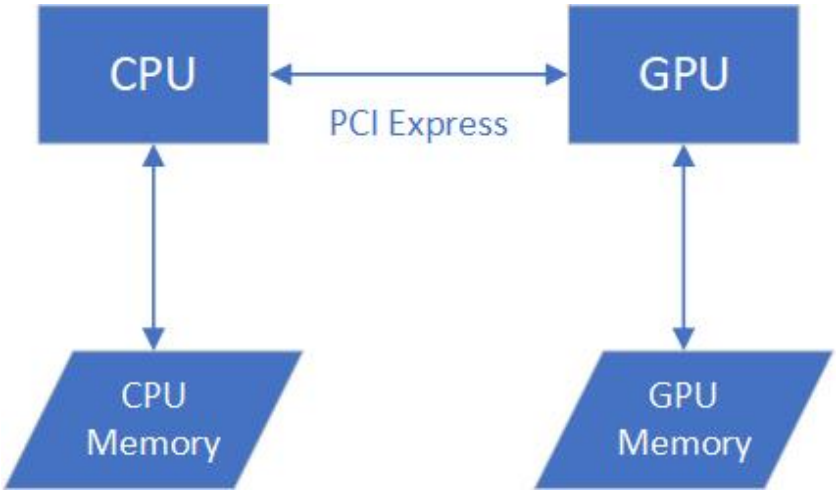

Figure 2.2: PCIe Architecture

to the according processing unit. Therefore CUDA provides several methods to allow interactions between host and device on different levels. First would be the *Pinned Host Memory*. This memory is part of the CPU memory which can be directly accessed by the GPU, but cannot be moved [34]. Secondly, CUDA provides *Command Buffers*, which are managed by the CUDA drivers. These buffers are used to send commands to the GPU [34]. As these command buffers are streamed asynchronous between CPU and GPU CUDA provides a method to *synchronize* both. These last methods also enables the CPU to track the progress of the GPU commands [34].

Besides those interactions, CUDA keeps the separation between CPU and GPU regarding memory resources. Therefore, it is distinct between *device memory*, *host memory* and *shared memory* [34]. Additionally to gain optimal performance result, the device memory itself is segmented in different levels according to [34]. The topmost level provides the *global memory*. It holds global load and store instruction and can be programmatically accessed by pointers. The next level below would be the *constant memory*. This cached, read only memory (ROM) is used to optimize broadcasts between multiple threads. The *local memory* holds local variables which cannot be fit into registers, parameters and subroutine addresses. To improve data access in up to three dimensions in a spatial locality, the *texture memory* is used. This memory is a cached ROM and uses arrays. Additionally, the *shared memory* enables fast data exchange between threads within a block [34].

Moreover the necessary computations in CUDA are implemented in form of *kernels*. These kernels are launched asynchronously, therefore the developer has to explicitly handle synchronization between host and device to avoid errors. Once a kernel is launched, it runs as grid of blocks. Inside each block runs a set of threads, as shown in Figure 2.3. Each block gets assigned to a Streaming Multiprocessor (SM), which can maintain the

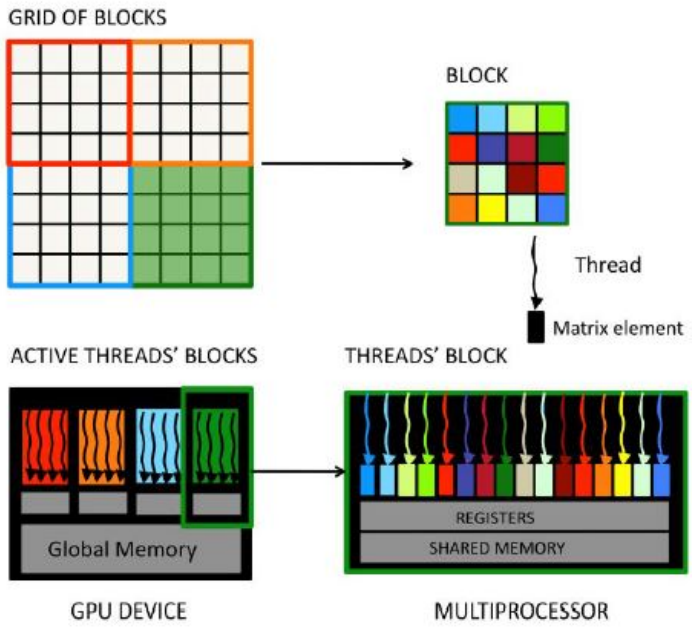

Figure 2.3: Schematics Grid of Blocks of Threads according to [5]

context for multiple blocks [34].

To fully exploit the parallel architecture, those blocks should be divided into 2D or 3D grids [12].

Processing Power

Nowadays simulations tend to use GPU over CPU, as modern GPU technologies allows improved programmability and provides a high computation power, as stated in [17], [32] For example, standard GPUs using *NVIDIA Pascal* architecture, provide a processing power ranging from 27.6 GFLOPs[1] (NVIDIA GT 1030) up to 257.1 GFLOPs (NVIDIA GTX 1080). Certainly, high power GPUs provide a higher performance, like the NVIDIA GeForce Titan V with 6144.0 GFLOPs.

In comparison to this, modern CPUs using, e.g. Intel Coffee-Lake architecture, have a processing power of 307.2 GFLOPs (Intel Core i7-8700) [16]. For high computation, Intel also offers more powerful CPUs, the *X-Series*, which provides up to 1152 GFLOPs (Intel Core i9-7980XE) [16].

[1] In computer science, the processor performance is measured in floating point operations per second (*FLOPS*). GFLOP = *Giga*FLOPS = 10^9 FLOPS

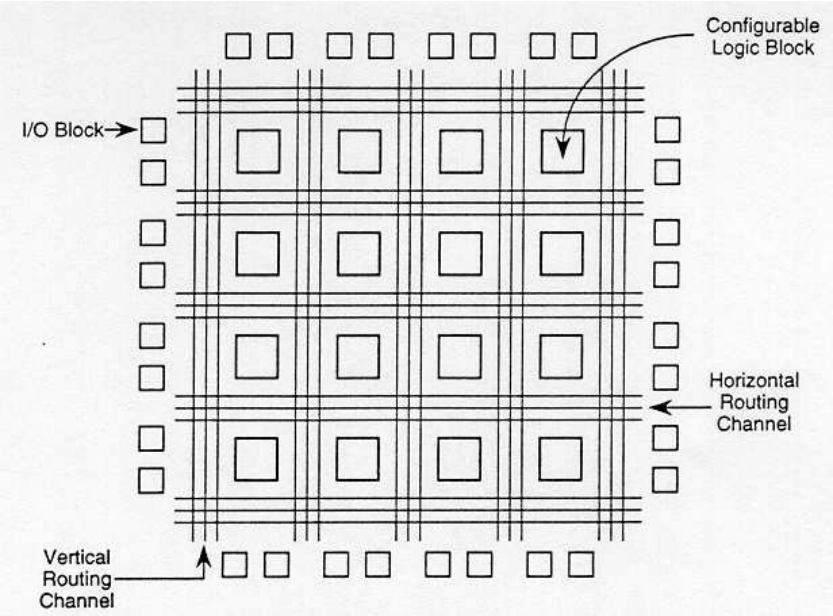

Figure 2.4: Architecture of a FPGA [26]

2.3.2 FPGA

Architecture

A Field Programmable Gate Array (FPGA) is a programmable hardware chipset. Inside logic blocks, predefined logical gates (e.g.: AND, nAnd, FlipFlops etc.) are used to create necessary arithmetic. Those logic blocks are then connected via switches to other logic blocks or I/O blocks, as seen in Figure 2.4. The detailed setup of a FPGA chipset depends on the chip type and the manufacturer [14]. The I/O blocks generally are configured to be used as both, input and output, the specific definition of the pins has to be performed by the developer. Additionally FPGA can be distinguished by structure and split into four classes: *Symmetric arrays*, *Series array*, *Sea-of-gates*, *Hierarchical Programmable Logic Device (PLD)* [27].

To store the configuration, basically two options are available. The first option is to use flash memory, whilst this option has the advantage of persistence, flash memory requires more space on the chip and has a cost disadvantage. In contrast to the first option stands the usage of Static RAM (SRAM). As this type of memory will reset on disconnecting the power source, additional concepts to load the configuration on startup are needed, like special EEPROMs. [14]

Besides I/O and logic blocks, an FPGA has several other basic elements, which will be explained shortly according to [14]. First the *Phase Locked Loop (PLL) and Clock Manager*. Usually the clock frequency of a FPGA is fixed. Using those components, the frequency can be adapted to the application. Secondly, the FPGA provides *Embedded*

Memory, which is dedicated on chip memory. Additionally, provide *Signal Processing blocks* optimized operations for signal processing. FPGA also contain a specific *infrastructure* for signal transport and routing. Finally, FPGA can be connected to several chips and communicate. To enable fast communication for such serial connections, transceiver elements can be used.

VHDL Programming Model

Designing FPGA always means designing a hardware circuit board. This, in combination with the highly complex architecture, is prone to fatal errors and eventually destroying the chip as a result. To avoid such errors, several tools for the generation of FPGA code are available. The code generated is either Very High speed Integratd Curcuits Hardware Description Language (VHDL) or *Verilog*. This work will focus on VHDL. Basically the concept of VHDL consists of three parts. The first part is the modelling of the circuit itself. This model can now either be processed by a synthesis program or verified via simulation. Those simulations, or test benches, decrease the error probability and show eventual problems beforehand [4]. One tool to generate and simulate VHDL is *Matlab Simulink*, which is also well represented in literature.

2.3.3 Related Work

Simulation of human tissue is a well discussed topic, especially when finding new technology or improve existing implementations. Studies and research clearly tend to the usage of high parallel solutions, which shows the foreseeable future of computational biology. Due to this tendency, GPUs became a highly important technology in this field, as they provide the perfect architecture and tools for parallel computing. In [12] several biological systems are implemented using GPU, to show the general applicability of GPU in biological system computation regarding the advantages and disadvantages compared to CPU. Although it shows that most biological models are fit for parallel architecture, it also demonstrates that the performance varies heavily depending on the underlying model in GPU.

Besides the general usage, GPUs are also used to accelerate cardiac cell simulation. Several papers compare CPU and GPU implementations regarding resource usage and elapsed time of the performed calculations to estimate the performance of each devices. The research conducted in [32], shows that the usage of GPUs in cardiac cell and tissue simulation is superior to CPU in most cases. Additionally [29] arrived to the same conclusion. Both studies compare the usage of CPU clusters to GPU clusters in different settings. They show, that the bottleneck of CPU implementations are the ordinary differential equations, as those are inherently parallel. Partial differential equations on the other hand, are the bottlenecks of GPU implementations.

To avoid those bottlenecks, several mathematical approaches to redesign the modelling can be taken. In [3] the usage of high finite elements in model simulation proofs to be more efficient compared to using linear finite elements. The study was conducted using

sequential CPU implementations, but also states, that the usage of parallel computing could provide even more efficient simulations. Based on [3] it should be possible to implement as a GPU application.

Another technology used to accelerate simulations are Field Programmable Gate Array (FPGA). As FPGAs are hardware solutions, designing those represents a challenge. To simplify the implementation of hardware solutions, Matlab Simulink offers a tool to design hardware implementations and generate HDL code. Moreover, FPGAs in cardiac simulation can be used to verify ECG signals as for example presented in [13]. When verifying ECG data, the signals are evaluated by an algorithm, to check for abnormalities. In [13] the FPGA implements the verifier to detect abnormal ECG signals. The signals are streamed into the FPGA circuit and the output flags normal or abnormal signals accordingly. But also full voltage clamp simulations are performed using FPGA as for example in [31]. Further, [13] as well as [31] use Matlab Simulink to design and implement the FPGA solution.

Although modern GPUs proof to be superior over CPUs, most complete simulation environments are CPU based, written in Phyton, C or C++. An example for such simulation environments would be *Brian*[20] and *GENESIS*[8], which simulate neuronal behavior.

2.4 Maxeler Technology

2.4.1 Introduction to Maxeler Dataflow

This chapter provides an overview of the technological terms used in this thesis and also give basic understanding of the technology itself.

Control Flow vs. Data Flow

Dataflow sets the focus on the optimal movement of data inside an application and highly parallelized computation [28]. The application in dataflow is typically represented as "dataflow graph". This graph is an abstract image of how the data inside the application moves, for an example see graph 2.5. Basically a dataflow graph has three actors:

Tokens Tokens are representation of the data inside the application. Those tokens are stored in the memory of the computing element. How the tokens are allocated in the memory depend on the underlying concept of the dataflow application [22].

Operators Operators represent the function performed on the data. These can be either numerical or logical functions [22].

Arcs Arcs represent the path of the data. If the arc points towards an operation, it is called an "input arc", if it carries the result of an operation, it is called an "output arc" [22].

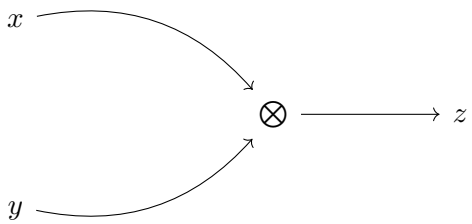

Figure 2.5: Example for a simple Dataflow Graph

The example graph 2.5 shows two input tokens x and y. Those tokens represent a value, for example two integers. The input arc of the operator are symbolized by arrows pointing to the operator node. In this example the operation performed on the tokens is a multiplication, as the asterisk in the node states. The operator has one output arc, which carries the output token z, the result of the multiplication [22].

In contrary to this approach stands control flow. Control flow applications are inherently sequential, as the instructions move sequentially through the processor and occasionally read or write onto the memory [28].

Example: 3-Point Moving Average The following example, derived from an example in [28], illustrates the difference between control flow and data flow programming for a better understanding. The example consists of a side-by-side pseudo implementation of a moving 3-point average filter in control flow and data flow. To line out the differences, the code snippets will be segmented and explained per segment.

Overall, the two programs will calculate the 3-point average over an integer array of ten elements.

Input In control flow, the array is used as it is, therefore to access the individual values, a loop is needed, see Listing 2.1. In contrast, the data flow program buffers the array on chip and streams the array element wise into the program body, as can be seen in Listing 2.2 [28].

```
array = [1 ,5 , 6, 13, 12, 5, 8, 3, 10, 2];
for(i<array) {
        .. do something
}
```

Listing 2.1: Array Representation in Control Flow [28]

```
x = io.input("array");
```

Listing 2.2: Array Representation in Data Flow [28]

Calculation As seen in Listing 2.3,in control flow the elements can be accessed at the neighboring locations during run-time, using indices.

```
array = [1 ,5 , 6, 13, 12, 5, 8, 3, 10, 2];

for(i<array-1) {
        prev = array[i-1];
        next = array[i+1];
        res[i] = (prev + next + array[i])/3;
}
```

Listing 2.3: Calculation Body in Control Flow [28]

In data flow, a window into the stream is needed to access non-current values. To do so, *offset* expressions are used. To access past values a negative offset is needed and for future values a positive offset, as shown in Listing 2.4 [28].

```
x = io.instream("array");
prev = stream.offset(x, -1);
next = stream.offset(x, 1);

res = (x + prev + next)/3;
```

Listing 2.4: Calculation Body in Data Flow [28]

Boundaries Using control flow, the elements in the array can be accessed at any arbitrary location by using the correct index. Therefore, the boundaries can be calculated at runtime at the correct index, as shown in Listing 2.5 [28].

```
array = [1 ,5 , 6, 13, 12, 5, 8, 3, 10, 2];

res[0] = (array[0] + array[1])/2
for(i=1;i<size-1) {
        prev = array[i-1];
        next = array[i+1];
        res[i] = (prev + next + array[i])/3;
}
res[size-1] = (array[size]+array[size-1])/2
```

Listing 2.5: Boundary Inclusion Control Flow [28]

In data flow, it is not possible to create a 2-point average at runtime depending on current position of the stream. Therefore all adders and dividers have to be implemented at compile time, to add them to the data flow logic [28]. The decision, which adder and divider to use, is done at run time, by evaluating logical conditions and counters, which keep track of the current stream position, as shown in Listing 2.6. The conditions in the pseudo code are evaluated with the ternary if-operator [28].

```
x = io.instream("array");
```

```
prevOrig = stream.offset(x, -1);
nextOrig = stream.offset(x, 1);

count = control.counter.add();

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

prev = aboveLowerBound ? prevOriginal : 0;
next = belowUpperBound ? nextOriginal : 0;

divisor = withinBounds? 3 : 2;

res = (x + prev + next)/divisor;
```

Listing 2.6: Boundary Inclusion in Data Flow [28]

Output After finishing the calculation in control flow, the result array is filled and can be returned or further processed. The full pseudo code example in control flow derived from examples in [28], is shown in Listing 2.7.

```
array = [1 ,5 , 6, 13, 12, 5, 8, 3, 10, 2];

res[0] = (array[0] + array[1])/2
for(i=1;i<size-1) {
        prev = array[i-1];
        next = array[i+1];
        res[i] = (prev + next + array[i])/3;
}
res[size-1] = (array[size]+array[size-1])/2

return res;
```

Listing 2.7: 3-point Moving Average Control Flow [28]

In data flow, the output is also a stream. In this example, it is assumed, that the data flow program consumes one input item per pass and produces one output item per pass. Based on this, the data flow program, will stream out the results value by value. The full pseudo code example in data flow derived from examples in derived from [28] is shown in Listing 2.8 [28].

```
x = io.instream("array");
prevOrig = stream.offset(x, -1);
nextOrig = stream.offset(x, 1);
```

```
count = control.counter.add();

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

prev = aboveLowerBound ? prevOrig : 0;
next = belowUpperBound ? nextOrig : 0;

divisor = withinBounds? 3 : 2;

res = (x + prev + next)/divisor;

io.outstream("res", res);
```

Listing 2.8: 3-point Moving Average in Data Flow [28]

Maxeler Technology Dataflow Engines (DFE)

To achieve better performance with data flow computing Maxeler Technology exploits loop level parallelism in a spatial and pipelined way. Thereby combining traditional synchronous data flow, vector and array processes [28]. This concept is based on the usage of Dataflow Engines (DFEs). These engines deploy the data flow concept on many levels of abstraction, using a set of different components, see Figure 2.6 [28].

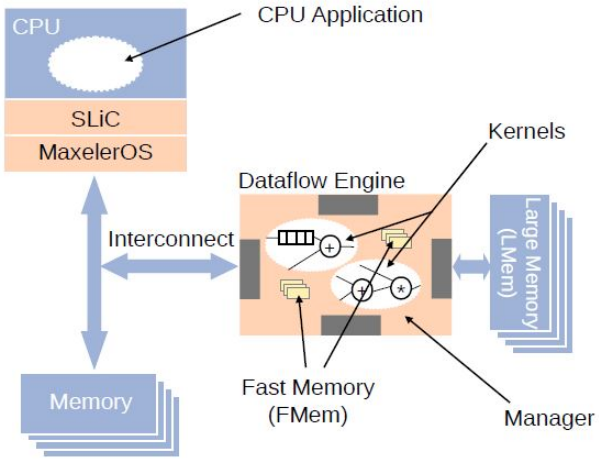

Figure 2.6: Architecture of a DFE) [28]

Kernel A DFE can have one or more kernels. These Kernels perform the computation on the data streams.

Large Memory in Maxeler DFE (LMem) A DFE has two kinds of memory, the LMem is one of them. It can hold several gigabytes of data off chip [28].

Fast Memory in Maxeler DFE (FMem) The second memory type is the FMem. It can hold several megabytes and has a high bandwidth for data transfer on chip [28].

Manager The manager coordinates how the data moves within the DFE [28].

Simple Live CPU (SLiC) This component connects the DFE with the CPU. Therefore the data can be prepared in the CPU and the calculations are performed separately in the Kernel of the DFE [28].

MaxCompiler This special compiler is part of the *MaxelerOs*. It generates the dataflow code from the implementation in the Kernels and the Manager. This dataflow code is then loaded into the CPU application via the SLiC [28].

Furthermore the MaxelerOS provides an Integrated Development Environment (IDE) designed for the implementation of DFE based applications. This IDE, called *MaxIDE* is based on the Eclipse IDE and has integrated support for the Maxeler programming language *MaxJ*. This programming language is used to configure the Kernels and the Manager. Furthermore *MaxJ* is a Java based language, which extends the basic functionalities with operator overloading semantics [28].

Basically those two components are classes in the implementation. An application can have several Kernels. These Kernels can have several input and output streams from and to the CPU. Usually the input from the CPU is a large data array. This array will be stored as buffer on the on chip memory. These buffers will then be streamed into the Kernel [28]. The computation of the values is then performed, as soon as the needed value is valid. After the finished computation, the result is streamed back to the CPU on the output stream [28].

The Manager class holds the configuration how and when data is streamed between the DFE and the CPU.

Although an DFE can have several Kernels, even the same instanced several times, it only has one Manager [28].

While using data flow programming, it is important to know which parts of the application should be reimplemented in a Kernel and which need to be implemented in the CPU. All read and write operations to and from the host are slow and should be performed at the same point. Additionally *MaxJ* has special data types, which enable the optimization while the data flow code is generated. This will be explained in more detail in Chapter 3, as each implementation step will be illustrated.

2.4.2 Technical Attributes

Pipelined Streaming to Exploit High Parallelism

The communication and data transfer on chip and off chip is a key feature for better performance. As there are several kinds of memory in use, the communication between the components differs. On one hand and we have streams from and to the CPU, as mentioned before. The data is generated in the CPU code and the SLiC automatically sets the array as input stream when calling the DFE. Now we differ between two ways of data movement. The first is the *programmer's view*, as the name states, this view describes how the programmer sees the data movement through the kernel [28]. After a certain amount of time, called a *tick*, the Kernel takes a new value of the input stream. Then the computations are performed on the data. As soon as all the computations are completed, the result is written onto output in the same tick [28].

To exploit parallelism the reality of the data movement is different. Supposed each computation node in the data flow graph needs one clock cycle to produce an output. In this case, as soon as an input for an computation node is available an output is produced. Using this pipelined style of streaming an operation can be performed parallel on several values in the stream [28]. Considering this style a latency regarding available input and output values should be expected, but due to the automatic management of MaxCompiler the latency can be disregarded [28].

Additionally, these streams do not necessarily need to connect the CPU and the DFE. A Kernel can have several input and output streams, which either connect to the CPU or another Kernel. As mentioned before, these connections are set and controlled in the Manager, the following example illustrates these in detail. Assuming an application, which repeatedly performs the same computation on a single, giant dataset [28]. To increase the parallelism of the application the input stream can, for example, be split between several Kernels. Therefore, the Kernel class is instanced several times inside the Manager [28]. Using the Manager the input stream is then split between the identical instances of the Kernel. Consequently, this would result in two output streams. In order to avoid this, the output stream of each Kernel is joined together in the Manager again. This concept results in a highly parallel computation, as the stream itself are synchronous, but the computation inside the Kernel is asynchronous [28].

Dataflow Variable Types

As mentioned before, Maxeler Technology uses the Java based programming language *MaxJ*. An addition in this language to the basic Java functionalities are special variable types, as variables are basically the entry point for the streams. By using the data flow variable types, it is possible to represent any number in any binary format wanted. This allows a fine tuning of the value precision and optimization to the bit level of the application [28]. Furthermore the declaration of the input and output variables is used to generate the SLiC.

The standard types for numerics used in computation (integer, floating point, etc.) are offered by the MaxCompiler as *DFEVars*. Variables from type *DFEVar* are represented by an object of *DFETypes* and usually get their specific datatype assigned automatically [28]. To ensure an uniform data typeset inside the application, the *DFEType* can be set globally for all *DFEVars*. Doing so, the programmer is in full control of the variable precision and the bits needed. Furthermore, even if the type is set globally, *DFETypes* can be casted during the flow. Nevertheless typecasting is high in resource costs, therefore the use of these operation should be minimized.

2.4.3 Possible Advantages and Disadvantages

One of the most outstanding characteristics of Maxeler Dataflow technology is the broad area of application. Based on this research further work can be conducted, not only for cardiac cells but also neural models. Furthermore, the high parallel technology used in Maxeler can accelerate the calculation of giant datasets, as the technology performs best with a high number of data.

On the other hand, redesigning an algorithm into Maxeler Dataflow requires a deeper understanding of the technology and therefore a long period of familiarization. Additionally, to exploit the full potential of Maxeler, it is necessary to design the application as optimal as possible.

Despite the complexity of Maxeler, it still offers a lot of advantages. As mentioned before, the precision can be set by the user. This might also influence the performance and resource usage of the application.

CHAPTER 3

Maxeler Implementation

This chapter will explain the implementation in Maxeler in detail. Therefore, we will focus on the Maxeler specific code in this chapter and do not discuss the model specific calculations in details, as the mathematical foundation was given in Chapter 2. The code examples used are from the Minimal Model implementation. We provide in Appendix A the full code of each model of the isolated single cell implementation and in Appendix B the full code of the homogeneous 1D cell cable . All implementations regarding this project are also available on Bitbucket[1].

Additionally, the project was created using MaxCompiler version 2017.2.1 and later on migrated to the latest MaxCompiler version 2018.2.1. The latest version separates the application in projects. These can either be from the type *C-SLiC* or *Maxeler DFE*. The configuration of these projects regarding MaxFiles and their generation, can be done using the *Maxeler IDE*.

3.1 Single Cell

DFE Part

As mentioned previously dataflow programs can be represented as graphs. Figure 3.1 shows a simplification of the dataflow graph for the single cell implementation. The calculation block is model specific and therefore only indicated, as the actual graph generated by Maxeler takes each operation within the calculation into account.

Further, we implemented several versions of the single cell, for different test cases. For example, we provide the option to generate the stimulus on CPU side using a Heaviside function and different spacing settings and stimulation times. This generated array of stimulus flags (1 if active, 0 if inactive) can then be streamed into the DFE as input.

[1] `https://bitbucket.org/lily93/thesis-src/`

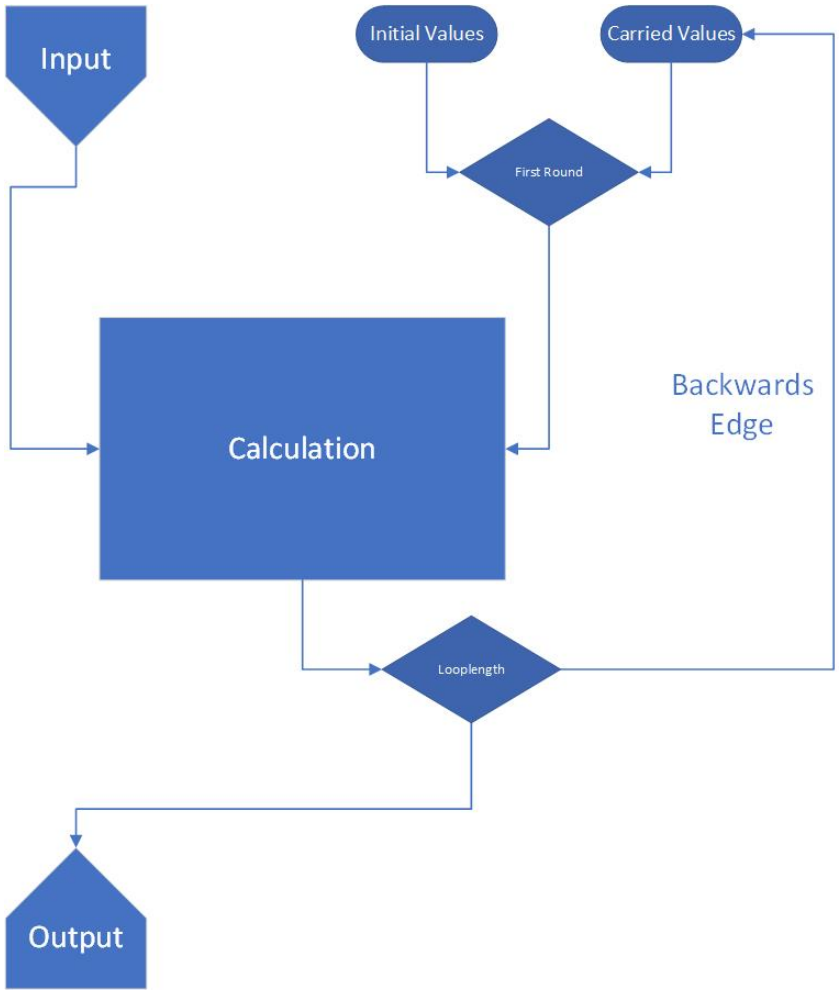

Figure 3.1: Simplified Dataflow Graph Single Cell

Doing so we provide the opportunity to simulate and evaluate different signals. To compare the accuracy between Maxeler and CPU we omitted this feature, which results in a graph without input.

The implementation consists of two loops. The first loop is the Kernel-loop itself, this loop starts with the first execution of code and ends when the Kernel finishes executing. The Kernel execution and therefore the created loop-length is managed and optimized by Maxeler. Therefore it is recommended to use the *autoloop offset* feature of Maxeler, to obtain controls for this loop, see Listing 1. Further optimization used within the design will be discussed in more detail later on.

The second loop present in the design, is the time dependency of the single cell calculation. This loop is more important for the calculation, as the results depend on previous values. This dependency is shown as *backwards edge* in Figure 3.1. Using the controls of the

```
OffsetExpr loopLength = stream.makeOffsetAutoLoop("loopLength");
DFEVar loopLengthVal = loopLength.getDFEVar(this, dfeUInt(11));
```

Listing 1: Autoloop offset

```
        CounterChain chain = control.count.makeCounterChain();
        final DFEVar nx =io.scalarInput("num_iteration", dfeUInt(32));

        DFEVar step = chain.addCounter(nx, 1);//counter for number of steps;
                                                                        //
        DFEVar loopCounter = chain.addCounter(loopLengthVal, 1); //counter
        ↪  for validation of values;
        DFEVar dt = io.scalarInput("dt", dfeFloat(11,53));
        dt = dt.cast(calculationType);
        //DFEVar stim_in = io.input("stim_in",scalarType,loopCounter ===
        ↪  (loopLengthVal-1));
```

Listing 2: Preparation

```
        DFEVector<DFEVar> statevars = vectorType.newInstance(this);
DFEVar ui = step===0? One : statevars[0];
```

Listing 3: Wrapper for State Variables

Kernel-loop we stream the calculated values back to the start of the calculation.

To do so, we need *counters* for each loop present to obtain some control. This is shown in Listing 2, where we use a *counter chain* to create a nested loop consisting the Kernel-loop and the time loop. We obtain the necessary constants set on CPU side as inputs, which is also shown in Listing 2.

The counter objects, automatically count from 0 to the given maximum value, then it wraps and restarts from 0. It is also possible to specify the wrapping behavior and increment of the counter if needed. Also, the example shows that we control the input array with the Kernel-loop, as we only need to read a new value after the Kernel finishes.

Since we prepared the controls for the design, we now need to establish the prerequisites for the backwards edge. As stated before, we stream the calculated values back to the top for further usage. To achieve this, we create source less streams at the beginning of the loop. These streams basically behave like input streams, but are not connected to the CPU. In our case we use a *DFEVector* to achieve this, as shown in Listing 3 DFEVectors are basically which is basically a wrapper for DFEVariables. A multiplexer is then used to choose between the carried value of the backwards edge and the initial value for t_0.

```
DFEVar uOffset = stream.offset(ui, -loopLength);
 statevars[0] <== uOffset;
```

Listing 4: Creating Stream Offset

```
io.output("u_out", statevars[0], scalarType,loopCounter ===
↪  (loopLengthVal-1));
```

Listing 5: Controlled Stream to CPU

```
KernelBlock k = addKernel(new
↪  MinimalModelKernel(makeKernelParameters(s_kernelName)));

DFELink y1 = addStreamToCPU("u_out");

y1 <== k.getOutput("u_out");
```

Listing 6: Creating Link Kernel-CPU

At the end of the loop, after the implementation of the model specific state variable calculation, the streams need to be connected to the top of the loop. To do so, we create an offset according to the *loopLength* within the stream, see Listing 4. These are then connected to the source less stream within the DFEVector and conclude the backwards edge. Naturally, all these operations are done for each state variable, as we generalize the Maxeler properties, the examples refer to the voltage variable u.

Finally, we stream the values back to the CPU. Here we also use the Kernel-loop control, as we only want to write valid results to the stream, as shown in Listing 5.

This concludes the Kernel implementation of the DFE part. The Manager implementation takes care about the data movement between DFE and CPU. First, we need to define a Link between the Kernel and the DFE, which is done in the Manager constructor, shown in Listing 6.

We define the scalar inputs set by the CPU and the streams within the *Engine Interface*, as shown in Listing 7.

The Manager calls the constructor and builds the *MaxFile* according to the build configuration in the main method.

```
private static EngineInterface interfaceDefault() {
                EngineInterface ei = new EngineInterface();

                InterfaceParam length = ei.addParam("length", CPUTypes.INT);
                InterfaceParam dt = ei.addParam("dt", CPUTypes.DOUBLE);
                InterfaceParam num_iteration = ei.addParam("num_iteration",
                ↪  CPUTypes.INT);
                InterfaceParam lengthInBytes = length *
                ↪  CPUTypes.FLOAT.sizeInBytes();
                InterfaceParam loopLength =
                ↪  ei.getAutoLoopOffset(s_kernelName, "loopLength");
                ei.ignoreAutoLoopOffset(s_kernelName, "loopLength");
                ei.setTicks(s_kernelName, length * loopLength);

                ei.setScalar(s_kernelName, "dt", dt);
                ei.setScalar(s_kernelName, "num_iteration", num_iteration);


                ei.setStream("u_out", CPUTypes.FLOAT, lengthInBytes);



                return ei;
        }
```

Listing 7: Creating Link Kernel-CPU

3.1.1 SLiC Part

Basically this project, contains the configuration and initialization of the DFE. The simulation parameters are therefore parsed from a file and necessary memory is allocated. The stimulus is then generated according to the parameters using a function, shown in Listing 8.

```
void take_step(float *stim_in, const int time,int stimNum, int * stimTimes) {

        int sizeBytes = (stimNum) * sizeof(float);
        float t1 = 0.0;
        float t2 = 0.0;

        for(int i = 0; i<time; i++){


                        t1+=dt;
                        t2+=dt;

                        for(int j = 0; j<stimNum; j++){
                        float c1 = t1-stimTimes[j];
                        float c2 = t2-(stimTimes[j]+1);
                        stim_in[i] += heavisidefun(c1) * (1 -
                        ↪  heavisidefun(c2));
                        }

        }

}
```

Listing 8: Single Cell Take Step

After preparing the data needed for the DFE, the according *MaxFile* needs to be loaded. Using the functionalities provided by the *Advanced SLiC*, Listing 9 shows how to load the MaxFile into the C-Code.

Furthermore, the scalar inputs and streams are set using a DFE specific *action* Enum. Hereby, the name used in the Manager Class must be consistent with the names used in the Enum, as the names in the MaxFile are generated using the ones within the Manager.

After loading the MaxFile and setting the actions, the DFE can be started by calling the built-in run function.

To avoid memory leaks, the memory allocated at the beginning is freed at the end of the function. The DFE is unloaded via a built-in function call.

3.2 Homogenous 1D Cable

Based on the previous implementation, we used the Minimal Model by [9] to create a homogenous 1D cable. As the values are not only time-dependent but also space-dependent we need to add another loop to the existing design, as shown in Figure 3.2.

The constants and parameters used in the single cell implementation introduced previously stay the same, only additional parameters for the spacial properties are added.

```
max_file_t *myMaxFile = MinimalModelDFE_init();
max_engine_t *myDFE = max_load(myMaxFile, "local:*");


MinimalModelDFE_actions_t actions;

actions.param_length = num_iteration;
actions.param_dt = dt;
actions.param_num_iteration = num_iteration;
//actions.outstream_stim_out = stim;
actions.outstream_u_out = u_out;
actions.outstream_v_out = v_out;
actions.outstream_w_out = w_out;
actions.outstream_s_out = s_out;

MinimalModelDFE_run(myDFE, &actions);

        free(u_out);
        u_out = NULL;
        free(v_out);
        v_out = NULL;
        free(w_out);
        w_out = NULL;
        free(s_out);
        s_out = NULL;
        free(stim);
        stim = NULL;
        max_unload(myDFE);
```

Listing 9: Single Cell SLiC Maxeler Call

Furthermore, we add two integers as constant to the MaxFile, one to declare the maximum number of cells allowed and one to fix the number of cells of the stimulus. In this way, we can allocate enough memory to hold an arbitrary amount of cells below the maximum. These constants, are defined in the Manager, shown in Listing 10. Within the Kernel, the values are passed as parameters.

Figure 3.2 also visualizes the neighborhood dependency of the diffusion term. Due to this dependency, we need to preserve the values over multiple Kernel ticks. To achieve this we leverage the on-chip FMem of the DFE. So, at the beginning of the loop we allocate the memory within the FMem with the maximum amount of cells. We allocate a memory block for each state variable and additionally two variables holding the current time, which are used to calculate the stimulus, see Listing 11. We need to read the values at the correct position within the memory block. Therefore we generate an address using the Java's *MathUtils.bitsToAddress(int size)* and the maximum cell number. Additionally, the counter of the current cell serves as base for the address, so we automatically track the cells within the time-loop.

```
public static void main(String[] args) {
        EngineParameters params = new EngineParameters(args);
        CellCableDFEManager manager = new
        ↪  CellCableDFEManager(params);


        manager.createSLiCinterface(interfaceDefault());
        manager.addMaxFileConstant("X", X);
        manager.addMaxFileConstant("duration", duration);
        manager.build();
}
```

Listing 10: Adding Constants to MaxFile

```
Memory<DFEVar> uMem = mem.alloc(scalarType, X);
Memory<DFEVar> wMem = mem.alloc(scalarType, X);
Memory<DFEVar> vMem = mem.alloc(scalarType, X);
Memory<DFEVar> sMem = mem.alloc(scalarType, X);
Memory<DFEVar> t1Mem = mem.alloc(scalarType, X);
Memory<DFEVar> t2Mem = mem.alloc(scalarType, X);

DFEVar uFromMem =
↪  uMem.read(cellNumAddress.cast(dfeUInt(MathUtils.bitsToAddress(X))));
DFEVar vFromMem =
↪  vMem.read(cellNumAddress.cast(dfeUInt(MathUtils.bitsToAddress(X))));
DFEVar wFromMem =
↪  wMem.read(cellNumAddress.cast(dfeUInt(MathUtils.bitsToAddress(X))));
DFEVar sFromMem =
↪  sMem.read(cellNumAddress.cast(dfeUInt(MathUtils.bitsToAddress(X))));
DFEVar t1FromMem =
↪  t1Mem.read(cellNumAddress.cast(dfeUInt(MathUtils.bitsToAddress(X))));
DFEVar t2FromMem =
↪  t2Mem.read(cellNumAddress.cast(dfeUInt(MathUtils.bitsToAddress(X))));
```

Listing 11: FMem Allocation and Read

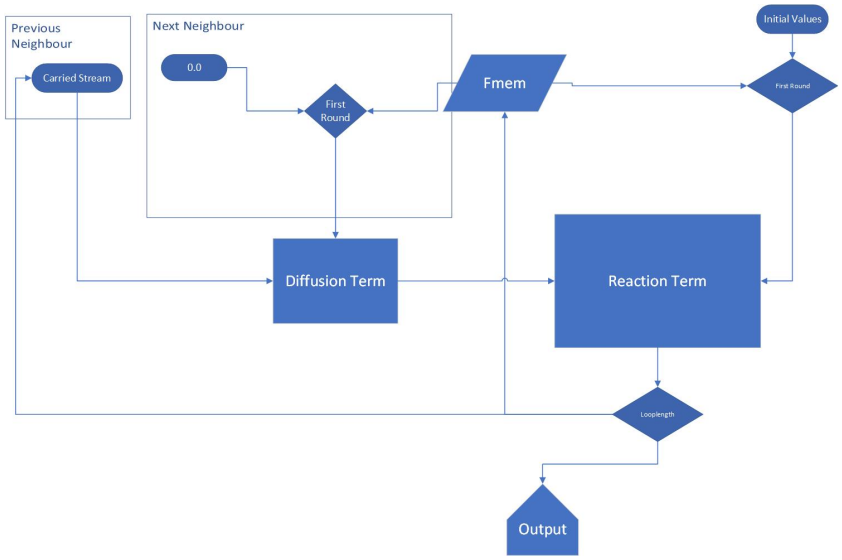

Figure 3.2: Simplified Dataflow Graph 1D Cable

```
DFEVar prevNeigh = stream.offset(carriedU, -1);

DFEVar neighbourAdress = cellNumAddress+1;
DFEVar nextNeigh = initCondition? 0.0:
↪  uMem.read(neighbourAdress.cast(dfeUInt(MathUtils.bitsToAddress(X))));
```

Listing 12: Diffusion Term Access Neighboring Cells

It is worthy to notice that the values in the allocated memory block are undefined until written to. Therefore, we create an additional source less stream for the voltage variable to access valid previous values. After the initial t_0 we use the values read from the memory for the state variables. The neighboring values are then used in the diffusion term to calculate the spacial propagation of the wave. We stated before, that we use a source less stream to access previous values, to access future values, we read the next value from the memory block, as shown in Listing 12.

The calculation stays the same as in the single cell, we only put some optimization operations in place. To run a design on a real DFE we have to meet some constraints regarding timing and logic utilization. Within the calculation we used two Maxeler functions to optimize the design, which are shown in Listing 13.

The first optimization functionality creates deeper pipelines for the operations chosen. In our case we want to deepen the pipeline for all operations. The default for the first parameter of this function is 1.0, which creates fully pipelined operations. As we need to be careful with the latency in our design, due to the backwards edge, we decreased it to 0.5. Secondly, we try to add more registers to variables which are used frequently in the

```
optimization.pushPipeliningFactor(0.5, PipelinedOps.ALL);

tvm =   (u >    EPI_THVM) ?    EPI_TV2M :    EPI_TV1M;
twm =     EPI_TW1M + (  TW2M_TW1M_DIVBY2)*(1+tanh(  EPI_KWM*(u-  EPI_UWM)));
tso =     EPI_TSO1 + (  TSO2_TSO1_DIVBY2)*(1+tanh(  EPI_KSO*(u-  EPI_USO)));
ts  = (u >    EPI_THW)  ?    EPI_TS2  :    EPI_TS1;
to  = (u >    EPI_THO)  ?    EPI_TO2  :    EPI_TO1;

optimization.popPipeliningFactor(PipelinedOps.ALL);

tvm = optimization.pipeline(tvm);
twm = optimization.pipeline(twm);
tso = optimization.pipeline(tso);
ts = optimization.pipeline(ts);
to = optimization.pipeline(to);
```

Listing 13: Optimization Code

```
DFEVar memOffset = stream.offset(cellNumAddress, -loopLength);

uMem.write(memOffset.cast(dfeUInt(MathUtils.bitsToAddress(X))), uOffset,
↪  (loopCounter === (loopLengthVal-1)));

io.output("u_out", u, scalarType,loopCounter === (loopLengthVal-1));
```

Listing 14: Writing Results

calculation.

These optimization techniques also influence the accuracy of the calculations. Using too much optimization might improve the performance, but it may lead to wrong results.

At the bottom of the outer-loop we again create an offset for each variable, as in the single cell. Then we write the values to the FMem as well as the CPU when they are meeting the validity constraint, as shown in Listing 14. To write to the FMem an additional offset is used, which holds value of the cell counter, to ensure we write to the right location in the allocated memory block.

CHAPTER 4

Results

4.1 Alternative Implementation

The aim of this work is the evaluation of using Maxeler Dataflow Computing in contrast to existing State-Of-The-Art technologies. We have compared our implementation of Maxeler with two alternative implementations in GPU and CPU. Further, to guarantee correct results, the alternative implementations are based on reviewed papers. The whole source code of this project can be accessed on BitBucket[1].

4.1.1 CPU Implementation

The implementation was done in C and is completely sequential CPU. In contrast to data flow and parallel programming, the data is accessible directly, therefore no streaming is needed. Further, to reduce unnecessary memory usage, most functions perform the operation directly on the array values, using pointer arithmetic. Another difference is, as the CPU implementation is using the control flow concept; the data dependency is implemented using for-loops. The cable structure makes the state variables dependent on time and the next neighboring cells, therefore iteration over time and each cell is necessary. For validating and evaluating the values computed, the results are printed to a file.

4.1.2 GPU Implementation

For the comparison and experiments, the code in [5] was adapted to gain similar properties. These adaptions are needed, as the original code considers a 2D cable, whereas the Maxeler implementation considers a 1D cable. One of the models used in [5] is the same 4-variable model by [9] used in this work. The model is implemented using CUDA for Nvidia

[1] `https://bitbucket.org/lily93/thesis-src/`

GPUs. Further, in the original code, model-specific optimization techniques were used, to enable maximal performance gains. These techniques include the splitting of differential equations amongst several kernels. Further, the careful leverage of available memory levels in CUDA can also influence the performance. In the adapted design, we use the built-in CUDA function to achieve the maximum potential occupancy as we increase the number of elements in several tests.

4.2 Evaluation

The implementation of the isolated single cell and homogenous cell cable in Maxeler provide a new approach to influence the implementation of cardiac dynamics. To establish a scientific reference, the implementation was compared to state-of-the-art implementations in several aspects. More precisely, we used the isolated single cell to evaluate the correctness and model accuracy of the implementations. Further, we use the cell cable to evaluate the performance with respect to known technologies. For additional testing, the Maxeler code was compiled and tested on a DFE at the Maxeler Institute, as the *TU Wien* only provides a simulator so far.

In [5] a *spiral wave protocol* is used for initialization of the state variables. To produce a similar environment for the other alternative implementations, the last parameter specifies if the spiral wave protocol initialization should be used, or the binary protocol used in the single cell. Although, using the *spiral wave protocol* omits the usage of the stimulus, as the cells are already stimulated to generate a wave form.

To compare the cell cable implementations, we focus on resource usage and elapsed time of each device.

General Validity

Due to the differences regarding the scientific analysis implemented in the various resources, the correctness of the calculation is not as easy to evaluate. Nevertheless, the Matlab Code by [21] was extended by implementing the diffusion term. The results of this calculation, were then again compared to the results of the Maxeler and CPU execution. Figure 4.1 shows the propagation of a stimulus at t_0 for 2ms in a cable of 50 cells, which equals a length of approximately 1cm.

For this evaluation, Maxeler and CPU use single precision, whereas Matlab uses per default double precision. On the base of this we can conclude, that the calculation of the partial differential equations using Maxeler Technology is overall correct. We assume this, as the signal computed with Matlab, CPU and Maxeler are widely equal. Further differences will be discussed in detail in the following section.

Precision

In this work, we have used the 4-variable model by [9] for most of the tests. To evaluate the applicability of Maxeler Technologies to different models, we have also implemented

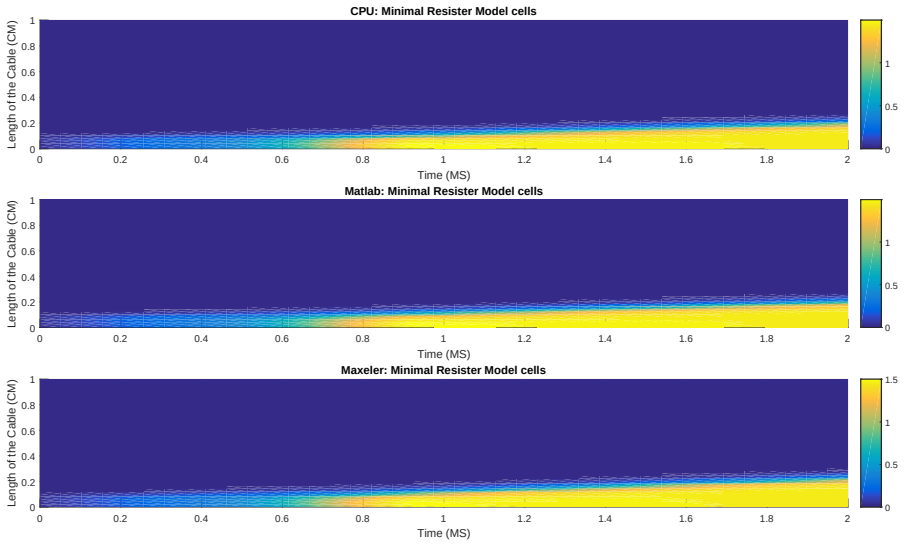

Figure 4.1: Comparison: Matlab, CPU and Maxeler Cell Cable

the 8-variable model by [6](Beeler Reuter Model) and the 2-variable model by [18] (Karma Model). All four models use the Euler Method as time step integration. By decreasing the step size (dt in ms), the calculated values of the model get more accurate, as more points in time can be simulated. This also results in a higher computation load, as the number of needed iterations increases with decreasing step size. To estimate the optimal step size several methods are possible. One of these methods is the least square method. This method of parameter fitting, estimates the optimal time step for a specific model, by minimizing the sum of squared residuals, as shown in Equation 4.1

$$S = \sum_{n}^{i=1} r_i^2 \tag{4.1}$$

A residual is defined as the difference between the actual value and the value predicted by the model: $u_i - \widehat{u}_i$.
As the reference values represent points within two steps, the formulation is adapted to include the step size, resulting in Equation 4.2. The original calculation is then later used to evaluate the precision in Maxeler.

$$r_i^2 = \left(u(\widetilde{t}_i)dt - [u(\widetilde{t}_j)\frac{dt}{2} - u(\widetilde{t}_{j+1})\frac{dt}{2}] \right)^2 \tag{4.2}$$

For this work, the datasets are computed values of the voltage state-variable of the isolated single cell within a whole cycle. More precisely, we compute the least square error between the values of a chosen step size and the results of two transient steps with half the step size, as illustrated in Figure 4.2.

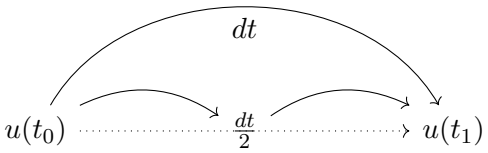

Figure 4.2: Double Step Method

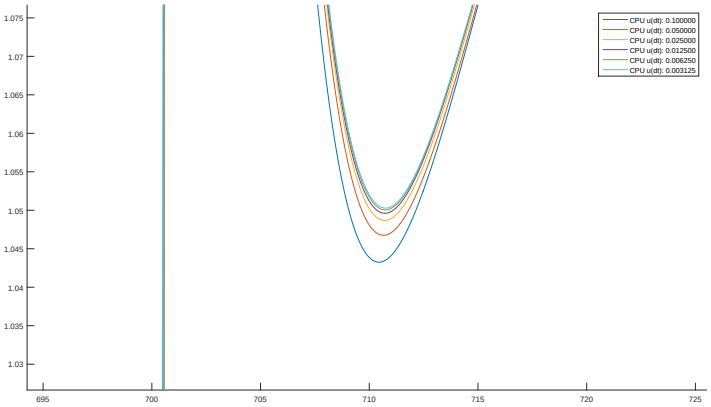

Figure 4.3: Overlapped Plotted Signals (Zoomed)

Using this method, the most accurate step size can be estimated, as shown in Figure 4.4, Figure 4.3 shows the approximation of the overlapping signals of the Minimal Model.

The accuracy of the computation and therefore the optimal step size also depends on the model used. As mentioned before, we tested the accuracy using three different cardiac models. For the Karma and the Minimal Model, we started with a dt of 0.1ms, while the Beeler-Reuter Model could only produce valid values starting by a dt of 0.025ms. The respective step sizes were then decreased and compared using the *least square error* method explained previously. The errors were then plotted against the step size used as reference. Figure 4.7 presents the error for the Minimal Model, Figure 4.6 for the Karma Model and Figure 4.5 for the Beeler-Reuter Model. As these figures show, the error approaches 0 with decreasing step size. In Figure 4.8 we have plotted the error curve of the Karma Model versus the error curve of the Minimal Model, as these Models use the same start step size. The scale of the error indicates, that a smaller step size is needed, when using a higher number of differential equations.

The errors according to Model and dt are further shown in Table 4.1.

However, each time we reduce the step size of a half we double the number of iterations. The optimal step size should balance computation load and accuracy. To bypass the increased computation load, some features of Maxeler can be leveraged. As mentioned

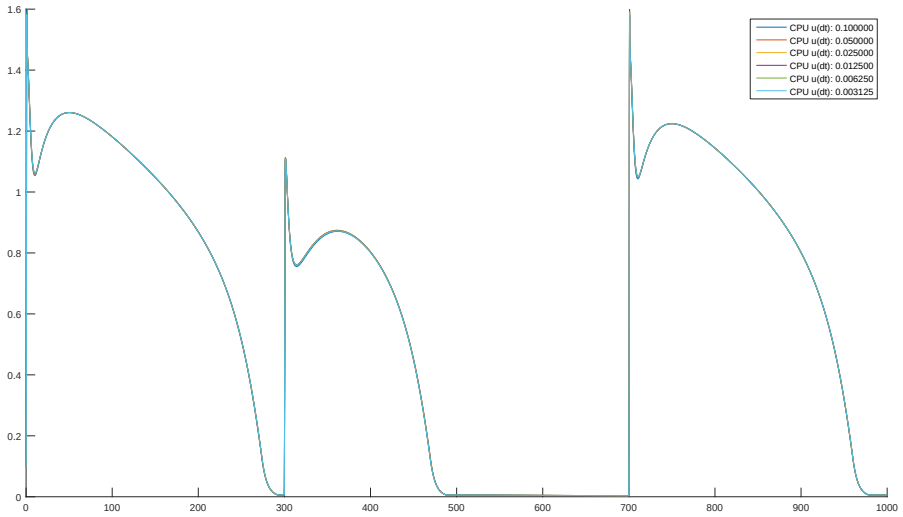

Figure 4.4: Overlapped Plotted Signals

	Karma Model	**Minimal Model**	**Beeler Reuter Model**
0.1 vs. 0.05	0.000216981	0.000140867	-nan
0.05 vs. 0.025	2.73066e-05	1.48384e-05	-nan
0.025 vs. 0.0125	3.40385e-06	1.70065e-06	0.35568
0.0125 vs. 0.00625	4.27584e-07	2.03606e-07	0.0225861
0.00625 vs. 0.003125	5.32403e-08	2.49297e-08	0.00267601
0.003125 vs. 0.0015625	6.68467e-09	3.0784e-09	0.000326343
0.001563 vs. 0.000781	8.35661e-10	3.82951e-10	4.03099e-05

Table 4.1: LSE Per Model and Step Size

before, this technology enables setting the variable precision itself. Generally floating point describes a method to represent real numbers, by approximating the number using the exponential format: $x = s \cdot m \cdot b^e$. Basically, the format uses a *sign bit* s, *exponent* e and *mantissa* m and a *bias* b. Depending on the size (in bits) of mantissa and exponent the range and magnitude of the underlying real number can be approximated. Further, the width of the *mantissa* defines the precision of the represented number. In the C-programming language the IEEE 754 floating point format is used, which uses a bias of 2 [35]. Further, this format defines single and double precision for the according C data types *float* and *double*. The single precision is a 32bit representation and splits into an 8bit exponent and a 24bit mantissa, where 23bits are explicitly stored and one bit is used as significant bit [35]. In the double-step experiments above, we use double precision. The double precision in IEEE 754 has an overall size of 64bits, using 11bits for

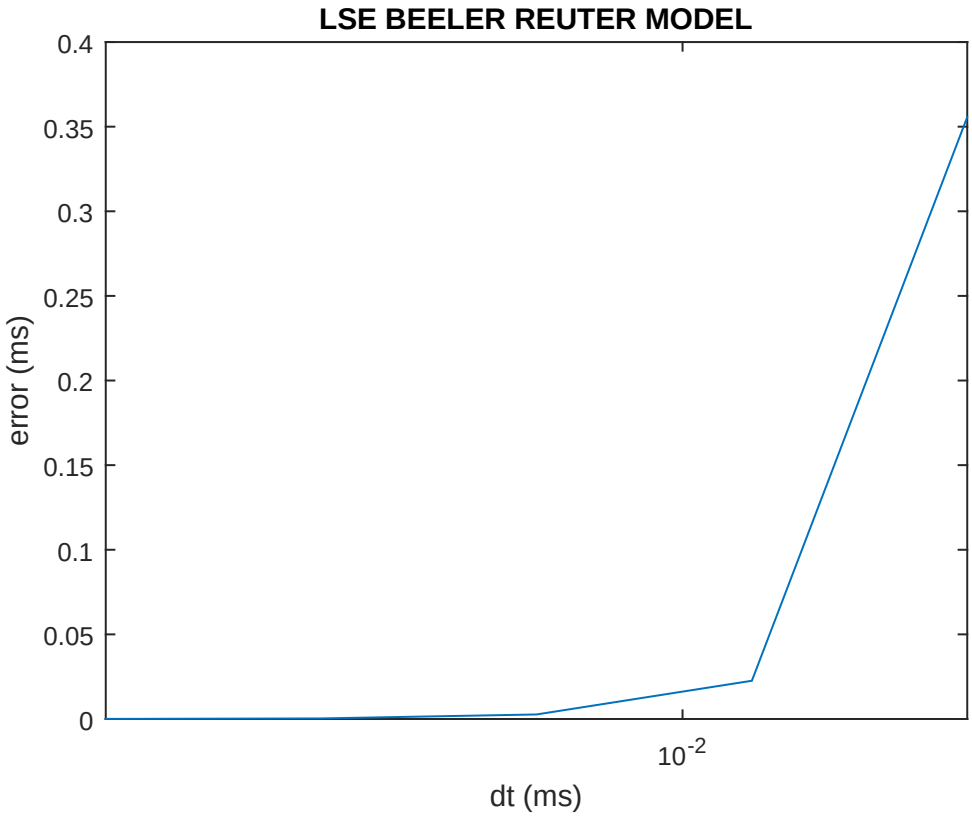

Figure 4.5: Least Square Error: Beeler-Reuter

the exponent and 53 for the mantissa including the significant bit [35].

To evaluate the accuracy of the Maxeler computations we computed the voltage variable for each model and decreased the bit size of the floating point representation on Maxeler side. The computed values were then transformed back to an IEEE representation, which can be processed on CPU side by using the typecast method in Maxeler. Using the least square error calculation of the form $u_i - \widehat{u}_i$, we have then computed the error by model and bit size between the computations of the double precision C of the double-step experiment before using the same *dt*. As we want to keep the accuracy of the different models, we choose a *dt* of $\frac{0.1}{64}$(0.001563), as this results in the smallest error within the experiments before. We have used 64bit according to the IEEE 754 representation, with an 11bit exponent and a 53bit mantissa. As shown in Table 4.2, the error in all models is nearly negligible. Although it is evident that the Maxeler computation is not exactly as precise as the C implementation. We conclude, this is due to different methods in rounding and Maxeler uses partly fixed point representation to accelerate the calculation. When decreasing the precision to 56bit, we have used one representation with an 11bit exponent and a 45bit mantissa, as well as a representation with an 8bit exponent and a 48bit mantissa. As these are not IEEE conform floating point types, we have used

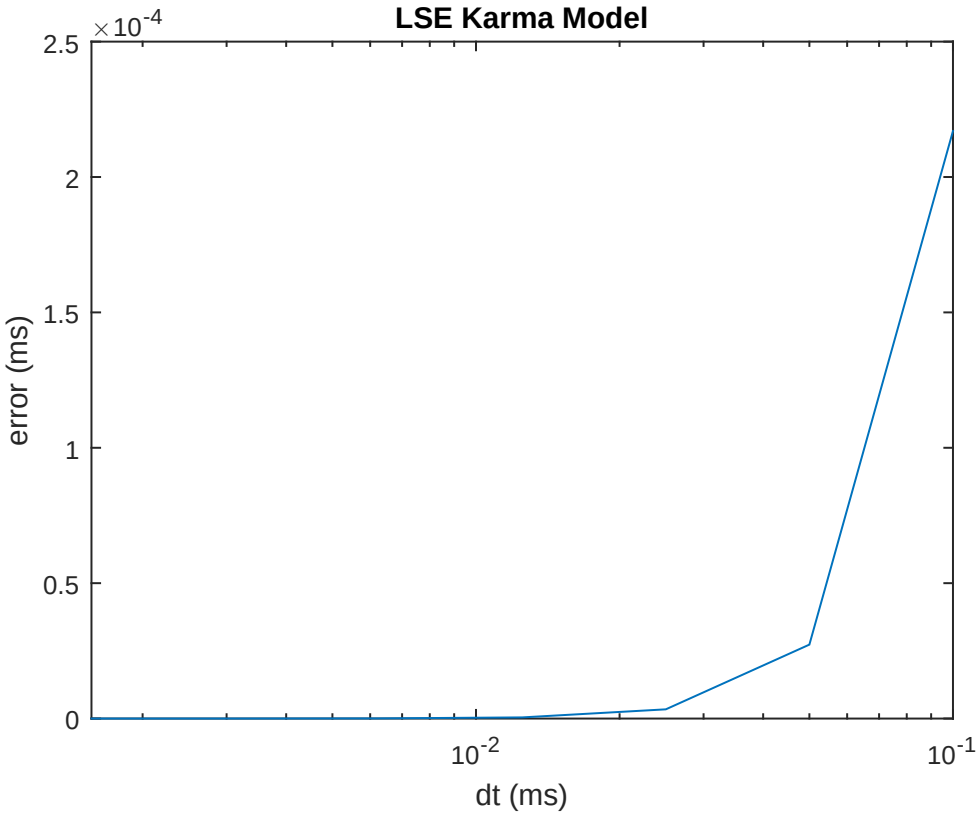

Figure 4.6: Least Square Error: Karma

	Karma Model	Minimal Model	Beeler-Reuter Model
64bit (11,53)	7.45323e-27	5.84172e-27	4.16243e-23
56bit (11,45)	2.73089e-16	1.89358e-18	5.12611e-15
56bit (8, 48)	6.04859e-19	7.93174e-20	9.53619e-17
40bit (8,40)	4.37402e-09	2.47023e-10	2.4158e-06
32bit (8,24)	0.00211634	0.000375366	667.813
20bit (8,12)	1.49361e+06	722628	5.40494e+09

Table 4.2: LSE CPU vs. Maxeler varying Floating Point

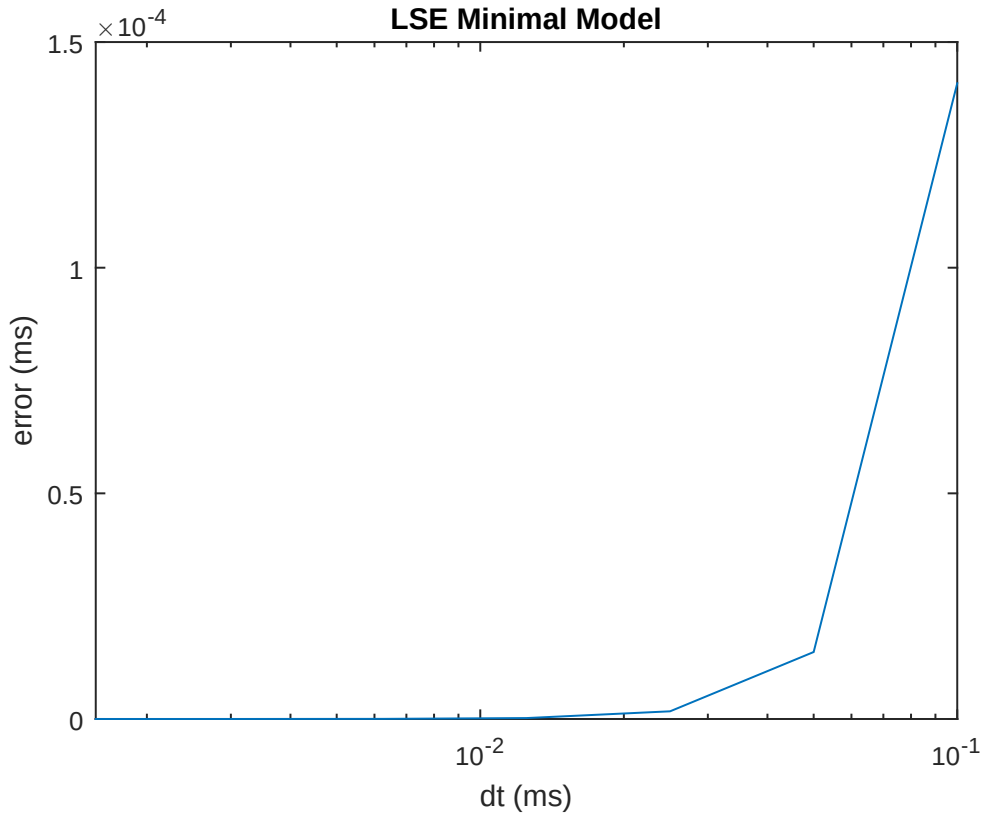

Figure 4.7: Least Square Error: Minimal

Maxeler's typecasting method to convert the results. Furthermore, we need to fit the bits of the representations accordingly to the data type, as we will create an overflow or underflow otherwise. Therefore, we have used IEEE 754 double precision on the CPU side for the streams, as single precision results in an overflow.

The error between the 64bit CPU computation and the converted 56bit representation is still within a negligible range.

Starting with the 48bit (8bit exponent, 40bit mantissa) representation, we switched to single precision on CPU side, as using double precision results in an underflow and non-valid numbers. Although the results will be rounded to a valid single precision representation, the error is still relatively small.

The 32bit representation is again according to the IEEE standards using an 8bit exponent and a 24bit mantissa. As we use double precision for comparison, the error is now significantly higher than before. Basically the representation size was cut in half, which leads to a loss in precision on a larger scale. Especially the Beeler-Reuter Model shows an high error between the different precisions.

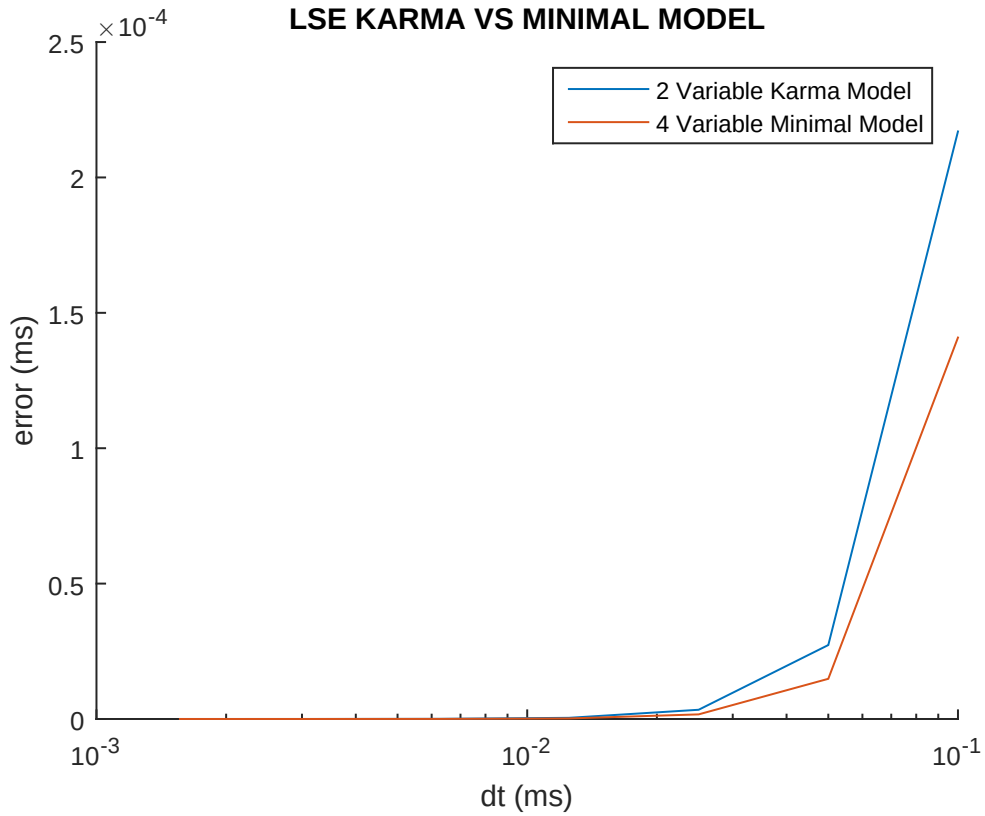

Figure 4.8: Least Square Error: Karma VS Minimal

The last test only uses a 20bit representation with an 8bit exponent and a 12bit mantissa. The IEEE half-precision standard defines a 5bit exponent and an 11bit mantissa, this results in an error within Maxeler, as some library functions need at least 20bits for their fixed point operations. Therefore we halved the mantissa size of the 32bit representation to create a 20bit floating point type. Due to the size and type conversion, the calculated results with the 20bit floating point type are invalid. This is shown in Figure 4.9, as the values do not approximate the initial value as supposed to. We have overlapped the values of the 20bit computation with the double precision of the CPU as well as the single precision of Maxeler. The curve created by the single precision of Maxeler nearly matches the double precision of the CPU, but it is evident that the 20bit precision is invalid. These invalid values are most likely due to a truncation within the type conversion.

Additionally Table 4.2 shows, that the Minimal Model by [9] has the smallest error. The Karma Model by [25] still has a relatively small error, whereas the Beeler-Reuter Model by [6] exceeds both with a magnitude of 10^3.

We have also computed the least square error for the double precision in CPU and Maxeler, to estimate the accuracy within the cable. The overall error is 0.886ms within

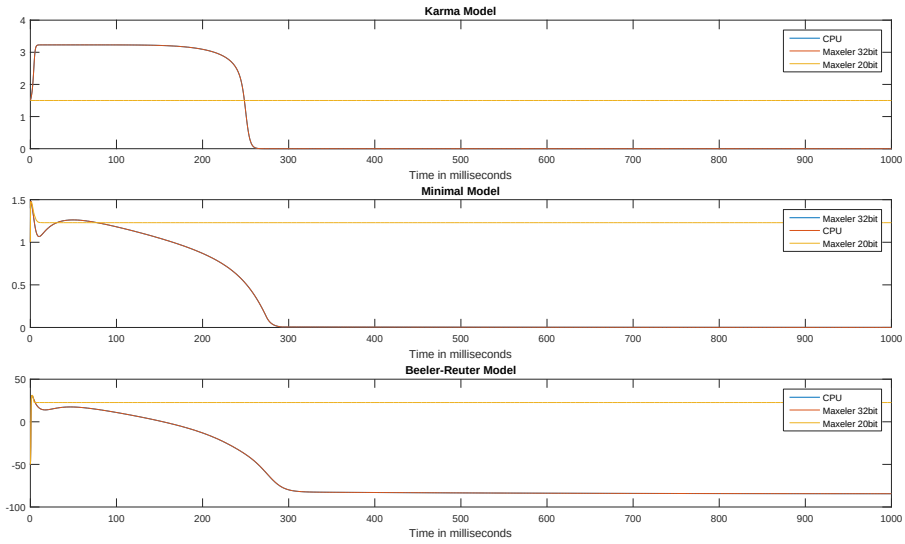

Figure 4.9: 20Bit Precision Maxeler vs. 32Bit Precision Maxeler vs. 64Bit Precision CPU

the simulator using 256 cells, 2ms simulation time and a stimulus at t_0. This is significantly higher as in the single cell implementation, the cable code is based on. It is also interesting to note, that the error is the same for single and double precision. Therefore, we assume the error is most likely caused by the read/write operations on the FMem in Maxeler to retrieve past results. We tried different settings to access those values, but the error remains higher than in the single cell. Further, we know that the design for the cable is not optimal yet and needs more improvement to achieve more accurate calculations and even better performance. Besides the overall error, we have calculated the least square error per cell, which is represented for the first 50 cells in Table 4.3. The error decreases, as the voltage value decreases, also, the highest error is within the first few cells, which supports the assumption that the read/write operation on the FMem are faulty.

Performance

In this work, we also want to provide insights of the performance of Maxeler Technology. To do so, several aspects have been considered. First we have compared the performance of the Maxeler implementation to state-of-the-art CPU and GPU, using the implementations presented in the previous section. Secondly we want to test what the maximum number of cells is we can fit on the Maxeler chip. The evaluated performance aspects include the elapsed time of each technology in single and double precision. Before discussing the experiments in detail, a quick survey of the technical specification of the hardware we used. The tests for CPU were performed using an Intel® Xeon® Processor E5-1650 with approximately 153.6 GFLOPS according to [15]. As stated before, we use a Nvidia GPU for computation and evaluation, specifically a Nvidia GTX 1080. This GPU has

#	Error (double)	Error (single)
1	0,0167969151309800	0,0167966950861100
2	0,0154203464744700	0,0154202625567400
3	0,0208988732982000	0,0208977765592000
4	0,0286829759507301	0,0286821124827301
5	0,0436149882650100	0,0436123127200100
6	0,0778836336627777	0,0778803260058655
7	0,140781528526321	0,140776635410311
8	0,210837307600424	0,210826195943424
9	0,244888703586280	0,244876914539315
10	0,0642377885430561	0,0642359667852288
11	0,0175840114192383	0,0175837206008926
12	0,00453843488561558	0,00453838384733089
13	0,000551844764177748	0,000551840573578065
14	6,41268019342880e-05	6,41263690064864e-05
15	7,19906169253803e-06	7,19902963565275e-06
16	7,83560989223594e-07	7,83557870449281e-07
17	8,23814225766203e-08	8,23812589531544e-08
18	8,30441744568466e-09	8,30441142468528e-09
19	7,97092498415700e-10	7,97091956377211e-10
20	7,24807990301287e-11	7,24807054454869e-11
21	6,22344789548838e-12	6,22345007066092e-12
22	5,03644050072605e-13	5,03644050131583e-13
23	3,83853918268983e-14	3,83854030115307e-14
24	2,75522737974811e-15	2,75522751057722e-15
25	1,86351402203399e-16	1,86351705745980e-16
26	1,18877989074805e-17	1,18878279392810e-17
27	7,16114747844688e-19	7,16115696095940e-19
28	4,07908020910658e-20	4,07909296800315e-20
29	2,20027996295321e-21	2,20028652924478e-21
30	1,12558573760112e-22	1,12558649925758e-22
31	5,46921159317134e-24	5,46924398175680e-24
32	2,52793548864371e-25	2,52794377226583e-25
33	1,11312852312921e-26	1,11313038193081e-26
34	4,67620945846979e-28	4,67621079762535e-28
35	1,87674406333610e-29	1,87674724275388e-29
36	7,20560272477178e-31	7,20561704641875e-31
37	2,64999375190809e-32	2,64999477021351e-32
38	9,34689937925274e-34	9,34697864581466e-34
39	3,16561964737889e-35	3,16562244033944e-35
40	1,03061693170021e-36	1,03062775602653e-36
41	3,22900023783121e-38	3,22901129818564e-38
42	9,74583710489656e-40	9,74585478628063e-40
43	2,83637143904863e-41	2,83643969528066e-41
44	7,96825124647610e-43	7,96833751927583e-43
45	2,16251466233565e-44	2,16252941250123e-44
46	5,67487692058170e-46	5,67495264779772e-46
47	1,44116356854242e-47	1,44117592322828e-47
48	3,54475548457387e-49	3,54481562810316e-49
49	8,45104306355951e-51	8,45121004163940e-51
50	1,95433490769663e-52	1,95437236455486e-52

Table 4.3: Per Cell Error for the first 50 Cells with 2ms Simulation Time

approximately 8.873 GFLOPS in single precision according to [2]. Further, we were able to use a Maxeler card provided by Ivan Milankovic from Maxeler Technologies. The card model is a MAX5C card and uses a Xilinx VU9P FPGA.

To run a DFE project, it needs to be compiled first, to generate the MaxFile according to the design described with the Manager and the Kernel. The compiled and generated MaxFile is then loaded and run via the SLiC. This compilation can take several hours and depends heavily on the design, optimization techniques in place and precision. Therefore, this process was used to evaluate the effect of different precisions within Maxeler and to test the maximum number of cells. We started with 1million cells for 32bit, 48bit, 56bit and 64bit. This value is set within the Manager, as discussed in Chapter 3 to allocate the BRAM we use for the dependency implementation.

Figure 4.12 shows the hardware compilation results for the project using single precision and a maximum of 1million cells. Furthermore, when compiling the 64bit version of

```
ERROR: [Synth 8-5834] Design needs 243645 RAMB18 which is more than device capacity of 4320
```

Figure 4.10: Error 64bit Maxeler 1Million Cells

.

```
ERROR: [Place 30-640] Place Check : This design requires more RAMB18 and RAMB36/FIFO cells than
are available in the target device. This design requires 4359 of such cell types but only 4320
compatible sites are available in the target device.
```

Figure 4.11: Error 64bit Maxeler 600 000 Cells

.

```
FINAL RESOURCE USAGE
Logic utilization:       71833 / 1182240 (6.08%)
  LUTs:                  45926 / 1182240 (3.88%)
  Primary FFs:           51814 / 2364480 (2.19%)
DSP blocks:                151 / 6840    (2.21%)
Block memory (BRAM18):    3695 / 4320    (85.53%)
Block memory (URAM):       615 / 960     (64.06%)
```

Figure 4.12: Successful Build 32Bit Maxeler 1million cells

the cell cable, we had to reduce the maximum cell count, as the design would require more resources than available, the according error message is presented in Figure 4.10. The error in Figure 4.11 was the result of reducing the maximum cell count to 600 000. Therefore, we have reduced the cell count once more and where able to successfully build the 64bit version with 500 000. We also got insights how the precision affects the compilation time. As the 32bit version not only can fit more cells on the chip, but also was faster than the 64bit. We have also tried to build the designs with 48bit and 56bit, which failed, due to the same reason as the 64bit version. Further, the runtime of the DFE is not affected by the bit size of the floating point type used, as the runtime of the design only depends on the data amount and frequency of the Kernel-ticks. Therefore we did not proceed to build the 48bit and 56bit project.

The second part of the evaluation is to compare our Maxeler design to given State-Of-The-Art implementations. This evaluation consists of several test cases. First, we have simulated the cable using CPU in single and double precision for 2ms, a step-size of 0.0125ms and the maximum of 1million cells. We then run the simulation on the 32bit and the 64bit version of the Maxeler design to gain a general insight about the performance. Note, that we have a different maximum using double precision in Maxeler and therefore one result less. The times are presented in Table 4.4. Evidently, the Maxeler implementation is slower than expected. Using the 32bit version of our design for the cable, we cut the double precision using CPU in half, but cannot compete with the times of the single precision. Still, both the single precision in Maxeler and the double precision have better times than the double precision in CPU.

It is evidently that the Maxeler is not as efficient as we assumed. Concluding on these

Cell Count	CPU (single)	CPU (double)	Maxeler (single)	Maxeler (double)
2^8	16	293	64	103
2^9	25	380	127	207
2^{10}	40	738	254	413
2^{11}	63	1415	508	826
2^{12}	82	2806	1016	1652
2^{13}	150	5628	2032	3306
2^{14}	300	11375	4066	6615
2^{15}	606	22512	8135	13232
2^{16}	1128	45030	16274	26466
2^{17}	2257	90524	32535	52919
2^{18}	4508	180414	65082	105838
2^{19}	9010	329860	130125	

Table 4.4: Comparison Computation Time (MS) with increasing Cell Number 2ms Simulation Time

results, the design for the simulation of a cell cable using Maxeler presented in this work is not optimal. The runtime performance of a DFE heavily depends on the data amount, the design and the frequency of the Kernel-tick. Due to these results and some revision of the generated dataflow graphs, we discovered the bottleneck within the Maxeler design. As mentioned in Chapter 3 we have used the "autoloop offset" to loop back the data into the Kernel. Further, we already know that using the "autoloop offset" can result in performance deficiencies, which we clearly underestimated. After examining the generated graphs and the MaxFile, we have noticed, that we only produce a value every 155th Kernel tick within the 32bit design and ever 252th Kernel tick within the 64bit design. To fully leverage the acceleration of Maxeler, it is necessary to produce a valid result every Kernel tick. As we were not able to redesign the implementation and maintain a similar accuracy we tried to focus on the data amount and the frequency to achieve an acceleration. Therefore we rerun the tests, with a reduced maximum number of cells in the 32bit and 64bit version. Additionally we have increased the frequency within the Manager. The results of this test case where then compared to the times of the GPU, the according times are presented in Table 4.5.

Using a higher stream frequency, increases the values outputted by the Kernel. The design still cannot compete with GPU and CPU but we decreased the runtime of the Maxeler design significantly. Additionally, we have improved the resource utilization, as we use less cells for allocation. The final build results for the improved design are shown in Figure 4.13 for the 32bit design and Figure 4.14 for the 64bit design.

We have used the 2ms simulation to estimate the general speed of our design. To compare the general performance of Maxeler with a large dataset, we have used the improved design and increased the simulation time to 1000ms, reusing the other parameters. These

Cell Count	Maxeler (single)	Maxeler (double)	GPU (single)	GPU (double)
2^8	53	86	7.88899	4.74992
2^9	106	172	10.7437	7.00208
2^{10}	211	344	15.2052	10.2452
2^{11}	423	688	21.0274	13.8904
2^{12}	847	1376	30.761	21.3341
2^{13}	1693	2753	47.5963	37.3473

Table 4.5: Comparison Computation Time (MS) with increasing Cell Number 2ms Simulation Time (improved)

```
FINAL RESOURCE USAGE
Logic utilization:       56877 / 1182240 (4.81%)
  LUTs:                  33302 / 1182240 (2.82%)
  Primary FFs:           47151 / 2364480 (1.99%)
DSP blocks:                151 / 6840    (2.21%)
Block memory (BRAM18):     163 / 4320    (3.77%)
Block memory (URAM):        10 / 960     (1.04%)
```

Figure 4.13: Build Results 32Bit Maxeler 10 000 cells

```
FINAL RESOURCE USAGE
Logic utilization:      130839 / 1182240 (11.07%)
  LUTs:                  87267 / 1182240 (7.38%)
  Primary FFs:           87144 / 2364480 (3.69%)
DSP blocks:                360 / 6840    (5.26%)
Block memory (BRAM18):     192 / 4320    (4.44%)
Block memory (URAM):        18 / 960     (1.88%)
```

Figure 4.14: Build Results 64Bit Maxeler 10 000 cells

simulations were also performed on CPU and GPU to compare all three technologies regarding computation performance. The results are presented in Table 4.6. For a better visualization we compared the results in a bar graph, presented in Figure 4.15 for the single precision and Figure 4.16 for the double precision test.

As shown in Figure 4.16 the Maxeler design failed to compute valid values with 2^{13} cells.

Cell Count	Maxeler (single)	Maxeler (double)	GPU (single)	GPU (double)	CPU (single)	CPU (double)
2^8	26462	43028	2224.2	2168.96	2571	2021
2^9	52927	86047	2937.91	2968.86	4826	3636
2^{10}	105859	172121	4417.29	4680.35	9666	7272
2^{11}	211718	344253	6047.39	6439.36	19331	14547
2^{12}	423382	688513	9602.48	10046.6	38660	29151
2^{13}	846828		16179	17624	77303	58299

Table 4.6: Comparison Computation Time (MS) with increasing Cell Number 1s Simulation Time

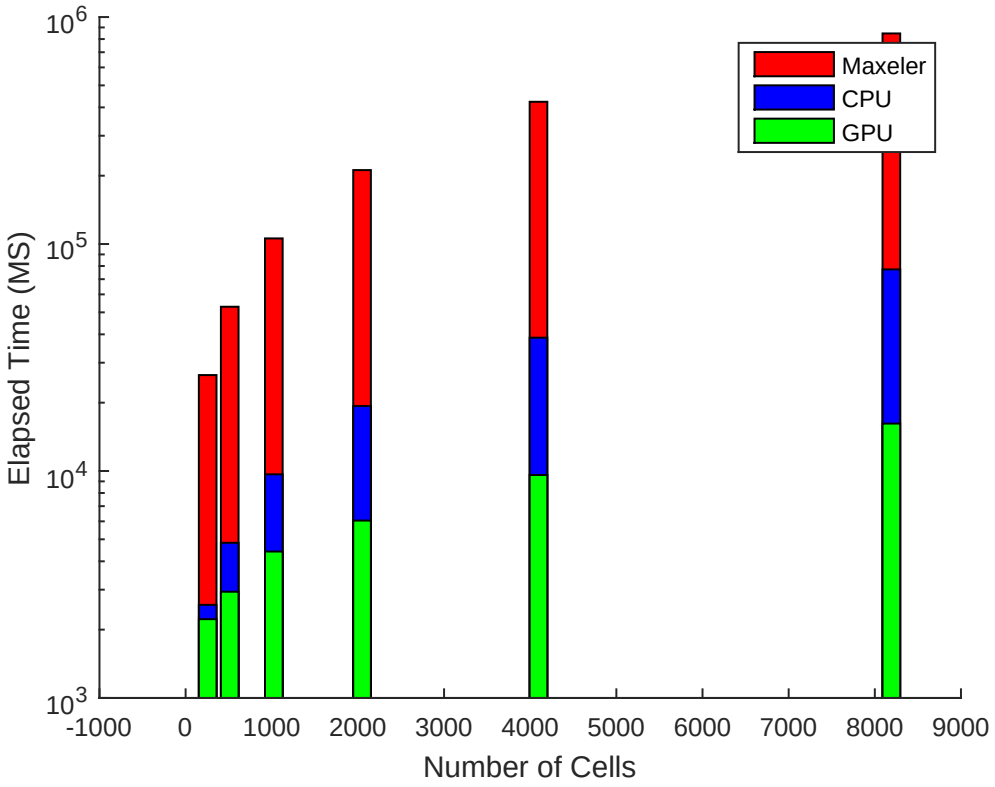

Figure 4.15: Performance Comparison Single Precision

Additionally, we have measured the calculation time of the single cell. This should give insights, how the base design is affected by the usage of the "autoloop offset". First, the build results for the single cell in 32bit are presented in Figure 4.18 and for the 64bit result in Figure 4.17. We then examined the generated MaxFile to receive the values of the loop length of each design. For the 32bit design the loop length is 255 and for the 64bit 304. These are not only higher than for the cell cable, but also means we only produce a value once the Kernel loop counter reaches this value.

For the evaluation of the isolated single cell, we simulated the cell for 1 cycle (1000ms) and started with the same step size as in the experiments for the cell cable, 0.0125ms. Then we decreased the step size to achieve a higher computation load, as the number of iteration increases with decreasing step size. The run time of both designs compared to CPU are presented in Table 4.7.

As presented in Table 4.7, the isolated single cell, which is also the basis for the cell cable, does not perform well either. This leads to the conclusion of revising the isolated single cell and reorganize our data flow. Thereby we can improve the design without using "autoloop" and accelerate the design. Based on the improved isolated single cell, we then

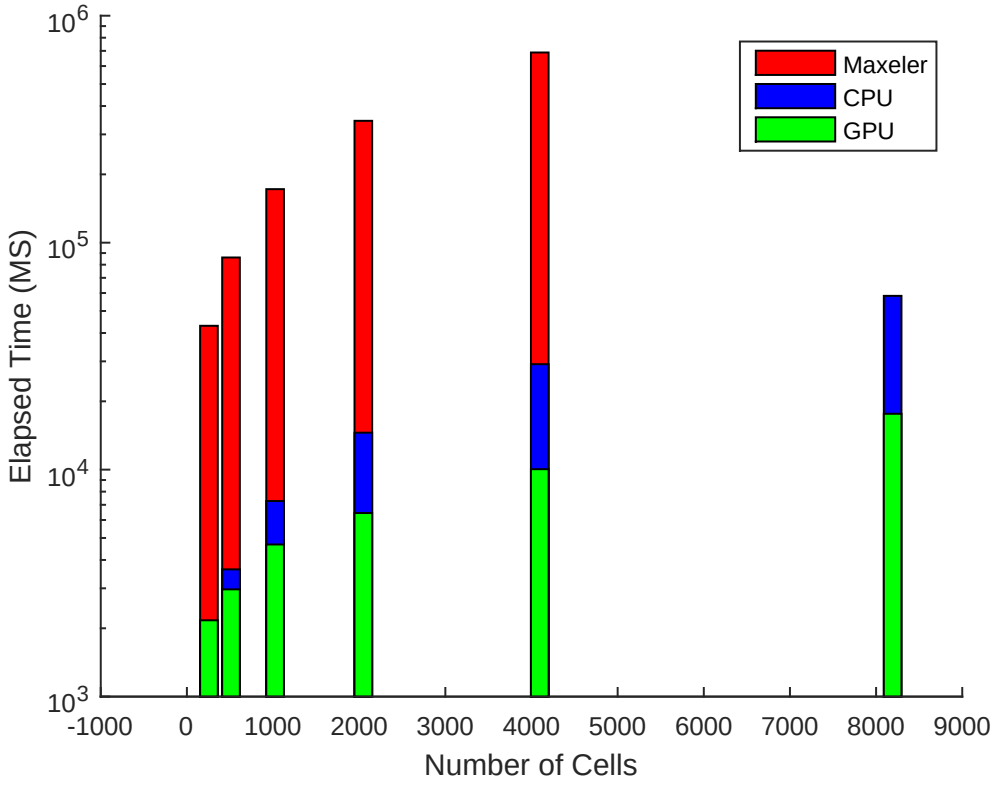

Figure 4.16: Performance Comparison Double Precision

```
FINAL RESOURCE USAGE
Logic utilization:      120653 / 1182240 (10.21%)
  LUTs:                  76003 / 1182240 (6.43%)
  Primary FFs:           89300 / 2364480 (3.78%)
DSP blocks:                264 / 6840    (3.86%)
Block memory (BRAM18):     138 / 4320    (3.19%)
Block memory (URAM):         5 / 960     (0.52%)
```

Figure 4.17: Build Results 64Bit Maxeler Isolated Single Cell

```
FINAL RESOURCE USAGE
Logic utilization:      120653 / 1182240 (10.21%)
  LUTs:                  76003 / 1182240 (6.43%)
  Primary FFs:           89300 / 2364480 (3.78%)
DSP blocks:                264 / 6840    (3.86%)
Block memory (BRAM18):     138 / 4320    (3.19%)
Block memory (URAM):         5 / 960     (0.52%)
```

Figure 4.18: Build Results 32Bit Maxeler Isolated Single Cell

DT	Maxeler (single)	Maxeler (double)	CPU (single)	CPU (double)
0.0125	204	244	20	14
0.00625	409	487	35	25
0.003125	817	974	55	42

Table 4.7: Performance Single Cell Evaluation

can build an improved 1D homogeneous cell cable.

CHAPTER 5

Conclusion and Future Work

5.1 Conclusion

In this work we wanted to provide insights regarding the applicability of Maxeler Technologies for simulating cardiac dynamics. Therefore we re-implemented known state-of-the-art algorithms for different models as single cell and one as 1D cell cable in data flow. We have tested those implementations in the simulator as well as a real DFE regarding different aspects. The conducted experiments should provide a survey of disadvantages and advantages regarding Maxeler in comparison to Stat-Of-The-Art technologies. First we successfully implemented three different models as single cell using Maxeler Dataflow Technology. We then compared the accuracy of those models with CPU and proved a sufficient precision with several floating point types. We could also estimate the applicability and the efficiency using different models, according to the error. As a model with higher complexity will produce a higher error compared to CPU. This provides several advantages, like setting the accuracy arbitrarily in the implementation.

In the first experiments, we used the simulator provided by *Maxeler Technologies*. We then discovered that the simulator should not be used to compare performance and resource usage. Since, the simulator will assume an infinite amount of resources but will run in a time out error when the actual resources are exhausted or the computation load is too high. Therefore we have used the simulator for general tests and to experiment with the precision. As presented in the previous chapter, the accuracy within Maxeler Technology is quite good, if the chosen precision does not differ too much from the reference. For example, using *double* precision in CPU and Maxeler has a neglectable error. Whereas mixing *single* precision and double precision results in a quite high error. Choosing a too small bit size results in completely wrong calculation results. Although, these features can be leveraged in different points of the calculation. For example, using a smaller precision to accelerate calculations can be a performance advantage.

We know now that the design of a data flow application requires deep knowledge of the underlying algorithm. Acquiring this knowledge can be a disadvantage, as extensive code reviews are necessary to create an optimal design. Additionally, accessing previous values within a calculation can be problematic. Maxeler Technology provides different ways to access values of past ticks and even CPU site-data, but the programmer needs to know exactly how to access those values. Otherwise, the calculations will not be correct or even result in undefined behavior. The design presented in this work for the cell cable is definitely not optimal. Due to the use of the "autoloop offset" generated by Maxeler, we have created a computation bottleneck, which compromises the acceleration. Although the performance of the design does not prove to be as expected, we successfully created a simulation of a simple 1D homogenous cell cable and an isolated single cell in Maxeler. Compared to State-Of-The-Art technologies, our design does not prove to be more efficient, but at least as precise and not necessarily less efficient. Based on the results, we conclude there is more research necessary regarding the usage of Maxeler to simulate cardiac dynamics.

5.2 Future Work

To fully evaluate if Maxeler Dataflow Technology is applicable to use in cardiac simulation further research is needed. In this work we presented an approach to simulate a 1D cable using data flow. An important part of researching cardiac dynamics is the simulation of more complex cell structures. Therefore future work includes the implementation of multidimensional structures, even tissue with different cell types. Additionally, we have only compared the Maxeler implementation to GPU and CPU. The implementation of cardiac dynamics using FPGA is a challenge due to its complexity. Therefore it was omitted in this work, but as this technology is also State-Of-The-Art in simulation acceleration, a comparison would be interesting.

The design of the Maxeler implementation is still not perfect, especially due to the dependencies within the cell cable. Therefore the design has to be optimized to achieve more an accurate and accelerated simulation. The essential bottleneck within the design is the "autoloop offset" we have used to connect the depended parts within the design. Restructuring the data flow of the computed values and omitting this offset will accelerate the design drastically. The runtime of the cable design can be calculated by $number\_of\_cells * number\_of\_iterations * loopLength$, omitting the loopLength in this equation, would result in an acceleration of 155 times within the 32bit design and 252 times within the 64bit design. In combination with an increased number of pipelines, which means more parallelism within the design, we can even achieve an higher acceleration of the design. This requires a whole reimplementation and redesign of the implementation presented in this work.

Moreover, we only used the Minimal Model by [9] for the cell cable. In this work we already tested different Models using a single cell implementation, to evaluate the accuracy and applicability of Maxeler. Based on this, a multidimensional implementation of a

cable or even tissue using different models should be possible using an improved design. These can give further insights of the performance of Maxeler DFE, as the computation load changes with the number of state-variables.

List of Figures

List of Tables

Appendix A: Maxeler Isolated Single Cell

Minimal Model

Manager Class Minimal Model

```
package minimal.model.cell;

import com.maxeler.maxcompiler.v2.build.EngineParameters;
import com.maxeler.maxcompiler.v2.managers.custom.DFELink;
import com.maxeler.maxcompiler.v2.managers.custom.blocks.KernelBlock;
import com.maxeler.maxcompiler.v2.managers.engine_interfaces.CPUTypes;
import com.maxeler.maxcompiler.v2.managers.engine_interfaces.EngineInterface;
import com.maxeler.maxcompiler.v2.managers.engine_interfaces.InterfaceParam;
import com.maxeler.platform.max5.manager.MAX5CManager;

public class MinimalModelManager extends MAX5CManager {

        public static final String s_kernelName = "MinimalModelDFEKernel";

        public MinimalModelManager(EngineParameters arg0) {
                super(arg0);
                KernelBlock k = addKernel(
                                new
                                ↪  MinimalModelKernel(makeKernelParameters(s_kernelName)));

                DFELink y1 = addStreamToCPU("u_out");
                y1 <== k.getOutput("u_out");
                DFELink y2 = addStreamToCPU("v_out");
                y2 <== k.getOutput("v_out");

                DFELink y3 = addStreamToCPU("w_out");
                y3 <== k.getOutput("w_out");

                DFELink y4 = addStreamToCPU("s_out");
                y4 <== k.getOutput("s_out");

                //DFELink y5 = addStreamToCPU("stim_out");
                //y5 <== k.getOutput("stim_out");
```

```
    }


    public static void main(String[] args) {
        EngineParameters params = new EngineParameters(args);
        MinimalModelManager manager = new
        ↪  MinimalModelManager(params);

        // Instantiate the kernel


        manager.createSLiCinterface(interfaceDefault());
        manager.build();
    }

    private static EngineInterface interfaceDefault() {
        EngineInterface ei = new EngineInterface();

        InterfaceParam length = ei.addParam("length", CPUTypes.INT);
        InterfaceParam dt = ei.addParam("dt", CPUTypes.DOUBLE);
        InterfaceParam num_iteration = ei.addParam("num_iteration",
        ↪  CPUTypes.INT);
        InterfaceParam lengthInBytes = length *
        ↪  CPUTypes.FLOAT.sizeInBytes();
        InterfaceParam loopLength =
        ↪  ei.getAutoLoopOffset(s_kernelName, "loopLength");
        ei.ignoreAutoLoopOffset(s_kernelName, "loopLength");
        ei.setTicks(s_kernelName, length * loopLength);

        ei.setScalar(s_kernelName, "dt", dt);
        ei.setScalar(s_kernelName, "num_iteration", num_iteration);

        //ei.setStream("stim_out", CPUTypes.DOUBLE, lengthInBytes);
        ei.setStream("u_out", CPUTypes.FLOAT, lengthInBytes);
        ei.setStream("v_out", CPUTypes.FLOAT, lengthInBytes);
        ei.setStream("w_out", CPUTypes.FLOAT, lengthInBytes);
        ei.setStream("s_out", CPUTypes.FLOAT, lengthInBytes);


        return ei;
    }
}
```

Kernel Class Minimal Model

```
package minimal.model.cell;

import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.Optimization.PipelinedOps;
import com.maxeler.maxcompiler.v2.kernelcompiler.op_management.MathOps;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.KernelMath;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.CounterChain;
import
↪  com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.Stream.OffsetExpr;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEType;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.composite.DFEVector;
import
↪  com.maxeler.maxcompiler.v2.kernelcompiler.types.composite.DFEVectorType;

public class MinimalModelKernel extends Kernel {

        static final DFEType scalarType = dfeFloat(8, 24);
         static final DFEType calculationType = dfeFloat(8, 24);
         static final DFEVectorType<DFEVar> vectorType =
         new DFEVectorType<DFEVar>(calculationType, 4);


        //Constants
         DFEVar TVP = constant.var(calculationType,      1.4506); //The same
         ↪  with Flavio's paper
         DFEVar TV1M  =  constant.var(calculationType,  60)    ;//The same
         ↪  with Flavio's paper
         DFEVar TV2M  =  constant.var(calculationType,  1150)  ;//The same
         ↪  with Flavio's paper

         DFEVar TWP   =constant.var(calculationType,    200.0);   // We have
         ↪  TWP1 and TWP2 = TWP in Flavio's paper

         //#define TW1P     200.0
         //#define TW2P     200.0

         DFEVar TW1M =   constant.var(calculationType,   60.0);    //The same
         ↪  with Flavio's paper
         DFEVar TW2M =constant.var(calculationType,      15);     //The same
         ↪  with Flavio's paper
         DFEVar TS1 = constant.var(calculationType,       2.7342);  //The
         ↪  same with Flavio's paper
         DFEVar TS2 =  constant.var(calculationType,    16);     //The same
         ↪  with Flavio's paper
         DFEVar TFI = constant.var(calculationType,      0.11);    //The same
         ↪  with Flavio's paper
         DFEVar TO1 =  constant.var(calculationType,      400);    //The same
         ↪  with Flavio's paper
         DFEVar TO2 =    constant.var(calculationType,  6);      //The same
         ↪  with Flavio's paper
```

```
DFEVar TSO1 =   constant.var(calculationType,  30.0181); //The same
↪  with Flavio's paper
DFEVar TSO2 =   constant.var(calculationType,  0.9957);  //The same
↪  with Flavio's paper

DFEVar TSI  =    constant.var(calculationType, 1.8875 ); // We have
↪  TSI1 and TSI2 = TSI in Flavio's paper
//#define TSI1     1.8875
//#define TSI2     1.8875


DFEVar TWINF  =  constant.var(calculationType, 0.07);    //The same
↪  with Flavio's paper
DFEVar THV  =    constant.var(calculationType, 0.3);     //EPUM //
↪  The same of Flavio's paper
DFEVar THVM  =    constant.var(calculationType, 0.006);   //EPUQ //
↪  The same of Flavio's paper
DFEVar THVINF  = constant.var(calculationType, 0.006);   //EPUQ //
↪  The same of Flavio's paper
DFEVar THW   =   constant.var(calculationType, 0.13);    //EPUP //
↪  The same of Flavio's paper
DFEVar THWINF  = constant.var(calculationType, 0.006);    //EPURR //
↪  In Flavio's paper 0.13
DFEVar THSO =   constant.var(calculationType,  0.13 );   //EPUP //
↪  The same of Flavio's paper
DFEVar THSI =    constant.var(calculationType,  0.13 );   //EPUP //
↪  The same of Flavio's paper
DFEVar THO =    constant.var(calculationType,  0.006);    //EPURR //
↪  The same of Flavio's paper
//#define KWP      5.7
DFEVar KWM  =    constant.var(calculationType, 65);    //The same of
↪  Flavio's paper
DFEVar KS  =    constant.var(calculationType,  2.0994);  //The same
↪  of Flavio's paper
DFEVar KSO =    constant.var(calculationType,  2.0458);  //The same
↪  of Flavio's paper
//#define KSI      97.8
DFEVar UWM  =    constant.var(calculationType, 0.03);    //The same
↪  of Flavio's paper
DFEVar US  =    constant.var(calculationType,  0.9087);  //The same
↪  of Flavio's paper
DFEVar UO =     constant.var(calculationType,  0);     // The same
↪  of Flavio's paper
DFEVar UU =     constant.var(calculationType,  1.55);   // The same
↪  of Flavio's paper
DFEVar USO  =     constant.var(calculationType, 0.65);   // The same
↪  of Flavio's paper
DFEVar SC  =    constant.var(calculationType,  0.007);
//#define WCP      0.15

DFEVar WINFSTAR =  constant.var(calculationType, 0.94);   // The
↪  same of Flavio's paper
```

```
 DFEVar TW2M_TW1M =  constant.var(calculationType, -45.0);
 DFEVar TSO2_TSO1 = constant.var(calculationType, -29.0224);

 DFEVar TW2M_TW1M_DIVBY2 = constant.var(calculationType, -22.5);
 DFEVar TSO2_TSO1_DIVBY2 = constant.var(calculationType, -14.5112);

 final DFEVar Zero = constant.var(calculationType, 0.0);
final DFEVar One = constant.var(calculationType, 1.0);



public MinimalModelKernel(final KernelParameters parameters) {
        super(parameters);

                //create autoloop offset to create backwards edge for
                ↪  calculation
                OffsetExpr loopLength =
                ↪  stream.makeOffsetAutoLoop("loopLength"); //
                DFEVar loopLengthVal = loopLength.getDFEVar(this,
                ↪  dfeUInt(11));

                CounterChain chain =
                ↪  control.count.makeCounterChain();
                final DFEVar nx =io.scalarInput("num_iteration",
                ↪  dfeUInt(32));

                DFEVar step = chain.addCounter(nx, 1);//counter for
                ↪  number of steps;
                                                                    //
                DFEVar loopCounter = chain.addCounter(loopLengthVal,
                ↪  1); //counter for validation of values;
                DFEVar dt = io.scalarInput("dt", dfeFloat(11,53));
                dt = dt.cast(calculationType);
        //        DFEVar stim_in =
        ↪  io.input("stim_in",scalarType,loopCounter ===
        ↪  (loopLengthVal-1));

                DFEVector<DFEVar> statevars =
                ↪  vectorType.newInstance(this);
                 DFEVar ui = step===0? One : statevars[0];
                 ↪  //initCondition? 0.0 : carriedU;
             DFEVar    vi = step===0? One : statevars[1];
             DFEVar   wi = step===0 ? One : statevars[2];
             DFEVar     si = step===0? Zero : statevars[3];
                  DFEVar tvm, vinf, winf;
                  DFEVar jfi, jso, jsi;
                  DFEVar twm, tso, ts, to, ds, dw, dv;


                   ↪  optimization.pushPipeliningFactor(1.0,PipelinedOps.ALL);

                   tvm =  (ui >   THVM) ?   TV2M :   TV1M;
```

```
twm =    TW1M + (  TW2M_TW1M_DIVBY2)*(1+tanh( KWM*(ui-
↪  UWM)));
tso =     TSO1 + (  TSO2_TSO1_DIVBY2)*(1+tanh(
↪  KSO*(ui-   USO)));
ts  = (ui >    THW)  ?    TS2  :    TS1;
to  = (ui >    THO)  ?    TO2  :    TO1;

vinf = (ui >     THVINF) ? Zero : One;
    winf = (ui >     THWINF) ?   WINFSTAR: (1.0-ui/
    ↪  TWINF);
winf =  (winf > One) ? One : winf;

dv = (ui >    THV) ? -vi/   TVP : (vinf-vi)/tvm;
dw = (ui >    THW) ? -wi/   TWP : (winf-wi)/twm;
ds = (((1+tanh(   KS*(ui-   US)))/2) - si)/ts;

//winf = (*ui > 0.06) ? 0.94: 1.0-*ui/0.07;
//twm  =  60 + (-22.5)*(1.+tanh(65*(*ui-0.03)));
//dw   = (*ui > 0.13) ? -*wi/200 : (winf-*wi)/twm;



    //tvm =  (*ui >   THVM) ?   TV2M :   TV1M;
    //one_o_twm = segm_table[0][th] * (*ui) +
    ↪  segm_table[1][th];
    //vinf = (*ui >    THVINF) ? 0.0: 1.0;
    //winf = (*ui >    THWINF) ?   WINFSTAR *
    ↪  one_o_twm: (segm2_table[0][th2] * (*ui) +
    ↪  segm2_table[1][th2]);
    //if (winf >one_o_twm) winf = one_o_twm;
    //dv = (*ui >   THV) ? -*vi/  TVP : (vinf-*vi)/tvm;
    //dw = (*ui >   THW) ? -*wi/  TWP : winf - *wi *
    ↪  one_o_twm;
    //ds = (((1.+tanh(  KS*(*ui-  US)))/2.) - *si)/ts;

optimization.popPipeliningFactor(PipelinedOps.ALL);



    //Update gates

 vi = vi +dv*dt;
 wi = wi + dw*dt;
 si = si+ ds*dt;

    //Compute currents
jfi = (ui >    THV)  ? -vi * (ui -    THV) * (   UU -
↪  ui)/   TFI : Zero;
/*if (*ui >   THV){
            if (((*vi - dv*dt) > 0.0) && *vi <
↪  0.0001){
```

```
                                        printf("vi=%4.20f, ui=%f,
↪  jfi=%f\n", *vi, *ui, jfi);
                                    }
                                }*/
                        jso = (ui >    THSO) ? 1/tso : (ui-   UO)/to;
                        jsi = (ui >    THSI) ? -wi * si/   TSI : 0.0;

                        ui = ui  - (jfi+jso+jsi)*dt;
                       // optimization.popDSPFactor();


                          DFEVar uOffset = stream.offset(ui, -loopLength);
                        DFEVar vOffset = stream.offset(vi, -loopLength);
                        DFEVar wOffset = stream.offset(wi, -loopLength);
                        DFEVar sOffset = stream.offset(si, -loopLength);

                //        DFEVar stimOffset = stream.offset(stim,
                ↪  -loopLength);

                statevars[0] <== uOffset;
                statevars[1] <==vOffset;
                statevars[2] <== wOffset;
                statevars[3] <== sOffset;

                        io.output("u_out", ui.cast(scalarType),
                        ↪  scalarType,loopCounter === (loopLengthVal-1));
                        io.output("v_out", statevars[1].cast(scalarType),
                        ↪  scalarType,loopCounter === (loopLengthVal-1));
                        io.output("w_out", statevars[2].cast(scalarType),
                        ↪  scalarType,loopCounter === (loopLengthVal-1)) ;
                        io.output("s_out", statevars[3].cast(scalarType),
                        ↪  scalarType,loopCounter === (loopLengthVal-1));



                }

                /***
                 *
                 * @param x value to check
                 * @return heaviside(x) returns the value 0 for x < 0, 1 for
↪  x > 0, and 1/2 for x = 0.
                 */

                protected DFEVar heavisidefun(DFEVar x) {

//                        //check if the value is below or above zero
                        DFEVar c = x<0.0? constant.var(dfeBool(), 1) :
                        ↪  constant.var(dfeBool(), 0);
        //
//                        //check if the value is zero
```

```
                        DFEVar z = x===0.0? constant.var(dfeBool(), 1) :
                        ↪  constant.var(dfeBool(), 0);
        //
//                        //assign values regarding to checks - first if
↪  value is below or above zero
                        DFEVar ret = c?constant.var(scalarType, 0.0) :
                        ↪  constant.var(scalarType, 1.0);
        //
//                        //if x is zero, z is true, as c would not consider
↪  this case, we can disregard the value of c if z is true
                        ret = z? constant.var(scalarType, 0.5) : ret;


                        return ret;



                }


                protected DFEVar tanh(DFEVar x) {
                        //tangens hyperbolicus: 1-(2/(e^(2*x)+1))

                                DFEVar v = KernelMath.exp(2*x);
                                DFEVar approximation = 1- (2/(v+1));
                                return approximation;

                }




        }
```

Karma Model

Manager Class Karma Model

```
package karma.model.cell;

import com.maxeler.maxcompiler.v2.build.EngineParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.KernelConfiguration;
import com.maxeler.maxcompiler.v2.managers.custom.CustomManager.Config;
import com.maxeler.maxcompiler.v2.managers.custom.DFELink;
import com.maxeler.maxcompiler.v2.managers.custom.blocks.KernelBlock;
import com.maxeler.maxcompiler.v2.managers.engine_interfaces.CPUTypes;
import com.maxeler.maxcompiler.v2.managers.engine_interfaces.EngineInterface;
import com.maxeler.maxcompiler.v2.managers.engine_interfaces.InterfaceParam;
import com.maxeler.platform.max5.manager.MAX5CManager;

public class KarmaManager extends MAX5CManager{
        private static final String s_kernelName = "KarmaKernel";
        public KarmaManager(EngineParameters arg0) {
                super(arg0);
                KernelBlock k = addKernel(
                                new
                                ↪  KarmaKernel(makeKernelParameters(s_kernelName)));

                addStreamToCPU("u_out") <== k.getOutput("u_out");
                addStreamToCPU("v_out") <== k.getOutput("v_out");


                //config.setDefaultStreamClockFrequency(50);
        }



        public static void main(String[] args) {
                EngineParameters params = new EngineParameters(args);
                KarmaManager manager = new KarmaManager(params);
                KernelConfiguration con = manager.getCurrentKernelConfig();

                // Instantiate the kernel


                manager.createSLiCinterface(interfaceDefault());

                manager.build();
        }

        private static EngineInterface interfaceDefault() {
                EngineInterface ei = new EngineInterface();

                InterfaceParam length = ei.addParam("length", CPUTypes.INT);
                InterfaceParam dt = ei.addParam("dt", CPUTypes.DOUBLE);
                InterfaceParam num_iteration = ei.addParam("num_iteration",
                ↪  CPUTypes.INT);
```

```
                InterfaceParam lengthInBytes = length *
                ↪  CPUTypes.FLOAT.sizeInBytes();
                InterfaceParam loopLength =
                ↪  ei.getAutoLoopOffset(s_kernelName, "loopLength");
                ei.ignoreAutoLoopOffset(s_kernelName, "loopLength");
                ei.setTicks(s_kernelName, length * loopLength);

                ei.setScalar(s_kernelName, "dt", dt);


                ei.setScalar(s_kernelName, "num_iteration", num_iteration);

                ei.setStream("u_out", CPUTypes.FLOAT, lengthInBytes);
                ei.setStream("v_out", CPUTypes.FLOAT, lengthInBytes);



                return ei;
        }
}
```

Kernel Class Karma Model

```
package karma.model.cell;

import com.maxeler.maxblox.funceval.MathPow;
import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.Optimization.PipelinedOps;
import com.maxeler.maxcompiler.v2.kernelcompiler.RoundingMode;
import com.maxeler.maxcompiler.v2.kernelcompiler.op_management.MathOps;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.KernelMath;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.CounterChain;
import
↪  com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.Stream.OffsetExpr;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEFix;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEFix.SignMode;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEFloat;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFERawBits;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.composite.DFEVector;
import
↪  com.maxeler.maxcompiler.v2.kernelcompiler.types.composite.DFEVectorType;
import com.maxeler.maxcompiler.v2.utils.MathUtils;

public class KarmaKernel extends Kernel {
        static final DFEFloat scalarType = dfeFloat(8, 24);
         static final DFEFloat calculationType = dfeFloat(8, 24);

         final DFEVar TAUE =constant.var(calculationType, 2.5);
         final DFEVar TAUN   = constant.var(calculationType, 250);
         final DFEVar ONE_O_TAUE =constant.var(calculationType, 0.4);
         final DFEVar ONE_O_TAUN   = constant.var(calculationType, 0.004);
         final DFEVar EH =constant.var(calculationType,3);
         final DFEVar EN = constant.var(calculationType, 1);
         final DFEVar ESTAR = constant.var(calculationType, 0.8);  //1.5415
         final DFEVar EPS = constant.var(calculationType,0.01); //TAUE/TAUN
         final DFEVar RE = constant.var(calculationType, 1.204);
         ↪  //VARIED BETWEEN 0.5 AND 1.4
         final DFEVar EXPRE = constant.var(calculationType,  0.299991841);
         ↪  //EXP(-RE)
         final DFEVar M  = constant.var(calculationType,  10);
         final DFEVar GAMMA    = constant.var(calculationType, 0.0011);
         final DFEVar ONE_O_ONE_MINUS_EXPRE   = constant.var(calculationType,
         ↪  1.428554778);


        public KarmaKernel(final KernelParameters parameters) {
                super(parameters);

                // Input
                //create autoloop offset to create backwards edge for
                ↪  calculation
                OffsetExpr loopLength =
                ↪  stream.makeOffsetAutoLoop("loopLength"); //
```

```
    DFEVar loopLengthVal = loopLength.getDFEVar(this,
    ↪  dfeUInt(11));


    CounterChain chain = control.count.makeCounterChain();
    final DFEVar nx =io.scalarInput("num_iteration",
    ↪  dfeUInt(32));

    DFEVar step = chain.addCounter(nx, 1);//counter for number of
    ↪  steps;

                                                                    ↪  //
    DFEVar loopCounter = chain.addCounter(loopLengthVal, 1);
    ↪  //counter for validation of values;
    DFEVar dtIn = io.scalarInput("dt", dfeFloat(11,53));
    DFEVar dt = dtIn.cast(calculationType);

    DFEVar carriedU = calculationType.newInstance(this);
    DFEVar carriedV = calculationType.newInstance(this);

    DFEVar u = step===0? 1.5 : carriedU;
    DFEVar v = step===0? 0 : carriedV;

      optimization.pushPipeliningFactor(1.0, PipelinedOps.ALL);
        DFEVar  dfunc = pow(v, 10);
        DFEVar  rfunc = 1.428554778 - v;

        DFEVar  hfunc = (1-tanh(u-EH)) * u* u * 0.5;
                         DFEVar   ffunc = -u + (ESTAR-dfunc) *
                         ↪  hfunc;
DFEVar h = (u > EN).cast(calculationType);
          //(a -n)*h  - (1-h)*n
          //a*h -n*h  - (n - n*h)
        DFEVar  gfunc = 1.428554778*h - (v);
        DFEVar  de    = ffunc * ONE_O_TAUE;
         DFEVar dn    = gfunc * ONE_O_TAUN;

     DFEVar uNext = u + dt*de;
    DFEVar vNext  = v + dt*dn;
    optimization.popPipeliningFactor(PipelinedOps.ALL);

        DFEVar uOffset = stream.offset(uNext, -loopLength);
           DFEVar vOffset = stream.offset(vNext, -loopLength);

           carriedU<== uOffset;
           carriedV<==vOffset;


           DFEVar uout = uNext.cast(scalarType);
           DFEVar vout = vNext.cast(scalarType);
           //optimization.popEnableSaturatingArithmetic();
        io.output("u_out", uout, scalarType,loopCounter ===
        ↪  (loopLengthVal-1));
```

```
                io.output("v_out", vout, scalarType,loopCounter ===
                ↪  (loopLengthVal-1));

        }


        DFEVar pow (DFEVar a, int b) {
                DFEVar temp;

                if (b == 0)
                return constant.var(calculationType, 1);

            temp = pow(a, (b / 2));
            if ((b % 2) == 0)
                return temp * temp;
            else
                return a * temp * temp;
        }


        protected DFEVar tanh(DFEVar x) {
                //tangens hyperbolicus: 1-(2/(e^(2*x)+1))
                        optimization.pushPipeliningFactor(0,
                        ↪  PipelinedOps.ALL);
                        optimization.pushEnableSaturatingArithmetic(true);
                        x = (2*x);
                        DFEVar v = KernelMath.exp(x);

                        DFEVar approximation = 1- (2/(v+1));
                        optimization.popPipeliningFactor(PipelinedOps.ALL);
                        optimization.popEnableSaturatingArithmetic();

                        return approximation.cast(calculationType);

        }

}
```

Beeler-Reuter Model

Manager Class Beeler-Reuter Model

```
package beelerreuter.model.cell;

import com.maxeler.maxcompiler.v2.build.EngineParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.managers.custom.CustomManager;
import com.maxeler.maxcompiler.v2.managers.custom.CustomManager.Config;
import com.maxeler.maxcompiler.v2.managers.custom.DFELink;
import com.maxeler.maxcompiler.v2.managers.custom.blocks.KernelBlock;
import com.maxeler.maxcompiler.v2.managers.engine_interfaces.CPUTypes;
import com.maxeler.maxcompiler.v2.managers.engine_interfaces.EngineInterface;
import com.maxeler.maxcompiler.v2.managers.engine_interfaces.InterfaceParam;
import com.maxeler.maxcompiler.v2.managers.standard.Manager;
import com.maxeler.maxcompiler.v2.managers.standard.Manager.IOType;
import com.maxeler.platform.max5.manager.MAX5CManager;

public class BeelerReuterManager extends MAX5CManager{
        private static final String s_kernelName = "BeelerReuterKernel";
        public BeelerReuterManager(EngineParameters arg0) {
                super(arg0);
                KernelBlock k = addKernel(
                                new
                                ↪  BeelerReuterKernel(makeKernelParameters(s_kernelName)));

                DFELink y1 = addStreamToCPU("u_out");
                y1 <== k.getOutput("u_out");
                DFELink y2 = addStreamToCPU("x1_out");
                y2 <== k.getOutput("x1_out");

                DFELink y3 = addStreamToCPU("m_out");
                y3 <== k.getOutput("m_out");

                DFELink y4 = addStreamToCPU("h_out");
                y4 <== k.getOutput("h_out");

                DFELink y5 = addStreamToCPU("j_out");
                y5 <== k.getOutput("j_out");

                DFELink y6 = addStreamToCPU("d_out");
                y6 <==k.getOutput("d_out");

                DFELink y7 = addStreamToCPU("f_out");
                y7 <== k.getOutput("f_out");

                DFELink y8 = addStreamToCPU("ca_out");
                y8 <== k.getOutput("ca_out");

                //config.setDefaultStreamClockFrequency(50);
        }
```

```
public static void main(String[] args) {
        EngineParameters params = new EngineParameters(args);
        BeelerReuterManager manager = new
        ↪  BeelerReuterManager(params);

        // Instantiate the kernel


        manager.createSLiCinterface(interfaceDefault());
        manager.build();
}

private static EngineInterface interfaceDefault() {
        EngineInterface ei = new EngineInterface();

        InterfaceParam length = ei.addParam("length", CPUTypes.INT);
        InterfaceParam dt = ei.addParam("dt", CPUTypes.DOUBLE);
        InterfaceParam num_iteration = ei.addParam("num_iteration",
        ↪  CPUTypes.INT);

        InterfaceParam u = ei.addParam("u_in", CPUTypes.FLOAT);
        InterfaceParam ca = ei.addParam("ca_in", CPUTypes.FLOAT);
        InterfaceParam h = ei.addParam("h_in", CPUTypes.FLOAT);
        InterfaceParam x1 = ei.addParam("x1_in", CPUTypes.FLOAT);
        InterfaceParam j = ei.addParam("j_in", CPUTypes.FLOAT);
        InterfaceParam d = ei.addParam("d_in", CPUTypes.FLOAT);
        InterfaceParam f = ei.addParam("f_in", CPUTypes.FLOAT);
        InterfaceParam m = ei.addParam("m_in", CPUTypes.FLOAT);

        InterfaceParam lengthInBytes = length *
        ↪  CPUTypes.FLOAT.sizeInBytes();
        InterfaceParam loopLength =
        ↪  ei.getAutoLoopOffset(s_kernelName, "loopLength");
        ei.ignoreAutoLoopOffset(s_kernelName, "loopLength");
        ei.setTicks(s_kernelName, length * loopLength);

        ei.setScalar(s_kernelName, "dt", dt);

        ei.setScalar(s_kernelName, "u_in", u);
        ei.setScalar(s_kernelName, "ca_in", ca);
        ei.setScalar(s_kernelName, "h_in", h);
        ei.setScalar(s_kernelName, "x1_in", x1);
        ei.setScalar(s_kernelName, "j_in", j);
        ei.setScalar(s_kernelName, "d_in", d);
        ei.setScalar(s_kernelName, "f_in", f);
        ei.setScalar(s_kernelName, "m_in", m);

        ei.setScalar(s_kernelName, "num_iteration", num_iteration);

        ei.setStream("u_out", CPUTypes.FLOAT, lengthInBytes);
```

```
                ei.setStream("x1_out", CPUTypes.FLOAT, lengthInBytes);
                ei.setStream("m_out", CPUTypes.FLOAT, lengthInBytes);
                ei.setStream("h_out", CPUTypes.FLOAT, lengthInBytes);
                ei.setStream("j_out", CPUTypes.FLOAT, lengthInBytes);
                ei.setStream("d_out", CPUTypes.FLOAT, lengthInBytes);
                ei.setStream("f_out", CPUTypes.FLOAT, lengthInBytes);
                ei.setStream("ca_out", CPUTypes.FLOAT, lengthInBytes);


                return ei;
        }
}
```

Kernel Class Beeler-Reuter Model

```
package beelerreuter.model.cell;

import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.Optimization;
import com.maxeler.maxcompiler.v2.kernelcompiler.Optimization.PipelinedOps;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.KernelMath;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.CounterChain;
import
↪  com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.Stream.OffsetExpr;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.memory.Memory;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEType;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.composite.DFEVector;
import
↪  com.maxeler.maxcompiler.v2.kernelcompiler.types.composite.DFEVectorType;
import com.maxeler.maxcompiler.v2.utils.MathUtils;
import com.maxeler.statemachine.statements.SimPrintf;


public class BeelerReuterKernel extends Kernel {
         static final DFEType scalarType = dfeFloat(8, 24);
         static final DFEType calculationType = dfeFloat(8, 24);
         static final DFEVectorType<DFEVar> vectorType =
                         new DFEVectorType<DFEVar>(calculationType, 8);


        //Constants
        final DFEVar EPI_TVP = constant.var(calculationType, 1.4506);

final DFEVar GS = constant.var(calculationType,  0.09);
final DFEVar        ENA = constant.var(calculationType, 50);
final DFEVar        GNAC = constant.var(calculationType, 0.003);
final DFEVar        GNA = constant.var(calculationType , 4);


        final DFEVar Zero = constant.var(calculationType, 0.0);
        final DFEVar One = constant.var(calculationType, 1.0);




        BeelerReuterKernel(KernelParameters parameters) {
                super(parameters);

                //time given by cpu
```

```
        //create autoloop offset to create backwards edge for
        ↪  calculation
        OffsetExpr loopLength =
        ↪  stream.makeOffsetAutoLoop("loopLength"); //
        DFEVar loopLengthVal = loopLength.getDFEVar(this,
        ↪  dfeUInt(11));




        CounterChain chain = control.count.makeCounterChain();
        final DFEVar nx =io.scalarInput("num_iteration",
        ↪  dfeUInt(32));

        DFEVar step = chain.addCounter(nx, 1);//counter for number of
        ↪  steps;
                                                                //
        DFEVar loopCounter = chain.addCounter(loopLengthVal, 1);
        ↪  //counter for validation of values;
        DFEVar dt = io.scalarInput("dt", dfeFloat(11,53));
        dt = dt.cast(calculationType);

//        DFEVar stim_in = io.input("stim_in",scalarType,loopCounter
↪  === (loopLengthVal-1));
        DFEVector<DFEVar> valueVector = vectorType.newInstance(this);
        DFEVar currentU = step===0 ? io.scalarInput("u_in",
        ↪  calculationType): valueVector[0];
        DFEVar currentCa = step===0 ? io.scalarInput("ca_in",
        ↪  calculationType) : valueVector[1];
        DFEVar currentX1 = step===0 ? io.scalarInput("x1_in",
        ↪  calculationType): valueVector[2];
        DFEVar currentM = step===0? io.scalarInput("m_in",
        ↪  calculationType) :valueVector[3];
        DFEVar currentF = step===0? io.scalarInput("f_in",
        ↪  calculationType): valueVector[4];
        DFEVar currentH = step===0? io.scalarInput("h_in",
        ↪  calculationType) : valueVector [5];
        DFEVar currentD = step===0 ? io.scalarInput("d_in",
        ↪  calculationType): valueVector[6];
        DFEVar currentJ = step===0? io.scalarInput("j_in",
        ↪  calculationType) :valueVector[7];

        optimization.pushPipeliningFactor(0, PipelinedOps.ALL);

         DFEVar ax1 = dt *((5.0*Math.pow(10,-4)) *
         ↪  KernelMath.exp(0.083 * ((currentU) + 50.0)) /
         ↪  (KernelMath.exp(0.057 * ((currentU) + 50.0)) + 1.0));
```

```
DFEVar        bx1 =  dt * (0.0013 * KernelMath.exp(-0.06 *
↪  ((currentU) + 20.0)) / (KernelMath.exp(-0.04 *
↪  ((currentU) + 20.0)) + 1.0));

DFEVar        am = dt* (-((currentU) + 47.0) /
↪  (KernelMath.exp(-0.1 * ((currentU) + 47.0)) - 1.0));
DFEVar        bm = dt *(40.0 * KernelMath.exp(-0.056 *
↪  ((currentU) + 72.0)));

DFEVar        ah = dt * (0.126 * KernelMath.exp(-0.25 *
↪  ((currentU) + 77.0)));
DFEVar        bh = dt *(1.7 / (KernelMath.exp(-0.082 *
↪  ((currentU) + 22.5)) + 1.0));

DFEVar        aj = dt * (0.055 * KernelMath.exp(-0.25 *
↪  ((currentU) + 78.0)) / (KernelMath.exp(-0.2 *
↪  ((currentU) + 78.0)) + 1.0));
DFEVar        bj = dt * (0.3 / (KernelMath.exp(-0.1 *
↪  ((currentU) + 32.0)) + 1.0));

DFEVar        ad = dt * (0.095 * KernelMath.exp(-0.01 *
↪  ((currentU) - 5.0)) / (KernelMath.exp(-0.072 *
↪  ((currentU) - 5.0)) + 1.0));
DFEVar        bd =dt * (0.07 * KernelMath.exp(-0.017 *
↪  ((currentU) + 44.0)) / (KernelMath.exp(0.05 *
↪  ((currentU) + 44.0)) + 1.0));

DFEVar        af = dt * (0.012 * KernelMath.exp(-0.008 *
↪  ((currentU) + 28.0)) / (KernelMath.exp(0.15 *
↪  ((currentU) + 28.0)) + 1.0));
DFEVar        bf = dt *(0.0065 * KernelMath.exp(-0.02 *
↪  ((currentU) + 30.0)) / (KernelMath.exp(-0.2 *
↪  ((currentU) + 30.0)) + 1.0));

        currentX1 = (currentX1) + (ax1 * (1.0 -
        ↪  (currentX1)) - bx1 * (currentX1));
        currentM = (currentM) + (am * (1.0 - (currentM)) -
        ↪  bm * (currentM));
        currentH =  (currentH)+ (ah * (1.0 - (currentH)) -
        ↪  bh * (currentH));
        currentJ = (currentJ) + (aj * (1.0 - (currentJ)) -
        ↪  bj * (currentJ));
        currentD = (currentD) + (ad * (1.0 - (currentD)) -
        ↪  bd * (currentD));
        currentF = (currentF) + (af * (1.0 - (currentF)) -
        ↪  bf * (currentF));


       DFEVar es = (-82.3 - 13.0287 *
       ↪  KernelMath.log((currentCa), calculationType));
       DFEVar is
       ↪  =0.09*(currentD)*(currentF)*((currentU)-es);
```

```
        currentCa = currentCa
        ↪  +dt*(-0.0000001*is+0.07*(0.0000001-(currentCa)));

  DFEVar  xik1 = (KernelMath.exp(0.08 * ((currentU) +
  ↪  53.0)) + KernelMath.exp(0.04 * ((currentU) +
  ↪  53.0)));
DFEVar       xik2 = (1.0 - KernelMath.exp(-0.04 *
↪  ((currentU) + 23.0)));
xik1= (xik1 === Zero)? 0.001: xik1;
xik2 = (xik2 ===Zero)?  0.001: xik2;
DFEVar       ik1 = (0.35 * (4.0 * (KernelMath.exp(0.04 *
↪  ((currentU) + 85.0)) - 1.0) / xik1 + 0.2 * ((currentU)
↪  + 23.0) / xik2));

     DFEVar  gix1 = (0.8 * (KernelMath.exp(0.04 *
     ↪  ((currentU) + 77.0)) - 1.0) /KernelMath.exp(0.04
     ↪  * ((currentU) + 35.0)));
       DFEVar ix1 = (gix1 * (currentX1));
       DFEVar ina = ((GNA * (currentM) * (currentM) *
       ↪  (currentM) * (currentH) * (currentJ) + GNAC) *
       ↪  ((currentU) - ENA));
       currentU = currentU -( dt * (ik1 + ix1 + ina +
       ↪  is));
       optimization.popPipeliningFactor(PipelinedOps.ALL);
       DFEVar uOffset = stream.offset(currentU,
       ↪  -loopLength);
       DFEVar caOffset = stream.offset(currentCa,
       ↪  -loopLength);
       DFEVar jOffset = stream.offset(currentJ,
       ↪  -loopLength);
       DFEVar hOffset = stream.offset(currentH,
       ↪  -loopLength);
       DFEVar x1Offset = stream.offset(currentX1,
       ↪  -loopLength);
       DFEVar fOffset = stream.offset(currentF,
       ↪  -loopLength);
       DFEVar mOffset = stream.offset(currentM,
       ↪  -loopLength);
       DFEVar dOffset = stream.offset(currentD,
       ↪  -loopLength);

       valueVector[0] <== uOffset;
       valueVector[1] <== caOffset;
       valueVector[2] <==x1Offset;
       valueVector[3] <== mOffset;
       valueVector[4] <== fOffset;
       valueVector[5] <== hOffset;
       valueVector[6] <==dOffset;
       valueVector[7] <==jOffset;
```

```
                                io.output("u_out", currentU.cast(scalarType),
                                ↪  scalarType,loopCounter ===
                                ↪  (loopLengthVal-1));
                                io.output("ca_out", currentCa.cast(scalarType),
                                ↪  scalarType,loopCounter ===
                                ↪  (loopLengthVal-1));
                                io.output("x1_out", currentX1.cast(scalarType),
                                ↪  scalarType,loopCounter ===
                                ↪  (loopLengthVal-1));
                                io.output("m_out", currentM.cast(scalarType),
                                ↪  scalarType,loopCounter ===
                                ↪  (loopLengthVal-1));
                                io.output("f_out", currentF.cast(scalarType),
                                ↪  scalarType,loopCounter ===
                                ↪  (loopLengthVal-1));
                                io.output("h_out", currentH.cast(scalarType),
                                ↪  scalarType,loopCounter ===
                                ↪  (loopLengthVal-1));
                                io.output("d_out", currentD.cast(scalarType),
                                ↪  scalarType,loopCounter ===
                                ↪  (loopLengthVal-1));
                                io.output("j_out", currentJ.cast(scalarType),
                                ↪  scalarType,loopCounter ===
                                ↪  (loopLengthVal-1));
        }



        /***
         *
         * @param x value to check
         * @return heaviside(x) returns the value 0 for x < 0, 1 for x > 0,
↪  and 1/2 for x = 0.
         */

        protected DFEVar heavisidefun(DFEVar x) {

//                //check if the value is below or above zero
                DFEVar c = x<0.0? constant.var(dfeBool(), 1) :
                ↪  constant.var(dfeBool(), 0);
//
//                //check if the value is zero
                DFEVar z = x===0.0? constant.var(dfeBool(), 1) :
                ↪  constant.var(dfeBool(), 0);
//
//                //assign values regarding to checks - first if value is
↪  below or above zero
                DFEVar ret = c?constant.var(scalarType, 0.0) :
                ↪  constant.var(scalarType, 1.0);
//
//                //if x is zero, z is true, as c would not consider this
↪  case, we can disregard the value of c if z is true
```

```
                ret = z? constant.var(scalarType, 0.5) : ret;

                return ret;


        }

        protected DFEVar tanh(DFEVar x) {
                //tangens hyperbolicus: 1-(2/(e^(2*x)+1))

                        DFEVar v = KernelMath.exp(2*x);
                        DFEVar approximation = 1- (2/(v+1));
                        return approximation;

        }


}
```

Appendix B: Maxeler Minimal Model Homogeneous 1D Cable

Manager Class

```
package minimal.thritytwo.cable;

import com.maxeler.maxcompiler.v2.build.EngineParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.managers.custom.DFELink;
import com.maxeler.maxcompiler.v2.managers.custom.blocks.KernelBlock;
import com.maxeler.maxcompiler.v2.managers.engine_interfaces.CPUTypes;
import com.maxeler.maxcompiler.v2.managers.engine_interfaces.EngineInterface;
import com.maxeler.maxcompiler.v2.managers.engine_interfaces.InterfaceParam;
import com.maxeler.maxcompiler.v2.managers.standard.Manager;
import com.maxeler.maxcompiler.v2.managers.standard.Manager.IOType;
import com.maxeler.platform.max5.manager.MAX5CManager;

public class CellCableDFE32Manager extends MAX5CManager{
        private static final String s_kernelName = "CellCableDFE32Kernel";
        private static final int X = 10000; // maximum number of values
        private static final int duration = 5;
        public CellCableDFE32Manager(EngineParameters arg0) {
                super(arg0);
                KernelBlock k = addKernel(
                                new
                                ↪  CellCableDFE32Kernel(makeKernelParameters(s_kernelName),
                                ↪  X, duration));
                DFELink y1 = addStreamToCPU("u_out");
                y1 <== k.getOutput("u_out");
                DFELink y2 = addStreamToCPU("v_out");
                y2 <== k.getOutput("v_out");

                DFELink y3 = addStreamToCPU("w_out");
                y3 <== k.getOutput("w_out");

                DFELink y4 = addStreamToCPU("s_out");
                y4 <== k.getOutput("s_out");

        }
```

```
    public static void main(String[] args) {
        EngineParameters params = new EngineParameters(args);
        CellCableDFE32Manager manager = new
        ↪  CellCableDFE32Manager(params);

        manager.createSLiCinterface(interfaceDefault());
        manager.addMaxFileConstant("X", X);
        manager.addMaxFileConstant("duration", duration);
        manager.setDefaultStreamClockFrequency(120);
        manager.build();
    }

    private static EngineInterface interfaceDefault() {
        EngineInterface ei = new EngineInterface();

        InterfaceParam length = ei.addParam("length", CPUTypes.INT);
        InterfaceParam dt = ei.addParam("dt", CPUTypes.FLOAT);
        InterfaceParam ddt_o_dx2 = ei.addParam("ddt_o_dx2",
        ↪  CPUTypes.FLOAT);
        InterfaceParam simTime = ei.addParam("simTime",
        ↪  CPUTypes.INT);
        InterfaceParam nx = ei.addParam("nx", CPUTypes.INT);
        InterfaceParam lengthInBytes = length *
        ↪  CPUTypes.FLOAT.sizeInBytes();
        InterfaceParam loopLength =
        ↪  ei.getAutoLoopOffset(s_kernelName, "loopLength");
        ei.ignoreAutoLoopOffset(s_kernelName, "loopLength");
        ei.setTicks(s_kernelName, (length) * loopLength);

        ei.setScalar(s_kernelName, "dt", dt);
        ei.setScalar(s_kernelName, "ddt_o_dx2", ddt_o_dx2);
        ei.setScalar(s_kernelName, "simTime", simTime);
        ei.setScalar(s_kernelName, "nx", nx);

        ei.setStream("u_out", CPUTypes.FLOAT, lengthInBytes);
        ei.setStream("v_out", CPUTypes.FLOAT, lengthInBytes);
        ei.setStream("w_out", CPUTypes.FLOAT, lengthInBytes);
        ei.setStream("s_out", CPUTypes.FLOAT, lengthInBytes);

        return ei;
    }
}
```

Kernel Class

```
package minimal.thritytwo.cable;

import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.Optimization.PipelinedOps;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.KernelMath;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.CounterChain;
import
↪  com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.Stream.OffsetExpr;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.memory.Memory;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEType;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.composite.DFEVector;
import
↪  com.maxeler.maxcompiler.v2.kernelcompiler.types.composite.DFEVectorType;
import com.maxeler.maxcompiler.v2.utils.MathUtils;


public class CellCableDFE32Kernel extends Kernel {


         static final DFEType scalarType = dfeFloat(8,24);
         static final DFEVectorType<DFEVar> vectorType =
                        new DFEVectorType<DFEVar>(scalarType, 4);
        //Constants
        final DFEVar EPI_TVP = constant.var(scalarType, 1.4506);
        final DFEVar EPI_TV1M = constant.var(scalarType, 60);
        final DFEVar EPI_TV2M = constant.var(scalarType,1150);
        final DFEVar EPI_TWP = constant.var(scalarType,200);
        final DFEVar EPI_TW1M = constant.var(scalarType,60);
        final DFEVar EPI_TW2M = constant.var(scalarType,15);
        final DFEVar EPI_TS1 = constant.var(scalarType,2.7342);
        final DFEVar EPI_TS2 = constant.var(scalarType,16);
        final DFEVar EPI_TFI = constant.var(scalarType,0.11);
        final DFEVar EPI_TO1 = constant.var(scalarType,400);
        final DFEVar EPI_TO2 = constant.var(scalarType,6);
        final DFEVar EPI_TSO1 = constant.var(scalarType,30.0181);
        final DFEVar EPI_TSO2 = constant.var(scalarType,0.9957);
        final DFEVar EPI_TSI = constant.var(scalarType,1.8875);
        final DFEVar EPI_TWINF = constant.var(scalarType,0.07);
        final DFEVar EPI_THV = constant.var(scalarType,0.3);
        final DFEVar EPI_THVM = constant.var(scalarType,0.006);
        final DFEVar EPI_THVINF = constant.var(scalarType,0.006);
        final DFEVar EPI_THW = constant.var(scalarType,0.13);
        final DFEVar EPI_THSO = constant.var(scalarType,0.006);
        final DFEVar EPI_THSI = constant.var(scalarType,0.13);
        final DFEVar EPI_THO = constant.var(scalarType,0.006);
        final DFEVar EPI_KWM = constant.var(scalarType,65);
        final DFEVar EPI_KS = constant.var(scalarType,2.0994);
        final DFEVar EPI_KSO = constant.var(scalarType,2.0458);
        final DFEVar EPI_UWM = constant.var(scalarType,0.03);
```

```
final DFEVar EPI_US = constant.var(scalarType,0.9087);
final DFEVar EPI_U0 = constant.var(scalarType,0);
final DFEVar EPI_UU = constant.var(scalarType,1.55);
final DFEVar EPI_USO = constant.var(scalarType,0.65);
final DFEVar TW2M_TW1M_DIVBY2 = constant.var(scalarType,-22.5);
final DFEVar TSO2_TSO1_DIVBY2 = constant.var(scalarType,-14.5112);
final DFEVar  WINFSTAR = constant.var( scalarType,0.94);
final DFEVar  EPI_THWINF = constant.var(  scalarType,0.006);

final DFEVar Zero = constant.var(scalarType, 0.0);
final DFEVar One = constant.var(scalarType, 1.0);



CellCableDFE32Kernel(KernelParameters parameters, int X, int
↪  duration) {
        super(parameters);

        //time given by cpu
        final DFEVar dt = io.scalarInput("dt", scalarType);
        final DFEVar ddt_o_dx2 = io.scalarInput("ddt_o_dx2",
        ↪  scalarType);
        final DFEVar nx = io.scalarInput("nx", dfeUInt(32));
        final DFEVar simTime = io.scalarInput("simTime",
        ↪  dfeUInt(32));

        //create autoloop offset to create backwards edge for
        ↪  calculation
        OffsetExpr loopLength =
        ↪  stream.makeOffsetAutoLoop("loopLength"); //
        DFEVar loopLengthVal = loopLength.getDFEVar(this,
        ↪  dfeUInt(8));




        CounterChain chain = control.count.makeCounterChain();
        DFEVar step = chain.addCounter(simTime+1, 1); //counter for
        ↪  simulation time..
        DFEVar cellNumAddress = chain.addCounter(nx, 1);//counter for
        ↪  number of cells ; resets automatically when reaching X;
        ↪  -> inner loop
        DFEVar loopCounter = chain.addCounter(loopLengthVal, 1);
        ↪  //counter for validation of values;




        Memory<DFEVar> uMem = mem.alloc(scalarType, X);
        Memory<DFEVar> wMem = mem.alloc(scalarType, X);
```

```
    Memory<DFEVar> vMem = mem.alloc(scalarType, X);
    Memory<DFEVar> sMem = mem.alloc(scalarType, X);
    Memory<DFEVar> t1Mem = mem.alloc(scalarType, X);
    Memory<DFEVar> t2Mem = mem.alloc(scalarType, X);
    DFEVar uFromMem =
    ↪  uMem.read(cellNumAddress.cast(dfeUInt(MathUtils.bitsToAddress(X))));
    DFEVar vFromMem =
    ↪  vMem.read(cellNumAddress.cast(dfeUInt(MathUtils.bitsToAddress(X))));
    DFEVar wFromMem =
    ↪  wMem.read(cellNumAddress.cast(dfeUInt(MathUtils.bitsToAddress(X))));
    DFEVar sFromMem =
    ↪  sMem.read(cellNumAddress.cast(dfeUInt(MathUtils.bitsToAddress(X))));
    DFEVar t1FromMem =
    ↪  t1Mem.read(cellNumAddress.cast(dfeUInt(MathUtils.bitsToAddress(X))));
    DFEVar t2FromMem =
    ↪  t2Mem.read(cellNumAddress.cast(dfeUInt(MathUtils.bitsToAddress(X))));
    DFEVar nextNeighAdd = cellNumAddress+1;
    DFEVar uNext =
    ↪  uMem.read(nextNeighAdd.cast(dfeUInt(MathUtils.bitsToAddress(X))));

    //create sourcless streams for variables

    DFEVar carriedU = scalarType.newInstance(this);


    //define initCondition - easier to read;
    DFEVar initCondition = (step===0);


    //init variables needed;

    DFEVar ui = initCondition? 0.0 : uFromMem; //initCondition?
    ↪  0.0 : carriedU;
DFEVar    vi = initCondition? 1.0 :  vFromMem;
DFEVar   wi = initCondition ? 1.0 :  wFromMem;
DFEVar     si = initCondition? 0.0 :sFromMem;
    DFEVar t1 = step===0? 0.0 : t1FromMem;
    DFEVar t2 = step===0? 0.0 : t2FromMem;
    DFEVar t1Curr = t1+dt;
    DFEVar t2Curr = t2+dt;

     DFEVar active = dfeBool().newInstance(this);


    optimization.pushPipeliningFactor(0.5,PipelinedOps.ALL);
     DFEVar stimFlag = heavisidefun(t1Curr - 0)*(1 -
     ↪  heavisidefun(t2Curr - 1)) + heavisidefun(t1Curr -
     ↪  300)*(1 - heavisidefun(t2Curr - 301)) +
     ↪  heavisidefun(t1Curr - 700)*(1 - heavisidefun(t2Curr -
     ↪  701));
    optimization.popPipeliningFactor(PipelinedOps.ALL);

     active = (stimFlag===1)&(cellNumAddress<duration);//#
```

```
DFEVar stim = active? constant.var(scalarType,0.66) :
↪  constant.var(scalarType, 0.0);
       DFEVar prevNeigh = stream.offset(carriedU, -1);
       DFEVar nextNeigh = initCondition? 0.0: uNext;

    optimization.pushPipeliningFactor(0.5,PipelinedOps.ALL);

     DFEVar lap = cellNumAddress===0?           ddt_o_dx2 *
     ↪  (-2.0*ui + 2.0*nextNeigh) :
                        cellNumAddress===(nx-1)?  ddt_o_dx2 *
                        ↪  (-2.0*ui + 2.0*prevNeigh) :









  optimization.popPipeliningFactor(PipelinedOps.ALL);

    DFEVar tvm, vinf, winf;
    DFEVar jfi, jso, jsi;
    DFEVar twm, tso, ts, to, ds, dw, dv;

    optimization.pushPipeliningFactor(0.5, PipelinedOps.ALL);
    tvm =  (ui >   EPI_THVM) ?   EPI_TV2M :   EPI_TV1M;
twm =    EPI_TW1M + (  TW2M_TW1M_DIVBY2)*(1+tanh(  EPI_KWM*(ui-
↪  EPI_UWM)));
tso =    EPI_TSO1 + (  TSO2_TSO1_DIVBY2)*(1+tanh(  EPI_KSO*(ui-
↪  EPI_USO)));
ts  = (ui >   EPI_THW)  ?   EPI_TS2  :   EPI_TS1;
to  = (ui >   EPI_THO)  ?   EPI_TO2  :   EPI_TO1;

tvm = optimization.pipeline(tvm);
twm = optimization.pipeline(twm);
tso = optimization.pipeline(tso);
ts = optimization.pipeline(ts);
to = optimization.pipeline(to);


vinf = (ui >    EPI_THVINF) ? Zero : One;
    winf = (ui >    EPI_THWINF) ?   WINFSTAR: (1.0-ui/
    ↪  EPI_TWINF);
winf =  (winf > One) ? One : winf;

vinf = optimization.pipeline(vinf);
winf = optimization.pipeline(winf);


dv = (ui >   EPI_THV) ? -vi/  EPI_TVP : (vinf-vi)/tvm;
```

```
            dw = (ui >   EPI_THW) ? -wi/  EPI_TWP : (winf-wi)/twm;
            ds = (((1.+tanh(  EPI_KS*(ui-  EPI_US)))/2.) - si)/ts;

            dv = optimization.pipeline(dv);
            dw = optimization.pipeline(dw);
            ds = optimization.pipeline(ds);
            optimization.popPipeliningFactor(PipelinedOps.ALL);

            //winf = (*ui > 0.06) ? 0.94: 1.0-*ui/0.07;
            //twm  =  60 + (-22.5)*(1.+tanh(65*(*ui-0.03)));
            //dw   = (*ui > 0.13) ? -*wi/200 : (winf-*wi)/twm;

                //tvm =  (*ui >   THVM) ?   TV2M :   TV1M;
                //one_o_twm = segm_table[0][th] * (*ui) +
                ↪  segm_table[1][th];
                //vinf = (*ui >    THVINF) ? 0.0: 1.0;
                //winf = (*ui >    THWINF) ?   WINFSTAR * one_o_twm:
                ↪  (segm2_table[0][th2] * (*ui) + segm2_table[1][th2]);
                //if (winf >one_o_twm) winf = one_o_twm;
                //dv = (*ui >   THV) ? -*vi/  TVP : (vinf-*vi)/tvm;
                //dw = (*ui >   THW) ? -*wi/  TWP : winf - *wi * one_o_twm;
                //ds = (((1.+tanh(  KS*(*ui-  US)))/2.) - *si)/ts;

                //Update gates

             vi += dv*dt;
             wi += dw*dt;
             si += ds*dt;

                //Compute currents
             optimization.pushPipeliningFactor(0.5, PipelinedOps.ALL);
            jfi = (ui >   EPI_THV)  ? -vi * (ui -   EPI_THV) * (  EPI_UU -
            ↪  ui)/  EPI_TFI : Zero;
            /*if (*ui >   THV){
                        if (((*vi - dv*dt) > 0.0) && *vi < 0.0001){
                                printf("vi=%4.20f, ui=%f, jfi=%f\n", *vi,
↪  *ui, jfi);
                        }
                 }*/
            jso = (ui >   EPI_THSO) ? 1/tso : (ui-  EPI_U0)/to;
            jsi = (ui >   EPI_THSI) ? -wi * si/  EPI_TSI : 0.0;

            ui = ui  - (jfi+jso+jsi-stim)*dt + lap;
            optimization.popPipeliningFactor(PipelinedOps.ALL);

              //generate offset for backward edge
              DFEVar uOffset = stream.offset(ui, -loopLength);
              DFEVar vOffset = stream.offset(vi, -loopLength);
              DFEVar wOffset = stream.offset(wi, -loopLength);
              DFEVar sOffset = stream.offset(si, -loopLength);
              DFEVar t1Offset = stream.offset(t1Curr, -loopLength);
              DFEVar t2Offset = stream.offset(t2Curr, -loopLength);
```

```
                // At the foot of the loop, we add the backward edge

                carriedU <== uOffset;


                DFEVar memOffset = stream.offset(cellNumAddress,
                ↪ -loopLength);

                 ↪ uMem.write(memOffset.cast(dfeUInt(MathUtils.bitsToAddress(X))),
                 ↪ uOffset, (loopCounter === (loopLengthVal-1)));

                 ↪ vMem.write(memOffset.cast(dfeUInt(MathUtils.bitsToAddress(X))),
                 ↪ vOffset, (loopCounter === (loopLengthVal-1)));

                 ↪ wMem.write(memOffset.cast(dfeUInt(MathUtils.bitsToAddress(X))),
                 ↪ wOffset, (loopCounter === (loopLengthVal-1)));

                 ↪ sMem.write(memOffset.cast(dfeUInt(MathUtils.bitsToAddress(X))),
                 ↪ sOffset, (loopCounter === (loopLengthVal-1)));

                 ↪ t1Mem.write(memOffset.cast(dfeUInt(MathUtils.bitsToAddress(X))),
                 ↪ t1Offset, (loopCounter === (loopLengthVal-1)));

                 ↪ t2Mem.write(memOffset.cast(dfeUInt(MathUtils.bitsToAddress(X))),
                 ↪ t2Offset, (loopCounter === (loopLengthVal-1)));


                //write to outputstreams;
                io.output("u_out", ui, scalarType,loopCounter ===
                ↪ (loopLengthVal-1));
                io.output("v_out", vi, scalarType,loopCounter ===
                ↪ (loopLengthVal-1));
                io.output("w_out", wi, scalarType,loopCounter ===
                ↪ (loopLengthVal-1)) ;
                io.output("s_out", si, scalarType,loopCounter ===
                ↪ (loopLengthVal-1));
                io.output("stim_out", stim, scalarType,loopCounter ===
                ↪ (loopLengthVal-1));


        }

        /***
         *
         * @param x value to check
         * @return heaviside(x) returns the value 0 for x < 0, 1 for x > 0,
↪  and 1/2 for x = 0.
         */
```

```
        protected DFEVar heavisidefun(DFEVar x) {


//                //check if the value is below or above zero
                DFEVar c = x<0.0? constant.var(dfeBool(), 1) :
                ↪  constant.var(dfeBool(), 0);
//
//                //check if the value is zero
                DFEVar z = x===0.0? constant.var(dfeBool(), 1) :
                ↪  constant.var(dfeBool(), 0);
//
//                //assign values regarding to checks - first if value is
↪  below or above zero
                DFEVar ret = c?constant.var(scalarType, 0.0) :
                ↪  constant.var(scalarType, 1.0);
//
//                //if x is zero, z is true, as c would not consider this
↪  case, we can disregard the value of c if z is true
                ret = z? constant.var(scalarType, 0.5) : ret;


                return ret;




        }


        protected DFEVar tanh(DFEVar x) {
                //tangens hyperbolicus: 1-(2/(e^(2*x)+1))

                        DFEVar v = KernelMath.exp(2*x);
                        DFEVar approximation = 1- (2/(v+1));
                        return approximation;

        }




}
```

Acronyms

APD action potential duration. 8, 9

CPU Central Processing Unit. 2, 10–16, 21, 22, 25–29, 34–37, 43–48, 50, 53, 54, 57

CUDA Compute Unified Device Architecture. 11–13, 36

CV conduction velocity. 9

DFE Dataflow Engine. 20–22, 25, 28, 30, 31, 33, 45, 46, 53, 57

DFEs Dataflow Engines. 20

EEPROM Electrically Erasable Programmable ROM. 15

FMem Fast Memory in Maxeler DFE. 20, 31, 32, 34, 44

FPGA Field Programmable Gate Array. 11, 14–16, 54, 57

GPU Graphics Processing Unit. 10–13, 15, 16, 35, 36, 44, 45, 47, 48, 54

IDE Integrated Development Environment. 21

LMem Large Memory in Maxeler DFE. 20

PCIe Peripheral Component Interconnect Express. 11

PLD Programmable Logic Device. 14

PLL Phase Locked Loop. 15

SLiC Simple Live CPU. 21–23, 25, 28, 30, 31, 45

SM Streaming Multiprocessor. 13

SRAM Static RAM. 14

VHDL Very High speed Integratd Curcuits Hardware Description Language. 15

Bibliography

[1] URL: http://www.theeplab.com/B-The-Members-Center/C-Cardiac-AnatomyPhysiology/F-Action-Potential/CF00-Action-Potential.php.

[2] URL: https://www.techpowerup.com/gpu-specs/geforce-gtx-1080.c2839.

[3] Christopher J Arthurs, Martin J Bishop, and David Kay. "Efficient simulation of cardiac electrical propagation using high order finite elements". In: *Journal of computational physics* 231.10 (2012), pp. 3946–3962.

[4] Peter J Ashenden. *The designer's guide to VHDL*. Vol. 3. Morgan Kaufmann, 2010.

[5] Ezio Bartocci et al. "Toward real-time simulation of cardiac dynamics". In: *Proceedings of the 9th International Conference on Computational Methods in Systems Biology*. ACM. 2011, pp. 103–112.

[6] GW Beeler and H Reuter. "Membrane calcium current in ventricular myocardial fibres". In: *The Journal of physiology* 207.1 (1970), pp. 191–209.

[7] Philip Bittihn. *Complex structure and dynamics of the heart*. Springer, 2014.

[8] J. M. Bower and D. Beeman. "GENESIS (simulation environment)". In: *Scholarpedia* 2.3 (2007). revision #89006, p. 1383. DOI: 10.4249/scholarpedia.1383.

[9] Alfonso Bueno-Orovio, Elizabeth M Cherry, and Flavio H Fenton. "Minimal model for human ventricular action potentials in tissue". In: *Journal of theoretical biology* 253.3 (2008), pp. 544–560.

[10] John Charles Butcher. *Numerical methods for ordinary differential equations*. John Wiley & Sons, 2016.

[11] Fulong Chen et al. "Identification of the parameters of the Beeler-Reuter ionic equation with a partially perturbed particle swarm optimization". In: *IEEE Transactions on Biomedical Engineering* 59.12 (2012), pp. 3412–3421.

[12] Lorenzo Dematté and Davide Prandi. "GPU computing for systems biology". In: *Briefings in bioinformatics* 11.3 (2010), pp. 323–333.

[13] Mohamed G Egila et al. "FPGA-based electrocardiography (ECG) signal analysis system using least-square linear phase finite impulse response (FIR) filter". In: *Journal of Electrical Systems and Information Technology* 3.3 (2016), pp. 513–526.

[14] C. Elias. *FPGAs für Maker: Eine praktische Einführung in programmierbare Logik*. Dpunkt.Verlag GmbH, 2016. ISBN: 9783864901737. URL: https://books.google.at/books?id=PXHwnQEACAAJ.

[15] *Export Compliance Metrics for Intel Microprocessors Intel Xeon Processors*. Tech. rep. Revision 2. 2200 Mission College Blvd. Santa Clara, CA 95054-1537 USA: Intel Corporation, July 2018.

[16] *Export Compliance Metrics for Intel Microprocessors, Intel Core Processors*. Tech. rep. Revision 1. 2200 Mission College Blvd. Santa Clara, CA 95054-1537 USA: Intel Corporation, Apr. 2018. URL: https://www.intel.com/content/www/us/en/support/articles/000005755/processors.html.

[17] Martin Falk et al. "Parallelized agent-based simulation on CPU and graphics hardware for spatial and stochastic models in biology". In: *Proceedings of the 9th International Conference on Computational Methods in Systems Biology*. ACM. 2011, pp. 73–82.

[18] Flavio Fenton and Alain Karma. "Vortex dynamics in three-dimensional continuous myocardium with fiber rotation: Filament instability and fibrillation". In: *Chaos: An Interdisciplinary Journal of Nonlinear Science* 8.1 (1998), pp. 20–47.

[19] Du-guan Fu. "Cardiac arrhythmias: diagnosis, symptoms, and treatments". In: *Cell biochemistry and biophysics* 73.2 (2015), pp. 291–296.

[20] D. F. M. Goodman and R. Brette. "Brian simulator". In: *Scholarpedia* 8.1 (2013). revision #129355, p. 10883. DOI: 10.4249/scholarpedia.10883.

[21] Radu Grosu et al. "From cardiac cells to genetic regulatory networks". In: *International Conference on Computer Aided Verification*. Springer. 2011, pp. 396–411.

[22] Jayantha Herath et al. "Dataflow computing models, languages, and machines for intelligence computations". In: *IEEE Transactions on Software Engineering* 14.12 (1988), pp. 1805–1828.

[23] Alan L Hodgkin and Andrew F Huxley. "A quantitative description of membrane current and its application to conduction and excitation in nerve". In: *The Journal of physiology* 117.4 (1952), pp. 500–544.

[24] Meyer B Jackson. *Molecular and cellular biophysics*. Cambridge University Press, 2006.

[25] Alain Karma. "Spiral breakup in model equations of action potential propagation in cardiac tissue". In: *Physical review letters* 71.7 (1993), p. 1103.

[26] Tanaya Katakkar. *Field Programmable Gate Arrays (FPGA)*. URL: https://www.engineersgarage.com/articles/field-programmable-gate-arrays-fpga.

[27] Dietmar PF Möller. "Guide to computing fundamentals in cyber-physical systems". In: *Computer Communications and Networks. Springer, Heidelberg* (2016).

[28] *Multiscale Dataflow Programming*. Version 2015.1.1. Maxeler Technologies. Aug. 2015.

[29] Venkata Krishna Nimmagadda et al. "Cardiac simulation on multi-GPU platform". In: *The Journal of Supercomputing* 59.3 (2012), pp. 1360–1378.

[30] Denis Noble. "A modification of the Hodgkin—Huxley equations applicable to Purkinje fibre action and pacemaker potentials". In: *The Journal of physiology* 160.2 (1962), pp. 317–352.

[31] Norliza Othman, Nur Atiqah Adon, and Farhanahani Mahmud. "FPGA in-the-loop simulations of cardiac excitation model under voltage clamp conditions". In: *AIP Conference Proceedings*. Vol. 1788. 1. AIP Publishing. 2017, p. 030105.

[32] Daisuke Sato et al. "Acceleration of cardiac tissue simulation with graphic processing units". In: *Medical & biological engineering & computing* 47.9 (2009), pp. 1011–1015.

[33] Raymond J Spiteri and Ryan C Dean. "On the Performance of an Implicit–Explicit Runge–Kutta Method in Models of Cardiac Electrical Activity". In: *IEEE Transactions on Biomedical Engineering* 55.5 (2008), pp. 1488–1495.

[34] Nicholas Wilt. *The cuda handbook: A comprehensive guide to gpu programming*. Pearson Education, 2013.

[35] Dan Zuras et al. "IEEE standard for floating-point arithmetic". In: *IEEE Std 754-2008* (2008), pp. 1–70.