University of Saskatchewan

CMPT 405 Final Report

# Real-Time Heart Simulation

*Submitted By:*
Yinsheng Dong
Yuechegn Rong

*Supervisor:*
Dr. Raymond.J Spiteri
Professor
Director of Centre for High-Performance Computing

Department of Computer Science

April 2019

## Abstract

Heart simulation is becoming more and more important on cardiac electrophysiology since heart disease is taking people's lives every year. According to the most recent report, it claims that thousands of lives are taken by heart disease every year in Canada[4]. Computer simulation of cardiac electrophysiology could help doctors analyze the human's electrical activities of the heart of their patients, and it is also useful for studying and treating with heart diseases such as cardiac dynamic modeling which is useful for studying and treating with arrhythmia[6].

It is difficult to solve the mathematical models of the electrical activities in the heart models. The reason is the complexity of capturing the electrochemical details of the organ. Cardiac cell models describe the potential differences across the cell membrane, and they depend on the complexity of the model, the movement of ions through the cell membrane and the Ion concentrations[7].

Over the past 6 years, there has been a marked increase in the performance and capabilities of graphics processing units. GPUs have performed an integral part of today's mainstream computing system. The modern GPU is containing a powerful graphics engine. Compare to CPUs, GPUs have more superiority such as the high-parallel programmable processor, featuring peak arithmetic and memory bandwidth. GPUs have been positioned to alternatives of traditional microprocessors in the high-performance computer system of the future[11].

On Scientific Computation side, CPUs can solve about $10^8$ ordinary differential equations (ODEs) per second, but GPUs can solve $3 \times 10^{10}$ to $4 \times 10^{10}$ ODEs per second by using parallel programming. Using GPUs could have better performance than the CPUs[6].

In this project, we (Yinsheng and Yuecheng) focused on simulating the heart activities (a piece of heart tissue) and tried to use GPU to accelerate the calculation, to make the tissue active as "real-time" as possible. The tissue consists of vast copies of cell models represented by mathematical heart models solved by numerical methods.

# Acknowledgements

We would like to express my sincerest gratitude to the following people for their guidance and support towards the accomplishment of this project.

**Professor Raymond J. Spiteri**, who is our project supervisor, guided us during the whole period of this project. His comments, feedbacks and encouragement have greatly helped us accomplish this project. Also, his opinion on the difference between academia and industry shaped our mind and helped us understand ourselves better.

**Mr. Wenxian Guo** whose Master Thesis helped us to understand the basic methodologies of our project, and he shared his experience on different approaches to clear our goal when we fell in trouble.

Thank you all.

# Contents

**1 Introduction** **5**

**2 Concepts, Results and System Design** **7**

2.1 Basic concepts . . . 7

2.2 Results, Testing and Evaluation . . . 12

2.3 System Design . . . 15

**3 Experience** **17**

3.1 Our Attempts through this project . . . 17

3.1.1 Code fetching from CellML . . . 17

3.1.2 MATLAB . . . 18

3.1.3 WebGL and GPU programming . . . 18

3.1.4 Python CUDA and OpenCL . . . 19

3.1.5 Julia Programming . . . 19

3.1.6 User Interface . . . 20

3.2 What we learned from this project . . . 21

3.2.1 Mathematics and Electrophysiology . . . 21

3.2.2 Languages and Technologies . . . 21

3.2.3 GPU Programming . . . 21

3.2.4 Pair Programming . . . 21

**4 Conclusion and Future** **22**

4.1 Conclusion . . . 22

4.2 Future Works . . . 22

4.2.1 To achieve GPU computing for more heart cell models . . 22

4.2.2 To implement more mathematical solver methods . . . 23

4.2.3 Database connection . . . 23

4.2.4 To reach the real-time . . . 23

4.2.5 Using 2D/3D modeling . . . 23

# List of Figures

1 Fitzhugh-Nagumo From CellML . . . . . . . . . . . . . . . . . . . . . 12
2 Fitzhugh-Nagumo From Our Project . . . . . . . . . . . . . . . . . . 12
3 Beeler-Reuter From CellML . . . . . . . . . . . . . . . . . . . . . . . . 13
4 Beeler-Reuter From Our Project . . . . . . . . . . . . . . . . . . . . . 13
5 Luo-Rudy From CellML . . . . . . . . . . . . . . . . . . . . . . . . . . 14
6 Luo-Rudy From Our Project . . . . . . . . . . . . . . . . . . . . . . . . 14
7 Ten Tusscher From CellML . . . . . . . . . . . . . . . . . . . . . . . . 14
8 Ten Tusscher From Our Project . . . . . . . . . . . . . . . . . . . . . 14
9 System Structure . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
10 WebGL for Fitzhugh-Nagumo . . . . . . . . . . . . . . . . . . . . . . 18
11 MATLAB for Fitzhugh-Nagumo . . . . . . . . . . . . . . . . . . . . . 18
12 Julia GPU programming for Beeler-Reuter Model 1 . . . . . . . . . 20
13 Julia GPU programming for Beeler-Reuter Model 2 . . . . . . . . . 20
14 User Interface with Vue.js and WebGL . . . . . . . . . . . . . . . . . 20

# 1 Introduction

This project focuses on using numerical methods to solve vast copies of cell models in the computer and trying to reach real-time. Because GPUs contain powerful graphics engine and could perform highly-parallel functional computing, it is possible to reach the real-time simulation by using GPUs to accelerate the calculation. The other goal we want to achieve is to fetch the code of electrophysiological cell models from CellML.org and store into a database so we could use them to do the simulation instead of hard code every cell models we need. Furthermore, Dr.Spiteri also mentioned that single GPU may not reach the real-time goal, thus in this project, we are supposed to use multiple GPUs as well.

The learning objectives of this project are WebGL programming, GPU computing, heart simulation, numerical methods, and learning electrophysiology background knowledge.

We first studied the concepts of heart cell models. The contents include heart cell models' ordinary differential equations, partial differential equations, and numerical methods. In this project, we studied and implemented 4 different heart cell models including Fitzhugh-Nagumo (1964), Beeler-Reuter (1977), Luo-Rudy (1991) and Ten Tusscher (2006, epi), and we studied and implemented 3 numerical methods (Forward Euler, Runge-Kutta 2nd Order, Rush-Larsen), and the Monodomain model that represents the effect a single cell will cause to its neighbor cells. All these models and methods are written in Julia language, and a static figure or gif could be plotted for each model and each method.

To fetch the electrophysiology cell models from CellML.org is also one of the project approaches. We wanted to use existing heart cell models instead of writing the cell models by hand. We attempted CellML official SDK and 3rd party tool kits in this part.

We tried different ways for GPU programming. Our first attempt is WebGL, which is a JavaScript API for rendering interactive 3D and 2D graphics within any compatible web browser without the suing of plug-ins. We also attempted OpenCL and CUDA by using Python. However, all of the language we mentioned above have different limitations. We finally tried Julia with CUDA (CuArrays) for doing GPU computations. Julia not only retains friendly scientific computing environments but also has flexibilities to support using other computer languages such as Python, C, and MATLAB.

In the following chapters, we will first introduce the concepts of the project, the software design and the results we get. Next, we will talk about our experience during the development of this project, including our attempts, the challenges we faced and the things we learned. Finally, we will conclude this project and talk about future works.

# 2 Concepts, Results and System Design

We will introduce the basic concepts of the project, and we will give the results and testing and evaluation of the project. Then we will talk about the system design.

## 2.1 Basic concepts

A piece of tissue consists of vast copies of cells. Thus, to simulate the activity of a piece of tissue, people could build many copies of cell models (which is represented by mathematical models), simulate the activity of each model and combine them together to reach the goal.

In this project, we build many copies of cell mathematical models, use numerical methods to solve the mathematical equations in the models to simulate their activities separately and use the monodomain model to combine them together. The whole project could be split into three parts: cell models, the monodomain model, and numerical methods.

Cells could be represented by mathematical models, which consists of many constants and ordinary differential equations.

Each cell model could be separated into two parts: transmembrane potential ($v$) and states ($s$). The transmembrane potential variable is the main variable we need, which represents the electrical activity of the cell over time. The state variables are the supporting variables that help to calculate the transmembrane potential. It could also be separated into two parts: gating variables and non-gating variables. Each variable has an ordinary differential equation.

There are many different cell models. For example, Fitzhuge-Nagumo (1964) model has 2 ordinary differential equations in total. It is written as:

$$I = \begin{cases} -80 & \text{if } (t \geq 0) \; and \; (t \leq 0.5) \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

$$\tfrac{dv}{dt} = 1 \times (v(v-\alpha) \times (1-v) - w + I) \tag{2}$$

$$\tfrac{dw}{dt} = 1 \times \varepsilon \times (v - \gamma \times w) \tag{3}$$

The equations could be found using OpenCOR.

One important step in this project is to hard code the cell models in Julia. Here is an example code of the fast sodium current J gate in Luo-Rudy model in Julia language.

```
# Right hand side of fast sodium current j gate
function J_RHS(j, v)
    if v < -40.0
        alpha = (-127140.0 * exp(0.2444 * v) - (0.00003474 *
        exp(-0.04391 * v))) * (v + 37.78) / (1.0 +
        exp(0.311 * (v + 79.23)))
        beta = 0.1212 * exp(-0.01052 * v) / (1.0 + exp(-0.1378 *
        (v + 40.14)))
    else
        alpha = 0.0
        beta = 0.3 * exp(-0.0000002535 * v) / (1.0 + exp(-0.1 *
        (v + 32.0)))
    end
    return alpha * (1 - g) - beta * g
end
```

We could also do the GPU programming with a little modification

```
# Right hand side of fast sodium current j gate by using
# GPU programming
using CuArrays, CUDAnative

# An GPU array which store the J variable
gpu_array = cufills(j_initial, N)
function J_RHS(j, v)
    # the math operations need to declear the CUDA
    if v < -40.0
        alpha = (-127140.0 * CUDAnative.exp(0.2444 * v)
        - (0.00003474 *
        CUDAnative.exp(-0.04391 * v))) * (v + 37.78)
        / (1.0 + CUDAnative.exp(0.311 * (v + 79.23)))
        beta = 0.1212 * CUDAnative.exp(-0.01052 * v)
        / (1.0 + CUDAnative.exp(-0.1378 *
        (v + 40.14)))
    else
        alpha = 0.0
        beta = 0.3 * CUDAnative.exp(-0.0000002535 * v)
        / (1.0 + CUDAnative.exp(-0.1 * (v + 32.0)))
    end
    j = alpha * (1 - g) - beta * g

    #push the j value to gpu_array
    push!(gpu_array, j)

    #since kernel cannot return value, then just nothing
    nothing
end
```

In total, 4 models are hardcoded in this project: Fitzhuge-Nagumo (1964), Beeler-Reuter (1977), Luo-Rudy (1991) and Ten Tusscher (2006, epi).

Monodomain model could be used to represent the communication between cells. In the real world, cells in tissue do not act alone; the activity of a cell will affect its neighbors.

Monodomain model, which is a mathematical equation, is applied to every cell in this project. It takes the transmembrane potential of the cell and its neighbors' as inputs, calculate and give a new transmembrane potential to the cell.

The monodomain model could be written as:

$$\begin{gathered}\chi C_m v_t = \chi I_{ion}(t,x,v,s) + I_{stim}(x,t) + \tfrac{\lambda}{1+\lambda}(\sigma_i v_x)_x \\ s_t = g(t,x,v,s)\end{gathered} \tag{4}$$

where $v_t$ is the transmembrane potential and $s_t$ is the state of the cell. [3]

Numerical methods are used to solve ordinary differential equations (and system of ODEs). Given a step size t and the variable value $x_i$ at $t = i$, it could calculate the approximate value of $x_i + 1$ at $t = i + t$.

One important thing we need to consider in this project is the stability of the numerical methods. Numerical methods with higher stability might be more expensive than those with lower stability in execution time, however, it could significantly enlarge the step size and thus reduces the recursion time.

3 numerical methods are applied in this project to each model: Forward Euler method, Runge-Kutta second Order (Heun's) method, and Rush-Larsen method.

- Forward Euler Method:

$$\begin{gathered}y^{'}(t) = f(t, y(t)), \quad y(t_0) = y_0 \\ y_{n+1} = y_n + hf(t_n, y_n)\end{gathered} \tag{5}$$

- Runge-Kutta 2nd Order (Heun's) Method:

$$\begin{gathered}y^{'}(t) = f(t, y(t)), \quad y(t_0) = y_0 \\ \tilde{y}_{i+1} = y_i + hf(t_i, y_i) \\ y_{i+1} = y_i + \tfrac{h}{2}[f(t_i, y_i) + f(t_{i+1}, \tilde{y}_{i+1})\end{gathered} \tag{6}$$

- Rush-Larsen Method:

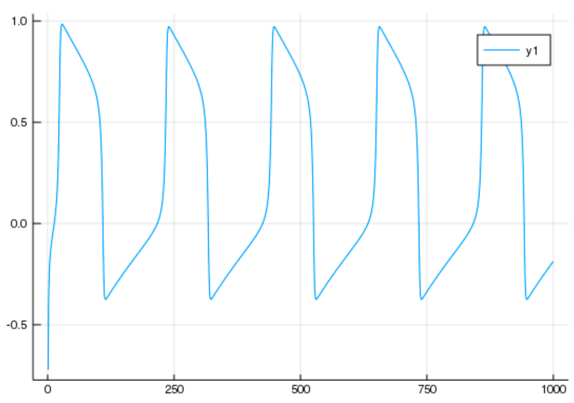
$$\frac{dg}{dt} = \frac{g_\infty - g}{\tau_g}$$
$$g_\infty = \frac{\alpha_g}{\alpha_g + \beta_g}, \quad \tau_g = \frac{1}{\alpha_g + \beta_g}$$
$$g_n = g_\infty + (g_{n-1} - g_\infty)e^{-\frac{\Delta t}{\tau_g}} \tag{7}$$

To do a simulation of a piece of heart tissue, we first build many copies of a cell model, set a time range and a step size. We apply the numerical method to each ODE in a single cell, calculate the new value of the variables and then calculate the monodomain model in each recursion. The total number of recursions the system need to execute is the duration of the time range divided by the step size (which is the simulation process for one model), times the number of copies of the cell model. If the process could be done faster than the duration of the time range, then we can say we reach real-time.

The number of cell models must be large enough to simulate a piece of tissue.

## 2.2 Results, Testing and Evaluation

We used our home computer for testing. The Processor is Intel Core i7-4770K CPU, the graphics card we used is Nvidia GeForce GTX 670. We run our code on Ubuntu 18.04.2 LTS. The benchmark tool we used is Julia "Tests.jl" and "Benchmark.jl" packages for testing CPU running time and memories usages. In this part, we will give a plot from our project and another plot from CellML for each model. The CellML plot is screenshot by hand from OpenCOR software, which is a official toolkit of CellML.org.



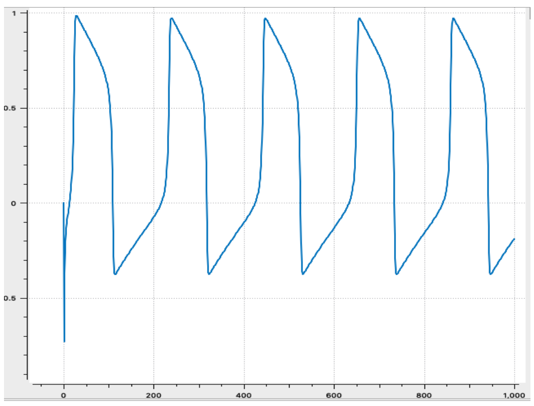
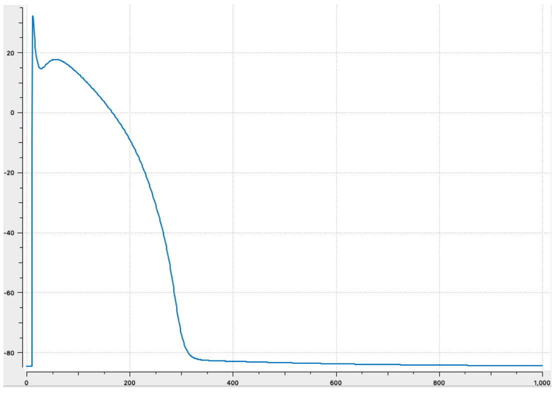


**Figure 1:** Fitzhugh-Nagumo From CellML

**Figure 2:** Fitzhugh-Nagumo From Our Project

Fig 1 (left) and Fig 2 (right) Fitzhugh-Nagumo model, 1964, on $timerange = 1,000ms$, $stepsize = 0.01ms$, $pointinterval = 1ms$, method we used Forward Euler. Fig 1 is plotted from CellML, and Fig 2 is plotted from our project.

Fitzhugh-Nagumo model, 1964 is fairly simple, it only has two ODEs in total. In the test, numerical method was Forward Euler, $numberofmodels = 10,000$, $timerange = 10,000ms$ and $stepsize = 0.01ms$. Thus, the recursion time in the test is $10,000 \times 10,000 \div 0.01 = 10$ billion times, and the calculation process was finished in 35.35844 seconds.



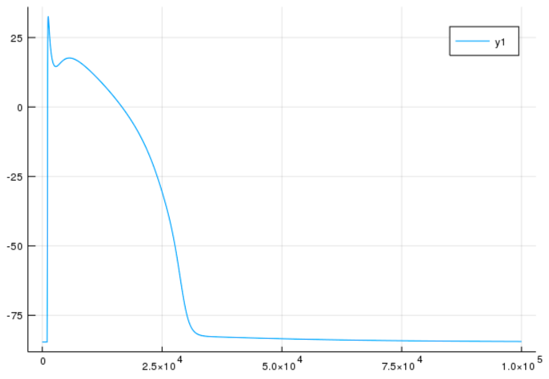
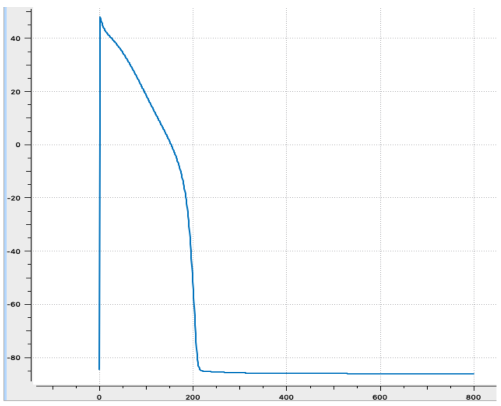


**Figure 3:** Beeler-Reuter From CellML

**Figure 4:** Beeler-Reuter From Our Project

Fig 3 (left) and Fig 4 (right). Beeler-Reuter model, 1977, on $timerange = 1,000ms$. Figure 3 plotted from CellML, method = CVODE, point interval = 1ms, and Figure 4 plotted from our project, method = Rush-Larsen, step size = 0.1ms, point interval = 0.1ms

Since Rush-Larsen method has high stability, it could have a smaller step size than the Forward Euler method. We set the step size to 0.1ms for Rush-Larsen (Rush-Larsen could bear step $size = 1.0ms$ and the model could still give the correct plot, however, the precision is fairly low in this case) and 0.01ms for the Forward Euler since any step size larger than 0.0253ms (result from experiment we made in our project) will lead to the wrong plot when processing Beeler-Reuter model, 1977. The recursion time for Rush-Larsen method is $10,000 \times 100 \div 0.1 = 10$ million times, finished in 6.144499 seconds, while the recursion time for Forward Euler method is $10,000 \times 100 \div 0.01 = 100$ million times, finished in 34.613153 seconds. We could see that applying the Rush-Larsen method is almost six times faster than the Forward Euler method, even it is more complex than the Forward Euler method.



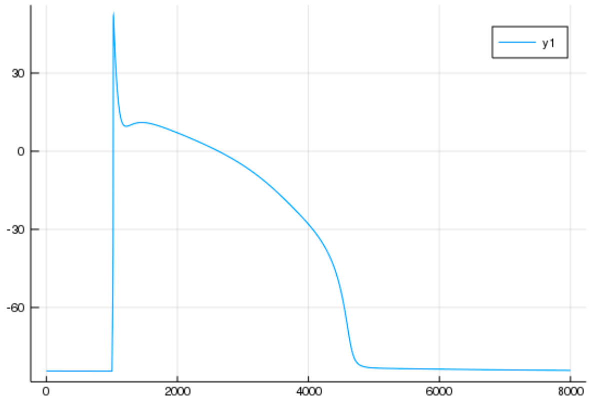
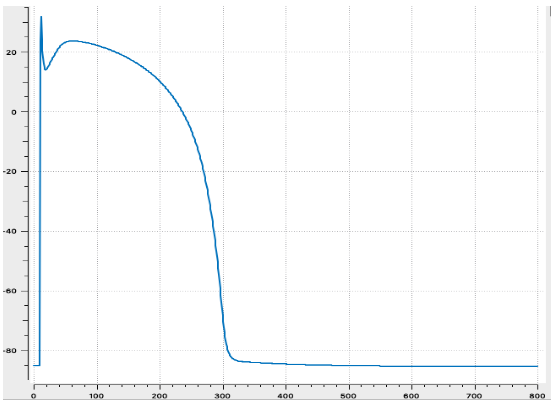


**Figure 5:** Luo-Rudy From CellML

**Figure 6:** Luo-Rudy From Our Project

Figure 5(left), 6(right). Luo-Rudy model, 1991, on time $range = 800ms$. Fig 5 plotted from CellML, $method = CVODE$, $pointinterval = 1ms$ Fig 6 plotted from our project, $method = Rush - Larsen$, $stepsize = 0.1ms$, $pointinterval = 0.1ms$

The model size, time range, step size and recursion time are the same as in the Beeler-Reuter model. Forward Euler in Luo-Rudy model cost 35.028249 seconds, while Rush-Larsen cost 6.890409 seconds.



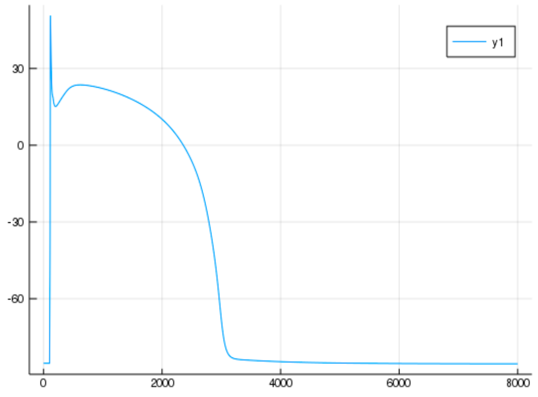


**Figure 7:** Ten Tusscher From CellML

**Figure 8:** Ten Tusscher From Our Project

Fig 7 (left) and Fig 8(right). Ten Tusscher model (epi), 2006, on time range = 800ms. Figure 7 plotted from CellML, $method = CVODE$, $pointinterval = 1ms$ Figure 8 plotted from our project, $method = Rush - Larsen$, $stepsize = 0.1ms$, point $interval = 0.1ms$

The Forward Euler in the Ten Tusscher model faces a more crucial stability problem. The step size has to be set to maximum 0.0022 milliseconds, otherwise, it will lead to the wrong result. The number of models is set to 100, the time range is still 10,000ms, but the step size is set to 0.001ms. The total calculating using Forward Euler cost 95.7535419 seconds. The Rush-Larsen could still bear step size = 0.1ms, and it cost 12.87450 seconds to process Rush-Larsen for the Ten Tusscher.

## 2.3 System Design

We use Julia as our backend to do the data analyzation and computation. We choose Julia because it provides easy and expressive high-level numerical computing. Also, developers could easily call other languages in Julia such as C, Fortran, and Python. What's more, Julia has many powerful packages such as "PyCall", "CuArrays", "CLArrays" and so on, so it perfectly aligns our project purposes.

Our second goal is to fetch the electrophysiology cell modes from CellML.org. We found CellML.org stores their cell models in CellML format, and people can easily find and download the model they want from their official website. We used Myokit to read .cellml files downloaded from their website. Myokit is a Python-based software package to simplify the use of numerical models in the analysis of cardiac myocytes. By using Myokit, we can read and categorize the .cellml files and we could store the fetched variables into our database.

Our idea is to composite a CellML file to 5 parts, "Components", "Constant variables", "Non-constant variables", "Units" and "Equations". A "Component" acts as a container of variables which are in the same functionality, such as a gating component contains a gating variable and its helper variables and equations. The "Constant variables" and "Non-constant variables" contains all constant and non-constant variables in the model. The "Units" contains all of the units that the model variables used, and the "equations" store all of the equations includes ODEs and linear equations of the model.

We used two different packages from Julia, "CuArrays.jl" and "CUDAnative.jl", for our GPU programming. These two packages could help us to do high-level GPU programming. "CuArrays.jl" performs the container to store the data which is readable from GPUs, and "CUDAnativejl" is using for GPU computation. Julia also provides "CLArrays.jl" and "OpenCL.jl", and we could use them to implement programs for non-Nvidia GPUs in the future. Unfortunately, we did not figure out the final solution using GPU.

Our project structure is shown in the figure below. We simplify our project into 4 layers. The first layer is the Python layer, we used Myokit(Python) to read the CellML files. The second layer is the Julia-Python layer. We used Julia to invoke all the Python environment (include Myokit) via "PyCall.jl" so we can decompose the CellML files in 5 categories, "Components", "constant variables", "non-constant variables", "units" and " equations". These categorical data would be stored in the Julia database. The third layer is Julia-GPU layer. In this part, we reintegrated all the variables in the database to GPU arrays ("CuArray") to interact with GPUs via "CuArrays.jl", and we redefined the mathematical operations from general written formats to GPU written formats via "CUDAnative.jl". We also defined the numerical methods in this layer. The 4th layer is the display and output layer. Since the GPU data is not printable and readable, we had to copy the data from GPU containers to the general containers to allow us to display or print the data.

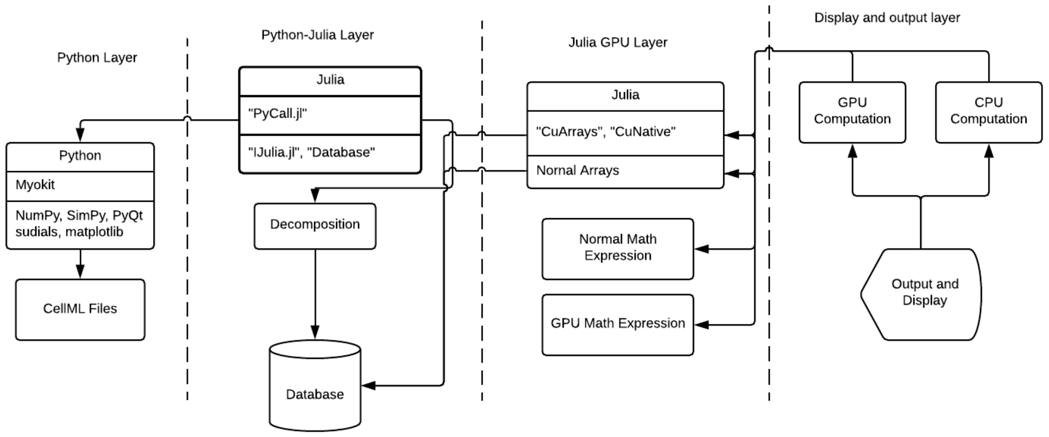


**Figure 9:** System Structure

# 3 Experience

We spent two semesters to build our projects. In this chapter, we will talk about the attempts we tried, the experience we gained and the knowledge we learned from this project.

## 3.1 Our Attempts through this project

During these two semesters, we did several different attempts because we never touched scientific computation and GPU programming before. We need to explore everything by ourselves. We faced a lot of challenges on fetching the CellML files to the computer programmable code and GPU programming.

### 3.1.1 Code fetching from CellML

The first challenge we met is to fetch the CellML files into the computer programmable languages. We tried multiple tools such as CellML official API (SDK), libCellML and Myokit.

We first tried CellML Official API [12]. However, it has not been updated since 2012 and the API is not compatible in the current software development environment. For example, it does not support Java 7 or later version, and it also cannot adapt to the 64-bit operating system. We tried to install it on different operating systems (OSX, Window 10, Ubuntu Linux), but unfortunately, none of them works.

As the CellML Official API performed not well, we started to find another tool to solve the problem. We then tried libCellML [1], which is recommended by CellML official website. The libCellML tool is focusing on CellML 2.0 and beyond. However, this tool is still constructing. Its current version does not contain useful functions to the project such as reading the CellML files or simulation supports, thus we decide to find another tool.

Finally, we found Myokit[2]. Myokit is a Python-based open source software package designed to simplify the use of numerical models in the analysis of cardiac myocytes. It is created by Michael Clerx as part of his Ph.D. Thesis at Maastricht University and is being developed future at the University of Oxford. This tool package could import models from CellML and SBML. It includes almost all the functionalities we need, and it has integrated documents and tutorials on its website. Thus, we decided to use Myokit as the code fetching tool in our project.

### 3.1.2 MATLAB

On the other hand, we started hardcode at the same time to make sure that we have an additional plan when the code fetching fails. We first tried MATLAB. MATLAB is easy to use and have high precision for scientific calculation and computation. We hardcoded Fitzhugh-Nagumo (1964) model, used "ode45" function to solve it and plotted a "heatmap" and the data. MATLAB is good at doing the calculation and does have parallel computation use GPU, but it is not that good on programmings such as the limitation on using your own numerical method and the limitation on exporting animation. Thus, is not a good choice for our project.

### 3.1.3 WebGL and GPU programming

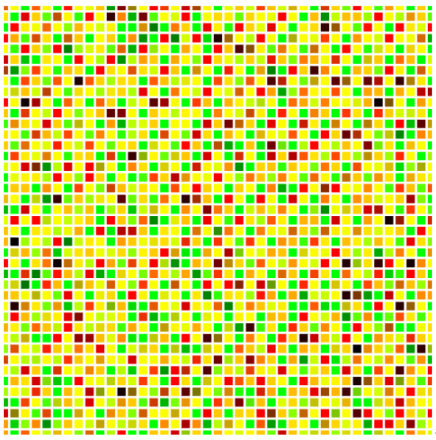
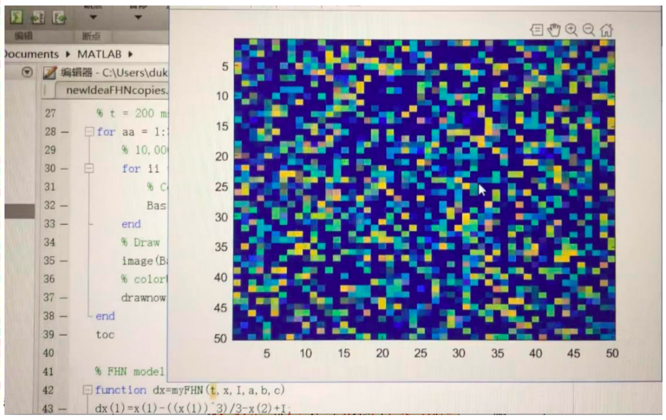


**Figure 10:** WebGL for Fitzhugh-Nagumo



```
% t = 200 m
for aa = 1:
    % 10,00
    for ii
        % C
        Bas
    end
    % Draw
    image(B
    % color
    drawnow
end
toc

% FHN model
function dx=myFHN(t, x, I, a, b, c)
dx(1)=x(1)-((x(1)).^3)/3-x(2)+I;
```

**Figure 11:** MATLAB for Fitzhugh-Nagumo

Another challenge we faced is GPU programming. Our initial goal is to implement our project on browsers, so we considered using WebGL by introducing an API that closely conforms to OpenGL ES 2.0 that can be used in HTML5 ¡canvas¿ element. If we could use WebGL to simulate the electrophysiology, then the project could run on different platform browsers. So we gave it a try. We first hardcoded Fitzhugh-Nagumo (1964) model using WebGL and tried to use the Forward Euler method to solve it. However, during this period, we met the I/O problem. Since WebGL uses GLSL (OpenGL shading language) for computations, it is not easy to input the cell models' parameters. For example, when we use WebGL to implement a cell model, we have to translate all variables from the model to the color channels (R,G,B,A), transport those colors into the shaders, write the differential equations and numerical methods and do the calculation in the shaders using GLSL and input the result into the textures, then we need to generate the new color channels from the texture, translate it back to cell model variables and record the value and keeps going. To follow those steps, we have to hard code every model, which is not our goal. To solve this problem, we tried multiple WebGL based Javascript libraries such as Tensorflow.js and GPU.js for implementing the computations on a high-level and we did find the solution. Those two libraries are high-level and could automatically use WebGL as backend. Although we still need to hard code, the process is now easier to understand. We plotted a "heatmap" for using Forward Euler method solving 128*128 Fitzhugh-Nagumo models [Figure 10] (left) and compared the data with the results from MATLAB [Figure 11] (right) to ensure the accuracy. However, we found another important issue that finally stopped us from using WebGL. Because it currently does not support double precision, but we are supposed to use double precision on ODEs to get accurate results, we have to find another solution that supports double position[9].

### 3.1.4 Python CUDA and OpenCL

Then, we started another attempt of the GPU programming. Since we decided to use Myokit as our CellML files reader, we started using Python. We found two ways of using Python to do GPU programming: "PyCUDA" and "PyOpenCL". However, the biggest problem for Python GPU programming is the kernel. In the CUDA and OpenCL implementation in Python, kernel files are the most important part in interacting with GPU, but the kernel is difficult to use. For example, a kernel cannot call other kernels, host functions or making the recursive call. If we used the CUDA and OpenCL on Python, we have to write a translator that translates the Python codes to the kernel codes, which is pretty difficult for beginner GPU programmers. It is not a good idea to spend time exploring kernel languages; thus, we decide to move on to another solution.

### 3.1.5 Julia Programming

Finally, Dr. Spiteri advised us to try Julia. Julia is a fairly new language, and it is useful in scientific computations[10]. We spent a few days to learn the language and tried hardcode models and numerical methods. Furthermore, it natively supports Python so we could use Myokit to invoke the CellML files. We spent

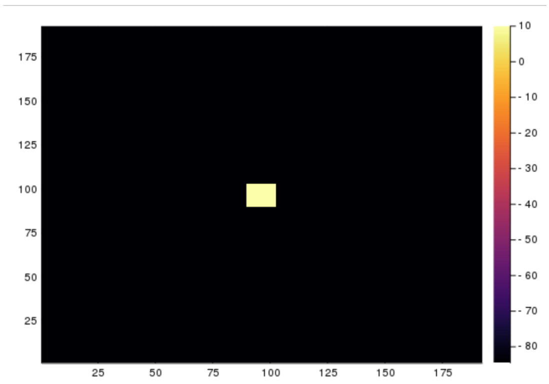


Figure 12: Julia GPU programming for Beeler-Reuter Model 1

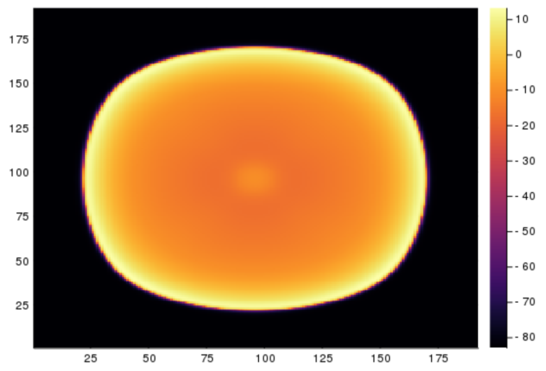
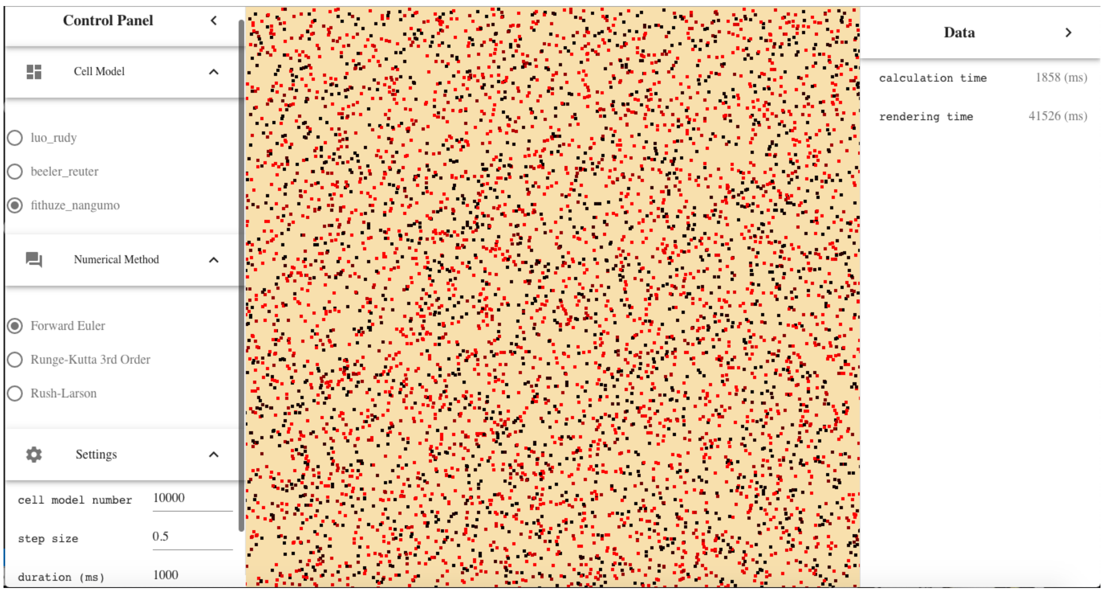


Figure 13: Julia GPU programming for Beeler-Reuter Model 2

another few days to figure out how: to use Python in Julia, we could simply install "PyCall.jl" package on Julia environment and set up a correct Python environment path. Also, we found some useful examples and packages for our project on their website. For instance, we found a Beeler-Reuter model written by Julia language and using "DifferentialEquation.jl"[5][8].The Figure 12 and 13 show the Beeler-Reuter models by uisng the laplacian and the Rush-Larsen method. We then hardcoded several heart models by using Julia, and we got positive results.

### 3.1.6 User Interface



Cell Model: luo_rudy, beeler_reuter, [✓ fithuze_nangumo]

Numerical Method: [✓ Forward Euler], Runge-Kutta 3rd Order, Rush-Larson

| | |
|---|---|
| cell model number | 10000 |
| step size | 0.5 |
| duration (ms) | 1000 |

| | |
|---|---|
| calculation time | 1858 (ms) |
| rendering time | 41526 (ms) |

Figure 14: User Interface with Vue.js and WebGL

When we were still trying WebGL, we designed a web application in Vue.js to plot a "heatmap" of the results we got. However, we finally decided to use Julia instead of WebGL and we did not find a way to transfer data from Julia to JavaScript locally, thus we did not use the application.

## 3.2 What we learned from this project

We learned a lot from this project. In addition to the technologies mentioned in section 3.2.1, we also learned mathematics and electrophysiology knowledge.

### 3.2.1 Mathematics and Electrophysiology

We only know a little bit about the electrophysiology and relative mathematics knowledge before we start the project. Thus, the prerequisite before we start this project is to study the electrophysiology. We read many thesis about electrophysiology and numerical methods and we attempted several different languages to realize our goal using the knowledge we learned from those materials.

### 3.2.2 Languages and Technologies

We tried 7 different languages (MATLAB, WebGL, Python, OpenCL, CUDA, Julia and vueJS) and several libraries in total. Those knowledge and experience will help us in our future life.

### 3.2.3 GPU Programming

Although we did not success for running all our code on GPU, we read many researches and tried our best to make it work. We also find several fundamentals about GPU programming in different languages as mentioned above. It is pity that we did not make it work. However, we still learned a lot about GPU programming, which will be surely useful in our life.

### 3.2.4 Pair Programming

We do pair programming averagely twice per week, to solve problems together and divide the work into a proper way. It is a good experience of pair programming for us, we tried to explore how to solve problems together and what we should do when we have conflicts and divergence.

# 4 Conclusion and Future

## 4.1 Conclusion

To conclude, Real-Time Heart Simulation project is meaningful because it is not only a project in Computer Science but also a research area in biology. Reaching the real-time heart simulation by using modern technologies could help doctors analysis and treat cardiac disease and indirectly save people's lives.
We contributed in hardcoded 4 cell models (Fitzhugh-Nagumo, 1964; Beeler-Reuter, 1977; Luo-Rudy, 1991 and Ten Tusscher, 2006, epi) including monodomain model and implement 3 numerical methods (Forward Euler, Runge-Kutta 2nd Order (Heun's), and Rush-Larsen) to each model in Julia language, which could be executed on CPU. Each model and methods could plot a static figure or a gif with the set time range. We also made some progress on GPU programming and code fetching from CellML files.

Through this project, we learned new knowledge in different aspects. On the Computer Science side, we studied the concepts of GPU programming and learned many new languages to realize it, including MATLAB, WebGL, OpenCL, CUDA, and Julia. We also learned how to design problem-solving software environment. On mathematics side, we learned ordinary differential equation, partial differential equation, and numerical methods, and we learned cell models and monodomain model on biology side, Even though it is not easy to implement scientific computations on GPUs since GPU programming is totally different from CPUs, we made some progress on it. We also met a lot of challenges such as CellML files fetching issues, GPU programming problems and failed several times, but we learned experience from our attempts. We appreciate everyone's help for us.

## 4.2 Future Works

There are a few more features that could be implemented in this project in the future.

### 4.2.1 To achieve GPU computing for more heart cell models

Because we did not finish GPU computation in the project, we need to make sure our GPU code could solve the equations of heart cell models.

### 4.2.2 To implement more mathematical solver methods

Right now, we only have Forward Euler, RK2, and Rush-Larsen to solve ODEs, hopefully, we could implement more methods such as RK4, RKC, etc in the future.

### 4.2.3 Database connection

Right now, we did not perfectly connect with our database because some problems are required to solve. For example, how to translate the normal mathematical equations to GPU's equations. How to reach the real-time database.

### 4.2.4 To reach the real-time

Our current project has not reached the real-time computation yet, in the future, we need to optimize our codes, and we might use multiple GPUs attempting to reach the real-time.

### 4.2.5 Using 2D/3D modeling

Our project is using 1D modeling cell models right now, it might perform too easy on GPU, we will try to implement 2D and 3D to analysis the complex situations.

## References

[1] T. Ahmadi, R. Blake, M. Clerx, J. Cooper, A. G. andDavid Ladd, G. Mirams, D. Nickerson, and H. Sorby.

[2] M. Cler, “Myokit,” http://myokit.org/, 2018.

[3] K. R. GREEN and R. J. SPITERI, “Extended bacoli: Solving one-dimensional multi-scale parabolic pde systems with error control,” 2010.

[4] W. Guo, “Efficient cardiac simulations using the rungekuttachebyshev method,” 2017.

[5] S. Iravanian, “An implicit-explicit cuda-accelerated solver for the 2d beeler-reuter model,” http://juliadiffeq.org/DiffEqTutorials.jl, 2019.

[6] A. Kaboudian and F. H. Cherry, Elizabeth M andFenton, “Real-time interactive simulations of large-scale systems on personal computers and cell phones: Toward patient-specific heart modeling and other applications,” 2019.

[7] M. E. Marsh, “An assessment of numerical methods for cardiac simulation,” 2012.

[8] C. of DifferentialEquation.jl.

[9] C. of hvidtfeldts.net, “Double precision in opengl and webgl,” http://blog.hvidtfeldts.net, 2012.

[10] C. of Julia, “Julia 1.1 documentation,” https://docs.julialang.org/en/v1/, 2019.

[11] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips.

[12] P. Project, “An api for processing and manipulating mathematical models in cellml,” http://cellml-api.sourceforge.net/.