UiO : Department of Mathematics
University of Oslo

Hybrid CPU-GPU computing for simulating calcium handling in the heart

Altanaite Neringa

Master's Thesis, Spring 2017

This master's thesis is submitted under the master's programme *Computational Science and Engineering*, with programme option *Computational Science*, at the Department of Mathematics, University of Oslo. The scope of the thesis is 60 credits.

The front page depicts a section of the root system of the exceptional Lie group E_8, projected into the plane. Lie groups were invented by the Norwegian mathematician Sophus Lie (1842–1899) to express symmetries in differential equations and today they play a central role in various parts of mathematics.

Abstract

Calcium plays a vital role in the normal functioning of a healthy human heart. Any disturbances in calcium handling can alter the excitation-contraction coupling and cellular properties, which lead to cardiac arrhythmias. Models of the electrophysiology and calcium handling in a cell can help in understanding the mechanisms of arrhythmias. Realistic simulations of the cellular and subcellular processes require a lot of computational power.

The aim of this thesis is to investigate the heterogeneous CPU-GPU computing as an approach to increase the performance of a realistic 3D Tissue-Scale simulator. We study a multiscale cardiac ventricular myocyte model which reproduces local calcium release processes and electrical activity. The cardiac cell model consists of 10000 calcium release units which contain 100 ryanodine receptors and 15 L-type calcium channels.

The most time consuming dyad-level computations are implemented on GPU using CUDA API. In order to achieve high efficiency of the simulator, we apply several optimizations to the code. Due to a large number of load and store operations, it is important to have a fast access to the memory. The completely optimized implementation demonstrates a significant speedup of the simulation. Numerical experiments showed that in order to fully utilize a single GPU, multiple cells must be involved in the computation.

Due to a large number of cells required for the realistic simulation, we implement the 3D Tissue-Scale simulator on multiple GPUs. Simulations of multiple cardiac cells are performed using multiple compute nodes equipped with two GPUs each. A good scalability was indicated by weak and strong scaling tests.

Scientific experiments demonstrated that physiological processes in a cell are correctly reproduced using the computational cell model. Thus, we are able to simulate arrhythmogenic patterns which arise from disturbances in calcium handling. This provides a possibility to understand the cause of cardiac ventricular arrhythmias and develop the preventive mechanisms.

Acknowledgements

First and foremost I would like to thank my supervisor, Johannes Langguth. Without your guidance and constant feedback this thesis would not have been completed or written. Your constructive comments and suggestions were an enormous help to me.

I would like to express my gratitude to my main supervisor, Professor Xing Cai, for advices and insightful comments. I would also like to thank Geir Kleivstul Pedersen, for being my internal supervisor.

I would like to express the deepest appreciation to Namit Gaur for numerous helpful e-mail correspondences.

Special thanks to the friendly and cheerful group of fellow students from Simula Research Labaratory. I want to thank the High Performance Computing group, which has provided an excellent working environment. I would particularly like to thank Jeremie Lagraviere who has been very supportive during my master thesis.

Finally, I thank my parents for supporting and encouraging me throughout all my studies at the University.

Contents

Contents **3**

1 Introduction **5**
1.1 Motivation . . . 5
1.2 Previous work . . . 6
1.3 Thesis outline . . . 6

2 Physiological background. The overview of mathematical and numerical methods **8**
2.1 Physiologically detailed cell modelling . . . 9
2.2 Mathematical modelling of dyadic calcium concentrations . . . 11
2.3 Mathematical modelling of calcium concentrations per cell . . . 13
2.4 Mathematical modelling of cardiac action potential . . . 14
2.5 Numerical modelling . . . 14
2.6 Summary . . . 16

3 Background and overview of heterogeneous computing **17**
3.1 Heterogeneous computing . . . 17
3.2 Graphics Processing Unit . . . 18
3.3 A platform for heterogeneous computing . . . 18
3.4 Overview of the NVIDIA KEPLER GK 110 architecture . . . 21
3.5 Hardware . . . 22
3.6 Message Passing Interface . . . 22
3.7 Performance metrics . . . 23
3.8 A brief introduction to the Roofline model . . . 25
3.9 Summary . . . 26

4 Modelling a cardiac simulator on a hybrid CPU-GPU cluster **27**
4.1 Computation distribution between CPU and GPU . . . 27
4.2 Basic approach: a single cardiac cell . . . 28
4.3 Multiple cells . . . 29
4.4 Thread configuration . . . 32
4.5 Tissue-Level parallelization . . . 34
4.6 Evaluation of the correctness of the implementation . . . 36
4.7 Summary . . . 36

5 Implementation and optimization of the dyad-level computations on GPU **37**
5.1 Common optimization techniques . . . 37

5.2 Implementation of the reduction kernel . . . 41
5.3 Stencil computation on the dyad level . . . 44
5.4 L-type channel simulation . . . 49
5.5 RyR probability calculation . . . 50
5.6 RyR opening computation . . . 51
5.7 Ca concentration computation . . . 51
5.8 Summary . . . 53

6 Experimental results and evaluation 54
6.1 Experimental setup . . . 54
6.2 Comparing results from different reduction implementations . . . 55
6.3 Results of different implementations of the diffusion computation 57
6.4 L-type channel simulation . . . 61
6.5 RyR probability calculation . . . 62
6.6 RyR opening computation . . . 63
6.7 Ca concentration computation . . . 64
6.8 Optimization impact on the distinct functions . . . 65
6.9 Cell computation speed on a single GPU . . . 67
6.10 Scaling experiments, results and analysis . . . 68
6.11 Summary . . . 69

7 Cardiac simulations 70
7.1 Conduction velocity . . . 70
7.2 A possible defibrillation strategy that targets RyR openings . . . 75

8 Discussion and Conclusion 80
8.1 Discussion . . . 80
8.2 Conclusion . . . 82

List of Tables 83

List of Figures 84

Bibliography 85

Chapter 1

Introduction

1.1 Motivation

Today, heart diseases are one of the most common causes of death. In 2013, one in nine deaths in the United States was caused by a heart failure [35]. In the recent years, a lot of research has been done to understand the cause of heart diseases. The research requires a closer look into the physiology of the heart and better understanding of the relation between the pathological condition and the myocyte at microscopic and nanoscopic levels.

Calcium is an essential component for the normal functioning of the healthy human heart. It is believed that calcium handling dysfunctions can implicate cardiac pathological conditions. Divergence in calcium release processes, various mutations in calcium buffering and changes in calcium interaction with ryanodine receptors (RyRs) are involved in cardiac arrhythmias [18, 26]. Any structural changes in the dyadic properties can affect the excitation-contraction coupling.

At present, it is ambitious to study microscopic processes in subcellular compartments and understand the relation between the processes in the subcellular level and the whole-cell using experimental approaches. However, the development in computing technologies and advances in numerical methods made it possible to study these phenomena through computer modelling. The advanced stochastic models of the cardiac cell, which integrate calcium release processes in the dyadic spaces and electrophysiology of a cell, were developed to study the dyadic dysfunctions and membrane potential abnormalities [22, 38]. This study can help to understand the cause of various cardiac pathological conditions and develop preventive mechanisms.

Due to the large computational demands, it is challenging to perform tissue scale cardiac simulations. A human heart has around 2×10^9 cells [1]. Each cell has about 10^6 RyRs and 10^5 L-type calcium channels that are distributed among 10^4 calcium release units. Recently, realistic simulations with detailed cell models of calcium handling at the tissue or organ level started to become computationally achievable due to the increased computational power of supercomputers.

Supercomputing or High Performance Computing (HPC) is the use of parallel processing techniques and supercomputers for solving large and demanding computational problems [48]. Over the years the number of processors in su-

percomputers increased from a few to thousands, which increased the capacity of supercomputing [8]. This made it possible to perform large-scale scientific and engineering applications in a reasonable amount of time.

As the Graphic Processing Unit (GPU) turned into a more powerful, programmable and highly parallel unit [43], it became an important component in HPC systems and the most prominent type of hardware accelerator. Many supercomputers consist of multicore CPUs and manycore accelerators, such as GPU. The high computational throughput and high parallelism made GPUs suitable for general purpose computing. Researches adopted GPU for diverse range of scientific applications [21].

Due to their massive parallelism, tissue-level simulations can take advantage of hardware accelerators. The detailed 3D Tissue-Scale model of electrical activity and calcium handling in the human cardiac ventricle was implemented on CPU [30] and later accelerated using heterogeneous computing [31, 32].

Motivated by the computational power of the GPU, we present a method to enable massive parallel simulations of a calcium circulation on the dyad-level on GPUs. Our aim is to implement the 3D Tissue-Scale model of calcium handling and electrical activity in the human cardiac ventricle on heterogeneous clusters that consist of multicore CPUs and manycore GPUs. Furthermore, we seek to show the scientific goals of the simulator by carrying out physiologically realistic simulations.

1.2 Previous work

This thesis is a part of the Tissue-Scale 3D cardiac simulations project that has been developed by High Performance Computing researches at Simula research laboratory. The model and computations are based on previous work described in [22, 30, 31] and [32]. In [30] a detailed human cardiac cell model [22] along with the parallel simulator was implemented on the Abel supercomputer [42]. Numerical experiments showed that various physiological behaviours are correctly reproduced using this model. In [31] the same model was implemented using a CPU-Xeon Phi heterogeneous design. Due to the various optimizations, described in the aforementioned studies, it became possible to perform a large scale simulations in a reasonable time. In [32] the optimizations were made using SIMD code vectorization and combination of OpenMP and MPI programming. Preliminary experiments have shown that GPUs can provide better performance than CPUs, which is why the main focus of this thesis is GPU implementation.

1.3 Thesis outline

The main part of this thesis is directed to the implementation and analysis of the 3D Tissue-Scale simulator and the rest of the thesis consists of the scientific experiments, verification and validation of the model. The structure of the thesis is as follow:

Chapter 2 describes a physiologically detailed cardiac cell model and introduces the most important definitions of cell physiology. Further, it presents mathematical models of the processes in a cell, and introduces numerical and stochastic methods to solve them.

Chapter 3 presents the background of the heterogeneous computing with focus on the GPU architecture and programming platform, which is used for the implementation of the simulator. In addition, it introduces important concepts about the CUDA application programming interface (API) and metrics for the performance measurements.

Chapter 4 focuses on the development of the simulator and computation distribution. The process of the development of the simulator consists of three stages: (1) the cell model implementation and optimization on a single GPU, (2) multiple cells implementation and optimization on a single GPU, and (3) multiple cells implementation on multiple GPUs. Furthermore, this chapter describes the structure which is used for the communication between different cell domains.

Chapter 5 introduces various optimization techniques and algorithms, which is used to achieve better performance. In this chapter the implementation of each execution kernel function is described separately.

Chapter 6 contains presentation and analysis of the results obtained due to the optimizations of separate CUDA execution kernel functions. Performance measurements collected running the implementation on the supercomputer are reported.

Chapter 7 focuses on the scientific application of the tissue simulator. In this chapter cell model is verified, tested and results are compared with the published model of a human cell.

Chapter 8 concludes the thesis by discussing the achieved performance and results from the scientific simulations. Further it presents future prospects of porting implementation to more recent hardware accelerators and possible improvements of the implementation.

Chapter 2

Physiological background. The overview of mathematical and numerical methods

The human heart is a muscular organ with rhythmic electrical activity and mechanical muscle contraction. The electrical and mechanical coupling generates a heartbeat. A cardiac muscle is formed of cardiomyocytes that have the ability to contract after being stimulated by a spontaneous electrical impulse called action potential. This impulse causes depolarization and contraction of atria and ventricles. In the cardiac ventricle when a cellular membrane is depolarized by an action potential, L-type channels open, and calcium enters the dyadic space from extracellular space and activates a large calcium inflow from RyRs. This calcium-inducted calcium release is known as a Ca spark. It results in increasing amount of calcium across the cell, which is important for the contraction of the heart muscle [5]. Calcium flow from RyRs can also occur spontaneously without influx from voltage-gated channels. Closing of the L-type calcium channels and decreasing generation of calcium leads to the repolarization of the cardiac myocyte membrane, in other words voltage becomes negative as before the depolarization. Further, the cardiac myocyte cell membrane repolarizes to resting membrane potential. In the healthy heart, each electrical activity is followed by the contraction. Any abnormalities of initiation and conduction of electrical activity can lead to a heart failure [49].

A realistic simulation of a human heart tissue requires the precise replication of all the intracellular and intercellular processes including the electrophysiology of the heart. Modeling a cardiac cell is a challenging task because of its complex structure. However, due to the development in technology, the understanding of the cardiac cell processes and their properties has increased in the last decades. Models of cardiac cells of different species on different levels of detail have been developed to understand the cause of various heart diseases. The computational cardiac models range from a single cardiac myocyte [38] to ventricles [51] and the whole heart [50].

The model [22] that we chose in this study is an extended model of calcium cycling that includes sarcolemmal ionic currents, pumps and exchangers. At the local calcium release level, the model reproduces calcium spark properties, while at the whole-cell level it reproduces an action potential, calcium currents and calcium transients.

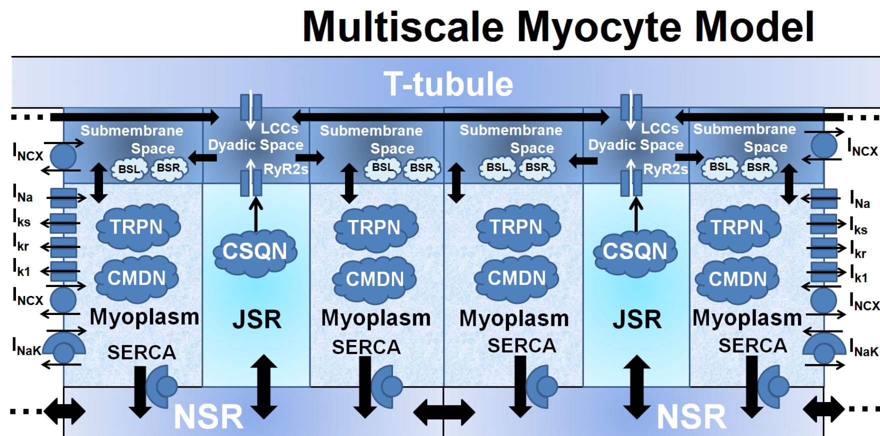

Figure 2.1: [22] A ventricular myocyte model that shows composition and communication of dyads. Calcium diffusion between dyads occurs in the dyadic space and NSR. White arrows represent interaction between L-type channels and RyRs. Each dyad contains five compartments with corresponding calcium buffers. Thin arrows indicate currents that affect membrane potential.

2.1 Physiologically detailed cell modelling

Detailed simulations of calcium handling in a human heart require a complex computational cell model that reflects calcium handling in a cardiac myocyte not only on the cell level, but also on the local calcium release level. The cardiac cell model that is used in this thesis is adopted from [30, 31, 32]. It uses the O'Hara-Rudy (ORd) model [41] to reproduce the cardiac ventricular action potential of a healthy human heart. Moreover, it applies a stochastic multiscale calcium cycling model [22] that replicates calcium release processes at the dyadic and the whole-cell level.

The multiscale cell model is shown in Figure 2.1. The cell model consists of 10000 calcium release units or dyads arranged as a 3D grid. Calcium release units are coupled by diffusion. Interaction between them occurs in a dyadic space and network sarcoplasmic reticulum (NSR). Diffusive calcium fluxes appear between myoplasm and the submembrane space in a dyad. Membrane currents, pumps, and exchangers that are calcium dependent perceive local calcium concentrations in their partition.

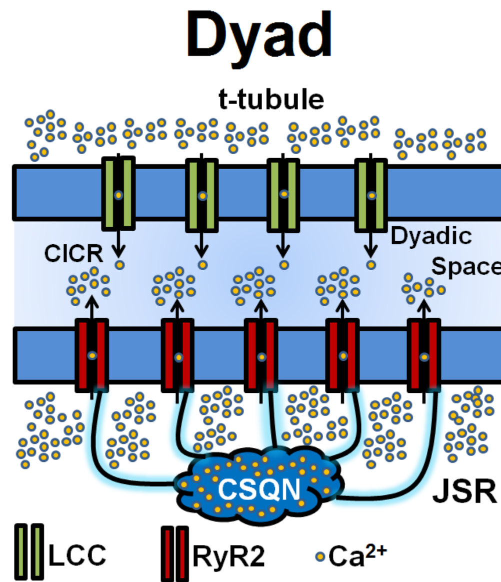

Figure 2.2: [22] Interaction between L-type calcium channels and RyRs in a dyad. L-type channels are voltage dependent calcium channels positioned along t-tubules. RyRs are calcium-induced channels with four states which are regulated by CSQN calcium buffer. JSR is the domain of CSQN distribution. Ca inflow from the RyR channels can occur spontaneously or can be activated by Ca influx from L-type channels into the dyadic space. JSR is the domain of CSQN distribution.

Dyad model

It is assumed that calcium release units are organized as a $100 \times 10 \times 10$ grid. Each dyad contains five calcium distribution compartments: myoplasm, submembrane space, dyadic space, network sarcoplasmic reticulum and junctional sarcoplasmic reticulum (JSR). Calcium concentration is assumed to be uniform in each of them. Calcium release occurs in the dyadic space, which consists of 15 L-type calcium channels (LCCs) and 100 RyRs. Figure 2.2 shows calcium release in the dyadic space. Calcium signals operate between RyRs and L-type channels that are positioned in the t-tubules [20]. Calcium inflow from the RyR channels can occur spontaneously or can be activated by calcium influx from L-type calcium channels into the dyadic space [5, 19]. In a healthy cardiac ventricular myocyte membrane depolarization causes a synchronous calcium release.

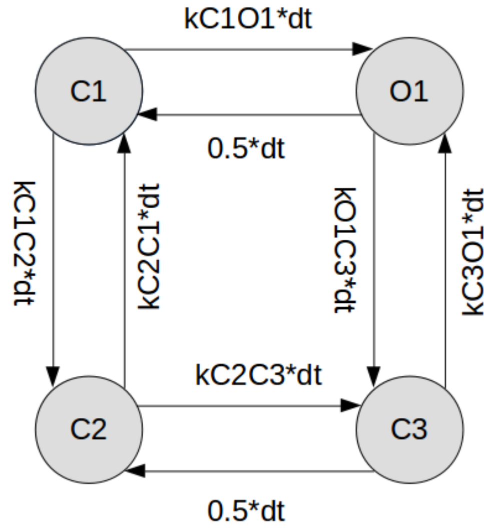

Figure 2.3: Possible transitions between four RyR states. The arrows indicate the direction of transitions with the probability shown in the labels on the arrows. The probability is related to the local calcium concentration.

RyRs model

The model of RyRs consists of four states and two tiers of modal gating. The lower tier is the refractory tier, and the upper tier is the activation tier. The model of possible transitions between states of RyR is shown in Figure 2.3. Here C1, C2, C3, and O1 are the four states that an RyR can have. The channel is open and calcium is released from the channel during the state O1. Each RyR can be in one of the four states at the time. The probability of the transition from one state to another is related to the local calcium concentration.

2.2 Mathematical modelling of dyadic calcium concentrations

The increasing level of knowledge of the processes in a cell contributes to the development of more accurate and complex mathematical models that are able to represent calcium handing at different cell levels. In the computational model [22] that we are using, calcium concentration in a dyad are modelled as ordinary and partial differential equations. Ca represents the global calcium concentration in a cell and ca represents concentration in a dyad. Calcium fluxes from one space to another in a dyad are shown in Figure 2.1.

Calcium concentration in the dyadic space

$$ca_{ds} = (J_{rel} + J_{lca} + \frac{ca_{ss}}{\tau_{eflflux}}) \times \tau_{efflux} \quad (2.1)$$

$$Ca_{ds} = (J_{rel} + J_{lca} + \frac{Ca_{ss}}{\tau_{eflflux}}) \times \tau_{efflux} \quad (2.2)$$

A subscript *ds* denotes dyadic space. J_{rel} and J_{lca} are the calcium release through RyRs and L-type calcium channels respectively. $\tau_{eflflux}$ is a diffusion constant between the dyadic space and the submembrane space.

Ca concentration in the submembrane space

$$\frac{dca_{ss}}{dt} = \overline{B_{ss}}(J_{NCX} + J_{diff-myo-ss} + J_{diff-ds-ss}) \quad (2.3)$$

$$\begin{aligned}\frac{\partial Ca_{ss}}{\partial t} &= \overline{B_{ss}}(J_{NCX} + J_{diff-myo-ss} + J_{diff-ds-ss} \\ &+ D_{Ca}\frac{\partial^2 Ca_{ss}}{\partial x^2} + D_{Ca}\frac{\partial^2 Ca_{ss}}{\partial y^2} + D_{Ca}\frac{\partial^2 Ca_{ss}}{\partial z^2})\end{aligned} \quad (2.4)$$

Subscript *ss* denotes submembrane space. J_{NCX} is the calcium flux through the Na-Ca exchange current into the submembarane space. $J_{diff-myo-ss}$ is the diffusive flux between myoplasm and submembrane space. $J_{diff-ds-ss}$ is the diffusive flux between dyadic and submembrane space. $\overline{B_{ss}}$ is a buffering factor, that includes both BSR and BSL buffers. Equation 2.4 includes a diffusion part which corresponds to an inner-dyad coupling.

Calcium concentration in the JSR

$$\frac{dca_{JSR}}{dt} = \overline{B_{JSR}}(J_{rel} + J_{diff-NSR-JSR}) \quad (2.5)$$

$$\frac{dCa_{JSR}}{dt} = \overline{B_{JSR}}(J_{rel} + J_{diff-NSR-JSR}) \quad (2.6)$$

$J_{diff-NSR-JSR}$ is the diffusive flux between NSR and JSR. $\overline{B_{JSR}}$ is the buffering factor by calsequenstrin (CSQN).

Calcium concentration in the NSR

$$\frac{dca_{NSR}}{dt} = J_{up} - J_{leak} - J_{diff-NSR-JSR} \quad (2.7)$$

$$\begin{aligned}\frac{\partial Ca_{NSR}}{\partial t} &= J_{up} - J_{leak} - J_{diff-NSR-JSR} \\ &+ D_{SR}\frac{\partial^2 Ca_{NSR}}{\partial x^2} + D_{SR}\frac{\partial^2 Ca_{NSR}}{\partial y^2} + D_{SR}\frac{\partial^2 Ca_{NSR}}{\partial z^2}\end{aligned} \quad (2.8)$$

J_{up} is the uptake Ca flux into NSR from myoplasm. J_{leak} is the leak flux from NSR into the myoplasm. $J_{diff-NSR-JSR}$ is the diffusive flux between NSR and JSR. Equation 2.8 includes the inner-dyad coupling.

Calcium concentration in the Myoplasm

$$\frac{dca_{myo}}{dt} = \overline{B_{myo}}(J_{cab} + J_{pca} + J_{NCX} - J_{up} + J_{leak} - J_{diff-myo-ss}) \quad (2.9)$$

$$\frac{\partial Ca_{myo}}{\partial t} = \overline{B_{myo}}(J_{cab} + J_{pca} + J_{NCX} - J_{up} + J_{leak} - J_{diff-myo-ss} + D_{Ca}\frac{\partial^2 Ca_{myo}}{\partial x^2} + D_{Ca}\frac{\partial^2 Ca_{myo}}{\partial y^2} + D_{Ca}\frac{\partial^2 Ca_{myo}}{\partial z^2}) \tag{2.10}$$

J_{cab} is the flux through the background calcium current. J_{pca} is the flux through the sarcolemmal calcium pump. J_{NCX} is the calcium flux through the Na-Ca exchange current. $J_{diff-myo-ss}$ is the diffusive flux between myoplasm and submembrane space. $\overline{B_{myo}}$ is the instantaneous buffering factor in myoplasm by calmodulin (CMDN) and troponin (TRPN). J_{up} and J_{leak} were defined in Equation (2.7) and Equation (2.8).

2.3 Mathematical modelling of calcium concentrations per cell

In order to get the calcium concentration value that represents the corresponding concentration of a particular cell, the average of dyadic concentrations must be calculated. In each calcium compartment we sum up the calcium concentration across dyads in the cell and divide the result by the number of calcium release units in the cell.

Calcium concentration in the dyadic space

$$Ca_{ds}^{cell} = \frac{\sum_{n=1}^{N} ca_{ds}^{n}}{N} \tag{2.11}$$

Ca concentration in the submembrane space

$$Ca_{ss}^{cell} = \frac{\sum_{n=1}^{N} ca_{ss}^{n}}{N} \tag{2.12}$$

Calcium concentration in the JSR

$$Ca_{JSR}^{cell} = \frac{\sum_{n=1}^{N} ca_{JSR}^{n}}{N} \tag{2.13}$$

Calcium concentration in the NSR

$$Ca_{NSR}^{cell} = \frac{\sum_{n=1}^{N} ca_{NSR}^{n}}{N} \tag{2.14}$$

Calcium concentration in the Myoplasm

$$Ca_{myo}^{cell} = \frac{\sum_{n=1}^{N} ca_{myo}^{n}}{N} \tag{2.15}$$

Here, subscripts *ds*, *ss*, *JSR*, *NSR* and *myo* denote the five compartments of a dyad. Index n represents a specific dyad. The number of dyads ranges from $n = 1$ to $n = N$, where N is the total number of dyads.

2.4 Mathematical modelling of cardiac action potential

Cardiac cells are excitable, which means that they have the ability to trigger an action potential in the neighbouring cells. Movements of ions across the cell membrane produces an action potential which causes depolarization or repolarization of the membrane. The action potential propagates through the cardiac cells, which leads to the contraction of the whole heart muscle.

In the computational cell model that we use in this study, tissue-scale electrical activity is modelled mathematically as a reaction-diffusion equation called monodomain model. This is a simplification of a more accurate binomial modeling assuming that conductivity in extracellular space is proportional to conductivity in the intracellular space.

$$\frac{\partial V_m}{\partial t} = \frac{-I_{ion}}{C_m} + D_x \frac{\partial^2 V_m}{\partial x^2} + D_y \frac{\partial^2 V_m}{\partial y^2} + D_z \frac{\partial^2 V_m}{\partial z^2}. \tag{2.16}$$

Here V_m is the membrane potential and I_{ion} is the algebraic sum of all the currents provided by the underlying multiscale cell model of calcium handling described above. C_m is the membrane capacitance of the cell and it is defined as $C_m = 1\mu Fcm^{-2}$, D_x, D_y, D_z are the voltage diffusion coefficients in x, y and z-dimensions respectively and they are defined as $D_x = D_y = D_z = 0.2mm^2/ms$. The solution domain is modeled as a 3D uniform grid made of cardiac cells.

2.5 Numerical modelling

Due to the increasing computational power of today's supercomputers, the most complex mathematical models can be solved numerically. In the previous sections, we have presented the real-world problem - calcium handling in a human heart, and described the mathematical model of the problem. In this section, we present numerical methods used to solve the equations and perform simulations.

Finite difference method

To solve differential equations (2.3)-(2.10) we use the finite difference method. Since in the tissue model dyads are represented as a 3D grid, we only need to discretize the temporal domain $[0, T]$, where T denotes the total time. The whole domain $[0, X] \times [0, Y] \times [0, Z] \times [0, T]$, where X, Y and Z denotes number of dyads in all three spatial dimensions, is represented as a set of uniform mesh points:

$$\begin{aligned}
x_i &= i\Delta x, \quad \text{for } i = 0, \ldots, Nx, \quad \text{where } x_{Nx} = X, \\
y_j &= j\Delta y, \quad \text{for } j = 0, \ldots, Ny, \quad \text{where } y_{Ny} = Y, \\
z_k &= k\Delta z, \quad \text{for } k = 0, \ldots, Nz, \quad \text{where } z_{Nz} = Z, \\
t_n &= n\Delta t, \quad \text{for } n = 0, \ldots, Nt, \quad \text{where } t_{Nt} = T.
\end{aligned} \tag{2.17}$$

The differential equations are fulfilled at the interior mesh points only; the boundary points need to be handled separately. To solve the equations in our model using the finite difference method, we need to replace all derivatives with finite differences. The first order derivatives are replaced by forward differences,

while the second order derivatives are replaced with centered finite differences. Equation (2.18) and Equation (2.19) are examples of forward difference and centered difference approximations applied to time derivatives.

$$u'(t_n) \approx \frac{u^{n+1} - u^n}{\Delta t} \tag{2.18}$$

$$u''(t_n) \approx \frac{u^{n+1} - 2u^n + u^{n-1}}{(\Delta t)^2} \tag{2.19}$$

Equations (2.20)-(2.21) show an examples of the finite difference method applied to the calcium concentration in the submembrane space presented in Equation (2.3) and Equation (2.4). Other calcium concentration equations are handled in the similar way.

$$ca_{ss}^{n+1} = ca_{ss}^{n} + \Delta t \overline{B_{ss}} (J_{NCX} + J_{diff-myo-ss} + J_{diff-ds-ss}). \tag{2.20}$$

$$\begin{aligned} Ca_{ss_{i,j,k}}^{n+1} = Ca_{ss_{i,j,k}}^{n} &+ \Delta t \overline{B_{ss}} (J_{NCX} + J_{diff-myo-ss} + J_{diff-ds-ss} \\ &+ \frac{Ca_{ss_{i-1,j,k}}^{n} - 2Ca_{ss_{i,j,k}}^{n} + Ca_{ss_{i+1,j,k}}^{n}}{\Delta x^2} \\ &+ \frac{Ca_{ss_{i,j-1,k}}^{n} - 2Ca_{ss_{i,j,k}}^{n} + Ca_{ss_{i,j+1,k}}^{n}}{\Delta y^2} \\ &+ \frac{Ca_{ss_{i,j,k-1}}^{n} - 2Ca_{ss_{i,j,k}}^{n} + Ca_{ss_{i,j,k+1}}^{n}}{\Delta z^2}). \end{aligned} \tag{2.21}$$

$$\text{for } 0 < i < x_{Nx}, \quad 0 < j < y_{Ny}, \quad 0 < k < z_{Nz}, \quad 0 \leq n < T.$$

The boundary conditions are treated in the same manner for all the calcium concentration equations:

$$\begin{aligned} &\text{For } i = 0 \quad i-1 = i+1, \quad \text{for } i = Nx \quad i+1 = i-1. \\ &\text{For } j = 0 \quad j-1 = j+1, \quad \text{for } j = Ny \quad j+1 = j-1. \\ &\text{For } k = 0 \quad k-1 = k+1, \quad \text{for } k = Nz \quad k+1 = k-1. \end{aligned} \tag{2.22}$$

Operator-splitting approach

To solve the monodomain equation (2.16) we use an operator-splitting approach [44]. This advanced algorithm is used for solving partial differential equations in cardiac conduction and it speeds up computation without loosing accuracy. This approach introduces two operators: one for the diffusion part and another for the non-diffusion part, and applies them on each of the corresponding parts separately. In our model Equation (2.16) is split into two parts: the diffusion part, which is solved using the finite difference method and the I_{ion} part which is computed by solving the cell model.

Stochastic methods

The states of L-type calcium release channels and RyRs in a dyad fluctuate stochastically, thus the stochastic method is needed to represent the state of calcium channels numerically. Since L-type calcium channels and RyRs are handled similarly, in this section we focus on the simulation of RyR state transitions. Recall that each dyad contains 100 RyRs, which can be in one of four

possible states (Figure 2.3). In [30] it is shown that the most efficient way of obtaining the number of RyRs that changed state in the current time step is to take two samples from binomial distributions for each of the four states, more precisely, one sample for each direction of the transition. In this thesis we apply the binomial distribution sampling method presented in [31].

The binomial cumulative distribution function is defined as follows:

$$F(k,n,p) = Pr(X \leq k) = \sum_{i=1}^{k} \binom{n}{k} p^k (1-p)^{n-k}, \quad \text{where } \binom{n}{k} = \frac{n!}{k!(n-k)!}. \tag{2.23}$$

In Equation (2.23), k is the number of successes in n trials with the individual probability of success p. We pick a random number r from a uniform distribution in the interval [0,1], then we find the smallest number k such that

$$r \leq F(k,n,p)$$

We obtain the number of RyRs in each state by sampling from binomial distributions. Let x_i be the number of RyRs in the state i and k_{ij} the number of RyRs transitioning from the state i to the state j. First we take one sample from binomial distribution $B(n,p)$ with $n = x_i$ and $p = p_{ij}$ to compute the number of RyRs transitioning from state i to state j. Then, we take another sample to compute the number of RyRs transitioning from state i to state l. Because only one of these transitions can happen, we have $n = x_i - k_{ij}$ and $p = p_{il}/(1-pij)$. To obtain the final number of RyRs in the next time step, we add the RyRs that transitioned from the neighbour states as shown in Equation 2.24. If no transition happens, the RyR remains in its original state.

$$x_i^{t+1} = x_i^t - k_{ij}^t - k_{il}^t + k_{ji}^t + k_{li}^n \tag{2.24}$$

Moreover, in [31] the optimized implementation of the binomial distribution sampling function was described. The distribution function is computed iteratively by subtracting from random number r. The computation stops when the smallest k satisfying the condition $r \leq F(k,n,p)$ is found.

2.6 Summary

To understand how modifications of calcium release can affect the human heart, we need to look at the multiscale model of cardiac myocytes. In this chapter we have presented a model that reproduces calcium handling in the cell on the dyad-level and action potential on the cell-level. The L-type channels and RyRs are of principal interest due to effect of calcium release on the action potential. Moreover, we described mathematical models of calcium concentrations in dyad compartments, inter-dyad couplings and the reaction-diffusion equation for the action potential computation. Numerical methods such as the finite difference method and operator-splitting approach were introduced as solutions to mathematical equations. Sampling from binomial distribution allows us to simulate the opening of L-type calcium channels and RyRs stochastically.

Chapter 3

Background and overview of heterogeneous computing

In the previous chapter we introduced the cardiac cell model of electrophysiology and calcium handling. Moreover, we presented numerical and stochastic approaches that allow us to perform cardiac simulations. Now we will introduce the specific hardware that is suitable for our implementation due to the large computational requirements, substantial parallelism and importance of the throughput. In this thesis we seek to utilize a hybrid CPU-GPU system, which is well-suited for the large and complex computations. Since our application contains data-parallel computation-intensive tasks, most of the computation is performed on GPU, whose description constitutes the main part of this chapter.

3.1 Heterogeneous computing

Heterogeneous computing [28] uses different processor architectures to execute an application by dividing it into tasks and assigning each task to the most suitable architecture. Until recently, computers consisted only of the central processing units (CPUs), which allow to perform general tasks. Over the last decade, High Performance Computing became oriented toward heterogeneous systems by including other processing units, such as GPUs, to their systems. A typical heterogeneous compute node consists of multicore CPU sockets and GPUs, both of them are discrete units connected by the PCI-Express bus, and it is widely used for scientific computing [2, 4]. GPU is designed for the applications with large parallel computation requirements and focus on the throughput [43], while CPU has its advantages on applications with a low level of parallelism, small data size and control intensive tasks. Thus, to utilize the computational power of the hybrid CPU-GPU system, the tasks should be divided and assigned to the processors according to their computational requirements.

3.2 Graphics Processing Unit

The first GPUs were designed as a fixed-function processors, built around the graphics pipeline. Over the years, the demand for more powerful and complex graphics increased, which yielded enhanced focus on the programmable parts of the pipeline. As a result, the GPU has evolved into a powerful programmable processor by changing its architecture from being pipelined task-parallel to being a singe unified data-parallel programmable unit [43].

Despite the fact that general-purpose computing on graphics processing units (GPGPU) is not related to graphics, applications still needed to be structured in terms of the pipeline. As a solution, the high level interfaces such as CUDA [11], which provides direct non-graphic interface to the hardware, were introduced. Most importantly, this programming model supports the hybrid CPU-GPU computing.

3.3 A platform for heterogeneous computing

CUDA is a parallel computing platform for GPGPU computing developed by NVIDIA. It uses a small set of extensions to the C programming language. A heterogeneous environment consists of *host* - the CPU and its memory, and *device* - the GPU and its memory. Both of them operate independently for most of the operations. The application is initialized on the host, which is responsible for the setup of the environment and managing the data before transferring it to the device. A function, which is performed on the device is called *execution kernel*, and it is specified and launched by the host. The code inside the kernel function is expressed as a sequential program, which is executed by a large number of threads generated during the kernel initiation. After the kernel launch, the control is returned immediately to the host, which allows the additional work to be performed on the CPU, while the parallel code is running on the GPU. When the computation on the device is completed, the results can be copied back to the host memory.

A function that is executed and launched only by the device is defined by a special qualifier __*device*__. An execution kernel is defined by using __*global*__ declaration specification, and it is launched by specifying execution configuration inside the triple-angle-brackets $<<< grid, block >>>$. The first parameter inside the brackets defines the number of blocks in a grid. The second parameter defines the number of threads in a block. *A grid* contains all threads generated by a single kernel launch. The grid is composed of *blocks* of threads. All threads in a block can access the same local *shared memory* and can be synchronized. Threads from different blocks cannot cooperate. Each block and each thread within a block has its unique ID, which can be accessed by the build-in variables *threadIdx* and *blockIdx*. In CUDA threads and blocks are organized in three dimensions, and the size of dimensions can be accessed by the build-in variables *gridDim* and *blockDim*. The global thread ID for a one-dimensional grid can be computed as

$$thread_idx = blockDim.x \times blockIdx.x + threadIdx.x. \tag{3.1}$$

For 3D grid, we compute thread ID in all three dimensions in the similar way. The hierarchy of threads is shown in Figure 3.1.

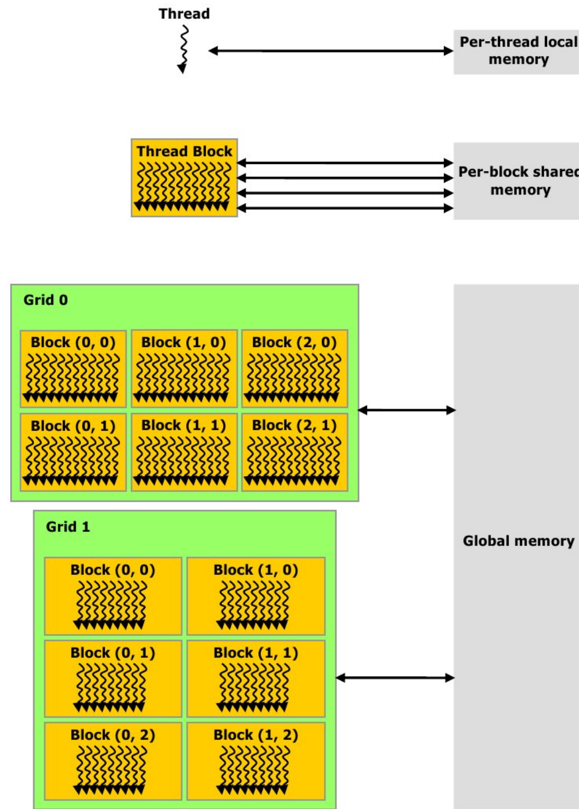

Figure 3.1: [11] CUDA memory and thread hierarchy. Here, threads are organized in 2D blocks which are assigned to the grid. Each block and each thread has unique ID. Each thread has access to its own local memory. All threads in a block has per-block shared memory. Each thread in the grids can access global memory.

3.3.1 Memory hierarchy

As defined in [9], there are three main memory places visible for threads: *per-thread local memory*, *per-block shared memory* and *global memory*. The memory hierarchy is shown in Figure 3.1. Per-thread memory is a local memory, which is private to each thread and can be accessed only by the thread. Variables that are eligible for the registers, but cannot fit into register space will be assigned to the local memory. This type of memory belongs to the same physical location as a global memory. Per-block shared memory is on-chip memory that can be accessed by all the threads in a block. It is private to each block, and it has a life-time of the block. Shared memory has a higher bandwidth and a lower latency than local or global memory. Moreover, the access to shared memory is approximately 10 times slower than to a register, but 100 times faster than to a global memory [54]. Shared memory is defined by the __*shared*__ qualifier. Global memory can be accessed by all the threads, because it is shared between threads, blocks and grids. It is the largest memory and has

a life-time of a program. A variable can be declared in global memory from the device by using __*device*__ qualifier, otherwise it is allocated from host. More information about the memory hierarchy and other memory types can be found in [6].

3.3.2 Thread execution on GPU

During the kernel launch all thread blocks are distributed among available Streaming Multiprocessors (SM) [39], such that all threads in a block execute concurrently on the same SM. Thousand of threads can perform simultaneously, because one GPU contains multiple SMs. Each SM partitions registers and shared memory among assigned threads, and schedules each thread block into groups of 32 threads, called *warps*. CUDA uses Single Instruction Multiple Threads (SIMT) architecture, which means that all threads in a warp execute the same instruction at the same time. It is possible that threads in a warp have different behaviour, but if one of the threads needs to execute a different instruction, all the threads perform that instruction as well. This is called *warp divergence* [55] and it can cause a degraded performance.

Ideally, we want to keep the cores of device busy. If one warp in SM is not able to issue an instruction, then SM executes another available warp. To measure the activity of the SM we look at the *occupancy* - a ratio between active warps and maximum warps.

3.3.3 Multiple devices

Multiple GPUs are used when problem domain size is too large for one GPU, because device has a limited amount of memory. Additionally, multiple GPUs can be used to increase speed of memory transfers and speed of the application by executing multiple tasks concurrently on the multiple devices.

To run the code on the multi-GPUs, we need to know how many devices are available and specify the target device. It can be done by calling the functions shown in Listing 3.1. The code is executed on the specified device until cudaSetDevice is called with a different id argument. If cudaSetDevice is not called before the first CUDA API call, then the code will be run on the default device with ID 0.

```
cudaError_t cudaGetDeviceCount(int* count);

cudaError_t cudaSetDevice(int id);
```

Listing 3.1: Multiple device management using CUDA. The first function returns the number of available devices. The second function specifies the device on which the computation will be performed.

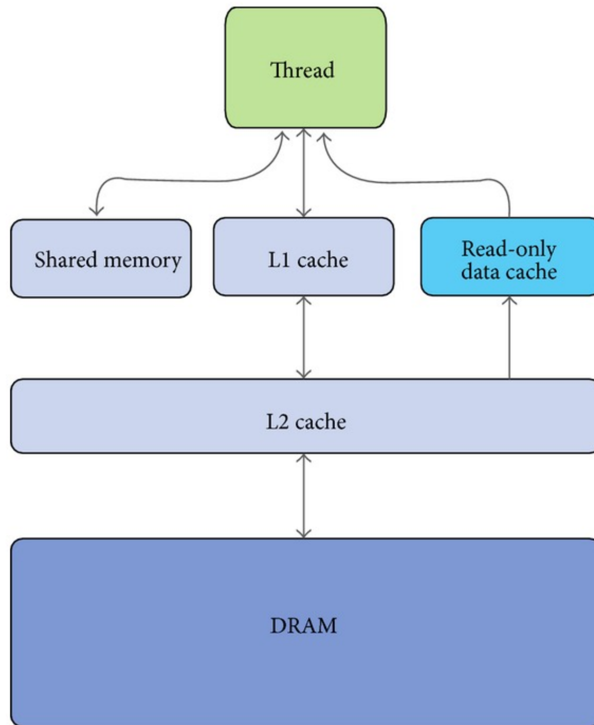

Figure 3.2: [39] Kepler Memory Hierarchy. Shared memory and L1 cache are accessible by all threads in a block. It is program-managed and configurable memory with low latency and high bandwidth. Read-only data cache is used for read only access and it is hardware managed. Global memory or DRAM is accessible by all threads, host and other GPUs in the same system. All accesses to the global memory go through L2 cache.

3.4 Overview of the NVIDIA KEPLER GK 110 architecture

The main GPU architecture used to perform our computations is Kepler GK 110 [39]. The specific GPU used in this study is Tesla K20m [12] that consists of a GK 110 processor equipped with 5 GB of GDDR5 memory. It is a high-end professional GPU developed by NVIDIA, launched in January 2013. It includes 13 Streaming Multiprocessors (SM), each of them consists of 192 single-precision CUDA cores and 64 double-precision units achieving 1174.78 GFLOPS in double-precision peak performance. CUDA cores are arithmetic logic units that perform the actual computation. Each SM implements four warp schedulers and eight instruction dispatch units, which allows four warps and two independent instructions per warp to be executed concurrently at each cycle.

Kepler memory hierarchy [39] consists of shared memory, L1 and L2 caches, read-only cache and global memory or DRAM. SM has 64 KB of on-chip memory that is used by shared memory and L1 cache. The on-chip memory can

be partitioned as 48 KB of the shared memory and 16 KB of the L1 cache or vice versa, also it can be divided equally. Stream cores inside an SM communicate using these low latency and high bandwidth memories. Each thread can access up to 255 registers, which is a compiler-managed local memory to each thread. For the data that can only be used for read operation, Kepler introduces a hardware-managed 48 KB read-only data cache, which can be directly accessed by the SM. L2 cache of size 1536 KB is a hardware-managed storage that is shared across the device. This cache is the primary point of data consolidation between the SM units. The majority of the memory in GPU is global memory, which is accessible by all threads, host and all GPUs in the same system. The loads and stores into and from the global memory are cached in L2 cache. Kepler's register, L1 and L2 caches, shared and DRAM memory are protected by Single-Error Correct Double-Error Detect ECC code [39]. The memory hierarchy implemented in the Kepler architecture is shown in Figure 3.2.

3.5 Hardware

We test our implementation on two different systems. To test the single GPU implementation we use Lizhi - a GPU system operated by Simula Research Laboratory. The system is configured with two NVIDIA Tesla K20m cards. To perform computations on multiple GPUs we use Abel [42], a supercomputer operated by the University of Oslo. Each compute node has a minimum of 64 GB RAM and 16 physical CPU cores. The interconnect is FDR (56 Gbps) Infiniband. Abel has a set of accelerated nodes equipped with Dual Intel Xeon(R) CPU E5-2609 processors with two NVIDIA Tesla K20X cards installed. Each card has 6 GB of device memory. The specifications of both graphic cards are shown in Table 3.1.

Specifications	K20	K20X
Double precision compute power	1.17 Tflops	1.31 Tflops
Memory size	5 GB	6 GB
Memory bandwidth	208 GB/s	250 GB/s
STREAM measured memory bandwidth	180 GB/s	

Table 3.1: Kepler K20 and K20X specifications taken from [13]

3.6 Message Passing Interface

Message Passing programming paradigm is the approach for programming parallel computers. The message passing platform consists of p processing nodes with exclusive address space. Each of the nodes can be a single processor or a shared address space multiprocessor. The interaction between processes executing on the different nodes is accomplished by message passing. MPI [23] is a standard library that is used to develop portable message-passing programs using C. The basic routines are initialization and termination of the MPI library, send and receive messages and get the information about the parallel environment such as number of processes and label of the calling process.

3.7 Performance metrics

To verify the performance of our implementation, we use different metrics: time measurement, floating-point operations per second (FLOPS) and memory bandwidth. Timing can be done by CPU and GPU timers. FLOPS and memory bandwidth can be found using NVIDIA profiling tools, which are explained in Section 3.7.1.

CPU timer measures the elapsed time of a CUDA call or kernel execution. This time includes the time required for launching the kernel. CUDA API functions are asynchronous, thus they return the control back to the CPU thread immediately after the call. In order to measure the execution time, we need to synchronize CPU thread with GPU. This is done by calling *cudaDeviceSynchronize()* function.

In this thesis we use two CPU timers: *gettimeofday()* and *MPI_ Wtime()*. The former returns the actual time of the day, while the latter returns an elapsed time on the calling processor.

CUDA GPU timer is an alternative to CPU timers provided via CUDA Event API [9]. It creates an event, records it and computes the difference between two recorded events into floating point value in milliseconds. The example of the CUDA event use is shown below in Listing 3.2.

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
float milliseconds;
cudaEventRecord(start,0);

KernelFunctionCall<<<grid,blocks>>>( );

cudaEventRecord(stop,0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&milliseconds, start, stop);
```

Listing 3.2: Example of usage of CUDA Event for measuring kernel execution time.

Number of floating-point operations per second is defined as a number of single-precision or double-precision floating point operations that are processed per second. It is a metric used to measure computational capability, and it is commonly used for scientific computing applications performed on a supercomputer.

Memory bandwidth is defined as an amount of data that is read or stored to memory per second. It is expressed in gigabytes per second. Bandwidth can be affected by the choice of the memory and the way data is stored and accessed. During the kernel evaluation we will measure bandwidth corresponding to the device memory. The theoretical bandwidth is given by hardware specifications while the actual memory bandwidth can be found using NVIDIA profiling tools (Section 3.7.1).

The peak memory bandwidth and the peak computational performance are often unachievable in practice. For this reason, during evaluation of our implementation we need to focus on the attained performance instead of the maximum performance.

Memory bandwidth depends on the Error Correction Code option (ECC)[39]. It adds a parity byte for every eight data bytes, which reduces available memory size by 12.5%. In addition, an extra byte must be read, so we lose at least 12.5% of the peak memory bandwidth because of the parity byte. This results in a performance difference between the operations with enabled and disabled ECC option.

3.7.1 NVIDIA profiling tools

To evaluate efficiency of the implementation, we use NVIDIA development platform for heterogeneous computing, called Nsight [14]. The two basic tools are NVIDIA Visual profiler and a command-line profiler, known as nvprof. NVIDIA Visual profiler is a graphical tool that displays a timeline of the application. It also includes an automated analysis engine, which identifies possible optimizations. To collect the data from the command-line we use nvprof tool. It provides a broad assortment of available measurements: memory transfer, kernel execution time, events and metrics [9]. In this thesis we use nvprof to extract the information about the performance and the NVIDIA Visual profiler tool to detect possible optimizations or existing defects of the code segments.

The metrics that we execute for the assessment of the performance via Nsight are listed below:

- Compute number of floating point operations
  −−metrics "flop_count_dp, flop_count_dp_add, flop_count_dp_fma, flop_count_dp_mul, flop_dp_efficiency"
- Compute the memory bandwidth
  −−metrics "dram_read_throughput, dram_write_throughput, dram_utilization, ecc_throughput"
- Perform guided analysis with Visual profiler
  −−analysis-metrics
- Gather different information, for instance number of registers, amount of shared memory, block size, grid size or execution time
  −−print-gpu-trace

CUDA Occupancy Calculator [11] is a spreadsheet provided in a CUDA toolkit. It helps to select optimal block and grid sizes that maximize occupancy for the kernel according to the amount of registers and shared memory required by the kernel function.

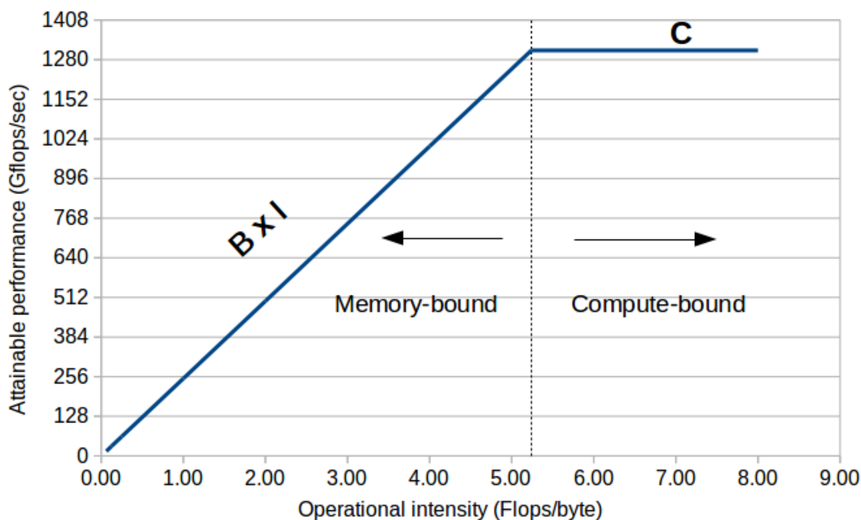

Figure 3.3: Roofline model for Kepler K20X. The dashed line indicates operational intensity for the system. The given kernel is compute-bound if its computational intensity is to the right from the dashed line and it is memory-bound if its computational intensity is to the left from the dashed line.

3.8 A brief introduction to the Roofline model

The Roofline model [53] offers performance guidelines for Multicore Architectures. It defines three types of potential bottlenecks: computation, communication, and locality. Moreover, it determines a realistic upper bound of the performance of the kernel depending on the operational intensity of the kernel function. The operational intensity (Equation (3.2)) is defined as a ratio of the number of operations performed by kernel to the amount of memory transfers carried out during the execution of the kernel.

$$\mathbf{I} = \frac{\mathbf{W}}{\mathbf{Q}}, \tag{3.2}$$

where $\mathbf{W}$ is measured in FLOPS, $\mathbf{Q}$ is measured in bytes, and $\mathbf{I}$ is expressed as FLOPS/byte.

The upper bound of the performance (Equation (3.3)) is defined as the minimum of peak floating-point performance, that is specified by the computer architecture and measured in GFLOPS/s, and the product of bandwidth, obtained from benchmarking, and operational intensity.

Let $\mathbf{C}$ define the peak floating-point performance and $\mathbf{B}$ define the memory bandwidth obtained by the benchmarking, then the attainable performance $\mathbf{P}$ is computed as follows:

$$\mathrm{P} = min \begin{cases} \mathbf{C} \\ \mathbf{B} \times \mathbf{I}. \end{cases} \tag{3.3}$$

The roofline model can be visualized as a 2D graph (see Figure 3.3), where x-axis represents an operational intensity of a kernel and y-axis express attainable Gflops/s. A roofline curve consists of two parts: the horizontal line of peak floating-point performance and the line of unit slope - a bound of maximum floating-point performance obtained by memory bandwidth. Both lines intersect at the point $\mathbf{I} = \mathbf{C}/\mathbf{B}$. The attainable performance can be found by drawing a vertical line through the point on the x-axis, that represents an operational intensity of the kernel. The point where the vertical line hits the Roofline curve defines the attainable performance. For a given kernel, we can find if the kernel is compute-bound or memory-bound by using its operational intensity [40].

$$\text{If} \quad \mathbf{I} \leq \mathbf{C}/\mathbf{B}, \text{then the kernel is memory-bound}$$

$$\text{If} \quad \mathbf{I} \geq \mathbf{C}/\mathbf{B}, \text{then the kernel is compute-bound}$$

3.9 Summary

In the last decades, a lot of developments in the High Performance Computing community have been done: GPUs have been adjusted to perform general-purpose computation and computing systems have started to include different types of processors. In this chapter we described a heterogeneous CPU-GPU programming platform, called CUDA. We introduced the main GPU architecture which we will use for the computation. Moreover, we described a message passing interface, called MPI which we will use for the communication on CPU. We defined performance metrics, profiling tools and a performance guidelines model that are used in this thesis.

Chapter 4

Modelling a cardiac simulator on a hybrid CPU-GPU cluster

In the previous chapters we have presented the cardiac tissue model and the hardware that we will be using for the implementation and simulation. This chapter concentrates on the data structures and computation distribution between CPU and GPU. In order to achieve good performance, we need to be aware of the memory transfer operations, thread-grid configuration and GPU utilization. Memory copying is an expensive operation, which can decrease overall performance of the program [24], especially with a large amount of data. One of the questions of interest here is how we can distribute computation without losing the performance due to unnecessary data copying. To get the maximum performance from the hardware, it is important to have a sufficient amount of data and threads, to keep GPU busy [45]. In this chapter, we also focus on the parallelization of the work at the tissue-level.

4.1 Computation distribution between CPU and GPU

The multiscale cell model consists of two types of computation: cell-level computation and dyad-level computation. At the cell-level we perform computation that involves whole-cell values and interaction between the cells, while at the dyad-level we concentrate on the computation that involves processes inside the dyad and interaction between dyads.

In the CPU implementation [30, 31, 32] the main tissue computation is executed as a time loop. At each time step we loop over all the cells individually and perform computation inside the current cell. The work of each cell consists of a loop over all dyads, the computation inside them and calcium diffusion between dyads after the last iteration. Before the next time step, we perform voltage diffusion (Equation (2.16)) between the cells. The loop that goes through the dyads in a cell contains the most time consuming operations [30][31][32]. Hence, performing these computations on the GPU in parallel can provide an essential improvement to the performance.

In our implementation, the GPU handles the computation at the dyad-level and the interaction between dyads, which means that each CUDA thread performs computation corresponding to the dyad assigned to it. Thus, there is

no need for the innermost loop and all the dyad computations are performed by parallel threads.

Since cell voltage diffusion does not consume a significant amount of time, it is performed on the CPU. All the voltage values of different cells are kept in an array on CPU and are updated after each computation step. This requires the communication between CPU and GPU, which is accomplished by CUDA memory copying operations. After each computation step the voltage values of the cells are copied from device to host and the updated voltage values are copied back to the device for the next time step.

4.2 Basic approach: a single cardiac cell

Initially, we focused on the implementation of a single cardiac cell on GPU. To implement a cardiac cell model, which corresponds to the actual human cardiac cell, we adopt the data structure that was used in [30, 31, 32]. A cell is represented as a structure - a user defined data type in the C programming language. It contains cell-level variables, such as voltage and global calcium concentrations in the different calcium distribution compartments, which are set to their initial values at the initialization step. Recall that a cardiac cell consists of smaller entities - calcium release units. Each of them is represented by its inherent currents and concentrations. The collection of dyads is kept in arrays - a data structure in the C programming language. In our model, each dyad contains 29 elements corresponding to currents and concentrations, and 10 random numbers. Thus, we need to allocate 29 arrays of the length equal to the number of dyads in the cell and one array of the length equal to the number of dyads multiplied by 10. Pointers of the arrays are included in the *struct* for the cell. In addition, the cell contains pointers to *dyadOuterVars* and *dyadInnerVars* structures, consisting of the cell-level information initialized during the computation. *dyadOuterVars* holds the values obtained during the dyadic computation, for example Na-Ca exchanger current in myoplasm. The structure is used to update the main cell values at each time step. *dyadInnerVars* contains data that is used only for the computation. One of the advantages of having these structures is the convenience of passing arguments to the functions, which require a large amount of parameters for the computation. The advantage of having these two structures is even more visible during computation on the device.

Before performing computations on GPU, we have to copy the necessary data from host to device. However, we are not able to copy the cell structure directly, because it contains pointers to the memory locations. Thus, *dyadOuterVars* and *dyadInnerVars* structures, and dyadic information need to be transferred from host to device separately.

Since the dyad-level computations are executed on device, the remaining cell-level computations and updates of the variables are performed on the host. The cell structure is updated by transferring *dyadOuterVars* back from device to host after each time step. As all cell-level variables are updated and initialized on the host, we need to copy the updated *dyadOuterVars* and *dyadInnerVars* from host to device at the beginning of each time step. The arrays with dyad-level information are transferred from host to device at the beginning of the program and transferred back after the last time step. Thus, we keep it

on the GPU during the whole computational process. The pseudo code of the computation for the single cell is shown in Listing 4.1.

However, one cell computation does not utilize the whole power of the GPU. The NVIDIA profiler reports low compute utilization, which means that some of the multiprocessors are idle due to the small amount of data. Moreover, inefficient size of data leads to low memory transfer throughput. Small amounts of data do not enable GPU to fully use host to device bandwidth. Furthermore, the kernel launch overhead cannot be hidden using only one cell. To obtain more parallelism, we need to invoke more threads and perform computation on the larger amount of data. This can be done by extending the single cell per GPU implementation to multiple cells per GPU.

```
Cell initialization on CPU
Memory allocation on GPU
Memory copying from host to device
for int t=0; t<time steps; t++
   Cell computation:
      Compute initialization values for the cell on CPU
      Update dyadOuterVars on CPU
      Update dyadInnerVars on CPU
      Copy dyadOuterVars from host to device
      Copy dyadInnerVars from host to device
      Dyad-level computations on GPU
      Copy dyadOuterVars from device to host
      Update cell values on CPU
   Cell diffusion on CPU
end for
```

Listing 4.1: The pseudo code of a single cell computation. At each time step the values for the main cell are computed, *dyadOuterVars* and *dyadInnerVars* structures are updated and copied to device memory. The dyadic computation is performed on device and the resulting *dyadOuterVars* is copied back to the host, where the main cell values are updated. In case of multiple cells, voltage diffusion between them is executed on the host before the next time step.

4.3 Multiple cells

4.3.1 Extending one cell implementation to multiple cells

Even though simulations involving only one cell are easier regarding the implementation, it is not enough to reproduce the real world problem. To perform a realistic simulation of a human heart, simulation of 2-3 billions cardiac cells is needed. Moreover, the single cell computation does not utilize the full power of the GPU. To attain more parallelism, we take our implementation one step further and implement the multiple cells approach. Therefore, we need to perform some changes in the data structures.

First of all, updating variables on the CPU and copying the structures before and after the main computation at each time step is expensive. Our approach is to transfer all the data needed for the computation from host

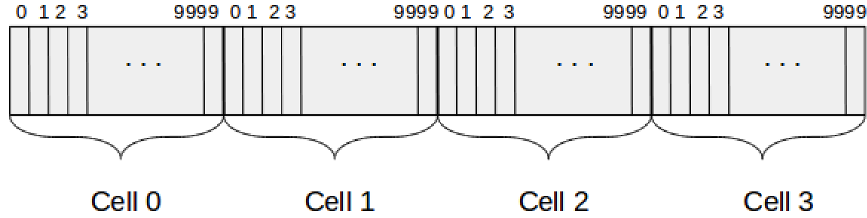

Figure 4.1: An example of the array of calcium release units for one specific element in the multiple cell model. This example shows four cells. The rectangles inside the cell represent calcium release units with their IDs above.

to device at the beginning of the program execution and transfer it back to the host at the end of the program. In this way, all the data needed for the computation remains on GPU for the complete execution time. Thus, we no longer need to update the main cell structure initialized on CPU at every time step.

To have control over the data that needs to be updated at every time step, we create third structure a *dyadGPUVars*. It contains all the cell values that previously were holden by the CPU cell structure and updated by the *dyadOuterVars* structure. Thus, instead of updating the main cell values, we update the *dyadGPUVars* values that remain on the GPU for the computation.

Finally, we create three arrays of of the same size as a number of cells, where each of them represents one type of structures *dyadOuterVars*, *dyadInnerVars* or *dyadGPUVars* inherent to a specific cell. Dyadic currents and concentrations corresponding to different cells are kept in the same array. There is a 10000 (or number of dyads) units interval that belongs to each cell in the array, which corresponds to the specific element of a calcium release unit. Thus, we have 29 arrays of size $10000 \times \textit{number of cells}$ that contain dyad-level values. Figure 4.1 shows an example of the data structure of dyadic information.

As mentioned before, a human heart contains billions of cells. Unfortunately, device memory is limited, so we can simulate only a finite number of cells on the GPU. In our model one cell contains 10000 calcium release units and each of them holds data of 29 values, which is

$$10000 \times 29 \times 8\text{byte} = 2.32\ \text{MB}$$

of double-precision data. Moreover, each calcium release unit requires 10 random numbers to perform binomial distribution: 2 numbers for L-type channel computation and 8 for the RyR channel computation, which gives

$$10000 \times (8 + 2) \times 8\text{byte} = 0.8\text{MB}.$$

In total, one cell requires about 3 MB of data. According to nvidia-smi [15], we have 4742 MiB of memory available on Kepler K20 without ECC, which gives us the maximum of 1580 cells. However, we are limited by the 1550 cells due to the CUDA random number generator (cuRAND), which is not able to perform memory allocation for more than 1550 cells. On the other hand, on Abel we

are able to simulate a larger number of cells. Since the NVIDIA Kepler K20X has 6 GB of device memory, we should be able to perform the computation of $6/0.003 = 2000$ cells. However, we have empirically established that we are able to simulate at most 1872 cells on one K20X GPU.

4.3.2 Structure of the implementation

The structure of multiple cells allows to transfer all the cells to the GPU at once, so that we do not need to iterate over the cells. As a result, cell-level computations and updates of cell-level variables that were required at the beginning and at the end of each time step for the single cell implementation (Listing 4.1) are directly performed on the device in parallel. Hence, the code contains only one loop - the main time loop.

In our implementation we launch all CUDA kernels from the host. After each kernel invocation, the control is given back to the host. In the pseudo code (Listing 4.2) one can see the relation between host functions and CUDA kernel functions. CUDA kernels are indicated with the specific syntax <<<>>> and are called from the corresponding host function.

As shown in the pseudo code, the CUDA initialization and memory transfer from host to device are performed before the time loop. Some of the cell values do not need to be computed at every time step. Therefore we initialize those values on device before the first time step. To avoid unnecessary memory transfers between host and device, the data needed for the computation remains on the device for the lifetime of the program. Due to the intercellular voltage diffusion, the cell voltage needs to be copied between host and device at every time step. At the beginning of each time step, we initialize and update the cell data, generate random numbers and copy the voltage values. Furthermore, we perform the dyad-level computation which is divided into five parts : (1) *L-type channel simulation*, (2) *RyR probability calculation*, (3) *RyR opening computation*, (4) *Ca concentration computation* and (5) *Dyad diffusion*. Each of these is executed on the device by launching one or several kernel functions. L-type function computes opening for the 15 L-type calcium channels. RyR opening addresses random state transitions for the 100 RyRs per dyad using probabilities computed in the RyR probability calculation function. The Ca concentration function computes local calcium concentrations in the dyads. Block reduction kernel in this function computes the whole-cell values of calcium currents. The dyad diffusion function computes the diffusion of intracellular calcium concentrations between the dyads using equations (2.4), (2.8) and (2.10) that were described in Section 2.2. The reduction function here computes the whole-cell value of the calcium concentrations in the dyad compartments. Before the next time step the cell diffusion function performs diffusion of intercellular voltage concentrations between the cells using Equation (2.16). Initialization kernel1, Initialization kernel2 and Final update compute Na and K concentrations and currents. This program structure is used further in our implementation.

```
Memory Copying from host to device
Compute first initialization
   Compute first initialization<<<blocks,threads>>
for(int t=0; t<time steps; t++){
    Cell computation:
        Compute initialization for the time step:
            Generate random numbers
            Copy voltage from host to device
            Initialization kernel1<<<blocks,threads>>>
            Initialization kernel2<<<blocks,threads>>>
        L-type channel simulation:
            L-type simulation<<<blocks,threads>>>
        RyR probability calculation:
            RyR probability<<<blocks,threads>>>
        RyR opening computation:
            RyR opening<<<blocks,threads>>>
        Ca concentration computation:
            Ca concentration<<<blocks,threads>>>
            Block Reduction<<<blocks,threads>>>
        Dyad diffusion:
            Dyad diffusion<<<blocks,threads>>>
            Reduction<<<blocks,threads>>>
            Block Reduction<<<blocks,threads>>>
        Final update for the time step:
             Final update <<<blocks,threads>>>
             Copy voltage from device to host
    Cell diffusion
    }
```

Listing 4.2: The pseudo code of the program. CUDA kernels are indicated with the specific syntax <<<>>> and are launched from the functions implemented on the host. The main computation consists of five parts : (1) *L-type channel simulation*, (2) *RyR probability calculation*, (3) *RyR opening computation*, (4) *Ca concentration computation* and (5) *Dyad diffusion*. Each of them is executed as one or more several CUDA kernel functions. The cell-level computations and updates are also performed on device, so that the data remains on device for the life time of the program.

4.4 Thread configuration

The configuration of threads and correct assignment of the data are important for the full utilization of the GPU. In our implementation the most of the computation on the dyad-level uses one dimensional (1D) data, which is the reason why we arrange our threads as 1D blocks assigned to a 1D grid.

By an empirical search, we have found that 128 threads per block is the optimal amount to perform computation on one cell with 10000 calcium release units. This requires 79 blocks of threads for a single cell choosing one thread to perform computation over one dyad. The size of the grid or number of blocks is computed by dividing the number of dyads in a cell by the number of threads in a block and rounding the result up to the nearest higher integer value (Equation (4.1)).

$$\text{number of blocks} = \left\lceil \frac{\text{number of dyads}}{\text{number of threads}} \right\rceil = \left\lceil \frac{10000}{128} \right\rceil = \lceil 78.12 \rceil = 79. \quad (4.1)$$

Most of the dyad-level computations are independent of each other, so any calculation performed on calcium release units of one cell cannot influence results of another cell. This means that computations performed by different threads do not overlap. Thus, threads that perform dyadic computations on the different cells can be placed in the same block. To find a number of blocks for multiple cell implementation, it is enough to compute the total number of calcium release units among all the cells and find the correct size of the grid in the same way as we did for a single cell. Hence, for multiple cells the number of blocks in a grid becomes:

$$\text{number of blocks} = \left\lceil \frac{\text{number of dyads in a cell} \times \text{number of cells}}{\text{number of threads}} \right\rceil \quad (4.2)$$

However, execution of the functions that involve more complex computations, such as diffusion, or usage of the shared memory, such as reduction, cannot have threads corresponding to the different cells in the same block, because it would cause data overlapping between cells. A specific mapping from the threads to the array of calcium release units, which does not allow threads representing dyads from distinct cells to be in the same block, is required. Thus, we choose to keep a single cell structure of thread blocks, but increase the grid according to the number of cells: we assign 79 blocks of threads to each cell. Because we have more threads than dyads in a cell, the last block in each cell contains some threads that are idle during the computation, but are involved in the operations on shared memory. Therefore, the dyad information from different cells does not overlap in the shared memory and we have a strong separation between cells.

To map a specific thread to a corresponding dyad in the function that requires a strong separation between the cells, we define a mapping called cell separation mapping. Figure 4.2 shows the core of the mapping, where the cell data is assigned only to a part of the threads. The mapping is computed as

$$\text{MyDyad} = \text{Ndyads} \times \text{myCell} + \text{blockDim} \times \text{localBlock} + \text{threadIdx}, \quad (4.3)$$

Ndyads defines the number of dyads in a cell, *blockDim* is the number of threads in a block and *localBlock* is the block ID in the cell (in this case it ranges from 0 to 78).

The calcium diffusion between the dyads requires a multidimensional grid, where dimensions depend on the selected implementation. Threads can be arranged into two dimensional (2D) blocks and assigned to a 2D grid or they can be arranged into 3D blocks and assigned to a 3D grid. The diffusion computation requires separation between threads corresponding to different cells. Thus, we define the optimal grid for one cell and then we increase it along the x-direction according to the number of cells. The optimal number of threads for the diffusion is presented in Section 5.3.

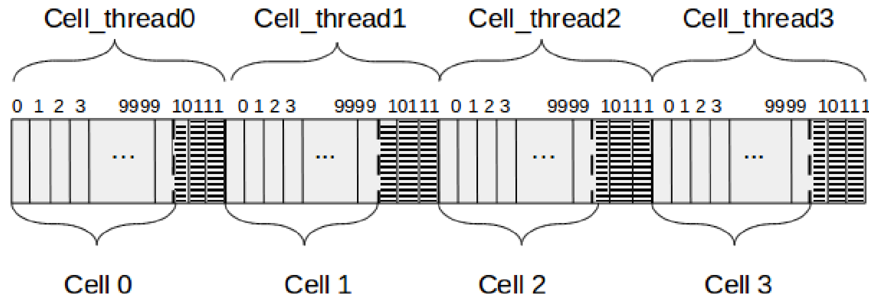

Figure 4.2: An example of a cell separation mapping from threads to original data for four cells. The rectangles represent threads in a cell defined via grid composition. Cell_thread ID represents the threads corresponding to a cell, while Cell ID represents original dyads in a cell. The thick vertical line defines the boundary from threads corresponding to two different cells. The dashed line defines the end of cell data. Threads in a striped area are idle.

4.5 Tissue-Level parallelization

The detailed tissue-scale cardiac simulation of the human heart requires a large amount of cells to be involved in the computation. However, as we have seen before, only a limited amount of cells, which form less than 1% of the cardiac tissue, can be simulated on one GPU due to its limited memory. Therefore, we extend our implementation from one GPU to multiple GPUs by enabling the usage of supercomputers, which allow us to perform large scale simulations on multiple GPUs.

The usage of multiple GPUs requires a particular data distribution among them. The cardiac ventricle tissue is represented as a 3D Cartesian grid of cardiac cells, which needs to be distributed among different GPUs. Since a supercomputer consists of many compute nodes, the data is distributed in two steps. Firstly, the grid is decomposed into smaller 3D domains which are assigned to the compute nodes of the supercomputer. To utilize every compute node equally, each domain contains a uniform number of cells. Secondly, we need to distribute the data among GPUs that belongs to the same compute node. Each compute node subdivides its domain into the smaller sub-domains of equal size. The number of sub-domains depends on the number of available GPUs on the node. Example of 3D Cartesian grid decomposition is shown in Figure 4.3.

In our implementation each sub-domain is assigned to one GPU by one MPI process. Since each compute node contains only two GPUs in the hardware that we use, the procedure to assign one GPU to one MPI process is quite simple. Each node invokes two MPI processes, the device with ID 0 is assigned to the MPI process with an even rank, and the device with ID 1 is assigned to the MPI process with an odd rank. Computation on each GPU are performed independently of each other and all the necessary communication between the cells is performed via the CPU.

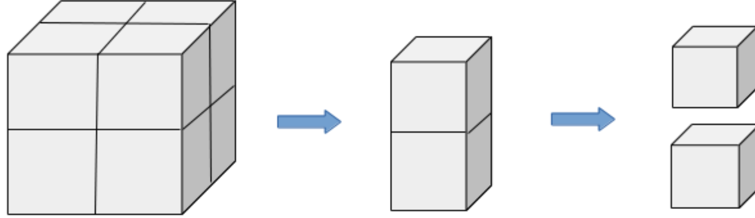

Figure 4.3: An example of the Tissue-Level parallelization having four compute nodes with two GPUs on each of them. The large rectangle represents a 3D cardiac ventricle tissue. The first arrow shows a part of the tissue assigned to one compute node. The second arrow shows how the domain is divided into two smaller sub-domains corresponding to two different GPUs on the same compute node.

The cells must be equally distributed among the GPUs, otherwise some GPUs would not be fully utilized which would slow down the whole simulation. Moreover, due to the required communication between the different MPI processes, some of the processes would need to wait for others to finish their computation to be able to exchange the data.

4.5.1 Communication

The only communication needed between cells is an exchange of voltage values required by the reaction-diffusion computation for Equation (2.16). To perform diffusion using a finite difference approach, each cell needs to have access to the voltage values corresponding to its neighbours in all three spatial dimensions. As mentioned earlier at each time step the computed whole-cell voltage values are transferred from device to host and the diffusion computation is performed on the CPU. However, since the global domain is decomposed into the smaller sub-domains that are assigned to the different MPI-processes, the diffusion computation is distributed between different processes. This requires communication between MPI-processes to provide access to the data from the neighbouring cells.

The communication method we use in this thesis is adopted from [32]. We introduce ghost cells at the boundary points that hold the values of the voltage that corresponds to the neighbour cells located in the sub-domain assigned to the neigbouring process. This makes it possible to exchange voltage values along the faces of the sub-domains. The data interchange is done via MPI messages [36] between the host CPUs. At each time step after the diffusion computation, the ghost cells are updated with new voltage values.

MPI processes are arranged as a 3D Cartesian grid by using the *MPI_ Cart_ create* function. For data exchange along the dimensions of the topology, the *MPI_ Cart_ shift* is used to compute the rank of the source and destination

processes. Moreover, each MPI process partitions the data and constructs a structure that contains all the relative information for the communication, such as buffers for sending and receiving the data, and information about the grid of cells that each process owns. In addition, each partition holds a structure containing MPI information, such as the number of processes in each dimension and the rank of neighbours.

Communication is performed after a process has completed its part of the voltage diffusion computation. So that all neighbours get the updated data. First, the process or partition assigns the voltage values that are required by its neighbour consecutively to a buffer and sends it using the non-blocking *MPI_Isend*. The non-blocking receive operation *MPI_Irecv* is performed in order to receive data from the neighbours. If the neighbour do not exists, the ghost cells are updated using only values from the same partition. After all given MPI requests are completed, the received data is unpacked and allocated to the ghost cells in the voltage array.

4.6 Evaluation of the correctness of the implementation

To evaluate the correctness of our implementation, we compare the obtained results of voltage, calcium concentrations in various spaces and calcium currents with the obtained results from previously implemented human heart simulators [30, 31, 32]. Due to the stochastic modelling of channel opening, the same random number generator and access pattern of random numbers is used during the evaluation of the correctness.

4.7 Summary

The main computation is distributed between CPU and GPU, such that the most expensive dyadic computations are performed on the GPU, while the voltage diffusion is performed on the CPU. A single cell on the GPU does not utilize the whole GPU capacity, as a consequence a multiple cell implementation was introduced. Further, due to the large number of cells, the implementation was extended to multiple GPUs. The required data distribution and communication is handled by MPI processes.

Chapter 5

Implementation and optimization of the dyad-level computations on GPU

The dyad-level computation consists of several computationally heavy and time-consuming functions: L-type channel simulation, the calculation of RyR channels opening, Ca concentration computation and the diffusion of intracellular calcium concentration. To increase the performance of the simulator, we have chosen to implement the dyad-level computation in parallel on the GPU. In addition to the previously mentioned functions, we will define reduction function which has not been implemented in the previous implementations [30, 31, 32]. Reduction operation constitutes a major part of our implementation. In this chapter, we will broadly present the CUDA implementations of diffusion and reduction functions, because of the significant difference from the CPU implementations. Functions with minor changes in the implementation will be described in terms of optimizations that we have used in order to increase the performance. Since our parallel implementation of the cell-level computations, such as initialization and final update, on the GPU is not different from the CPU implementation, they are omitted in this chapter.

5.1 Common optimization techniques

Random number generation

Sampling from binomial distribution requires drawing a randomly generated number from a uniform distribution in the interval between 0 and 1. Due to eight possible transitions in the RyR channels and two possible transitions in the L-type calcium channels, ten random numbers need to be generated for one dyad. Since in our model each cell contains 10000 calcium release units, 100000 random numbers need to be generated for one cell at each time step. For this task, we utilize NVIDIA CUDA Random Number Generation library (cuRAND) [10], which provides a high performance GPU-accelerated random number generation. The random number generator covers all the internal capacity necessary to generate a sequence of pseudo-random numbers. cuRAND provides two interfaces: host generation and device generation. The former

generates and stores all the random numbers on the CPU. The latter is called on the host, but all the random numbers are generated and stored on the device. In our application, we use cuRAND host interface and generate an array of uniformly distributed 10000 times *number of cells* random numbers. Each thread reads the corresponding random values from the global memory.

Shared memory usage

CUDA supports declarations of single shared memory variables and 1D, 2D and 3D shared memory arrays. Threads load the corresponding variable into the shared memory before performing computation on it. If there is no bank conflict, the data is loaded simultaneously.

Shared memory is one of the most frequent optimization techniques applied in our implementation. First of all, we can use it to get faster access to the global data that is reused several times by the thread. Secondly, shared memory can serve as a communication between the threads in a block, for instance in the diffusion computation. Finally, it can be used to store a global common variable for all the threads in a block.

Read-only cache

In this application, a significant amount of data is used as a read-only for the duration of the kernel function. This data can be stored in a read-only cache which is considerably faster to access than the global memory. The compiler can automatically access the stored data via read-only cache and minimize redundant access to global memory. The pointers used for loading the data should be marked with *const* and *__restrict__* annotations. In this way, read-only cache can benefit the performance of bandwidth-limited kernels.

Coalesced memory access

Global memory access pattern is essential for the performance of a CUDA application. To minimize DRAM bandwidth requirement the device coalesces reads and stores into global memory, such that the number of transactions would be minimized. During the memory load, the hardware identifies whether the threads access memory that is consecutively located. In order to carry out the best performance, all threads in a warp should execute the same instruction and access consecutive DRAM locations. The consecutive locations then are combined into a single entry. Memory access is coalesced if it is sequential and aligned.

To have coalesced memory access to random numbers, we need to take into consideration the structure of the grid. All computations involving random numbers are performed on the dyad-level and there is no possible overlap between the data belonging to the distinct cells. Therefore, the grid of threads is formed of consecutive threads. This allows us to store random numbers sequentially so that all threads in a warp have a coalesced access.

Improved power function

The power function is one of the most expensive operations. In cases where the exponent is known beforehand and it is not a large number, substituting power function with the direct multiplication can give a significant speed up to the program. However, performing binomial distribution sampling, the value of the exponent ranges from 0 to 100, which makes it cumbersome to use a simple direct multiplication. Recall that power can be expressed as a product of other powers that have the same base. For instance, $a^k = a^{m+n} = (a^m)(a^n)$ for $k = m + n$. To improve the implementation it is beneficial to adopt bit-wise operations. First, the exponent is bit-wise compared with the smallest possible exponent, which in this case is one. The binary add operator checks if the bit exists in both operands. If this is the case, the power becomes a part of the combination of power functions. The binary left shift is performed on the smallest exponent in order to compare the next bit of the exponent of interest, and then the next possible power, which is the previous power squared, is computed. To compute the power of 100, it is enough to perform this computation seven times, because $a^{100} = a^4 \times a^{32} \times a^{64}$. The code is shown in Listing 5.1.

```
int bit = 1;
double result = 1.0;
int exponent = n;
double base = 1 - p;
for(int j = 0;j < 7;j++ ){
  char tmp = bit&exponent;
  if(tmp)
    result*=base;
  bit = bit<<1;
  base*=base;
}
```

Listing 5.1: The code for the power computation using bit-wise operators

Double-precision floating point division

The device that we use is of CUDA compute capability 3.5 which means that it supports both single-precision and double-precision floating point operations. However, the same arithmetic operation can produce slightly different results due to the greater accuracy of the double-precision floating values and rounding issues. CUDA supports all arithmetic operations: addition, multiplication, subtraction and division. The last mentioned produces the floating point value closest to the correct mathematical result. GPU hardware does not have support for floating point division at the instruction level. Instead, it uses numerically implemented division based on other basic instructions. As an alternative for the standard division and other standard operations, CUDA provides intrinsic functions, which are only accessible from the device. Intrinsic functions use fewer instructions which makes them faster, but they are less precise than standard functions.

Performing experiments and assessing the outcomes of the NVIDIA profiler, we have noticed that the count of double-precision floating-point division

operations includes also single-precision operations. According to the data from NVIDIA profiler, one division is counted as one multiplication and five fused multiply-add operations, thus the device performs 11 double-precision floating point operations. In addition, the device executes one single precision fused multiply-add, which is 2 single precision floating point operations. Therefore, to keep double precision and speed up the computation, we interchange division by multiplication where it is possible. However, some of the denominators contain values that are not known at compile time. Thus, division operation cannot be interchanged by multiplication by precomputed reciprocal of the denominator. For instance, $x = y/2$ can be changed to $x = y \times 0.5$ while to change $x = y/(k1 + k2 \times k4 \times k6)$ is not that simple. As a consequence, we introduce an intrinsic reciprocal rounded to the nearest value in our implementation. The intrinsic is used by calling *__drcp_rn(double x)* function. Using CUDA intrinsic $x = y/(k1 + k2 \times sk4 \times k6)$ can be changed to $x = y \times __drcp_rn(k1 + k2 \times k4 \times k6)$, which speeds up the computation.

Redundant calculations

There is a considerable amount of variables involved in the dyadic computations. Several of them vary from dyad to dyad, while others remain the same for all dyads. To improve performance by avoiding computation of the same variable for each of 10000 dyads, we pre-compute all the common variables before each time step. Pre-computed variables are stored as whole-cell values into one of the structures depending on their purpose. Some of the variables remain the same for the whole execution time of the program. These variables are defined as constants and are known at the compiling time.

Decomposition of the computation

Several of the kernel functions, for instance, RyR probability calculation kernel and Ca concentration kernel, are limited by the number of registers. This restricts the number of concurrently executing threads. To improve the performance and increase occupancy, we need to lower the number of registers. This can be done by decomposing the computation inside the CUDA kernel function into the smaller *__device__* functions, which are called and executed on the device.

Merging the execution kernels

The main computation at the dyad-level is divided between several distinct execution kernels that perform computations on GPU. The amount of computation performed by each kernel varies from one function to another. Having one kernel that executes a larger amount of computation instead of several smaller kernels can increase performance significantly. First of all, since the most time - consuming operation is loading data from the global memory, separate kernels might need to perform several redundant memory accesses. For the second, having a lot of small kernels might introduce kernel launch overhead, which could be an important bottleneck for the performance of the program. Thus, we can improve performance by merging two or more small kernels together. The kernel merging was used to improve L-type calcium channel simulation, by merging L-type probability and L-type opening kernels.

5.2 Implementation of the reduction kernel

Majority of the computations corresponding to the estimation of the calcium concentration are executed on the dyad-level. However, to represent calcium level in a cell, the whole-cell value is needed. The calcium concentrations of the whole-cell are computed using equations (2.11)-(2.15). The whole-cell representations of calcium currents are computed in the same way. Equations (2.11)-(2.15) are known as reduction operations, which pass over $\mathcal{O}(N)$ input elements and generate $\mathcal{O}(1)$ results. Finding the best way to sum-up variables across the calcium release units and assign them to the corresponding cell is one of the most challenging problems we encountered working with our implementation.

NVIDIA has presented several ways of performing reductions. Since this part of the code has been improved many times during the programming process, we were able to try several approaches. Further in this paragraph, we will present a couple of possible implementations of reduction function.

5.2.1 Basic parallel reduction

Tree-based approach within each thread block is the mostly used solution for reduction computation. First, each thread loads the corresponding value to the shared memory. Then, the number of active threads is halved by adding the values from the upper half of the shared memory to the lower half. We continue to reduce the number of active threads until only one value within a block is left. Note that because of this step, we are required to have the block size of power of two. To accumulate the final value we launch kernel one more time, but this time we use only one block of threads.

In our implementation, we utilize several optimization techniques presented in [25]. Since the size of blocks is fixed in our implementation, we are able to unroll the loops. To decrease the number of registers, we cache the data into the read-only data cache. One of the main optimizations is performing multiple adds per thread. We let each thread load and add multiple elements into the shared memory before we start to perform the actual reduction algorithm. The pseudo code is shown in Listing 5.2.

5.2.2 Shuffle warp reduce

Kepler architecture provides a shuffle instruction [9, 39], which allows to interchange data among threads in a warp without going through shared or global memory. The benefits of the shuffle instruction are increased bandwidth and decreased latency.

The algorithm that we present here is adopted from [33]. First, we perform reduction within warps. At each iteration a thread calls *_shfl_down(var, offset)* instruction. It calculates source thread index within its warp, known as *lane ID*, by adding offset to the lane ID of the calling thread. Then it returns value held by the source thread. The returned value is added to the variable that the caller is holding. We continue until *lane 0* has the total reduced value. To perform reduction within blocks, each warp writes its result to the shared memory. Then the first warp performs reduction on the shared data. Finally, *thread 0* of each block contains the partially reduced variable. This is a two-pass reduction, which means that we need to execute the kernel one more time in order to perform the final reduction across the blocks. The pseudo code is shown in Listing 5.3.

```
__shared__ double input_s [BLOCK_WIDTH];
input_s[tid] = 0.0;
size_t i = getDyad(n,num_cells);
unsigned int tid = threadIdx.x;
if(i < n * (my_cell + 1))
   input_s[tid] += input_gpu[i];
//array size and grid size for one cell must be divisible 2
i = i + blockDim.x * (gridDim.x / num_cells);
if(i < n * (my_cell + 1))
   input_s[tid] += input_gpu[i];
__syncthreads();
if (blockSize >= 128)
    if (tid < 64)
        input_s[tid] +=input_s[tid+64];
__syncthreads();
  if (tid < 32){
     volatile double * input_w=input_s;
       if (blockSize >= 64)
               input_w[tid] += input_s[tid + 32];
       if (blockSize >= 32)
              input_w[tid] += input_s[tid + 16];
       if (blockSize >= 16)
              input_w[tid] += input_s[tid + 8];
       if (blockSize >= 8)
              input_w[tid] += input_s[tid + 4];
       if (blockSize >= 4)
              input_w[tid] += input_s[tid + 2];
       if (blockSize >= 2)
              input_w[tid] += input_s[tid + 1];
       if (tid == 0)
           output_tmp[blockIdx.x] = input_w[0];
   }
```

Listing 5.2: The basic parallel reduction implementation. A block consists of 128 threads. The number of threads needed for the one cell computation is halved at each step, such that at the beginning of the execution each thread adds two input values to the shared memory. Note that n represents the number of elements that need to be reduced per cell.

```
unsigned int tid = threadIdx.x;
int i = getDyad(n,num_cells);
__shared__ double input_s [num_part_sum];
double sum=0.0;
if(i < n * (my_cell + 1))
   sum1 += input_gpu[i];
i = i + blockDim.x * (gridDim.x / num_cells);
if(i < n * (my_cell + 1))
   sum1 += input_gpu[i];
int lane = threadIdx.x % warpSize;
int wid = threadIdx.x / warpSize;
for (int offset = warpSize/2; offset > 0; offset /= 2)
     sum += __shfl_down(sum, offset);
if(lane == 0)
   input_s[wid] = sum;
__syncthreads();
sum = (threadIdx.x < blockDim.x / warpSize) ? input_s[lane] : 0;
if (wid==0){
    for (int offset = warpSize/2; offset > 0; offset /= 2){
         sum += __shfl_down(sum, offset);
  }
}
if (tid == 0)
    output_tmp[blockIdx.x] = sum;
```

Listing 5.3: The shuffle warp reduce implementation. At the beginning of the execution, each thread adds two input values to the shared memory. Note that n represents the number of elements that need to be reduced per cell.

5.2.3 CUB library

CUB [16] is a library that provides the implementation with diversity of algorithmic strategies of the state-of-the-art algorithms in parallel for arbitrary data types and widths of parallelism. The library contains *BlockReduce* class that supports several functions which perform parallel reduction of items partitioned across the CUDA threads. We select *Reduce* function where each thread contributes with one element and computes a block-wide reduction for *thread 0* using a binary combining operator. It is possible to optimize the reduction by selecting an algorithm between *BLOCK_ REDUCE_ RAKING _COMMUTATIVE_ ONLY* and default *BLOCK_ REDUCE_ WARP_ REDUCTIONS*. Further in our implementation, we use the default algorithm.

Another possibility of implementation is to let each thread contribute with two elements and use only half of the grid size for the computation. In other words, each thread holds two values from different dyads before the actual reduction operation. The pseudo code is shown in Listing 5.4. Notice that this is a two-pass reduction, thus we need to perform the reduction on partially reduced data as well. The second kernel launch uses the one-element-per-thread solution described above due to the small amount of data. This approach is called CUB list.

```
int my_cell = getCell(num_cells);
int i = getDyad(num_of_dyads,num_cells);
double elements [2];
element [1] = 0;
typedef cub::BlockReduce <double,BLOCK_WIDTH_R > BlockReduceT;
__shared__ typename BlockReduceT::TempStorage temp_storage;
element[0] = input_gpu[i];
i = i + blockDim.x * (gridDim.x / num_cells);
if(i<n*(my_cell+1))
   elements [1] = input_gpu[i];
result = BlockReduceT(temp_storage).Reduce(elements, cub::Sum());
if(threadIdx.x == 0)
   outdata[blockIdx.x] = result;
```

Listing 5.4: The implementation of execution kernel that uses CUB list solution. First, we initialize the two values corresponding to the dyads per thread, then we perform block reduction operation using CUB library. Note that n represents the number of elements that need to be reduced per cell.

5.3 Stencil computation on the dyad level

Stencil computation is a fixed computational pattern on an n-dimensional grid, where each point in the grid is updated iteratively as a function of itself and its neighbouring points. This pattern is used to write finite difference approximations of derivatives at the grid points. To solve the differential equations (2.4), (2.8) and (2.10) in our application, we use the finite difference method which is expressed as a 7-point 3D stencil. A 7-point stencil means that the value of a point is computed as a combination of its own value and values of its six neighbours at the previous time step. A 3D stencil means that the computation is moving along the x, y and z spatial dimensions. Our main focus here is to find a method to accelerate the stencil computation on GPU.

5.3.1 Baseline implementation

The baseline implementation of the stencil is performed by letting each thread handle one separate grid point. Threads are organized as 3D blocks and assigned to a 3D grid, where cells are arranged in the x-direction. To keep track of the previous time step and updated values, we allocate two buffers. One buffer is allocated for reading values from previous time step and one for writing the results of the computation. At the end of each time step buffers are swapped. In our implementation we let one kernel function to execute all three diffusion operations due to the possible reuse of data and kernel launch overhead.

5.3.2 Two-kernel implementation

To improve our baseline implementation, we split the computation into two parts: (1) computation along the y-z plane on the left ($x = 0$) and right ($x = Nx-1$) boundaries, and (2) computation of inner points and the boundary points for $y = 0$, $y = Ny - 1$, $z = 0$ and $z = Nz - 1$. We implement two CUDA kernel functions to handle this computation. Both of them take care of the boundary conditions by introducing conditional statements, which create divergent branches. Figure 5.5 shows the pseudo code of (1), and Figure 5.6 shows the pseudo code of (2).

```
int y = blockIdx.y*blockDim.y + threadIdx.y;
int z = blockIdx.z*blockDim.z + threadIdx.z;
int my_cell=blockIdx.x/num_blocks_per_cell;
int x = threadIdx.x;

//check that treads are inside the computational region
if y < Ny_diff and z < Nz_diff
    c = y*Nx_diff + z*Nx_diff*Ny_diff + (Nx_diff - 1)*x;
    n = c - Nx_diff;
    s = c + Nx_diff;
    if Ny_diff == 1
       n = s = c;
    else if y == 0
       n = s;
    else if y == Ny_diff-1
       s = n;
    b = c - Nx_diff*Ny_diff;
    t = c + Nx_diff*Ny_diff;
    if Nz_diff == 1
       b = t = c;
    else if z == 0
       b = t;
    else if z == Nz_diff - 1
       t = b;
    if x == 0
       p = c + 1;
    else
       p = c - 1;
    n += Ndyads*my_cell;
    c += Ndyads*my_cell;
    s += Ndyads*my_cell;
    b += Ndyads*my_cell;
    t += Ndyads*my_cell;
    p += Ndyads*my_cell;

    new_diff[c] = old_diff[c] + 2*(old_diff[p] - old_diff[c])*r1
         + (old_diff[n] + old_diff[s]-2*old_diff[c])*r2 + 2*(
        old_diff[b] + old_diff[t] -2*old_diff[c])*r3;

endif
```

Listing 5.5: Pseudo code for diffusion computation along the y-z plane on the $x = 0$ and $x = Nx - 1$. First each thread computes its global ID in y and z directions, then it checks if it belongs to the region of the computation. If so, each thread computes its and its neighbouring indices, which will be used to read the data from 1D array. Before the computation, each thread checks boundary conditions. Finally, threads perform the diffusion computation.

```
int x=blockDim.x*(blockIdx.x%num_blocks_per_cell)+threadIdx.x;

if x>0 and x<Nx_diff-1 and z<Nz_diff and y<Ny_diff
   c = y*Nx_diff + z*Nx_diff*Ny_diff + x;
   n = c - Nx_diff;
   s = c + Nx_diff;
   if Ny_diff == 1
      n = s = c;
   else if y == 0
      n = s;
   else if y == Ny_diff - 1
      s = n;
   b = c - Nx_diff * Ny_diff;
   t = c + Nx_diff * Ny_diff;
   if Nz_diff == 1
      b = t = c;
   else if z == 0
      b = t;
   else if z == Nz_diff - 1
      t = b;
   n += Ndyads*my_cell;
   c += Ndyads*my_cell;
   s += Ndyads*my_cell;
   b += Ndyads*my_cell;
   t += Ndyads*my_cell;

   new_diff[c] = old_diff[c] + (old_diff[c-1] + old_diff[c+1] -
       2*old_diff[c])*r1 + (old_diff[n] + old_diff[s] - 2*
       old_diff[c])*r2 + (old_diff[b] + old_diff[t] - 2*
       old_diff[c])*r3;
endif
```

Listing 5.6: Pseudo code for diffusion on inner points and the $y = 0$, $y = Ny-1$,$z = 0$,$z = Nz-1$ boundary points. First each thread computes its global ID in all three dimensions, then it checks if it belongs to the region of the computation. If so, each thread computes its and its neighbouring indices, which will be used to read the data from 1D array. Before the computation, each thread checks boundary conditions. Finally, threads perform the diffusion computation.

Kernel (1) and (2) are independent of each other, so it is not important in which order we invoke them. For kernel (1) we use the block size of $2 \times 4 \times 4$. Thread with local $ID = 0$ computes values on $x = 0$ boundary and thread with local $ID = 1$ computes values on $x = Nx-1$ boundary. For kernel (2) an optimal block size which allows to compute on internal grid points is $32 \times 4 \times 4$. The block sizes are dependent on the arrangement of the dyads.

5.3.3 Two-dimensional implementation

Two-dimensional diffusion implementation method was adopted from [37] and [52]. The method was improved and adjusted to fit our model. The main idea is to organize threads into 2D blocks and use a loop to obtain the coordinate in the third dimension. Hence, the grid is divided into a 2D plane and each point is represented as a point in the x and y-direction. Each thread operates on several grid points which are placed $Nx \times Ny$ distance apart, assuming that z is the slowest varying dimension. In our implementation we use blocks of 32×4 threads. This size was selected among all valid configurations by empirical search.

Optimization techniques

Below we describe several optimization techniques that have been used to improve the two-dimensional implementation. The pseudo code is shown in Listing 5.7.

- **Usage of registers**

  Every time a thread iterates over z-direction, it reads a current value and its neighbours along z-direction from the global memory. Since the point (i, j, k) becomes (i, j, k-1) and the point (i, j, k+1) become (i, j, k) at the next iteration, we can cache the points (i, j, k-1) and (i, j, k) into the registers and read only (i, j, k+1) from the global memory. In this case we need only one global memory access at every iteration.

- **Blocking with read-only cache**

  Each thread reads 7 points of the data in order to produce one output value. Since the buffer containing input grid is used only for read operations, we can get faster access by using read-only cache instead of accessing the global memory at each time step.

- **Handling constants**

  To avoid unnecessary computations, whenever it is possible we compute constants beforehand and define them with *#define* macro.

- **Handling x and y boundary conditions**

  The loop over z-direction contains a significant number of if tests that check boundary conditions for x-y plane, which causes warp divergence. We create four new variables: *Nx_diff1*, *Nx_diff2*, *x_diff1*, *x_diff2*, which are used to compute the corresponding four neighbouring indices: $y-1$, $y+1$, $x-1$ and $x+1$ respectively. This allows us to check the boundary conditions and update the new variables according to the boundary before the for-loop. As a consequence, now indices over z-direction do not require boundary check and do not cause warp divergence.

- **Handling z boundary conditions**

  Since we use registers to cache values over z-direction, we can initialize values for the boundary $z = 0$ before the loop. For the other boundary, $z = Nz - 1$, we construct if test inside the loop and check the boundary condition at every iteration.

```
int blocksPerCell = gridDim.x /num_cells;
int y = blockIdx.y * blockDim.y + threadIdx.y;
int x = blockDim.x*(blockIdx.x%blocksPerCell)+threadIdx.x;
int my_cell = blockIdx.x /b locksPerCell;
int in_idx = y*Nx_diff + x;
int stride = Nx_diff*Ny_diff;
register double front, current, behind;
if x < Nx_diff and y < Ny_diff
 int Nx_diff1 = Nx_diff;
 int Nx_diff2 = Nx_diff;
 int x_diff1 = 1;
 int x_diff2 = 1;
 if(Ny_diff == 1)
   Nx_diff1 = Nx_diff2 = 0;
 else if(y == 0)
   Nx_diff1 = -Nx_diff;
 else if(y == (Ny_diff-1))
   Nx_diff2 = -Nx_diff;
 if(Nx_diff == 1)
   x_diff1 = x_diff2 = 0;
 else if(x == 0)
   x_diff2 = -1;
 else if (x == (Nx_diff-1))
   x_diff1 = -1
 if(Nz_diff == 1)
   stride = 0;
 in_idx += Ndyads*my_cell
 current = old_diff[in_idx+stride];
 front = old_diff[in_idx];
 for int i = 0; i < Nz_diff; i++
    behind = current;
    current = front;
    front = old_diff[in_idx+stride];
    if (i == (Nz_diff - 1))
      front = behind;

    n = in_idx - Nx_diff1;
    s = in_idx + Nx_diff2;
    t = in_idx + x_diff1;
    b = in_idx - x_diff2;

     new_diff[in_idx] = current + (old_diff[b] + old_diff[t] -
         2*current) * r1 + (old_diff[n] + old_diff[s] - 2*
         current)*r2 + 2*(behind + front - 2*current)*r3;
     in_idx += stride;
 end for loop
end if
```

Listing 5.7: Pseudo code for the two-dimensional dyad diffusion computation. Threads are arranged as 2D blocks. The loop over z-direction is introduced.

(Listing continued from previous page) First, each thread computes the corresponding index and checks the x and y boundary conditions. Then, previous and current z values are loaded into the registers. Each thread goes through the loop, updates z values and reads $z + 1$ value from the global memory. Then, it checks $z = Nz - 1$ boundary, computes the neighbouring indices and performs diffusion operation.

5.3.4 Spatial blocking with shared memory

Shared memory can be used to improve the performance of a stencil computation by reducing the access to the global memory. We apply spatial blocking with shared memory to the two-dimensional implementation. Each thread block allocates $(BDIMY+2)\times(BDIMX+2)$ chunk of shared memory, where $BDIMX$ and $BDIMY$ are the number of threads in the x and y-directions. Since a point computation requires access to its neighbour points, we introduce additional slices (halo points) at each border of the shared memory array. Firstly, each thread reads its own value from the previous grid and stores it into the shared memory. Then, the boundary threads read and load the halo data from the global memory to the shared memory.

5.3.5 Spatial blocking with shared memory and additional registers

Loading halo data can cause warp divergence, which can be reduced by keeping y halo values in the registers in the similar fashion as for the boundary conditions in the z-direction. In this case, every thread needs to allocate only $(BDIMY) \times (BDIMX + 2)$ chunk of shared memory, so that shared memory is used for the inner grid points and halo points in the x-direction. However, this requires two more registers per equation.

5.4 L-type channel simulation

L-type calcium channel simulation consists of two steps: probability computation and sampling from the binomial distribution. In our first approach, we construct two kernels: (1) L-type probability calculation and (2) L-type opening. In this paragraph, firstly we will describe the implementation of each of the kernel functions separately, then we will assess the possibility to join two separate kernel functions into one function, called L-type simulation.

5.4.1 L-type probability calculation

Although probability computation involves a sufficient number of operations, L-type probability calculation kernel does not allow for many optimizations. The structure of thread blocks and grid is plain. Since we are operating on dyad-level and there is no dependency between cells, calcium release units from contrasting cells are allowed to be assigned to the same block of threads. In our computation we take advantage of CUDA compiler and read-only cache. Other optimizations, such as the use of shared memory, had no impact on performance.

5.4.2 L-type opening

In this part we focus on the L-type calcium channels opening, which is sampling from the binomial distribution. The structure of the threads and grid is treated in similar fashion as in the (1) L-type probability computation. We use coalesced memory to access random numbers generated by the cuRAND random number generator. Read-only cache is used for faster access to the read-only data. Moreover, the standard power operation is changed to our implemented power function.

5.4.3 Joined L-type channel simulation kernel

The L-type opening computation depends on the probabilities computed in L-type probability calculation kernel. In the next chapter we will see that L-type opening function is compute-bound while L-type probability computation is memory-bound. Moreover, the probability computation requires 30 registers, while the sampling computation requires 38 registers. The number of registers required for the computation can be reduced by merging kernels. Therefore, the possible optimization is to join both kernel functions into one, called L-type simulation.

To increase the performance of the joined kernel function, several optimizations are performed in addition to the ones applied to distinct implementations. Because the computed probabilities and random numbers are used more than once by each thread, we store them into the shared memory. The use of shared memory helps to lower the number of registers. Furthermore, for the faster access, we load one of the structures which contain variables per cell into the local memory.

5.5 RyR probability calculation

The RyR probability calculation performs an extensive number of computations, some of them require a reduction operation. The basic approach would be to have separate kernels for the computation and for the reduction operation. In this case, we would need to perform three kernel launches because the reduction operation consists of two kernel launches. However, in our implementation we decided to partially reduce data inside the computational kernel. Thus, the RyR probability calculation kernel is composed of the actual computation and a partial-reduction on dyad-level. Considering that RyR opening (see Section 5.6) is independent of these operations and that we need to perform reduction in the RyR probability calculation function, we delay the final block reduction until the calcium concentration computations will be executed. In other words, we combine two final block reduction kernels into one, such that we are able to lower the kernel launch overhead. As a results, we need to launch only one kernel function to perform RyR probability calculation. The main computation is composed of several parts that are divided into individual device functions (see Listing 5.8). Note that due to the partial-reduction computation which requires to keep dyads of distinct cells in separate blocks, we need to use a cell separation mapping to assign each thread to a specific dyad.

```
RyR probability<<<blocks,threads>>>
     __device__ Compute Ca current through Ca Channel()
     __device__ Compute Na current through L-type channel()
     __device__ Compute K current through L-type channel()
     __device__ Compute RyR probabilities()
      Partial - reduction of Ca, Na and K currents
```

Listing 5.8: RyR probability computation structure. It shows all the computations executed inside the RyR probability kernel function. Compute Ca current, Compute Na current and Compute K current are defined as __device__ functions. CUDA kernel function is indicated by <<<>>>.

Similarly to the majority of the functions, we use read-only cache to increase the number of warps per SM and reduce the number of registers. Moreover, we put all the variables, that are common to all threads in the same block, into the shared memory. This lowers the number of registers as well. Finally, we take advantage of the CUDA intrinsic function that provides a faster computation of reciprocals, which is used to avoid the heavy division operation.

5.6 RyR opening computation

The RyR opening kernel function computes eight possible transitions between four RyR states. It is the second most time consuming kernel in our application. We keep implementation on device similar to the CPU implementation: each thread checks the four states and if the state is not empty, the thread reads two random numbers from the global memory and samples from the binomial distribution. We keep the same structure of the grid as for L-type simulation kernel, because all the computations are performed on the dyad-level and no reduction operation is needed.

A small amount of optimizations have been made to decrease the execution time of the RyR opening computation: coalesced access to the random numbers in the global memory, the read-only cache and the CUDA intrinsic reciprocal. The most significant improvement that has a great impact to the RyR opening kernel function is our implemented power function used in the binomial sampling method.

5.7 Ca concentration computation

Calcium concentration computation is especially complex. It contains several laborious calculations that also include reductions of one or more items. The implementation is divided into two parts: main kernel and reduction kernel. The latter performs reductions on partially reduced data across the blocks of threads and sends the final result as the whole-cell value to the corresponding cell. The main kernel consists of two parts: computation and partial-reduction. The structure of Ca concentration computation is shown in Listing 5.9. Because the final reduced value is a per-cell value, calcium release units from different cells need to be assigned to different blocks. Thus, we used a cell separation mapping. We have broadly presented reduction algorithm in Section 5.2, so we omit the reduction part here and concentrate mostly on the computational part.

```
Ca concentration<<<blocks,threads>>>
  __device__ Compute concentrations and SR Ca flux in a dyad ()
             Partial-reduction of SR Ca release flux
  __device__ Compute local Na-Ca exchanger current in myoplasm()
  __device__ Compute local Na-Ca current in submembrane()
  Compute Sarcolemmal Ca Pump, local background Ca current
  Compute local myoplasmic Ca concentration
  Partial - reduction of Na-Ca, Ca Pump, Ca currents
Block reduction<<<blocks,threads>>>
```

Listing 5.9: Structure of the Ca concentration computation. There are two main kernels: Ca concentration and Block reduction. All the computations and partial-reduction operations executed inside the Ca concentration kernel are shown in the code. CUDA kernel functions are indicated by <<<>>>, functions with __device__ syntax are called from kernel and performed on the device, while other computations are performed directly inside the kernel function.

5.7.1 Calcium concentration kernel

In this paragraph we declare all optimizations we have made on the Ca concentration kernel excluding partial-reduction of SR Ca release flux and partial-reduction of NA-Ca, Ca Pump and Ca currents. Our initial implementation was limited by the number of registers per thread. To lower the number of registers we apply several optimizations. Since fetching from the shared memory is faster than from the global memory, we let *thread 0* load variables that are common for cell into the shared memory which is accessible by all threads in a block. Moreover, we store data, which is accessed more than once into the shared memory. Secondly, we divide the kernel into smaller device functions. This reduces the number of registers significantly. Powers of two and division are costly operations that are substituted by a sophisticated multiplication operation. For divisions that cannot be interchanged with multiplications, we use the CUDA intrinsic function that computes the reciprocal of a variable in round-towards-nearest mode.

As mentioned above, the computation kernel performs two types of calculations: the main computation and the reduction within the dyads. In order to perform the reduction, we use one of the algorithms presented in Section 5.2. We select to use the block-reduction provided by the CUB library because of its high performance and effortless adjustment to our implementation. The partial-reduction is implemented as a part of the computation. It operates inside the device function which was called from the kernel function and it takes part in the main kernel computation. This is shown in Listing 5.9.

5.7.2 Reduction kernel

The calcium concentration computation requires the reduction of five values per cell. The partial reduction is performed during the main kernel launch and the remaining reduction is left to an independent kernel. To obviate unnecessary kernel launch overheads, we combine the reduction over partially reduced items from the RyR probability calculation function (Section 5.5) with the reduction from the computation of calcium concentration. Block reduction operates on the Ca, Na, K, Na-Ca, Ca Pump currents and on the SR Ca release flux. In our implementation we select the CUB provided block reduction to reduce all eight values. We invoke the kernel only once, specifying the grid size equal to the number of cells and the optimal block size equal to the number of threads (128 threads are used in our program) that allows each block to perform the reduction over cell variables.

5.8 Summary

In this chapter, we presented implementation and optimization strategies for the computation on the dyad-level. We broadly described several algorithms and implementation methods for the reduction operation and the intracellular diffusion computation. L-type simulation, RyR channel opening and Ca concentration computation were presented in terms of the composition and optimizations.

Chapter 6

Experimental results and evaluation

In the previous chapter we have presented various optimization techniques and algorithms used for the implementation of dyad-level computations. We begin this chapter by comparing different implementations for the reduction operation and stencil computation. Further, we evaluate each of the dyad-level function implemented on the device separately. To verify our implementation we will look at the execution time of the function, number of floating-point operations per second and memory bandwidth. Then, we compare the impact of the different optimization techniques for all the functions. We conclude this section by measuring scalability and the performance of the simulator.

6.1 Experimental setup

For all experiments, we use a fixed time step of 0.05 ms. To discretize the diffusion terms in Equation (2.16), we use a fixed spatial mesh resolution of 0.5 mm. For the diffusion terms in equations (2.4), (2.8) and (2.10) we use spatial mesh resolution of 0.2 mm. To evaluate all the implementations and optimizations of the separate kernel functions on the dyad-level, we run the tests for one time step. Unless otherwise noted, the number of cells is 1500 and each cell always contains 10000 calcium release units. Dyads are organized as a 3D grid of size $100 \times 10 \times 10$. Each kernel was executed five times and the fastest time was selected. For the evaluation we use kernel execution time excluding the cost of PCI data transfer. Floating-point operations per second and memory bandwidth were measured by nvprof profiler. The implementation and optimization experiments were performed with disabled ECC option on the Lizhi server (Section 3.5) using a single GPU. The scaling tests and the cell computation speed tests were performed on the Abel supercomputer (Section 3.5) for 10000 time steps, measuring the whole program execution time.

Implementation	Read	Write	Total
Basic parallel reduction	121 GB/s	2 GB/s	123 GB/s
Shuffle warp reduce	163 GB/s	3 GB/s	166 GB/s
CUB (commutative only)	169 GB/s	4 GB/s	173 GB/s
CUB (warp reduction)	177 GB/s	3 GB/s	180 GB/s
CUB (list)	179 GB/s	1 GB/s	180 GB/s

Table 6.1: Memory bandwidth for the reduction computation using different implementation strategies: basic parallel reduction, shuffle warp reduce and three reduction algorithms from the CUB library. The table contains read, write and total bandwidth of the first kernel launch.

6.2 Comparing results from different reduction implementations

In this section we will present the achieved performance of several different implementation techniques of the reduction operation: basic parallel reduction, shuffle warp reduce algorithm and three different variations of the CUB library algorithms. In this analysis, we measure the reduction operation contained in the dyad diffusion function (see Listing 4.2). It is the only function where the reduction is not merged with the computation. The reduction is performed by launching two execution kernels. It operates on six items, which implies $6 \times 10000 \times 8$ bytes = 0.48 MB of data in total.

Reduction is not a computationally heavy operation, but it requires a lot of memory read and write operations. Therefore, the reduction kernel is memory-bound and we focus on the memory bandwidth in this analysis. During the first kernel launch a large number of blocks performs reduction, while during the second launch the reduction is performed on already partially reduced data by a significantly smaller number of threads. The second kernel launch is so short, that it is omitted from our bandwidth analysis. Table 6.1 shows the memory bandwidth given by the NVIDIA profiler for the different implementations. The bandwidth of the basic parallel reduction reaches only 123 GB/s. The improved reduction with shuffle operations increases device memory bandwidth to 166 GB/s. The CUB reduction with the commutative only algorithm achieves 173 GB/s, which is more than 96% of the STREAM measured bandwidth. The CUB with the warp reduce algorithm and the CUB list reach the maximum practically achievable memory bandwidth, which is 180 GB/s.

Figure 6.1 visualizes how bandwidth depends on our selected implementation method and data size. From the graph we can see that all the implementations behave similarly. The bandwidth increases rapidly when applying the reduction for data ranging from 0.48 MB to 24 MB, then the growth slows down and the bandwidth reaches its maximum at the point where the data size is 240 MB. When further increasing the data size, bandwidth remains stable. The increased size of data yields the better usage of GPU resources and increased bandwidth. From the graph we can see that there is a large gap in memory bandwidth between the basic parallel reduction and other implementations. Moreover, the progress of the bandwidth of the basic parallel reduction is not as steep as for the other methods.

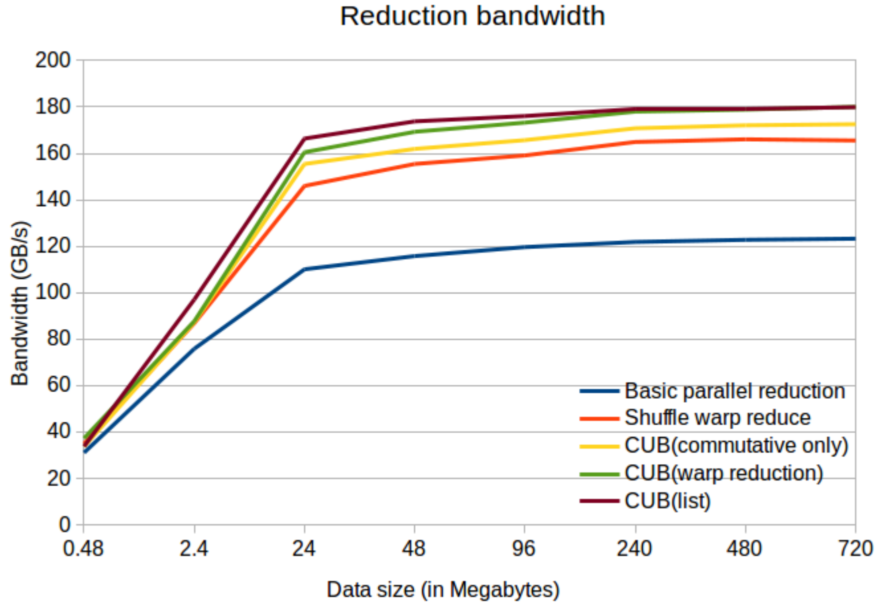

Figure 6.1: Memory bandwidth for the reduction computation. The graph shows how bandwidth depends on the selected implementation method and the data size. The bandwidth reaches its maximum when using 240 MB of data.

Implementation	First launch	Second launch	Total time
Basic parallel reduction	6.00 ms	0.15 ms	6.15 ms
Shuffle warp reduce	4.39 ms	0.09 ms	4.48 ms
CUB (commutative only)	4.27 ms	0.10 ms	4.37 ms
CUB (warp reduction)	4.09 ms	0.09 ms	4.18 ms
CUB (list)	4.02 ms	0.07 ms	4.09 ms

Table 6.2: Execution time of the reduction computation. The table shows the execution time of the first kernel launch, the second kernel launch and the total execution time for different implementations. CUB list implementation consists of first launching the kernel with the two-elements-per-thread approach and then launching the kernel with the one-element-per-thread approach.

The execution time of the reduction operation, which consists of two kernel launches, is shown in Table 6.2. We can see that the fastest algorithm takes 4.09 ms. Basic parallel reduction is 1.5 times slower than the fastest CUB list implementation. Shuffle warp reduce implementation execution takes 4.48 ms, which is 27% less than the execution time of basic parallel reduction, but 10% more than the execution time of the CUB list implementation. Both CUB commutative only and CUB warp reduction takes more time to perform reduction operation than the CUB list algorithm.

6.2.1 Discussion

We have tried several reduction algorithms during the process of the implementation. The advantage of using the CUB library to perform the reduction operation is undeniable. The drawbacks of our implemented basic parallel reduction are low occupancy and the need for thread synchronization. Increasing occupancy may not increase the performance because we are also limited by the shared memory. Even though we increased memory utilization of the basic parallel reduction by including multiple additions at the beginning of the kernel, our implementation does not reach the achievable bandwidth or the bandwidth obtained in [25]. Reduction with the shuffle instruction achieves 68.3% occupancy, which means that there is an acceptable number of warps executing on each SM. The problem of limited performance might be memory and execution dependencies. Although all the algorithms provided by the CUB library showed good performance, in our program we will be using the fastest approach - the CUB list implementation, which achieves the maximum STREAM measured bandwidth. Our achieved performance is close to the performance of the CUB reduction given in [33], where the reduction was executed on the Kepler K20X and on data set of 537 MB, which gave bandwidth of 175 GB/s. In addition, the whole data set was reduced by one CUB function while in our implementation we call the CUB reduce function 6 times and we use thread synchronization between the calls.

6.3 Results of different implementations of the diffusion computation

We evaluate the performance of the 3D stencil computation algorithms and compare it with the Roofline model [53]. Since the dyad diffusion kernel function computes not one, but three differential equations, we include all of them in our analysis.

First, we use the Roofline model analysis on the baseline implementation of the diffusion equations. An update of a single point requires 18 or 16 double-precision floating point operations depending on the selected equation. Therefore, at each time step the kernel function performs $(18{+}18{+}16){\times}Nx{\times}Ny{\times}Nz$ operations, where Nx, Ny and Nz represent the grid dimensions. To make our analysis easier, we assume that once we load a value, it remains in the cache memory for this time step. Hence, the size of the data that is loaded per time step is $Nx \times Ny \times Nz \times 8 \times 3$ bytes and the size of the data that is stored per time step is also $Nx \times Ny \times Nz \times 8 \times 3$ bytes. Here, 8 is the size of a double precision value in bytes. We multiply computations by the factor of 3, because we perform three differential equations that have the same amount of loads and stores. Now, we can compute the operational intensity **I**, which is defined as a floating-point operations per DRAM transfer in bytes and perform the Roofline model analysis that was presented in Section 3.8.

$$\mathbf{I} = \frac{\mathbf{W}}{\mathbf{Q}} = \frac{(18+18+16) \times Nx \times Ny \times Nz}{(Nx \times Ny \times Nz \times 8 \times 3) \times 2} = 1.0833 \text{ GFLOP/GB}$$

Peak floating-point performance is defined by the hardware specifications:

$$\mathbf{C} = 1174.78 \text{ GFLOPS}$$

Memory bandwidth measured by the STREAM benchmark is

$$\mathbf{B} = 180 \text{ GB/s}$$

$$\frac{\mathbf{C}}{\mathbf{B}} = \frac{1174.78\text{GFLOPS}}{180 \text{ GB/s}} = 6.5 \text{ GFLOP/GB} > \mathbf{I}$$

According to the Roofline model, this indicates that the performance is limited by memory bandwidth. Optimizing memory accesses is the most important approach in this case.

The attainable compute performance found by the Roofline model is

$$\mathbf{P} = \min \begin{cases} \mathbf{C} \\ \mathbf{B} \times \mathbf{I} \end{cases} = \min \begin{cases} 1174.78 \text{ GFLOPS} \\ 180 \times 1.0833 \text{ GFLOPS} \end{cases} = 194 \text{ GFLOPS}$$

Table 6.3 displays the execution times for one time step of all the optimizations applied to stencil computation function. The two kernel implementations described in Section 5.3.2 lead to the longest execution time, which is 17.35 ms. The two-dimensional solution from Section 5.3.3, where we loop along the z-direction and store redundant variables into the registers, decreases execution time by more than 50%. Usage of the read-only cache and x and y boundary optimizations decreases the execution time further. Thus, the completely optimized two-dimensional version obtains the best execution time, which is 6.99 ms. As we can see, this implementation is more than two times faster than our naive baseline implementation. Additional optimization techniques on the two-dimensional implementation, such as spatial blocking using shared memory with or without additional registers, increased the execution time.

As we have observed, the diffusion kernel function is memory-bound. None of the implementations achieve the STREAM measured bandwidth. For the optimized two-dimensional implementation, the bandwidth increases to 138 GB/s, whereas for the baseline and two-kernel implementation it is only 64 GB/s. The implementation that uses shared memory without additional registers achieves bandwidth of approximately 82 GB/s and the implementation with additional registers achieves 83 GB/s. All the measurements can be found in Table 6.4.

In Figure 6.2, the two-dimensional implementation leads to a significant increase in the number of floating-point operations per second compared to the baseline implementation. As discussed above, the computational intensity is 1.0833. This indicates that the attainable number of FLOPS is 194 GFLOPS. As shown in Figure 6.2, the baseline and the two-kernel implementations have a large gap compared to the estimated attainable peak on Kepler, which in ratio is 55% and 50%, respectively. In contrast to the two-kernel implementation, the two-dimensional implementation achieves 98% of the estimated number of FLOPS. This is the highest performance we were able to reach. Moreover, in our kernel function we perform additional calculations needed for the stencil computation, which probably lowers the performance. The figure shows, that blocking with shared memory does not yield performance improvements. On the contrary, it degraded performance by approximately 35%. Blocking with shared memory and the usage of addition registers in the y-direction reduces performance even more.

Our analysis with the NVIDIA profiler indicates that warp divergence and a high number of registers are the main bottlenecks of performance for all the

Optimization	time in ms	#registers	size of shared memory
Baseline implementation	16.14	46	0
Two-kernel solution	16.00 + 1.35	42 + 42	0 / 0
Two-dimensional implementation	8.12	64	0
Read-only cache optimization	7.37	65	0
X and Y boundary optimization	6.99	64	0
Spatial blocking with shared memory	10.52	78	4.7813KB
Spatial blocking with shared memory and additional registers	11.43	89	3.1875KB

Table 6.3: Execution time of diffusion computation on the dyad-level and kernel details. The time and the number of registers for the two-kernel solution are given as inner diffusion kernel plus x-boundary diffusion kernel.

Implementation	Read	Write	Total
Baseline implementation	41 GB/s	23 GB/s	64 GB/s
Two-kernel solution	41 GB/s	23 GB/s	64 GB/s
Two-dimensional implementation	87 GB/s	51 GB/s	138 GB/s
Spatial blocking with shared memory	48 GB/s	34 GB/s	82 GB/s
Spatial blocking with shared memory and additional registers	51 GB/s	32 GB/s	83 GB/s

Table 6.4: Device Memory Bandwidth of the optimized versions of the 3D 7-point stencil computation. The measurements are given by the NVIDIA Visual profiler.

implementations. The number of registers used is given in Table 6.3. As we can see spatial blocking with shared memory and additional registers requires the largest number of registers. However, in the two-dimensional implementation each thread uses 64 registers or in other words, each block uses 8192 registers, which means that each SM is limited to executing 8 blocks simultaneously. As a result, the achieved occupancy is no larger than 47%.

6.3.1 Discussion

Several different stencil computation implementations and optimizations were analyzed during this project. First, the two-kernel solution was introduced to avoid warp divergence during computations on the boundary. However, we saw that the kernel launch overhead cannot be hidden by the computation. Dividing the computation among two kernels is more expensive than having conditional statements inside the main computation. Having a loop over the z-direction and caching the data into the registers turned out to be the most significant improvement in the diffusion computation because of the fast access to the registers. However, thread divergence and a large number of registers per thread are significant issues. Because of the conditional statements caused by the unoptimized boundary conditions not all warps have the same branching behaviour, which leads to inefficient usage of GPU resources. Despite all the optimizations applied to the boundary conditions, the NVIDIA profiler

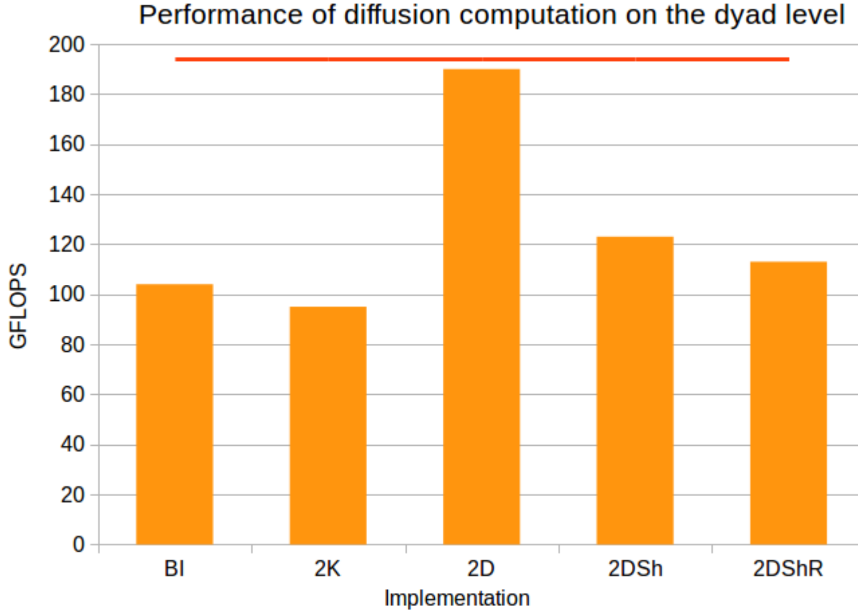

Figure 6.2: Number of floating-point operations per second of the different implementations of the 3D 7-point stencil computation. Note that BI is the baseline version that was described in Section 5.3.1. 2K is implementation that invokes two kernel functions: one for the inner computation and one for the x-boundary computation. 2D is a two-dimensional implementation, where we use a 2D grid for x and y-directions, and a loop over z-direction. Moreover, we use registers to cache points that vary over z-direction. The 2DSh implementation takes advantage of the 2D shared memory. In addition to the shared memory, the 2DShR implementation uses registers for keeping points that vary in the y-direction. The red line is the model-estimated maximum of the floating-point operations per second.

still reports warp divergence of 20.8% for the two-dimensional implementation. Other optimizations such as predefining constants and improving the $z = Nx - 1$ boundary conditions by performing the last iteration outside the loop do not increase the performance.

Loading halo data in the shared memory needs conditional operations at each iteration. This causes a significant execution overhead, since every thread in the warp needs to execute the same condition, which leads to 75% of warp divergence in the implementation that uses spacial blocking with shared memory. In fact, this futile use of the shared memory on stencil computations on the Kepler architecture was mentioned in [37].

We extended our shared memory optimization further by using registers along the y-direction, but even then the execution time increased. Although, additional usage of registers gave improvements on the Kepler GK-104 architecture [52], it does not show any benefits in performance on the Kepler GK110.

In our program we will be using fully optimized two-dimensional solution.

6.4 L-type channel simulation

In the previous chapter, we introduced two different implementations of the L-type computation on the device. One possible way is to have two separate CUDA kernel functions: (1) L-type probability calculation and (2) L-type opening computation. Another way is to join all the computations into one kernel function. In this section, we will present and compare results from both implementations.

First, we want to determine if the L-type probability calculation kernel is memory-bound or compute-bound. We evaluate two specific metrics: memory bandwidth and computational throughput expressed as the number of floating-point operations per second. Table 6.5 shows the resulting memory bandwidth and computational throughput. We know that for Kepler K20m the STREAM measured bandwidth is 180 GB/s. L-type probability calculation function attains 170 GB/s of memory bandwidth, which is more than 94% of the estimated peak. From the NVIDIA profiler we can get the exact number of the double-precision floating-point operations kernel is performing and measure the performance in GFLOPS. The execution time of the kernel is 4.22 ms and it performs 720×10^6 floating-point operations, which gives 171 GFLOPS. According to the NVIDIA profiler, the kernel utilizes only 50% of the compute capability. Hence, L-type probability calculation kernel is memory-bound. Note that block size, register usage, and occupancy allows us to fully utilize all warps on the GPU. However, the possible problem here could be that a load or store operation can not be performed, because the required resources are fully utilized.

The execution time of the L-type opening kernel is 5.18 ms and the computational throughput is 295 GFLOPS. According to the NVIDIA profiler, the kernel compute utilization is more than 80%, which means that the kernel is compute-bound. Memory bandwidth is 142 GB/s, which is 79% of the STREAM measured bandwidth.

We have observed that the L-type opening function is compute-bound, but the L-type probability computation is memory-bound. This means that we can consolidate these two kernels into one, without losing performance. In this case, we avoid 50% of the kernel launch overhead. Moreover, we lower the number of accesses to the global memory since the L-type probability computation provides values that are used in the L-type opening. After the consolidation, the L-type kernel execution time becomes 7.6 ms which is faster than executing the two kernels consecutively, but as expected the number of registers increases. The final result after all optimizations is a perfectly balanced memory and compute utilization and a significantly faster computation. The complete L-type computation time is 6.11 ms, the memory throughput is 158 GB/s which is 88% of the achievable peak bandwidth and the computational throughput is 334 GFLOPS. Thus, we have increased the utilization of the computation by the consolidation of the L-type probability calculation and the L-type opening computation kernels. According to the NVIDIA profiler, the L-type channel simulation utilization is 80%. We are not able to improve the joined L-type computation kernel further, because of the limited number of registers and amount of shared memory per multiprocessor. According to the CUDA occupancy calculator, lowering the number of registers further would not help to improve the occupancy and increase the performance.

Kernel function	GFLOPS	Bandwidth	Time	# registers
L-type probability calculation	171	170 GB/s	4.22 ms	30
L-type opening	295	142 GB/s	5.18 ms	38
L-type simulation	334	158 GB/s	6.11 ms	41

Table 6.5: Performance metrics of L-type channels computation. The table shows the number of floating-point operations per second in GFLOPS, device memory bandwidth, execution time and number of registers for L-type probability calculation kernel, L-type opening computation and a joined implementation - L-type simulation.

6.5 RyR probability calculation

The occupancy of the RyR probability calculation function was limited by the number of registers. Performing the first optimization technique, which is dividing computational kernel into several small device functions called from the execution kernel, reduced the number of registers used and increased a theoretical occupancy rate of the threads from 56% to 75%.

The RyR probability calculation is a complex function, where the main computation and partial-reduction are performed in the same execution kernel. We have already evaluated the reduction implementation and confirmed that it is a memory-bound operation. Thus, to perform a better evaluation, we will look only at the computational part. According to the NVIDIA visual profiler, compute utilization is significantly lower than memory utilization. Thus, the computational part of the RyR probability calculation is also bounded by memory bandwidth. To increase the performance, we need to lower the number of registers used. This was done by the use of the read-only cache, which decreased the number of register per thread from 37 to 34. To lower the number of registers further, we loaded a common variable into the shared memory. This optimization decreased the number of registers per thread to 32, which according to the NVIDIA occupancy calculator, is enough to achieve 100% occupancy. A disadvantage of the usage of shared memory is the required thread synchronization. In Table 6.6, we can see how the decrease in the number of registers results in decreased execution time.

Optimization	Time	# of registers
None	8.23 ms	37
Read-only	8.07 ms	34
Shared memory	7.95 ms	32

Table 6.6: Optimisation of the RyR probability calculation. The table shows that the register count per thread can affect the execution time of the CUDA kernel function. Note that the measurements are done just for the computational part of the kernel.

Kernel function	GFLOPS	Bandwidth	Time
Computational part	209	172 GB/s	7.95 ms
Full RyR probability calculation	210	142 GB/s	9.36 ms

Table 6.7: Performance metrics of the RyR probability calculation. The first row shows performance of the function without reduction. The second row shows performance of the full kernel function.

Now, we will look at the complete RyR probability calculation execution kernel, where both computation and reduction operation are executed. In this evaluation, we look at the fully optimized computational part. We use the CUB library one-element-per-thread approach to perform the reduction operation. The complete RyR probability kernel execution time is 9.36 ms. The compute utilization is slightly lower than the memory utilization. Otherwise it would be a perfectly balanced kernel function. Memory bandwidth exceeds 142 GB/s, which is 79% of the achievable bandwidth. Table 6.7 contains the performance of the RyR probability calculation. It is not surprising that joining reduction part almost does not effect the computational performance, but it lowers the memory utilization.

6.6 RyR opening computation

Several optimizations have been made to increase the performance of the RyR opening kernel function, but the most significant improvement is the reimplemented power function used in the binomial sampling method. It decreased execution time from 12.17 ms to 10.44 ms. According to the NVIDIA profiler, the RyR opening kernel function is bounded by memory bandwidth. The achieved bandwidth in this computation is 162 GB/s, which is 90% of the achievable bandwidth. Computational throughput reaches 142 GFLOPS. Results can be found in Table 6.8.

Kernel function	GFLOPS	Bandwidth	Time
RyR opening	142	162 GB/s	10.37 ms

Table 6.8: Performance of the RyR opening computation. The table shows the number of floating-point operations per second, device memory bandwidth and execution time of the RyR opening computation.

Kernel function	GFLOPS	Bandwidth	Time
Ca concentration only computation	274	119 GB/s	22.09 ms
Ca concentration kernel	244	98 GB/s	26.12 ms

Table 6.9: Performance of the Ca concentration computation. The table shows the number of floating-point operations per second, the device memory bandwidth and the execution time of the main kernel of the Ca concentration computation. *The Ca concentration kernel* defines the main computational kernel. *Ca concentration only computation* excludes the partial-reduction operations.

6.7 Ca concentration computation

The Ca concentration function consist of two kernel calls: the main kernel and the block-reduction kernel. As we have already evaluated the reduction operation, in this analysis we will concentrate on the main kernel. We evaluate the performance of the main kernel in the same way as we did for the RyR probability calculation function. We will look at the computational part separately from the partial-reduction operation.

We need to determine whether the computational part of the main kernel is memory or compute-bound. The kernel performs 6.045×10^9 floating-point operations and the execution time is 22.09 ms, which means that its performance is $6.045 \times 10^9/0.02209 = 274$ GFLOPS. According to the NVIDIA Visual profiler, kernel compute and memory utilizations are 53% and 55% respectively. Both compute and memory utilizations are lower than 60%, which means that the kernel is probably bounded by latency of arithmetic operations. The implementation achieves 55.2% occupancy which is close to the theoretical occupancy 56.2% of the kernel. Due to the grid structure and the number of registers we cannot expect higher occupancy rate.

Now, let us evaluate the results of the main kernel that includes the partial-reduction. The bandwidth of the whole main kernel is represented in the second line in Table 6.9. The kernel achieves a low compute throughput and memory bandwidth utilization compared to the achievable peak. A large amount of warp blocking before reduction and before writing to the shared memory is performed, which reduces the performance.

Merging block reduction kernels from the RyR probability calculation and the Ca concentration computation, increases the overall performance significantly. However, the achieved bandwidth of the block reduction kernel is 83 GB/s which is significantly lower than the STREAM measured bandwidth. Although thread block size and number of registers allows us to completely utilize warps on the GPU, the program does not reach peak performance due to the low amount of data involved in the reduction operation.

6.8 Optimization impact on the distinct functions

Figure 6.3 shows most of the optimization techniques that were applied to the dyad-level computation functions. All of the functions are distinctive: some of them perform more computations than others, some of them are compute-bound, while others are memory-bound, some of them have a simple grid structure, while others require the cell separation mapping. As a consequence, diverse optimization techniques have a different influence on distinct kernel functions and the same optimization might not lead to the same improvement. One example is the dyad diffusion function for which the use of shared memory leads to decreased performance while for other functions it lead to significantly better performance.

From the Figure 6.3 we can see that the calcium concentration computation is the most demanding kernel function in our application. It has the longest execution time, but it also reflects optimizations best. The use of shared memory decreased execution time by 5 ms alone. Other optimizations, such as dividing computation into smaller device functions, defining constants, the use of read-only cache and CUDA intrinsic functions, improved running time by 5 ms. The final implementation is 33% faster then the original. The execution time of other kernel functions shows slightly less improvement in the performance.

As we mentioned above, accessing items stored in the shared memory gives one of the largest improvements in the execution time of the functions. The use of the shared memory decreased L-type simulation time by 6%, the RyR probability calculation time by 1% and Ca concentration computation time by 17%. The sum of reductions over all the code segments with applied shared memory optimization is 5.75 ms.

Most of the computations require an extensive amount of complex division operations. Unfortunately, not all division operations were possible to interchanged with a simple multiplication. However, the CUDA's intrinsic reciprocal was used instead. It gave approximately 3 ms of reduction in execution time over all the functions. This optimization had the considerable impact on the calcium concentration computation. Usage of the CUDA's intrinsic reciprocal reduced calcium concentration computation time by 7%.

Read-only cache is the most common optimization, which had been used in every kernel function. This is the most obvious and easiest optimization. Moreover, it decreased execution time of every kernel function. The usage of read-only cache decreased execution time of the calcium concentration computation by 3%. The execution time of the dyad diffusion kernel decreased by 9%.

The replacement of the power function that uses bit-wise operations is used in the L-type simulation and RyR opening functions. It has no effect on the L-type opening due to a small exponent. In this case the MATH power function might be faster than the implemented power function. Contrary to the L-type binomial sampling, we get a meaningful improvement for the RyR opening binomial sampling. The execution time of the function decreased from 13.17 ms to 10.04 ms, meaning that for large exponents the new power function is significantly faster than the simple power function.

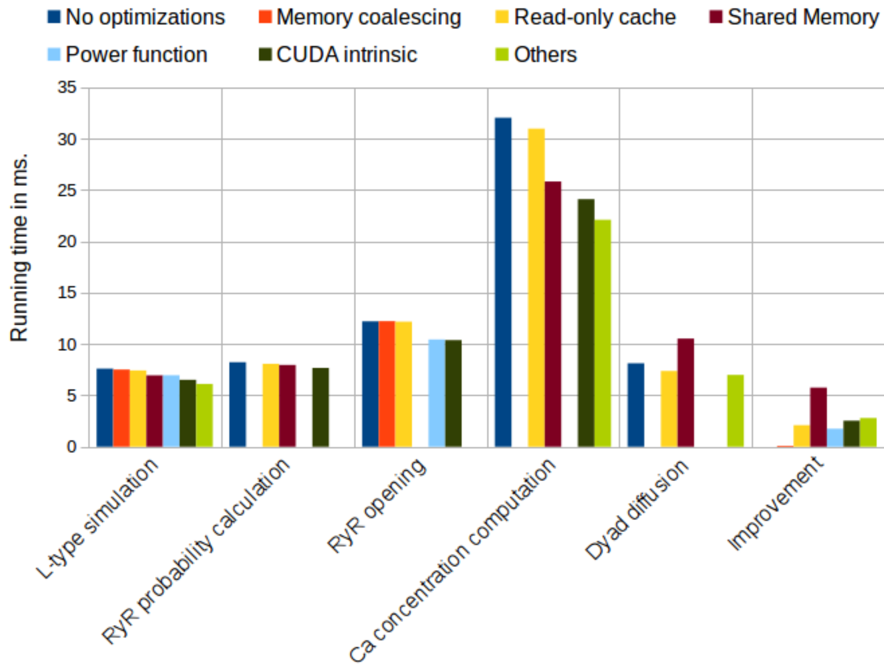

Figure 6.3: Performance improvements of the individual functions in the dyad-level computations due to the different optimization techniques. All optimizations are applied cumulatively. *Improvement* shows the sum of reduction in running time over all code sections due to each optimization. *Others* represents manifold optimization techniques, such as power by two or division operation interchanged with multiplication, defining constants, dividing kernel into smaller device function or loading variables into local memory. RyR opening and calcium concentration computation are counted without reduction operation.

After all the optimizations that are shown in Figure 6.3 the execution time of the L-type simulation decreased by 20%. The computation time of the RyR probability and the RyR opening kernel functions decreased by 7% and 15%, respectively. The calcium concentration computation time is reduced by 31% and the dyad diffusion computation time is reduced by 14%.

The optimization that is not shown and mentioned in the graph is the improved reduction operation. Faster reduction implementation yields faster RyR opening and Calcium concentration computation. Using the fastest reduction algorithm, time for the calcium concentration computation decreased from 46.93 ms to 26.12 ms, which is more than 50%. The RyR opening computation execution time decreased only by 0.2 ms. According to the results, we can conclude that the improved reduction has the largest impact on the performance of our simulator.

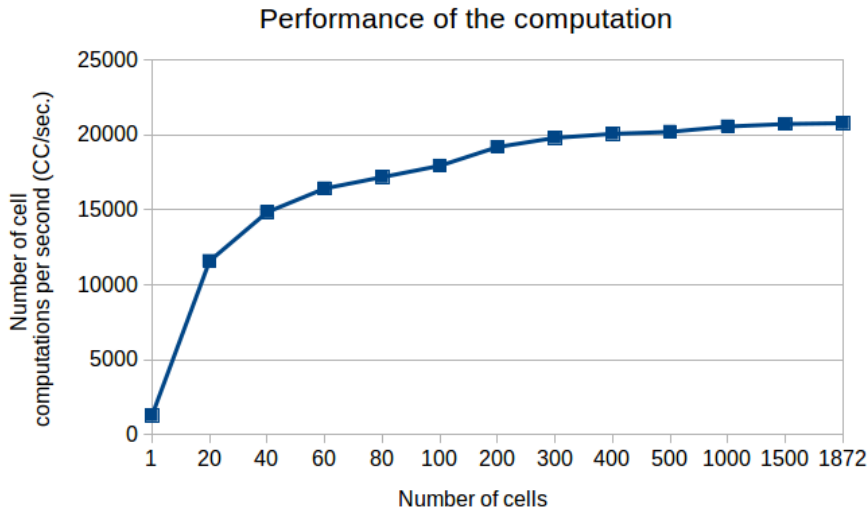

Figure 6.4: Speed of the cell computation on one GPU. The graph shows the dependency between the performance of the computation and the number of cells on one GPU. The performance is given in CC/sec. 1872 cells is the maximum number of cells that can be contained in the memory of our device.

6.9 Cell computation speed on a single GPU

A test on one GPU is performed to find out how the number of cells affects the speed of a cell computation. To compute the speed of the cell computation, we use a special metric called number of cell computations per second (CC/sec). The number of cell computations is defined as the number of time steps for a single cell. So the total number of cell computations for a given number of cells is *number of cells* × *number of time steps*. For the smallest number of cells the stride between cells used in the experiment is 20 cells. For the simulation, where number of cells ranges from 100 to 500 we choose the stride to be 100 cells and for the larger numbers of cells the stride is set to 500 cells. Graph 6.4 shows the obtained results. The speed of computation increases with the greater number of cells. Using a small amount of data we do not utilize GPU fully, thus single cell computation is extremely slow. The speed rises rapidly by increasing the number of cells from one to ten. The number of cell computations per second is 6 times larger for the simulation of 10 cells than for the simulation of one cell. Further expanding the number of cells, we increase the speed, but the increase in speed is not rapid. The speed becomes almost stable with the number of cells being larger than 1000. The reason for that is the full utilization of GPU using a large amount of data. In Figure 6.4 we can see that 1000 cells is enough to achieve the maximum speed. Therefore, increasing the number of cells for one GPU probably would not increase the speed.

6.10 Scaling experiments, results and analysis

To measure the efficiency of our application using an increasing number of nodes, we perform scaling experiments. Due to the limited number of available compute nodes during these tests, the number of nodes that we use ranges from 1 to 8, where each node has two NVIDIA GPUs.

As the cell simulation requires a lot of memory, weak scaling is a suitable way for testing the program. We perform weak scaling tests using 10000 dyads per cell. In this test we use grids of size $18 \times 13 \times 8$ which is 1872 cells for each GPU or 3744 cells for each compute node. We increase the tissue size and the number of compute nodes for each test. The tissue size for 8 nodes is $72 \times 26 \times 16$ cells.

Since simulation of a large number of cells takes a long time to run, a good way of evaluating the program is a strong scaling test. To perform the strong scaling we fix the tissue size at $18 \times 13 \times 16$, which utilizes the maximum memory of two GPUs. The tissue size remains the same, but we increase the number of compute nodes for each experiment.

We measure scalability in the compute cell computations per second (CC/sec). Note that in this experimental setup we use 10000 time steps. To verify scalability, we plot the number of cell computations per second versus the number of nodes.

The performance of weak and strong scaling tests is shown in Figure 6.5. We see that almost the same performance is achieved using both weak and strong scaling. The linearly increasing curve indicates a good scaling. The weak scaling attains almost 100% efficiency in every test case because of the low communication between the nodes and a large amount of computation. The efficiency of the strong scaling decreases with an increasing number of compute nodes. The communication overhead becomes more visible for the larger number of compute nodes in the strong scaling experiment because each node performs a smaller amount of the computation.

6.10.1 Analysis of the performance

The execution time of the simulation of the 29952 cells organized as a $72 \times 26 \times 16$ grid is 901.12 seconds using 16 GPUs. The time spend to perform cell computations is approximately equal to 894 seconds on each GPU. The diffusion computation takes 0.313 seconds for each process. The remaining time is spent on the MPI communication and data transfer.

To verify the speed of our implementation, we compare our achieved speed defined as a number of cell computations per second with the speed achieved in the CPU implementation of the same simulator described in [31]. Here, a dual Intel Xeon E5-2670 (Sandy Bridge) processor was used. For the grid of size $236 \times 2 \times 3$ or 1416 cells, the CPU implementation reached the speed of 5000 CC/sec. We performed the simulation of 1416 cells using our GPU implementation. The speed amounts to 20623 CC/sec, which is more than 4 times faster than the speed of the CPU version.

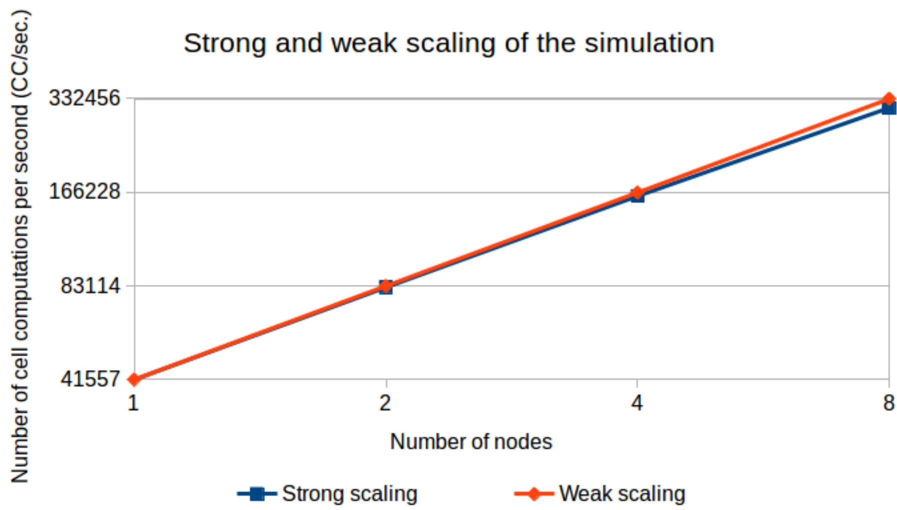

Figure 6.5: The performance of the weak and strong scaling tests. Performance is given in CC/sec. The number of nodes ranges from 1 to 8. Linearly increasing curves indicate a good scaling.

6.11 Summary

In this chapter we evaluated our implementation of the dyad-level computations. Firstly, we compared different algorithms for the reduction operation. The best performance was demonstrated by using the CUB library implemented reduction functions, which achieved the maximum bandwidth of 180 GB/s. Then, we compared different implementations and optimizations of the stencil computation, where the fully optimized two-dimensional solution showed the best performance. Several optimizations on the dyad-level computation functions were made. The performance of the calcium concentration computation kernel improved the most. Shared memory was the largest improvement source for all the functions. Finally, we measured the speed of the cell computations and performed scaling tests.

Chapter 7

Cardiac simulations

The number of computational models of action potential propagation and processes in a cardiac cell increased during the last years. Models of the cardiac electrophysiology and calcium handling make it possible to test and explain hypotheses that would be problematic to investigate experimentally. However, computational models are usually based on several assumptions of the parameters that could have important consequences for the accuracy and validation of the models. Any changes in the structural dyadic properties in the cell can lead to the heart failure, thus it is essential to investigate computational models of the heart and estimate the correct values of the parameters.

7.1 Conduction velocity

A conduction velocity (CV) is the speed at which action potential propagates through a given tissue region. The most important factors that determine CV are membrane excitability and the conductivities of cardiac tissue. In the normal cardiac ventricle, a cell by cell conduction velocity is approximately 0.5 m/sec [29]. CV depends on a lot of parameters, such as cell excitability, tissue conductivity, a degree of repolarization and pacing cycle length [7].

To validate CV first we record voltage of two cells in z-direction: we choose one cell ($cell_1$) to be at point $v1 = (Nx/2\ , Ny/2,\ Nz/4)$ and another cell ($cell_2$) to be at point $v2 = (Nx/2,\ Ny/2,\ 3 \times Nz/4)$. Then, we find the time $t1$ and $t2$ at which each of these cells triggers respectively. The cell has been triggered if its voltage is larger than -70 mV, which means that we are looking for the time points where the voltage of $cell_1$ and $cell_2$ is larger than -70 mV for the first time. To compute the conduction velocity in the z-direction, we compute the distance between cells $cell_1$ and $cell_2$ in the z-direction and divide it by the difference in time at which each cell triggers. Since a cell is represented as a point on a grid, we find the distance between cells by computing the distance between $v1$ and $v2$ and multiplying it by the mesh resolution in the z-direction.

$$
\begin{aligned}
distance &= \Big(\frac{3 \times Nz}{4} - \frac{Nz}{4}\Big) \times dz = \frac{Nz}{2} \times dz, \\
CV &= distance/(t2 - t1).
\end{aligned}
\tag{7.1}
$$

7.1.1 The impact of the number of dyads on the conduction velocity

In the following experiment tissue of size $6\ mm \times 6\ mm \times 24\ mm$ is stimulated at the entire x-y plane at time $t = 0\ ms$. Each GPU performs computations on 1728 cells. This is the largest possible amount of cells, containing 10000 dyads, that can be simulated on a single GPU on Abel supercomputer. The experiment is performed on two compute nodes. The size of a time step is set to 0.05 ms and diffusion terms in Equation (2.16) are discretized using a fixed spatial mesh resolution $dx = dy = dz = 0.5\ mm$. We simulate one cardiac beat of 500 ms and measure the conduction velocity under healthy conditions. The goal of the experiment is to verify the impact of the number of dyads in a cell on the conduction velocity.

Our experiment considers four different cases: simulation using cells with 10000 dyads, 1000 dyads, 100 dyads and 10 dyads. The experiment has shown that the time for which $cell_1$ and $cell_2$ are triggered is the same for 10000, 1000 and 100 dyads: $cell_1$ is triggered at $t1 = 10.65\ ms$ and $cell_2$ is triggered at $t2 = 33.6\ ms$. The CV is computed as

$$CV = (48/2 \times 0.5)/(33.6 - 10.65) = 0.523 \text{ m/sec.} \tag{7.2}$$

For the simulation of 10 dyads, the time for which $cell_1$ and $cell_2$ are triggered are $t1 = 10.65\ ms$ and $t2 = 33.70\ ms$ respectively. The CV in this case is

$$CV = (48/2 \times 0.5)/(33.6 - 10.70) = 0.524 \text{ m/sec.} \tag{7.3}$$

This means that CV is similar for different numbers of dyads. Thus, we can conclude that the number of dyads in a cell does not have an impact on conduction velocity. Moreover, the CV we obtained in our experiments is approximately 0.5 m/sec, which validates the correctness of our model.

7.1.2 Conduction velocity as a function of diffusion coefficient

One of the main parameters that determines the conduction properties of the tissue model are the diffusion coefficients (or conductivities), which are correlated with the conduction efficiency. The diffusion coefficients, given in our model, are estimated based on the experimental measurements, which lead to the reasonable CV. The exact biophysical mechanism of cardiac conduction is unknown, so the suitable parameters depend on the mathematical model and numerical scheme. The common approach of selecting the parameters is to change diffusion coefficients and make sure that it results in plausible CV. Our aim is to investigate how CV changes as a function of the diffusion coefficient.

We stimulate the x-y plane, which means that action potential propagates in the z-direction. Thus, to observe effects of changing diffusion coefficient, we need to change Dz parameter in Equation (2.16), while keeping Dx and Dy constant.

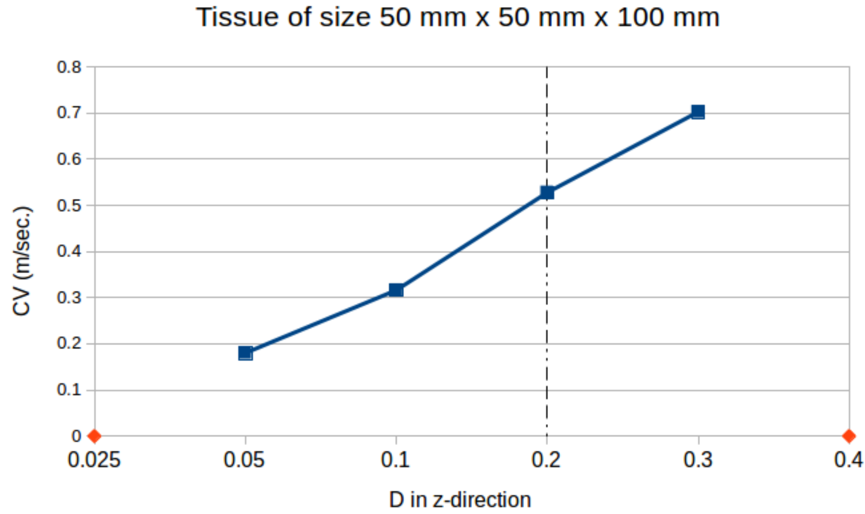

Figure 7.1: Conduction velocity as a function of diffusion coefficient for the tissue of size 50 $mm \times 50$ $mm \times 100$ mm, where each cell contains 10 dyads. The estimated value of Dz for our model is 0.2 and it is indicated by the dashed vertical line. With $Dz = 0.025$ and $Dz = 0.4$ conduction fails. Failing values of Dz are indicated by diamonds.

Tissue of size 50 mm x 50 mm x 100 mm

In this experiment we use a tissue of size 50 $mm \times 50$ $mm \times 100$ mm, where each cell contains 10 dyads, and we measure the conduction velocity for different values of the diffusion coefficient in the z-direction. A time step is defined as $dt = 0.05$ ms and a spatial mesh resolution is $dx = dy = dz = 0.5$ mm.

The results are shown in Figure 7.1, where CV is visualized as a function of voltage diffusion coefficient Dz. With Dz equal to 0.2, which is the estimated value for our model, we obtain CV equal to *0.53* m/sec, which is consistent with the reported value [29]. As expected, the conduction velocity increases with the higher values of conductivity and decreases with values lower than the estimated conductivity. With both low ($Dz = 0.025$) and high values ($Dz = 0.4$) of the diffusion coefficient conduction fails. In our experiment neither $cell_1$ nor $cell_2$ is triggered with the $Dz = 0.4$. At the high values electronic currents between cells are too fast and dissipate rapidly from the stimulated plane. Thus, the cells are unable to excite since there is not enough current to activate them. During pathological conditions CV decreases. In our experiment only $cell_1$ fired with the $Dz = 0.025$. The low values of CV may cause a propagation block and the conduction fails.

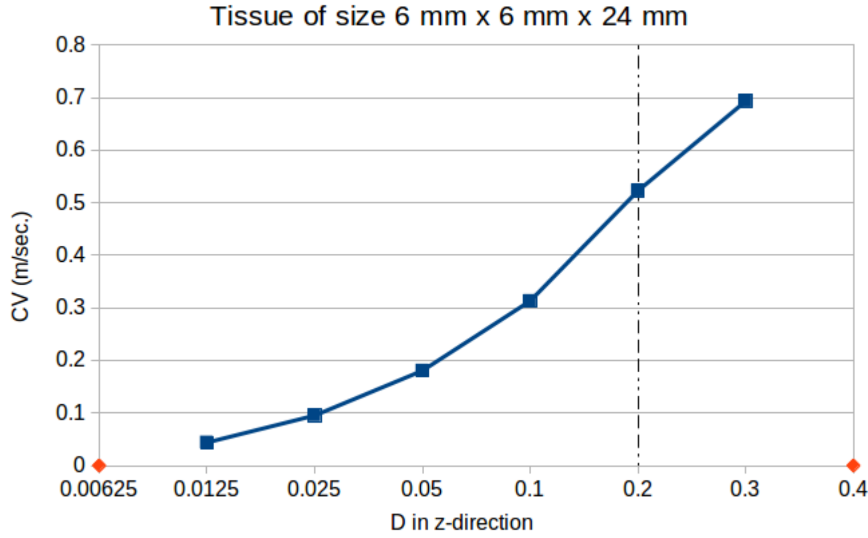

Figure 7.2: Conduction velocity as a function of diffusion coefficient for the tissue of size 6 $mm \times 6$ $mm \times 24$ mm, where each cell contains 10000 dyads. The estimated value of Dz for our model is0.2 and it is indicated by the dashed vertical line. With $Dz = 0.4$ and $Dz = 0.00625$ conduction fails. Failing values of Dz are indicated by diamonds.

Tissue of size 6 mm x 6 mm x 24 mm

We perform the same experiment on tissue of size 6 $mm \times 6$ $mm \times 24$ mm, where each cell contains 10000 dyads. The results are shown in Figure 7.2. The values and behavior of CV are similar to the previous experiment, where we used the tissue of size 50 $mm \times 50$ $mm \times 100$ mm. CV increases using higher values of conductivity and it slows down using lower values. Moreover, conduction fails with $Dz = 0.00625$ and $Dz = 0.4$. Thus, CV is independent of the dimensions of the tissue. For both small and large sizes of the tissue, conduction fails with $Dz = 0.4$. However, for the small size of tissue CV is able to exceed lower values and conduction fails only with $Dz = 0.00625$. The cells involved in the experiment are located closer to the stimulated plane and the distance between them is smaller than in the larger tissue. For this reason, we encounter propagation block for the smaller value of conductivity.

A closer look at the failing conduction

We perform tissue scale simulations using three different values of conductivity: $Dz = 0.025$, $Dz = 0.2$ and $Dz = 0.4$ to analyze the action potential propagation in case of the failing conduction. A tissue of size 50 $mm \times 50$ $mm \times 100$ mm, where each cell contains 10 dyads is used for this experiment. The xy-plane was stimulated at $t = 0$ ms, the selected size of a time step is $dt = 0.05$ ms and spatial mesh resolution is $dx = dy = dz = 0.5$ mm.

First, we perform the simulation of the tissue with the estimated value of Dz to see how the action potential propagates under normal conditions. The

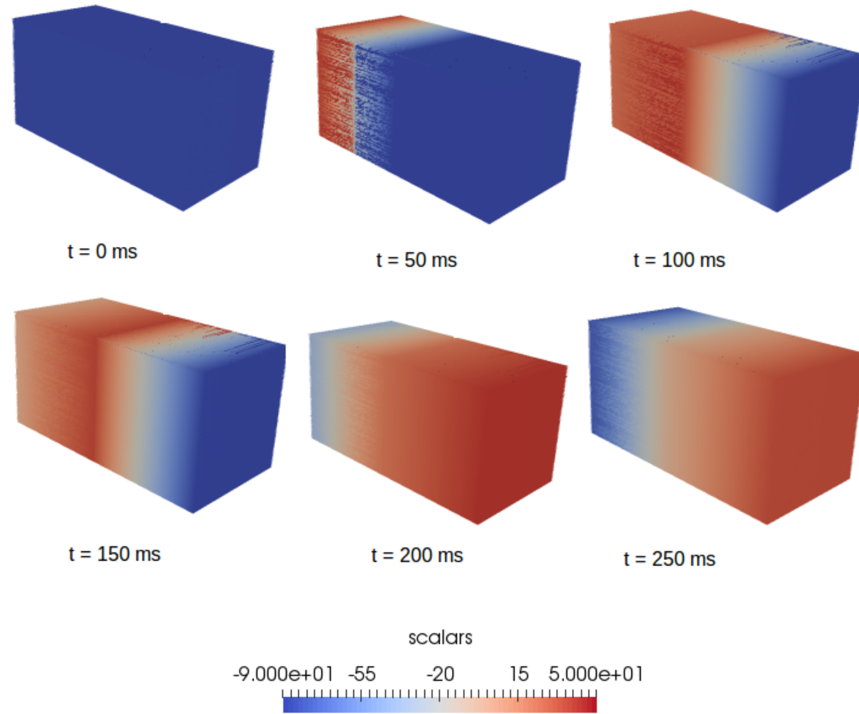

Figure 7.3: Activation pattern in 3D human ventricular tissue using diffusion coefficient value equal to *0.2*. The xy-plane is stimulated at $t = 0\ ms$. The excitation wave travels in the z-direction.

snapshots of membrane potential with $Dz = 0.2$ are shown in Figure 7.3. The entire tissue is depolarized between 150 ms and 200 ms, more precisely at $100\ mm\ /\ 0.53\ \frac{mm}{ms} = 189\ ms$. At 200 ms the membrane voltage values start to decrease and the entire tissue is repolarized.

Figure 7.4 visualizes snapshots of the membrane potential associated with $Dz = 0.4$. As we can see, there is a little increase in the voltage value at $t = 10\ ms$, but the membrane voltage does not reach -70mV. As a consequence, the cells are not triggered and therefore, the tissue is not depolarized.

Snapshots of the membrane potential associated with $Dz = 0.025$ are shown in Figure 7.5. From the figure, we clearly see the propagation block. The action potential is propagating until $t = 50\ ms$. Then it blocks leaving a part of the tissue not depolarized. At $t = 150\ ms$ we see the begining of the repolarization of a part of the tissue.

Results

The computational models of electrophysiology of the cardiac tissue incorporate information about excitability of the cells and conduction of the tissue. Exact values of the most of the parameters are unknown; they are dependent on

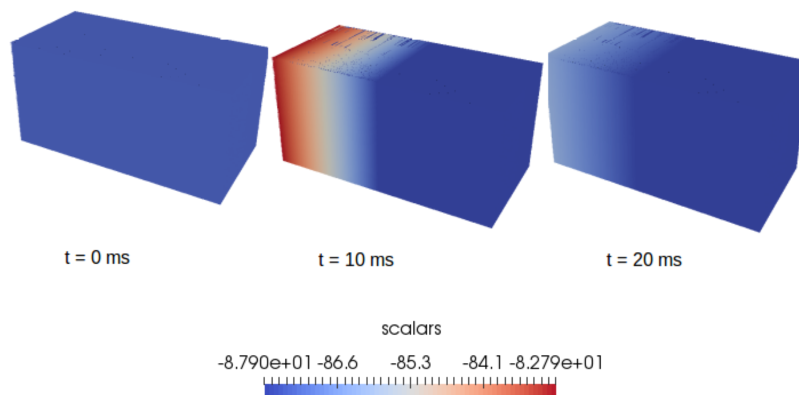

Figure 7.4: Activation pattern in 3D human ventricular tissue using diffusion coefficient value equal to 0.4. The xy-plane is stimulated at $t = 0\ ms$. The voltage does not reach -70mV and propagation fails.

the selected mathematical model and other parameters. The importance of the correctness of the computational model is crucial. We have validated our model and shown that CV is independent of the number of dyads in a cell and tissue dimensions. Thus, CV is only dependent on membrane properties. Moreover, CV decreases during the pathological conditions and fails at low and high values of the diffusion coefficient.

7.2 A possible defibrillation strategy that targets RyR openings

In this experiment we use a tissue of size $12\ mm \times 12\ mm \times 12\ mm$. At time $t = 50\ ms$ we open all RyRs for $100\ ms$, which causes a spontaneous action potential. The time step is defined as $dt = 0.05\ ms$, a spatial mesh resolution is $dx = dy = dz = 0.5\ mm$ and the cycle length is 500 ms. The goal of this experiment is to show the importance of having 10000 dyads in a cell and analyze the outcome of defibrillation that targets RyR openings.

Spontaneous and paced action potential

First, we perform a simulation of a cardiac tissue under normal conditions. To cause a paced action potential, we stimulated the y-z plane at $t = 50\ ms$. The paced action potential in a cell is shown in Figure 7.7 (A). During normal paced heart beat an action potential causes depolarization. When the membrane voltage reaches positive values, L-type channels became active and extracellular calcium flows into the dyad. The calcium activates RyRs, which release a greater calcium flow from the calcium stores inside the sarcoplasmic reticulum.

Now, instead of providing the stimulus to the tissue, we open RyRs at $t = 50\ ms$ for $100\ ms$ as we have described above. Open RyRs release a

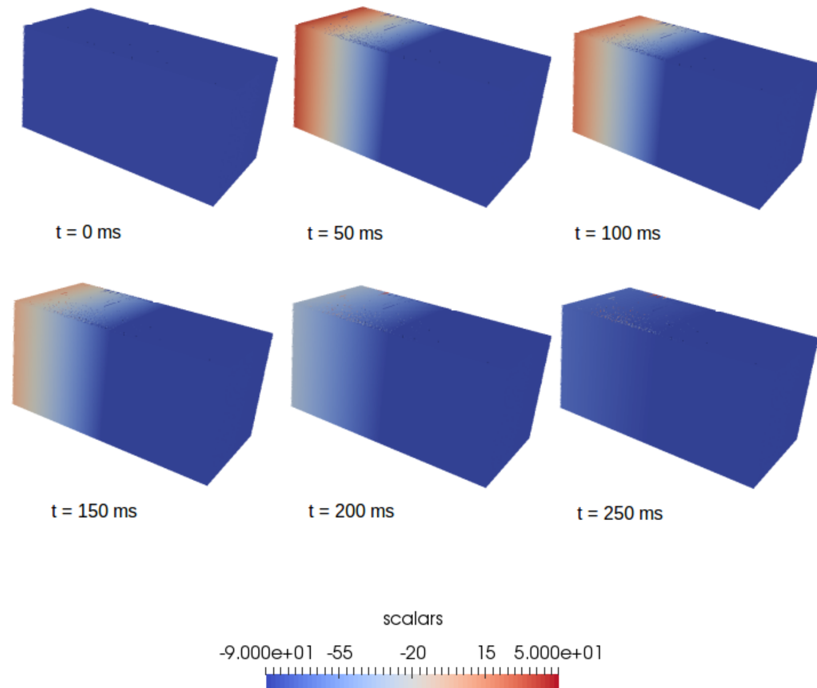

Figure 7.5: Activation pattern in 3D human ventricular tissue using diffusion coefficient value equal to 0.025. The xy-plane is stimulated at $t = 0\ ms$. The excitation wave travels in the z-direction, but it does not reach the edge of the tissue. The propagation stops inside the tissue.

high calcium flow, which causes a spontaneous action potential in a cell. The spontaneous action potential is shown in Figure 7.8 (A). Initially, intracellular calcium has a steep rise when there is a steep depletion of JSR. However, at $t = 50\ ms$ all RyRs in a cell are opened for 100 ms. Thus, we have a slower calcium release as JSR is refilled from NSR, and we have a linear increase of calcium concentration until the RyRs are closed. The calcium release activates Na-Ca exchange current, which slowly increases voltage at the beginning. When the voltage reaches a certain threshold, L-type Ca current and fast Na currents are activated and this forms a spontaneous action potential.

Spontaneous heart beat

In the previous paragraph, we have shown that abnormal RyRs openings can generate a spontaneous action potential in a cell. Figure 7.6 visualizes the results obtained at the tissue scale by opening all RyRs for 100 ms. As we can see all cells are depolarized and repolarized simultaneously, which forms a spontaneous heart beat. This proves that abnormal RyRs openings lead to the spontaneous heart beats.

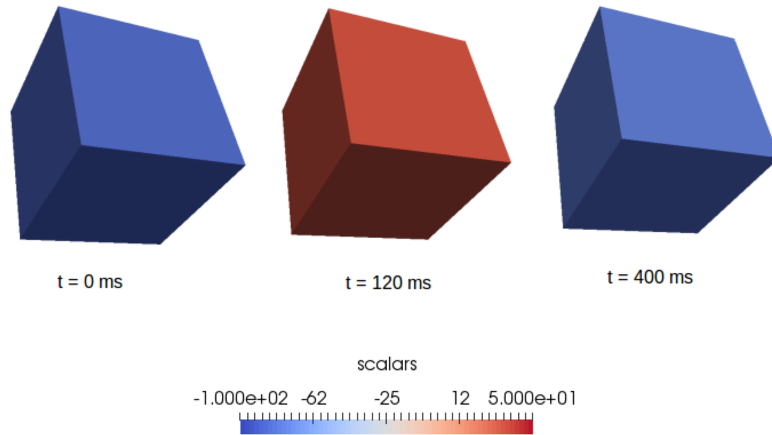

Figure 7.6: Spontaneous heart beat. At $t = 50$ msall RyRs are open for 100 ms in the tissue of size 12 $mm \times 12$ $mm \times 12$ mm, where each cell consists of 10000 dyads. The whole tissue is excited simultaneously at $t = 120$ ms.

Effect of the number of dyads

From Figure 7.8 we can see that the spontaneous action potential is fired when a cell contains 10000 dyads, but with a smaller amount of dyads, spontaneous action potential fails. A dyad is a small region in a cell where the calcium release occurs. When we employ more dyads in a cell, more RyRs are open. Thus, we have a larger calcium release, which can be seen from the plots of intracellular calcium concentrations. Having 10000 dyads and all of them open calcium concentration exceeds the normal range, which is between $0.1\mu M$ and $1\mu M$. On the other hand, small calcium release lowers the ability to trigger an action potential in a cell.

In Figure 7.7 we can also see that a larger amount of calcium release occurs with a larger number of dyads in a cell. As a consequence, with a smaller number of dyads, we have a shorter repolarization phase.

This reflects the importance of having all 10000 dyads in a computational cell model in order to get accurate simulation results.

Results

Under abnormal conditions, when RyRs are open for some time, calcium release from RyRs can activate a Na-Ca exchange current, which causes depolarization. When the voltage reaches a certain threshold, the Na current becomes activated and this forms an action potential. We have tested and proven that abnormal RyRs openings can cause the spontaneous action potential, which can lead to arrhythmias. Moreover, we have verified that in order to receive accurate results from the simulation, we indeed need to have a detailed cell model, which contains 10000 dyads.

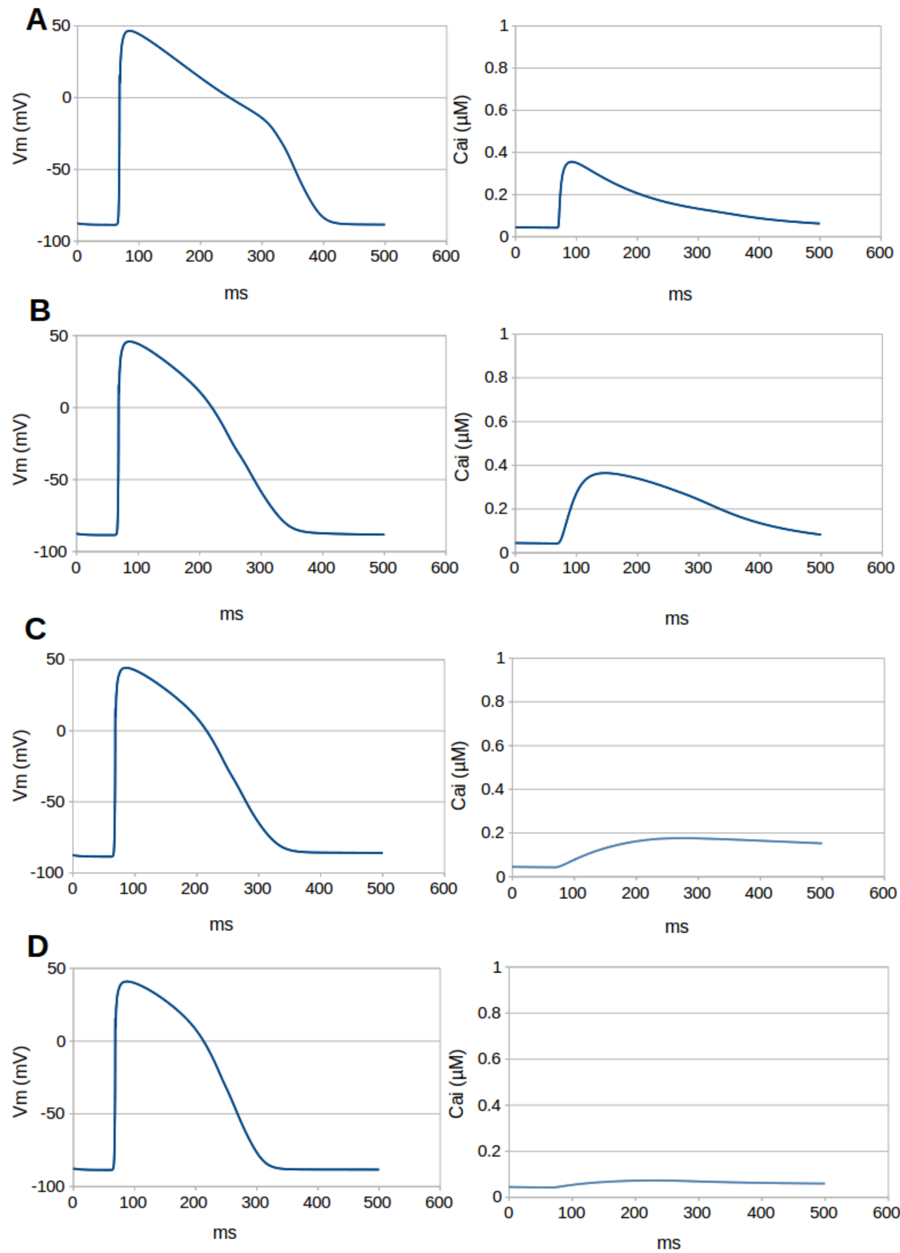

Figure 7.7: Paced beat of a cell. The figure shows paced action potential (left) and intracellular calcium concentration (right). A, B, C, D and E represent the results of simulations where 10000, 1000, 100 and 10 dyads were employed respectively. The tissue was stimulated at $t = 50\ ms$, basic cycle length is $500\ ms$ under healthy conditions.

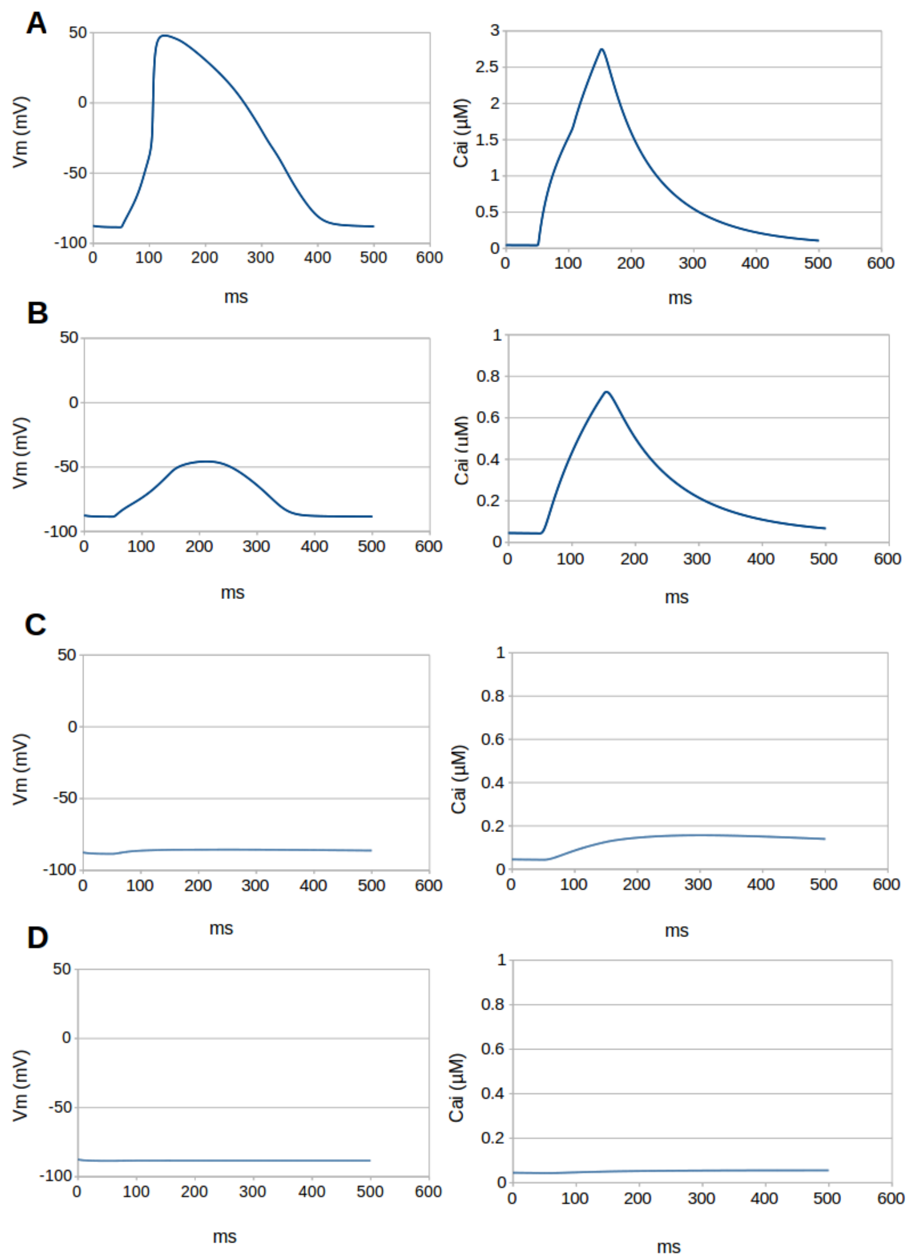

Figure 7.8: Spontaneous beat of a cell. The figure shows spontaneous action potential (left) and intracellular calcium concentration (right). A, B, C, D and E represent the results of simulations where 10000, 1000, 100 and 10 dyads were employed respectively. At $t = 50$ ms all RyRs were open for 100 ms.

Chapter 8

Discussion and Conclusion

8.1 Discussion

The general aim of using the computational models of the human heart is to develop a precise perception of the role played by disturbances of calcium handling in cardiac arrhythmias. The multiscale myocyte model [22] was adopted to develop a 3D Tissue-Scale model which can be used to investigate various dysfunctions of subcellular calcium release processes and action potential propagation. The same model was implemented and tested on CPU and Xeon Phis in previous research [30, 31, 32]. This thesis presents a novel approach of increasing performance of the computation by integrating GPU into the 3D Tissue-Scale model of calcium handling and electrical activity in the human cardiac ventricle. The high-efficiency simulator which uses hybrid CPU-GPU computing has been implemented and tested. Moreover, physiologically realistic simulations were carried out in order to evaluate the selected model and show the scientific purpose of the cardiac simulator.

During the last years, a lot of research has been done to investigate the efficiency of the computations, such as sparse matrix solvers [3], implemented using GPU accelerators. However, in most of the cases GPU was used for one specific computational task, while our application consists of several distinct computational strategies and algorithms. This requires different adjustments of the implementation and optimization techniques for each of the functions executed on the device. In this thesis GPU accelerator is applied to the detailed simulator that has been used in cardiac research and requires good computational precision.

The optimized implementation demonstrates a significant speedup of simulations of human cardiac ventricle tissue. We have shown that the optimized reduction and diffusion operations can result in significant performance gain. After performing various optimizations to the simulator, we have certified that the large performance improvements come from the usage of shared memory, which is not surprising because most of the kernel functions in our simulator are memory traffic bound. In addition, we have demonstrated the weakness of the power function and division operation. An algorithmic improvement, such as implemented power function, leads to decreased execution time. In this thesis we have demonstrated the dependency between the number of cardiac cells on the device and the computational performance. Having one cell on GPU is

inefficient, and thus in order to achieve full utilization of the GPU, we need to use large-scale data or multiple cells.

In [31] the same 3D Tissue-Scale model was implemented to run on heterogeneous CPU - Xeon Phi systems. Compared to the optimized CPU implementation results, which have been reported in [31], our GPU implementation is approximately 4 times faster. The usage of both CPU and Phi halves the running time, but it still remains 2 times slower than our GPU simulator. Even including two Phis is not enough to exceed the GPU performance. 3D cardiac Tissue-Scale simulator implemented on GPU shows better performance than all previous implementations of this simulator [30, 31, 32]. On the other hand, the preliminary experiments of the tissue simulator on the Knights Landing generation of Xeon Phis have demonstrated that the performance greater than obtained on Kepler GPU can be achieved.

The benefits of using GPU for cardiac computing have already been shown in other publications, such as simulations of cardiac electrophysiology equations in the tissue [46]. It confirmed GPU to be about $30 \sim 40$ times faster than CPU. However, our model is more complex involving not only electrophysiology of the heart but also calcium handling at subcellular level, which requires a larger amount of computations to be performed.

In [38] another model of cardiac cell and calcium signaling has been implemented using GPU accelerator. However, all the computations are performed on a Fermi GPU, which is an older architecture with lower CUDA compute capability than Kepler. In this implementation the reduction of Ca fluxes and concentration was also performed on the GPU. The basic parallel reduction approach, which as we have seen requires a lot of thread synchronizations, was used to implement reduction operation. The authors of the previously mentioned paper have chosen to perform the ODEs of the action potential on CPU while in our implementation we solve the PDEs of the action potential on CPU. In [38] computations are performed on the cellular scale, whereas our implementation contributes with computations on the tissue-scale.

The ability of the parallel 3D Tissue simulator to predict normal ventricle tissue patterns, action potential, calcium values and currents consistent with the published results were shown in [31]. In this thesis we have provided additional simulations that verified the correctness of the chosen cardiac simulator model and its structure.

The performed tissue simulations in this thesis have shown that the conduction velocity is independent of the number of dyads and dimension of the tissue. This confirms that conduction velocity depends on the excitability of the membrane as stated in [29]. Furthermore, we found out that conduction fails at low values of the diffusion coefficient and the tissue cannot be depolarized. At high values of the diffusion coefficient conduction fails, because there is not enough current to activate the cells. In addition, conduction slows down during pathological conditions, which is consistent with [17]. Finally, we have used our implementation of the computational model of the human heart tissue to test and prove the hypothesis that abnormal RyRs openings can cause spontaneous heart beats. Dysfunctions of RyRs and their relation to arrhythmogenesis have been examined extensively in the literature [27, 47]. Our findings suggest that the simulator can be used as a tool to investigate electrophysiology and calcium release processes in the heart, which will help to understand the multiscale mechanisms of cardiac arrhythmias.

Due to the limited GPU memory and our limited access to the large-scale GPU clusters, we were not able to test the cardiac tissue simulator on the large-scale, which would involve millions of cells containing 10000 dyads per cell. Even a single cell implementation requires hundreds of variables and significant amount of computation using our selected detailed cell model.

Further studies of the attainable performance of the implementation on the larger and novel architectures are needed. Our future work will focus on refining and testing simulator on the NVIDIA Pascal architecture, which is equipped with 16 GB of memory, 5.3 Tflops of double precision compute power and 480 GB/s memory bandwidth. Having 16 GB of memory we would be able to perform simulations of over 5000 cells on one GPU, which is almost three times more cells than we were able to initialize on Kepler. Moreover, we expect to test the simulator on the new generation GPU based on Volta architecture [34] with memory bandwidth reaching 900GB/s, which gives 95 % greater bandwidth efficiency than on Pascal architecture. Furthermore, we intend to incorporate CPU into the computation by distributing the tissue domain between multiple CPUs and GPUs in our future work. Another possible approach is to shuffle the data between CPU and GPU to increase the simulation size.

8.2 Conclusion

Models of electrophysiology and calcium handling in the human heart have played an important role in understanding the causes of arrhythmias that occur on tissue and organ scale. Detailed large scale models require a lot of computational power, which makes it challenging to perform realistic simulation in reasonable time. In this thesis we have shown that a detailed 3D Tissue-Scale simulator can be accelerated using GPU. We can firmly state that hybrid CPU-GPU computing is faster than a pure CPU computing. This provides a possibility to scientists to use this simulator in order to understand the multiscale mechanism of cardiac arrhythmias originating from the disturbances in calcium handling in a heart. Due to the increasing power of supercomputers, larger scale simulations will be feasible in the future.

List of Tables

3.1 Kepler K20 and K20X specifications 22

6.1 Memory bandwidth and utilization 55
6.2 Execution time of the reduction computation 56
6.3 Execution time of diffusion computation on the dyad-level and kernel details 59
6.4 Device Memory Bandwidth of different optimizations of stencil computation 59
6.5 Performance metrics of L-type channels computation 62
6.6 Optimisation of the RyR probability calculation 62
6.7 Performance metrics of the RyR probability calculation 63
6.8 Performance of the RyR opening computation 63
6.9 Performance of the Ca concentration computation 64

List of Figures

2.1 Ventricular Myocyte model 9
2.2 LCC-RyR interaction 10
2.3 RyR Scheme 11

3.1 CUDA memory and thread hierarchy 19
3.2 Kepler Memory Hierarchy 21
3.3 Roofline model for Kepler K20X 25

4.1 Multiple cell representation of dyadic information 30
4.2 A cell separation mapping 34
4.3 Tissue-Level parallelization 35

6.1 Memory bandwidth for the reduction computation 56
6.2 Number of floating-point operations per second of different optimizations of stencil computation 60
6.3 Effects of the code optimization 66
6.4 Speed of the cell computation on one GPU 67
6.5 Scaling Experiments 69

7.1 Conduction velocity as a function of diffusion coefficient 72
7.2 Conduction velocity as a function of diffusion coefficient 73
7.3 Activation pattern in 3D human ventricular tissue using diffusion coefficient value equal to 0.2 74
7.4 Activation pattern in 3D human ventricular tissue using diffusion coefficient value equal to 0.4 75
7.5 Activation pattern in 3D human ventricular tissue using diffusion coefficient value equal to 0.025 76
7.6 Spontaneous heart beat 77
7.7 Paced beat of a cell 78
7.8 Spontaneous beat of a cell 79

Bibliography

[1] CP Adler and U Costabel. "Cell number in human heart in atrophy, hypertrophy, and under the influence of cytostatics." In: *Recent advances in studies on cardiac structure and metabolism* 6 (1974), pp. 343–355.

[2] Peter Benner et al. "Using hybrid CPU-GPU platforms to accelerate the computation of the matrix sign function". In: *European Conference on Parallel Processing*. Springer. 2009, pp. 132–139.

[3] Jeff Bolz et al. "Sparse matrix solvers on the GPU: conjugate gradients and multigrid". In: *ACM Transactions on Graphics (TOG)*. Vol. 22. 3. ACM. 2003, pp. 917–924.

[4] Jun Chai et al. "Resource-efficient utilization of CPU/GPU-based heterogeneous supercomputers for Bayesian phylogenetic inference". In: *The Journal of Supercomputing* 66.1 (2013), pp. 364–380.

[5] Heping Cheng, WJ Lederer, and Mark B Cannell. "Calcium sparks: elementary events underlying excitation-contraction coupling in heart muscle". PhD thesis. University of Maryland, 1994.

[6] John Cheng, Max Grossman, and Ty McKercher. *Professional Cuda C Programming*. John Wiley & Sons, 2014.

[7] RH Clayton et al. "Models of cardiac tissue electrophysiology: progress, challenges and open questions". In: *Progress in biophysics and molecular biology* 104.1 (2011), pp. 22–48.

[8] Jeffrey S Cook and Neha Gupta. "History of Supercomputing and Supercomputer Centers". In: *Research and Applications in Global Supercomputing*. IGI Global, 2015, pp. 33–55.

[9] Nvidia Corporation. *CUDA Toolkit Documentation v8. 0*. 2017. URL: http://docs.nvidia.com/cuda/index.html.

[10] Nvidia Corporation. *CURAND LIBRARY Programming Guide*. 2016. URL: http://docs.nvidia.com/cuda/pdf/CURAND_Library.pdf.

[11] Nvidia corporation. URL: http://www.nvidia.com/object/cuda_home_new.html.

[12] Nvidia corporation. URL: http://www.nvidia.com/content/pdf/kepler/tesla-k20-passive-bd-06455-001-v07.pdf.

[13] Nvidia corporation. URL: https://computing.llnl.gov/tutorials/linux_clusters/gpu/Tesla-K10K40-datasheet.pdf.

[14] Nvidia corporation. URL: http://www.nvidia.com/object/nsight.html.

[15] Nvidia corporation. URL: https://developer.nvidia.com/nvidia-system-management-interface.

[16] Nvidia corporation. URL: https://nvlabs.github.io/cub/index.html.

[17] Paul F Cranefield, Herman O Klein, and Brian F Hoffman. "Conduction of the cardiac impulse". In: *Circulation Research* 28.2 (1971), pp. 199–219.

[18] DA Eisner et al. "From the ryanodine receptor to cardiac arrhythmias". In: *Circulation Journal* 73.9 (2009), pp. 1561–1567.

[19] ALEXANDRE Fabiato. "Time and calcium dependence of activation and inactivation of calcium-induced release of calcium from the sarcoplasmic reticulum of a skinned canine cardiac Purkinje cell." In: *The Journal of general physiology* 85.2 (1985), pp. 247–289.

[20] CLARA FRANZINI-ARMSTRONG, Feliciano Protasi, and Pierre Tijskens. "The assembly of calcium release units in cardiac muscle". In: *Annals of the New York Academy of Sciences* 1047.1 (2005), pp. 76–85.

[21] Michael Garland et al. "Parallel computing experiences with CUDA". In: *IEEE micro* 28.4 (2008).

[22] Namit Gaur and Yoram Rudy. "Multiscale modeling of calcium cycling in cardiac ventricular myocyte: macroscopic consequences of microscopic dyadic function". In: *Biophysical journal* 100.12 (2011), pp. 2904–2912.

[23] Ananth Grama. *Introduction to parallel computing*. Pearson Education, 2003.

[24] Chris Gregg and Kim Hazelwood. "Where is the data? Why you cannot debate CPU vs. GPU performance without the answer". In: *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*. IEEE. 2011, pp. 134–144.

[25] Mark Harris. *Optimizing Parallel Reduction in CUDA*. 2007. URL: http://docs.nvidia.com/cuda/samples/6_Advanced/reduction/doc/reduction.pdf.

[26] Dawei Jiang et al. "Enhanced store overload–induced Ca2+ release and channel sensitivity to luminal Ca2+ activation are common defects of RyR2 mutations linked to ventricular tachycardia and sudden death". In: *Circulation research* 97.11 (2005), pp. 1173–1181.

[27] Rodolphe P Katra et al. "Ryanodine receptor dysfunction and triggered activity in the heart". In: *American Journal of Physiology-Heart and Circulatory Physiology* 292.5 (2007), H2144–H2151.

[28] Ashfaq A. Khokhar et al. "Heterogeneous computing: Challenges and opportunities". In: *Computer* 26.6 (1993), pp. 18–27.

[29] Andre G Kleber and Yoram Rudy. "Basic mechanisms of cardiac impulse propagation and associated arrhythmias". In: *Physiological reviews* 84.2 (2004), pp. 431–488.

[30] Qiang Lan et al. "Towards Detailed Tissue-Scale 3D Simulations of Electrical Activity and Calcium Handling in the Human Cardiac Ventricle". In: *International Conference on Algorithms and Architectures for Parallel Processing*. Springer. 2015, pp. 79–92.

[31] Johannes Langguth et al. "Accelerating Detailed Tissue-Scale 3D Cardiac Simulations Using Heterogeneous CPU-Xeon Phi Computing". In: *International Journal of Parallel Programming* (2016), pp. 1–23.

[32] Johannes Langguth et al. "Enabling Tissue-Scale Cardiac Simulations Using Heterogeneous Computing on Tianhe-2". In: *Parallel and Distributed Systems (ICPADS), 2016 IEEE 22nd International Conference on.* IEEE. 2016, pp. 843–852.

[33] Justin Luitjens. "Faster Parallel Reduction on Kepler". In: *Parallel Forall (blog)* (2014). URL: https://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler/.

[34] Mark Harris Luke Durant Olivier Giroux and Nick Stam. URL: https://devblogs.nvidia.com/parallelforall/inside-volta/l.

[35] Dariush Mozaffarian et al. "Executive summary: Heart Disease and Stroke Statistics-2016 update: A report from the American Heart Association." In: *Circulation* 133.4 (2016), p. 447.

[36] MPICH. *High-Performance portable MPI.* URL: https://www.mpich.org.

[37] Maruyama Naoya and Takayuki Aoki. "Optimizing stencil computations for NVIDIA Kepler GPUs". In: ().

[38] Michael Nivala et al. "Computational modeling and numerical methods for spatiotemporal calcium cycling in ventricular myocytes". In: *Frontiers in physiology* 3 (2012), p. 114.

[39] C Nvidia. "NVIDIAs next generation CUDA compute architecture: Kepler GK110". In: *Whitepaper* (2012).

[40] Georg Ofenbeck et al. "Applying the roofline model". In: *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on.* IEEE. 2014, pp. 76–85.

[41] Thomas O'Hara et al. "Simulation of the undiseased human cardiac ventricular action potential: model formulation and experimental validation". In: *PLoS Comput Biol* 7.5 (2011), e1002061.

[42] University of Oslo: Abel. URL: http://www.uio.no/english/services/it/research/hpc/abel/.

[43] John D Owens et al. "GPU computing". In: *Proceedings of the IEEE* 96.5 (2008), pp. 879–899.

[44] Zhilin Qu and Alan Garfinkel. "An advanced algorithm for solving partial differential equation in cardiac conduction". In: *IEEE Transactions on Biomedical Engineering* 46.9 (1999), pp. 1166–1168.

[45] Shane Ryoo et al. "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA". In: *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming.* ACM. 2008, pp. 73–82.

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

[47] Mark Scoote and Alan J Williams. "The cardiac ryanodine receptor (calcium release channel)". In: *Cardiovascular research* 56.3 (2002), pp. 359–372.

[48] Charles Severance and Kevin Dowd. *High performance computing.* 2011.

[49] Daniel C Sigg et al. *Cardiac electrophysiology methods and models.* Springer Science & Business Media, 2010.

[50] NP Smith et al. "Multiscale computational modelling of the heart". In: *Acta Numerica* 13 (2004), pp. 371–431.

[51] KHWJ Ten Tusscher et al. "A model for human ventricular tissue". In: *American Journal of Physiology-Heart and Circulatory Physiology* 286.4 (2004), H1573–H1589.

[52] Anamaria Vizitiu et al. "Optimized three-dimensional stencil computation on Fermi and Kepler GPUs". In: *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*. IEEE. 2014, pp. 1–6.

[53] Samuel Williams, Andrew Waterman, and David Patterson. "Roofline: an insightful visual performance model for multicore architectures". In: *Communications of the ACM* 52.4 (2009), pp. 65–76.

[54] Nicolas Wilt. *The CUDA handbook.* A comprehensive guide to GPU programming. Pearson Education, 2013.

[55] Ping Xiang, Yi Yang, and Huiyang Zhou. "Warp-level divergence in GPUs: Characterization, impact, and mitigation". In: *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*. IEEE. 2014, pp. 284–295.