Thesis for Master of Engineering

Enhancing the efficiency of animal-alternative *in silico* drug cardiotoxicity prediction through CUDA-based parallel processing

December 2024

Graduate School
Kumoh National Institute of Technology

Department of IT Convergence Engineering

Iga Narendra Pramawijaya

Thesis for Master of Engineering

Enhancing the efficiency of animal-alternative *in silico* drug cardiotoxicity prediction through CUDA-based parallel processing

December 2024

Graduate School
Kumoh National Institute of Technology

Department of IT Convergence Engineering

Iga Narendra Pramawijaya

Enhancing the efficiency of animal-alternative *in silico* drug cardiotoxicity prediction through CUDA-based parallel processing

Supervisor Ki Moo Lim

This Thesis Presented for the Master of Engineering

December 2024

Graduate School
Kumoh National Institute of Technology

Department of IT Convergence Engineering

Iga Narendra Pramawijaya

Approval of the Thesis for the Master of Engineering Submitted by Iga Narendra Pramawijaya

December 2024

Chair of Committee 엄지용 (seal)

Committee 임기무 (seal)

Committee 김한준 (seal)

Graduate School
Kumoh National Institute of Technology

Enhancing the efficiency of animal-alternative *in silico* drug cardiotoxicity prediction through CUDA-based parallel processing

Iga Narendra Pramawijaya

Department of IT Convergence Engineering,
Graduate School
Kumoh National Institute of Technology

Abstract

Introduction: This research focuses on enhancing *in silico* cardiotoxicity prediction by utilising GPU-based parallel computing. Traditional CPU-based simulations are computationally expensive, especially for large-scale studies. By leveraging CUDA programming, this research aims to optimise simulation efficiency while maintaining the accuracy of cellular electrophysiological models.

Method: The study employed three well-established cardiac cell models: ORd 2011, ORd 2017, and ToR-ORd. Simulations were conducted using GPU-based implementations of ordinary differential equation (ODE) solvers, with the Rush-Larsen method applied for ORd 2011 and a Forward Euler approach for ORd 2017 and ToR-ORd. The simulations were validated against CPU-based OpenCOR results, with performance evaluated in both drug-free and drug-induced conditions.

Results: GPU simulations demonstrated equivalent accuracy to CPU-based results, effectively replicating action potential dynamics and key biomarkers across all cell models. However, the Forward Euler solver required more computation time compared to the Rush-Larsen method. Computational performance analysis revealed significant efficiency improvements in GPU-based simulations, particularly in handling large-scale datasets.

Conclusion: This research successfully validates GPU-based parallel computing as a reliable and efficient approach for *in silico* cardiotoxicity prediction. The findings support its potential for accelerating drug discovery processes while reducing reliance on animal testing. Future work will focus on expanding model complexity and variabilities to further enhance the system's applicability

CUDA기반 병렬처리를 통한 동물대체 인실리코 약물 심독성 예측 효율성 증대

Iga Narendra Pramawijaya

금오공과대학교 대학원 IT융복합공학과

요 약

소개 : 본 연구는 GPU 기반 병렬 컴퓨팅을 활용하여 *in silico* 심장 독성 예측을 향상시키는 데 초점을 맞추고 있습니다. 기존의 CPU 기반 시뮬레이션은 대규모 연구에서 계산 비용이 높아 비효율적입니다. CUDA 프로그래밍을 활용하여 세포 전기생리학 모델의 정확성을 유지하면서 시뮬레이션 효율성을 최적화하는 것을 목표로 합니다.

방법 : 본 연구에서는 ORd 2011, ORd 2017, ToR-ORd 모델이라는 세 가지 잘 확립된 심장 세포 모델을 사용했습니다. ODE(상미분방정식) 해석기를 GPU 기반으로 구현하였으며, ORd 2011에는 Rush-Larsen 방법을, ORd 2017 및 ToR-ORd에는 Forward Euler 방법을 적용했습니다. 시뮬레이션 결과는 CPU 기반 OpenCOR 결과와 비교하여 검증하였으며, 약물 없는 상태와 약물 유도 상태 모두에서 성능을 평가했습니다.

결과 : GPU 시뮬레이션은 모든 세포 모델에서 CPU 기반 결과와 동일한 정확성을 보였으며, 활동 전위 역학 및 주요 바이오마커를 효과적으로 재현했습니다. Forward Euler 방법은 Rush-Larsen 방법에 비해 계산 시간이 더 오래 걸렸습니

다. 계산 성능 분석에서는 특히 대규모 데이터셋 처리에서 GPU 기반 시뮬레이션이 상당한 효율성 개선을 보여주었습니다.

결론 : 이 연구는 GPU 기반 병렬 컴퓨팅이 신뢰할 수 있고 효율적인 인실리코 심장독성 예측 방법임을 성공적으로 입증했습니다. 연구 결과는 약물 개발 과정을 가속화하고 동물 실험 의존도를 줄이는 데 기여할 가능성을 뒷받침합니다. 향후 연구에서는 모델의 복잡성과 변동성을 확장하여 시스템의 적용 가능성을 더욱 향상시키는 데 중점을 둘 것입니다.

Contents

[List of Figures]

[Glossary]

CUDA	Compute Unified Device Architecture
CiPA	Comprehensive *in vitro* Proarrhythmia Assay
CPU	Central Processing Unit
GPU	Graphics Processing Unit
MPI	Message Parsing Interface
ODE	Ordinary Differential Equations
GP-GPU	General Purpose computing on Graphics Processing Unit
CaI	Intracellular calcium concentration.
IKr	Rapid delayed rectifier potassium current.
hERG	Human Ether-à-go-go-Related Gene
ICaL	L-type calcium current
IK1	Inward rectifier potassium current
IKs	Slow delayed rectifier potassium current
Ito	Transient outward current
INaL	Late sodium current
INa	Sodium current

[Acknowledgement]

In the name of God, Ida Sang Hyang Widhi Wasa, I offer my deepest gratitude. This thesis could not have been completed without His guidance and blessings.

First and foremost, I extend my heartfelt thanks to my supervisor, Professor Ki Moo Lim, for giving me the opportunity to be a part of the Computational Medicine Lab (CML). Thank you for belived in me and my skill, even when I am doubting myself. Your invaluable guidance and support have greatly contributed to my growth during my time at CML. I am also deeply grateful to my examiners, Professor Um Ji Yong and Professor Kim Hanjoon, for their time, encouragement, and insightful feedback.

I would like to express my appreciation to all my lab mates—Ali, Aulia, Marcell, Adnan, Lulu, Bharindra Ari, Ariyadi, Icha, Olivia, Latifa, Nurul, Ana, Yunen, 정래현, 박혜림, and 정혜주. Your support, advice, and companionship have been crucial to my journey.

To my fellow Indonesian Students Association in South Korea (Perpika), thank you for being a significant part of my life here. You have shaped me into a better person.

Also, my deepest gratitude goes to my family, my father, mother, and relatives, who have always provided me with unconditional love and encouragement. Thank you for your unwavering support.

Lastly, for everyone in distress, physically, mentally, financially, and such, there is a hope. I gave life its second chance. There is a hope if we open up our heart even just for a moment. This thesis would not even exist if I did not take the second chance. I hope this work can shed a little light in each ones tunnel of despair. Thank you for staying here.

Chapter 1. Introduction

Cardiovascular diseases are the leading global causes of death, which emphasizes the importance of effective methods for cardiac drug discovery. Traditionally, drug cardiotoxicity prediction is achieved using animal testing, which takes time due to ethcical clearance and effortful. Modern *in silico* or computer-based methods for drug cardiotoxicity prediction show promising results as an animal-alternative solution. Nevertheless, some of them are computationally inefficient due to large amount of sample it needs to compute to mimic natural variations. As the sample size increases, the complexity of the calculations grows, resulting in longer processing times and reduced efficiency.

This efficiency limitation makes it difficult for traditional computational approaches to handle large-scale simulation (such that uses multi-sample scenario or inter-individual variations) within a reasonable timeframe. This research introduces an updated solution to address the computational inefficiencies of current *in silico* drug cardiotoxicity simulations. By implementing NVIDIA's CUDA (Compute Unified Device Architecture)-based parallel programming on Graphics Processing Units (GPU) [1], this method can significantly accelerates overall computational process, enabling faster handling of large-scale simulations. By leveraging the power of parallel processing, computational time will be reduced and this can accelerate preclinical testing, potentially reducing drug development costs and reliance on animal testing.

1.1 *in silico* Cardiac Electrophysiology Simulation

In silico cardiac electrophysiology simulation is a computational approach used to model and analyse the electrical activity of heart cells. This method relies on mathematical calculations, often using ordinary differential equations (ODEs), to simulate biological responses under various conditions. By studying the electrophysiology of cardiac cells, researchers can gain valuable insights into cardiovascular diseases, evaluate the efficacy of drugs, and assess potential cardiotoxicity, or the harmful effects of drugs on the heart.

These simulations serve as a minimally invasive alternative to experimental methods, providing a detailed understanding of cardiac function, disease mechanisms, and potential treatments, while reducing reliance on traditional animal testing or invasive procedures. *In silico* cardiac electrophysiology simulation is a powerful computational tool used to model and study the electrical activity of the heart. These simulations provide insights into cardiac function, disease mechanisms, treatments and what might harmful for the heart with a minimum invasive approach to collect data.

1.2 Parallel Computing

Over recent decades, parallel computation has promised to accelerate overall computation speed. Parallel computing, from a technical standpoint, means performing many calculations simultaneously, based on the principle that large problems can often be divided into smaller tasks that

can be processed at the same time. For programmers, the main challenge is how to allocate these concurrent tasks across multiple computing resources.

Parallel computing has both hardware and software requirements that are deeply interlinked. Computer hardware architechture must ensure it has two or more computing cores, while parallel programming designs code for more than one computing core. The hardware aspect of computer architecture supports parallelism by providing an infrastructure that can handle multiple, simultaneous processes or threads. Meanwhile, parallel programming focuses on efficiently using this hardware to perform tasks concurrently.

Understanding the computer architecture is less crucial when developing a non-parallel program. However, in parallel programming, a solid understanding of multicore architectures becomes essential for developing efficient and correct parallel programs. This programming paradigm involves distributing tasks to these available cores to achieve simultaneous execution, ensure every core runs in harmony, and arranging output from each cores.

The effectiveness of parallel computing depends on overcoming challenges like coordinating tasks, managing data dependencies, and distributing work evenly among computing units. Tasks that rely on each other must be executed in the right order to minimise idle time. Data dependencies, where one task requires the output of another, necessitate careful coordination to avoid delays or errors. Ensuring that all computing units are working efficiently is crucial to prevent bottlenecks. Addressing these issues requires both sophisticated programming techniques and well-designed systems.

1.2.1 Central Processing Unit (CPU) for Parallel Computing

For decades, one key method of improving consumer computing performance was to increase the CPU's clock speed. However, due to power, heat, and physical limitations in making smaller transistors, this approach has reached its limits. As a result, manufacturers have shifted their focus from boosting clock speed to increasing the number of cores per CPU, inspired by supercomputers that achieve high performance by using large numbers of processors. Rather than relying solely on single-core performance, adding multiple cores also allows personal computers to improve processing power without clock speed increases. Widely-known standards to do parallel processing with a GPU is to use Open Multi-Processing (OpenMP) or the Message Passing Interface (MPI).

OpenMP (Open Multi-Processing) is a parallel programming model tailored for shared-memory architectures, commonly used to leverage the capabilities of multicore CPUs. By integrating simple compiler directives (pragmas) into C, C++, or Fortran code, OpenMP enables developers to parallelize loops and code sections with minimal changes [2]. Its approach involves dividing tasks among multiple threads that access a shared memory space, simplifying data management and parallel execution. Due to its ease of implementation, OpenMP is an optimal choice for high-performance applications running on single multicore processors, where shared memory facilitates efficient data handling.

MPI (Message Passing Interface), on the other hand is a standard designed for parallel computing in distributed-memory environments, such as clusters or supercomputers. MPI enables communication between processes operating in separate memory spaces by exchanging messages,

making it suitable for applications requiring parallel execution across multiple networked nodes [3]. Although MPI introduces complexity due to explicit data management requirements, it provides exceptional scalability and flexibility. This makes it the preferred solution for high-performance computing tasks that demand extensive data communication between distributed systems. OpenMP and MPI are two frameworks that serve distinct yet complementary roles in parallel computing. OpenMP is well-suited for shared-memory systems with straightforward parallelism, while MPI offers the scalability required for distributed systems that span multiple nodes.

In this research, CPU parallelisation will use MPI due to its multiple nodes capability [3]. CPU parallelisation using MPI will be compared with GPU parallelisation in terms of computational time. CPU Parallelisation in this research used OpenMPI, an open-source implementation of MPI standard.

1.2.2 Graphics Processing Unit (GPU) for Parallel Computing

Graphics processing unit initially designed to compute graphical calcucations that is repetitive, relatively more simple compared to what CPU calculates, but quantitatively much more calculations compared to CPU. Unlike CPUs, which are optimized for a wide variety of tasks and tend to have a smaller number of powerful cores, GPUs have thousands of smaller cores designed for high-throughput parallelism. This architecture allows GPUs to perform many calculations simultaneously, making them highly efficient for tasks that can be broken down into smaller, identical operations.

Beside of graphics computing, GPUs are also able to accelerate other computing purposes such as scientific simulation and machine learning. This demand creates new sub-field in computer programming, called GP-GPU programming, that stands for general-purposed graphics processing unit programming. Its high-throughput parallelism makes GPU suitable for scientific calculation that has massive datasets or extensive matrix calculation. With frameworks like CUDA and OpenCL, programmers can leverage GPU architectures to perform GP-GPU.

1.2.3 CUDA

CUDA, short for Compute Unified Device Architecture, is a parallel computing platform and programming model developed by NVIDIA. It enables developers to harness the immense computational power of NVIDIA GPUs for general-purpose processing tasks [1]. CUDA is built on the foundation of extending standard C/C++ programming with GPU-specific features, making it accessible for developers already familiar with these languages. It provides tools that allow detailed control over GPU resources, enabling efficient parallel execution of computationally intensive tasks across thousands of GPU cores.

The computational model of CUDA is structured around a hierarchical organization of cores, blocks, and threads. At the highest level, the GPU consists of multiple streaming multiprocessors, each containing numerous CUDA cores. These cores execute the smallest unit of work in CUDA, referred to as a thread. Threads are grouped into blocks, which can contain hundreds or thousands of threads, depending on the GPU's architecture.

Blocks are further organized into a grid, creating a hierarchy that allows the distribution of computational tasks across the entire GPU [1]. This hierarchy is essential for mapping complex problems onto the GPU efficiently. Developers can set the number of threads per block and blocks per grid to match the problem's needs and the GPU's hardware limits. Each thread gets a unique ID within its block, and each block gets a unique ID within the grid. These IDs allow CUDA programs to divide tasks or data evenly among threads, ensuring balanced workload distribution. This setup explained more in chapter 2.

1.2.4 CellML

CellML is an XML-based language created to represent mathematical models in a platform-independent format, facilitating model sharing between researchers and secure archival in repositories [4]. This standardisation in a machine-readable form is essential in bioinformatics, as it enhances scientific accuracy, accelerates model development, and enables the integration of multiple models into complex, combined systems [5]. CellML supports collaboration by allowing models to be easily exchanged and archived. Several public databases host extensive collections of CellML models, with the CellML Model Repository being one of the most prominent. Additionally, the BioModels database includes models converted from the Systems Biology Markup Language (SBML) into CellML, broadening accessibility and compatibility for researchers using these bioinformatics resources. [6][7].

1.3 Previous Study

Parallelisation in computational biology is not an entirely new concept. The Cells in Silico (CiS) framework presented by Berghoff et al. [8] offers a tool for simulating the growth and development of biological tissues. The modular and parallel design of CiS allows for flexible configuration of different model assumptions, making it applicable to a wide range of research questions. As demonstrated by the example of a 10003 voxel-sized cancerous tissue simulation at sub-cellular resolution, CiS can be used to explore complex biological processes at a high level of detail.

Utilisation of GPU in biological cell computing has been explored in previous researches. One of them is from Martinez, et al [9] in 2020. Miguel, et al. explored an adaptive parallel simulator to solve performance loss in massive parallel membrane computing devices known as membrane systems or P systems. The paper demonstrates the effectiveness of this approach by extending an existing simulator for Population Dynamics P systems. Experimental results show that this adaptive simulation can significantly improve performance, up to 2.5x on both GPUs and multicore processors.

Related to drug toxicity and discovery, other researchers tried to approach and optimise drug development process using parallel computing approach as well. Previously, McIntosh-Smith et, al. developed *in silico* drug screening method on multiple core processors. McIntosh-Smith et, al. developed BUDE (Bristol University Docking Engine), a drug discovery tool, simulating molecular docking. To speed up calculations on powerful processors with multiple cores, BUDE has been adapted to work with OpenCL, a common language for parallel programming [10]. As a result, McIntosh-Smith et, al. achieved of 46% at peak, or 1.43 TFLOP/s on a single

Nvidia GTX 680. Amar et, al. developed a parallelisation on biochemical simulation of metabolic pathways in their high-level computational simulation.

BUDE with parallel computing also allows Barth et, al. to run simulations with more complex models. This complexity increase features a greater number of chemicals and reactions. Hence, Barth et, al. can achieve more realistic, lifelike outcomes while using less computing time [11].

Recent studies highlight the growing use of *in silico* approaches to investigate specific cardiac conditions and evaluate pharmacotherapies. For example, Whittaker et, al. used computational modelling to assess the effects of mutations associated with Short QT Syndrome and their impact on atrial arrhythmias. Their findings illustrate how *in silico* simulations can explore drug responses and guide pharmacological strategies in addressing genetic cardiac disorders [12].

Furthermore, advancements in computer hardware and parallel processing techniques have significantly improved the speed and efficiency of *in silico* heart simulations. This technological progress allows researchers to analyse larger datasets and more complex models at an unprecedented pace, making computer simulations an essential tool in contemporary heart research. As computer technology continues to advance, computer-based heart simulations are poised to play an even more crucial role in uncovering new insights into heart function and developing targeted treatments for heart diseases.

1.4 Objectives

The objective of this study is to enhance the efficiency and scalability of *in silico* simulations for predicting drug cardiotoxicity by leveraging CUDA-based parallel processing. The research aims to:

1. Address computational bottlenecks caused by increasing sample sizes and complex calculations in traditional methods.
2. Optimize GPU resources for faster, large-scale simulations without compromising accuracy.
3. Validate the accuracy and reliability of GPU-based simulations compared to CPU-based methods.
4. Develop a practical and cost-effective approach suitable for real-world drug discovery applications, reducing reliance on animal testing.

Chapter 2. Methodologies

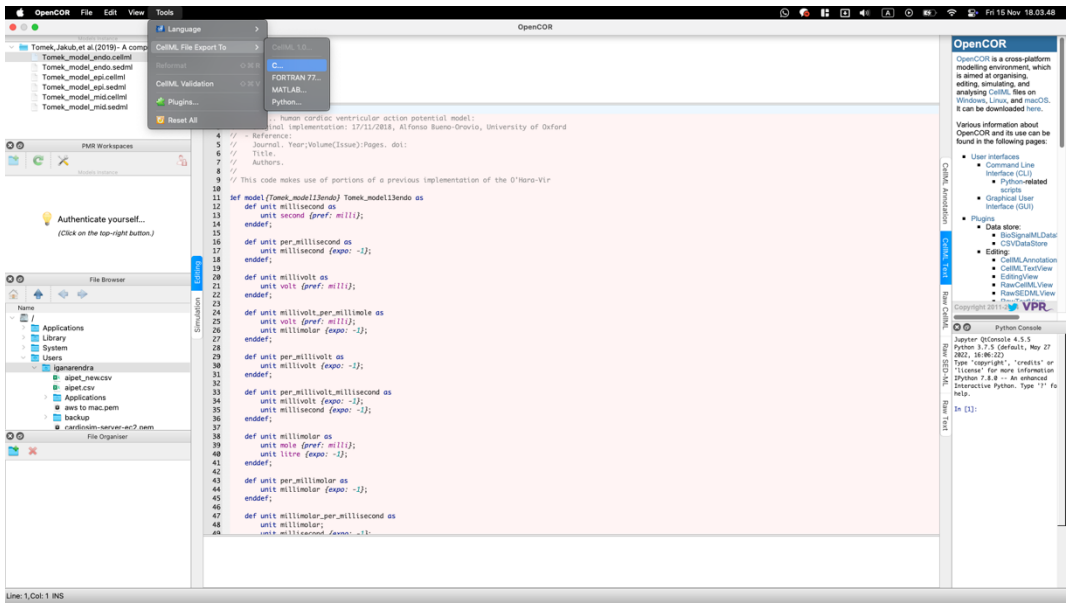
This chapter describes the development process of the GPU-based cardiac cell simulation software. The focus is on enabling multi-sample simulations, where each cardiac cell model is simulated in parallel. This chapter will guide readers through the process of converting CellML-based models into C code, modifying the generated code for GPU simulation, and implementing parallelisation techniques to handle multiple samples efficiently. Additionally, this chapter will explain how ordinary differential equations (ODEs) in the cell models are solved within this framework. The goal is to ensure that researchers or software engineers can follow the steps presented here to replicate the parallelisation process and build their own GPU-based multi-sample simulation platforms.

2.1 Generating C Code from CellML

Various third-party libraries or applications are available to convert CellML's XML format into programming script. One of the most popular application is OpenCOR. OpenCOR is a versatile software tool designed for modelling and simulation of biological processes, including those described in CellML. It provides functionality to parse CellML models and convert them into executable code in various programming languages, including C. Beside of generating C code, OpenCOR supports model editing, running simulation, and analysis, making it a comprehensive platform for working with CellML models. Its extensibility and integration with other tools, such as Python

scripting, further enhance its utility for researchers. By leveraging OpenCOR, users can streamline the process of implementing CellML models into broader computational workflows, such as those involving high-performance computing or *in silico* drug testing.

This research involves three different cell models. O'Hara-Rudy 2011 [13], O'Hara-Rudy 2017 [14] and ToR-ORd [15]. After installing OpenCOR, open the application and search for these three cell models in the search bar on top left corner. Select file with .cellml extension. Select tools, and export the CellML file to C code. Figure 2.1 shows the OpenCOR GUI on MacOS.



[Figure 2. 1] OpenCOR interface when selecting ToR-ORd model and converting it to C codes on MacOS.

Having the C code of the cell model is important because CUDA programming uses .cu format, that is similar to C. CUDA is built upon C/C++, extending it for GPU programming. Both use C syntax as their base language.

Many fundamental syntax elements like loops, conditionals, functions, and function definitions are similar in both CUDA and C. CUDA programs include host code that runs on the CPU, which is a standard C/C++ code. For programmers familiar with C/C++, CUDA maintains a relatively low learning curve by starting from familiar concepts, only adding memory allocation and parallel processing paradigms.

2.2 GPU Memory Adjustment and Offsetting

Efficient memory management is critical in GPU programming, as the performance of a CUDA-based application heavily depends on how data is transferred between the host (CPU) and device (GPU), as well as how it is organized within the GPU's memory. This research uses three types of GPU memory: global, shared, and constant memory.

In the GPU, global memory is the largest and most flexible, but with slower access speed. Shared memory is faster to access but has limited space (10 KB) and is restricted to threads within the same computing block. Constant memory is the fastest, with purpose to deliver execution commands to all threads. Constant memory's size is more suitable to store constants during execution due to its 64 KB limit.

Proper adjustment of these memory types can significantly enhance computational efficiency. In this research, the global memory is used for storing variables, constant memory for orchestrating commands from CPU, and shared memory used to optimise GPU to CPU feedbacks. Details of the GPU specification will be mentioned in chapter 3 and appendix.

In this research, selecting an optimal number of threads per block is crucial to maximising utilisation of shared memory. Two factors need to be

considered to optimise performance when selecting core per block value: thread grouping in CUDA, and number of samples. CUDA executes threads in groups called warps, which consist of 32 threads [16]. Using a block size that is a multiple of 32 ensures that all warps are fully utilised, minimising idle threads and maximising efficiency.

It is known that this configuration is not transferable across different GPUs. As the time writing this, NVIDIA 40xx series GPU supports 32 core per block, while older series like the 30xx only support 16 core per block. 30xx series also uses wraps, but it has less computing core. Therefore, for the 30xx series, 16 cores per block were selected, aligning well with the hardware's limitations, as $16 \times 2 = 32$ matches the warp requirements.

The choice of cores per block was also influenced by the number of samples. Each simulation sample usually comes in the multiplication of 2000 (2000, 4000, etc.). To optimise warp utilisation (with warps consisting of 32 threads), To optimize warp utilization, the number of cores should be close to 32 and evenly divisible by 2000. Through trial and error, 20 cores per block were found to provide the most efficient configuration. This adjustment ensures that each sample is allocated its own computing core.

By default, the code is configured to use 32 cores per block. However, hardware limitations may cause errors with this configuration. In such cases, changing the number of cores per block to 20 ensures the parallel processing process remains stable. Since 32 does not divide evenly into 2000, the code automatically rounds up the number of blocks to ensure that at least one additional core is available for each sample. Detailed information on this configuration is included in the Appendix.

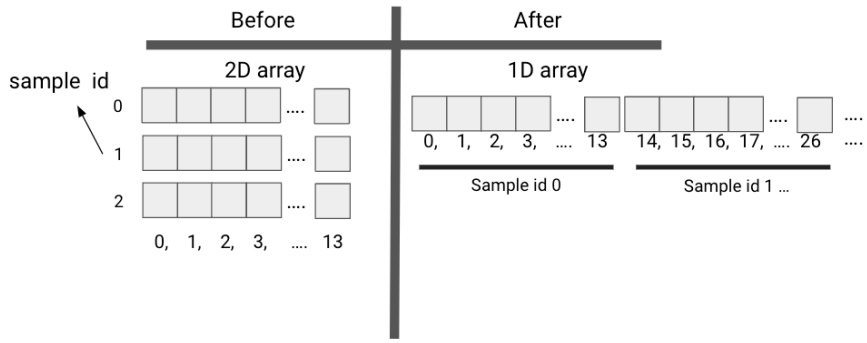
Offsetting is a technique used to manage data indexing efficiently, ensuring that thread indices correspond to the correct memory addresses.

This can reduce memory bank conflicts and improve overall performance. Proper offset calculation is also crucial when dividing large datasets across multiple threads and blocks, ensuring each thread processes its designated segment efficiently and correctly.

In this research, offsetting also used to simplify any multi-dimensional input. In the previous iteration of the simulation based on CPU, it uses vector of struct to temporarily store simulation results. In CUDA programming, there is no native multidimensional vector type like in higher-level programming languages. This research simplified all multi-dimensional vector used in the previous iteration into 1 dimensional (1D) array. Offsetting is mainly used for pointing the correct data in a flattened 1D array.

Since CUDA natively operates on linear memory, storing data in a flattened 1D format aligns well with the GPU's global memory, allowing for optimal performance while maintaining simplicity. In this method, a multidimensional array is represented as a single contiguous block of memory, and elements are accessed using calculated indices. For example, a 2D array with dimensions (rows, columns) can be indexed as current row multiplied by row size, added by current columns. This linearization simplifies memory allocation and transfer between the host and device, ensuring compatibility with CUDA's memory management functions.

Figure 2.2 explains graphically the flattening process and how multi-dimensional vector data differentiate samples (called as sample_id) in the previous iteration of drug toxicity *in silico* simulation.



[Figure 2. 2] Main difference in values storing paradigm after CUDA-parallelisation, assuming column size is 13.

The previous iteration of the code uses row to indicates different samples. Each sample will have their own identification number called the sample_id, so each row correlates to one sample_id. In the GPU version, instead of using rows to differentiate samples, it will utilise fact that each arrays have same number of columns.

For example, array STATES has 43 columns, then STATES[0] up to STATES[42] is reserved for sample_id = 0, STATES[43] up to STATES[84] is reserved for sample_id = 1, and so on. Adapting with this approach, the current index can be determined by knowing row dimension, sample_id and the number of specific columns selected. In the code from OpenCOR, most of the constants declarations, states calculations, rates calculations and

```
CONSTANTS[5] = 8314;
```

Or, for algebraic formulation

```
ALGEBRAIC[3] = 1.00000/(1.00000+exp((STATES[0]+87.6100)/7.48800))
```

At the top of the code from OpenCOR, it is mentioned how many CONSTANTS, STATES (similar to number of RATES), and ALGEBRAIC are there in the code. It looks like:

```
There are a total of 223 entries in the algebraic variable array.
There are a total of 43 entries in each of the rate and state variable arrays.
There are a total of 163 entries in the constant variable array.
```

These are going to be the row size, while number of sample will be the current row. Number of sample will be declared later in this chapter, with name 'sample_id'. Apply offsetting in every CONSTANTS, STATES (similar to RATES), and ALGEBRAIC array occur in the code by applying this to every array index:

```
new_index = (sample_id * row size) + columns.
```

Hence, all of the declarations and calculations in the code should look like:

```
CONSTANTS[(sample_id * 163) + 5] = 8314;
```

Or, for algebraic formulation

```
ALGEBRAIC[(sample_id * 223) + 3] = 1.00000/(1.00000+exp((STATES[(sample_id * 43) +
0]+87.6100)/7.48800));
```

Notice that the calculation for new index of each arrays are calculated within the declaration of each elements, and all arrays are treated the similar way.

2.3 Solving Ordinary Differential Equations

The model relies on algebraic calculations and dynamic functions expressed in the form of ordinary differential equations (ODEs), which are essential for simulating the complex behaviours of biological systems. To efficiently solve these ODEs within a CUDA-based parallel processing framework, two distinct numerical methods were employed depending on the specific cell model: the Rush-Larsen method and a custom implementation of the forward Euler method. These methods were chosen to balance computational efficiency, numerical stability, and compatibility

with the CUDA architecture. This research implements the ODE solver inside the cell model code as a function.

For the ORd 2011 model, the Rush-Larsen method was utilised due to its computational efficiency and computational stability in this context. This method effectively integrates stiff components of the equations, making it well-suited for the dynamic features of the ORd 2011 model. This method was implemented with a dynamic time-stepping mechanism by adjusting the time step during each iteration, while balancing acceptable numerical errors [17].

Initially, the Rush-Larsen method was meant to be used across all cell models. However, when applied to ORd 2017 and ToR-ORd models, this approach exhibited instability, failing to provide reliable results. To address this limitation, a simple forward Euler method was implemented for these two models. While the Euler method produced accurate and stable results, especially for the ORd 2017 and ToR-ORd models, it proved to be computationally intensive, significantly increasing the runtime.

Forward Euler is a simple method to solve ODE in particular time. The forward Euler method calculates the next value of a variable by taking its current value (STATES array) and adding the product of the rate (RATES array) of change and the time step (dt). This straightforward approach makes the method computationally simple and easy to implement. Mathematically, it is expressed as in Equation 1:

$$x_{n+1} = x_n + rate(x_n) \cdot \Delta t \quad (1)$$

where x_{n+1} is the current value, $rate(x_n)$ represents the rate of change at x_n, and Δt is the time step. In the converted code, a function should be added

to implement this calculation. This can be achieved by implementing for loop such as:

```
void solveEuler( double *STATES, double *RATES, double dt, int
sample_id)
{
    for(int i=0;i<43;i++){
    STATES[(43 * sample_id) + i] = STATES[(43 * sample_id) + i] +
RATES[(43 * sample_id) + i] * dt;
    }
```

The function uses a for loop to iterate over the 43 state variables associated with a single sample in ToR-ORd cell model. For each state variable, it calculates the new value using the forward Euler method formula. The formula is applied to the corresponding state variable and rate of the selected sample, determined by the sample_id. By multiplying sample_id by 43 (the number of state variables per sample in ToR-ORd cell model), the function accesses the correct block of memory in the flattened 1D array for both STATES and RATES. This ensures that updates are sample-specific and do not interfere with other samples.

2.4 Simulation Protocol and Code Organisation

To optimise the parallelisation process, algorithm was simplified, enables parallel threads to process multiple samples rather than multiple equations simultaneously. Despite the trade-off between computational speed and stability, this combination of methods ensures that the CUDA-based framework effectively supports the diverse requirements of different cell models, while maintaining accuracy and to reduce numerical errors.

The whole GPU simulation code repository consist of three main folders: bin, cell models, and modules. Bin folder stands for 'binary', means all compilation process starts here, and the executable will also available in this folder. This folder also has a folder for storing input data (named 'drugs' by default) and a folder named 'result' to collect all output from the simulation. This folder also contains a text file known as the input deck. Input deck holds changeable simulation parameters, without re-compiling the code. As some parts of the output are hard-coded, the 'result' folder cannot be changed or deleted, as it will result a segmentation fault or a crash at the end of the simulation.

The second folder is the 'cellmodels' folder. This folder holds codes of cell models like ToR-ORd, ORd 2011, ORd 2017, or others. Each cell model also requires a header file, so it can be run from another code. This folder also contains cell model header (cellmodel.hpp) which contains common functions and variables used in every of the cell models. Some of commonalities in these cell models are: 1) they have arrays of different sizes that need to be accounted for the offsetting, 2) initialisation function (initConsts function), and 3) ODE solver. As an addition to make identifying each gate, parameters, or variables in the cell model easier, enumeration was introduced as a header file. This header located inside the enums folder, in cell models folder. This header simply create enumerator for each index in each array. For example, in ToR-ORd cell model, CONSTANTS[1] is NaO (millimolar) in component extracellular. This header will enable CONSTANTS[1] aliased with CONSTANTS[nao] for easier tracing. Detail of all codes mentioned in this research will be attached in the Appendix.

Next, in the modules folder, there are a lot of utility codes stored in this folder. First, is the cipa_t header, that define custom struct datatype

that will store simulation biomarkers result. Then, there is global function script and header. Global function is used to read input deck, input flags, and some common variables. After that, there is global type script and header to declare custom data type to store drug data (IC50 and Hill fitting result). Next, there is script and header of parameters, named param.cpp and param.hpp. The parameters script acts as declarator to default simulation parameter, including the default input file directory. There is provided a default input file to act as 'failsafe', to ensure simulation still runs for checking even if user did not provide any input. The last one but the most important, is the gpu.cu and gpu.cuh.

The file `gpu.cu` is central to all parallel processing in this research. It is specifically designed to handle computational tasks leveraging GPU acceleration, ensuring efficient parallel execution. Within `gpu.cu`, a key function named 'kernel_DrugSimulation' manages the parallelisation process. This function is responsible for batching operations, determining the data to be processed, and managing memory sharing among threads. This primary function orchestrates the parallelisation by distributing tasks across thousands of GPU threads. Then, each thread runs the same function, the simulation function concurrently but processes a distinct portion of the data. The key function then calls the simulation function helper functions are called within this function, and these are effectively "multiplied" across the threads, each executing independently. This design ensures the workload is evenly distributed and processed simultaneously, fully utilising the computational power of the GPU. By encapsulating the core parallel processing logic within 'gpu.cu', the research achieves a streamlined and modular structure, making it easier to maintain, optimise the GPU-specific operations, and simplicity for future modifications.

The header for the key GPU code, ‘gpu.cuh’ help to declare three different functions, the key function, and two other simulation functions, named ‘kernel_DoDrugSim’ and ‘kernel_DoDrugSim_single’. Similar to their names, ‘kernel_DoDrugSim’ was designed to being run multiple times, and ‘kernel_DoDrugSim_single’ was designed to run only for one or two times to collect necessary simulation result. The ‘kernel_DoDrugSim’ will run for thousands of paces first, amplifying the drug effect in the simulation, then ‘kernel_DoDrugSim_single’ will run only once or twice (depending on the cell model) to capture and calculate important simulation results in the form of time series data and features called biomarkers. This research split the drug effect amplification and data capture process due to memory limitation.

The method in ‘kernel_DoDrugSim_single’ will take up to 60% more memory to save all necessary information. Because ‘kernel_DoDrugSim’ can save memory usage, the remain memory space can be utilised to put more samples, trading off some feature calculation away. The output from ‘kernel_DoDrugSim’ will be used as input for ‘kernel_DoDrugSim_single’. The ‘kernel_DoDrugSim_single’ measures the amplified drug effect after thousands of paces.

2.5 Output Format

The simulation produces two distinct types of output files, a biomarker file and time-series data files, along with one intermediate cache file. The cache file is generated as the output of the `kernel_DoDrugSim` function, which represents the initial phase of the simulation. During this phase, the function runs the simulation for thousands of cycles (referred to as paces) to amplify the drug effects within the model. After this initial phase,

the `kernel_DoDrugSim_single` function is executed, which generates the biomarker file and the time-series data files. All output files are organised into a dedicated folder within the `result` directory for efficient storage and retrieval.

The biomarker file provides a summary of key features extracted from the simulation for each sample. It includes data such as sample numbers, qNet, qInward, and action potential shape analysis result. These biomarkers represent crucial physiological parameters simulated under drug influence, and they are instrumental for downstream analyses, such as machine learning-based predictions. The time-series file offers a detailed temporal view of each sample's behaviour. Each sample has its own individual time-series file; thus, a simulation involving 2000 samples will result in 2000 time-series files. These files capture parameters such as time, action potential, voltage gradient over time, Cai, INa, INaL, ICaL, IKs, IKr, IK1, and Ito. Using this detailed data, it is possible to plot the drug-induced cellular responses over a single cycle, facilitating visualisation and deeper analysis of dynamic behaviours.

2.6 Compilation, Input Files, and Testing

This subchapter explains the methods used to compile the code developed in this research, the input files required for the simulation, and the testing procedures employed to validate the simulation results. The code repository for this research contains several scripts that must be compiled and linked to run the simulation. A Makefile was used to streamline the compilation process, ensuring that all files are compiled in the correct order and linked to CUDA's system library. Additionally, this subchapter details

the required input files, including how to define them when running the simulation. Finally, it outlines the testing process, where the simulation outputs are compared with results generated by OpenCOR to verify accuracy and confirm successful simulation.

2.6.1 Compiling with makefile

This research's code repository provided Makefile to outline a clear and structured approach to compiling the CUDA-based C++ scripts and headers. It incorporates key elements like specifying source files, dependencies, compilation flags, and cleaning commands. Key concepts, such as the use of := for immediate value assignment, are highlighted to ensure consistent behaviour during execution. Commentary further guides users on technical aspects. Overall, this Makefile is structured to handle complex dependencies, optimise for GPU-based execution, and maintain flexibility for evolving project requirements. Its modular approach, detailed flag definitions, and automated file management make it a robust tool for managing the compilation process in this drug simulation research project. The Makefile for this research will be provided in the appendix.

The .PHONY directive specifies that all and clean are symbolic targets rather than actual files or directories. This ensures that Make does not mistake them for real entities during execution. The PROGNAME variable is used to define the final executable's name (drug_sim), making it easy to change the output program's name if needed. These simple, high-level definitions contribute to a more modular and maintainable Makefile.

The Makefile specifies nvcc, NVIDIA's CUDA compiler, as the compiler, ensuring compatibility with GPU-based computation. It also

utilises nvlink for linking purposes. Key flags are defined for both compilation and linking. CPPFLAGS includes the path to CUDA header files, while CXXFLAGS incorporates options such as -Wall for warnings, -O2 for optimisation, and -fpermissive for handling relaxed syntax. The LDFLAGS variable defines linker flags, such as paths to CUDA libraries, GPU architecture specifications (-arch=sm_86), and relocatable device code support (-rdc=true), ensuring optimal performance on the targeted GPU architecture.

To manage source and header files, the Makefile uses wildcard functions to automatically detect files with relevant extensions (e.g., .cpp, .cu, .c for source files and .hpp, .h, .cuh for headers). This approach ensures that the build process dynamically adapts to new files added during development, eliminating the need for manual updates to the Makefile. Object files are generated by substituting the .cpp extension with .o, ensuring a seamless mapping from source to object files. The all target is set as the default goal, compiling the entire project by depending on the program name ($(PROGNAME)). The linking process combines all object files into the final executable using the specified compiler and linker flags. Explicit rules are defined for generating object files from source files, utilising CUDA-specific flags such as -x cu, -dc, and -arch=sm_86 to ensure compatibility with GPU-based parallel processing.

A ‘clean’ target is included to facilitate the removal of temporary files such as object files (*.o) and the executable. This ensures a clean workspace for subsequent builds. The @ symbol suppresses the command echo, while the - symbol allows the process to continue even if errors occur, such as when files are missing. Additionally, the LIBS variable specifies external libraries like OpenBLAS and CUDA-specific libraries (-lopenblas,

-lpthread, -lcudart, -lcublas) that are crucial for performing mathematical and parallel computations.

2.6.2 Required Input Files

To execute the simulation, two essential input files are required: an input deck and a drug data file (IC50 and Hill coefficient values in one file). These files serve as the foundation for configuring the simulation environment and defining the drug properties necessary for the analysis. Together, they ensure the simulator has the parameters and data needed to accurately model the effects of the drug on the cell samples.

The first file, known as the input deck, is a text file containing simulation parameters. This file specifies crucial settings, such as the number of simulation steps, time intervals ('dt'), cell model identifiers, and other key configurations that govern how the simulation is run. By modifying this file, researchers can adapt the simulation to different experimental conditions without altering the underlying code. The flexibility provided by the input deck allows for efficient experimentation and testing across a wide range of scenarios. By default, an input deck file contains:

- Basic_Cycle_Length (length of one cycle in millisecond) = 1000
- Number_of_Pacing (number of cycle) = 1000
- Simulation_Mode = 0
- Celltype = 0 (type of cell to simulate) (0: endo, 1: epi, 2: M
- Is_Dutta = 1 (means conductance scaling from Dutta et al. 2017)
    - Dutta's conductance scaling may vary. ToR-ORd cell model do not require this.

- Is_Post_Processing = 0 (set 0 to use 'kernel_DoDrugSim' or set 1 to use 'kernel_DoDrugSim_single', run mode 0 first, then mode 1)
- Use_Conductance_Variability = 0 (1: read additional file which contain individual conductance variability)
- Pace_Find_Steepest = 250 (timing to start searching steepest time gradient in repolarisation It means searching from last 250 cycles, or cycle number 750 to 1000) (minimum value: 2)
- Drug_Name = quinidine (change as needed)
- Concentrations = 3237.0 (concentration of the drug in mMol)
- GPU_Index = 0 (choose which GPU will run the simulation, a PC with 1 GPU should let it 0)

The second file is a CSV file containing drug-specific data, particularly the IC50 and Hill coefficient values. These pharmacological parameters are critical for modelling the drug's effect on ion channels and other cellular processes. The IC50 value represents the drug concentration at which 50% of its maximal inhibitory effect is observed, while the Hill coefficient describes the slope of the dose-response curve, indicating the cooperativity of drug binding [18]. This file provides the simulator with the 7 IC50 7 Hill coefficients data to simulate the drug's interaction with the cells accurately.

These required input files declared by adding flags in the running parameter. After compilation, the compiled simulator will be available in the 'bin' folder. Then add two flags when running drug_sim, -input_deck declares the location of the input deck file, and -hill_file declares the location of IC50 and hill file, such as:

```
./drug_sim -input_deck input_deck.txt -hill_file drug/IC50_file.csv
```

By using these two files, the simulator integrates user-defined configurations with drug-specific properties, creating a dynamic and adaptable environment for simulating drug-induced cellular responses. The modularity of this input system ensures that new parameters or drugs can be tested efficiently, making the simulation framework highly versatile and scalable for diverse research needs.

2.6.3 Testing and Result Validation Method

Testing and validating the results of the simulator is a critical step to ensure its accuracy and reliability. The process begins with compiling the code and resolving any errors that may arise during the compilation stage. This involves carefully examining the Makefile and codebase to ensure all dependencies are correctly linked and that no syntax or compatibility issues are present. Once the code is successfully compiled, it can then proceed to the testing phase. The initial testing phase involves running the simulator for 10 to 100 iterations. This step aims to confirm that the program executes correctly without unexpected crashes or errors. During this phase, intermediate outputs are monitored to verify that the calculations align with expected values. Any discrepancies observed at this stage are investigated and resolved before moving forward.

Once the basic functionality is verified, the simulator undergoes a more rigorous testing phase by running a full GPU-based simulation of 1000 paces. This comprehensive test ensures that the GPU's parallel processing capabilities are functioning as intended and that the system can handle the computational load efficiently. The outputs of this simulation, including both

biomarker files and time-series data, are then compared to the results obtained from running the same simulation in OpenCOR.

The comparison between the GPU-based simulator and OpenCOR serves as a key validation step. For the results to be considered accurate, the discrepancies in time-series data between the two methods should not be able to visually distinguished when stacked onto one plot. Any significant deviations are analysed to determine whether they result from numerical methods, precision differences, or potential errors in implementation. By following this systematic approach to testing and validation, the reliability of the simulator is established, providing confidence in its ability to produce accurate and meaningful results for drug-induced cellular response simulations.

Chapter 3. Results and Discussion

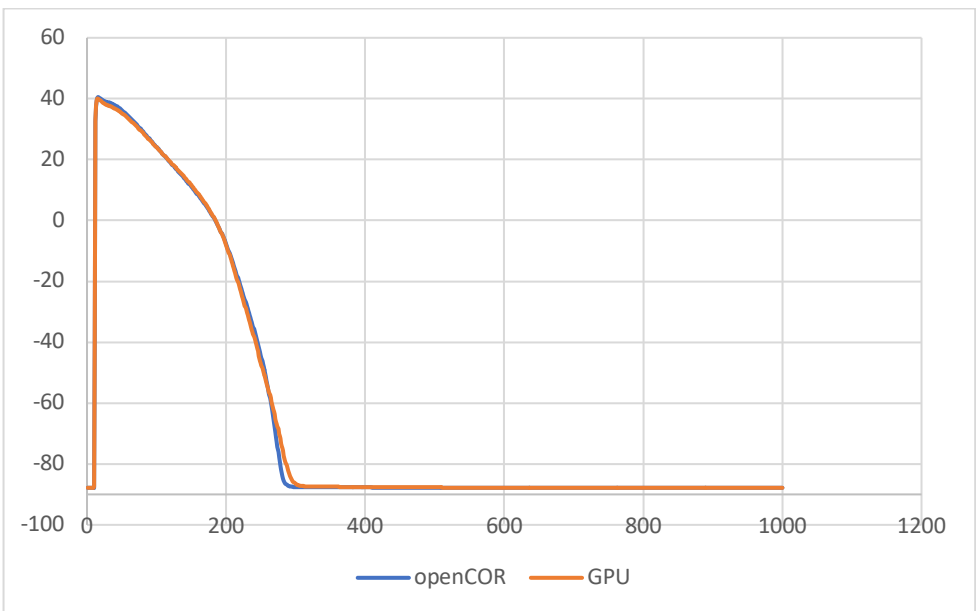
This chapter presents the findings of the GPU-based simulation for three different cell models: ORd 2011, ORd 2017, and ToR-ORd. Each section provides a detailed analysis of the results obtained from each cell model simulation result. In all simulations, the simulation accuracy is validated, the drug-induced changes are analysed, and computational performance is reviewed. There are two kind of simulation tested on these three cell models: Control, and drug-induced simulation. All control simulation results obtained in no-drug situation, and the drug bepridil was simulated to analyse drug-induced changes in the drug-induced simulation. This research uses bepridil with concentration of 33 mMol (cmax 1), 66 mMol (cmax 2), and 132 mMol (cmax 4). All result both with and without drug effect were run for 1000 pacing, and each pace lasts for 1000 milliseconds.

3.1 GPU Simulation Result Using ORd 2011 Cell Model

This section examines the results of GPU-based simulation using the ORd 2011 cell model. The ODE solver in ORd 2011 model was using the Rush-Larsen method which offers faster computational time by optimising the handling of gating variables in the equations. This approach not only accelerates simulations but also ensures sufficient numerical stability for this cell model.

3.1.1 Result Validation

To validate the results of the GPU simulation, this research compared the action potential outputs against reference solutions obtained from the OpenCOR, running under CPU. Visual comparisons of time-series plots for action potentials were performed to ensure qualitative agreement. Key electrophysiological biomarker, such as action potential duration were also compared. These biomarkers from CPU and GPU were compared under a same, no drug conditions, ensuring that the GPU-based simulation accurately reproduces the physiological dynamics from the ORd 2011 model. The findings revealed the GPU simulation result is almost exactly same with its CPU predecessor. Figure 3.1 shows visually action potential from both simulation platforms.

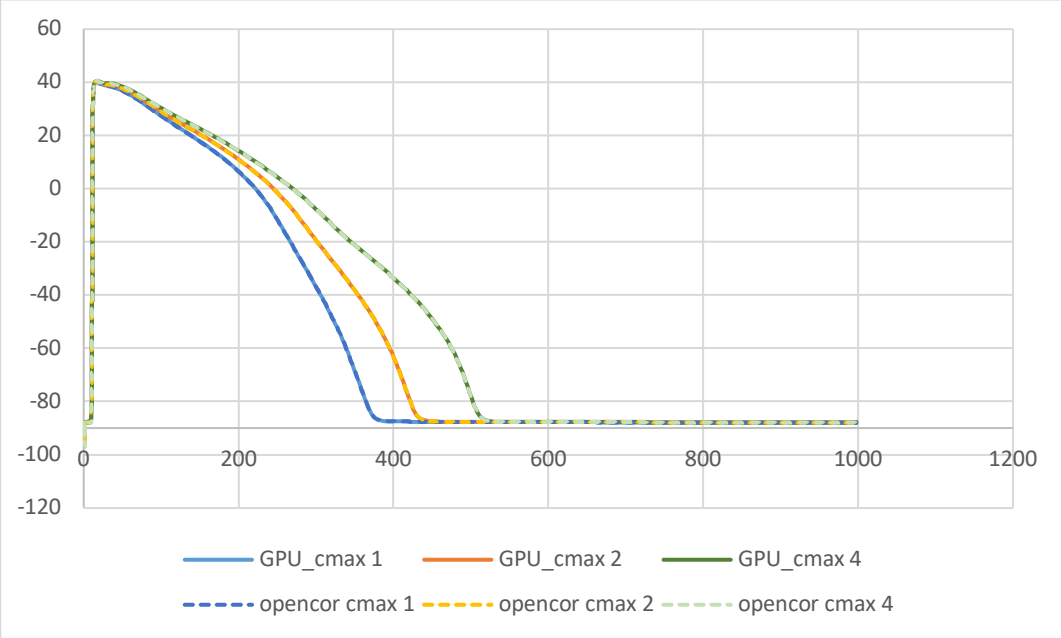


[Figure 3. 1] Action Potential (mV) Shape of both CPU (blue) and GPU (orange) Result Using ORd 2011

As shown, little to no difference from both of the result, indicating a valid result from the GPU-based simulation. Promising more efficient *in silico* drug cardiotoxicity prediction.

3.1.2 Result Validation Under Drug

This section evaluates the accuracy of the GPU-based simulation for the ORd 2011 cell model under drug conditions. Drug effects were modelled by adjusting ionic current parameters using IC50 and Hill coefficient values, applied consistently in both GPU and CPU (OpenCOR) simulations. Validation involved comparing action potential traces and key biomarker, such as action potential shape.

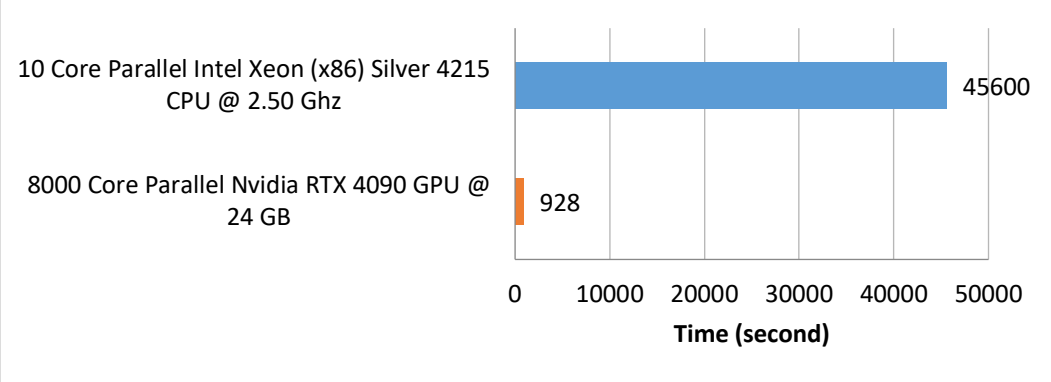


[Figure 3. 2] Action Potential Shape (mV) of both CPU (dashed) and GPU under drug effect Using ORd 2011

The results from figure 3.2 showed that despite the added complexity of drug effects, the GPU simulation closely matched the outputs of the CPU-based OpenCOR simulation. This confirms the GPU implementation's reliability in replicating physiological and pharmacological responses under drug conditions.

3.1.3 Computational Time and Efficiency Analysis

This analysis compares computational time between two hardware. GPU based simulation executed using NVIDIA RTX 4090 with 24 GB of memory. CPU based simulation also uses parallel processing of 10 cores Intel Xeon (x86) Silver 4215 CPU @ 2.50 GHz. The computational time compared for 8000 samples (each drug has 2000 samples, and 4 concentrations). In theory, GPU cores operate at lower clock speeds, making them inherently less powerful than CPU cores, which is why CPUs are typically preferred for single-sample simulations. The computation time for CPUs increases linearly with the sample size and pacing, meaning that as the number of samples grows, so does the time require for computation.



[Figure 3. 3] Simulation time comparison between GPU and CPU in ORd 2011

In contrast, GPU parallelisation eliminates this linear growth as seen in the figure 3.3. the time it takes to compute one sample is nearly the same regardless of how many samples are processed, due to its parallel computing architecture. GPU achieved a speedup of up to 40.91 times compared to a 10-core CPU. GPU requires about 928 seconds to complete the simulation, regardless of whether it handles a single sample or 8,000 samples. In contrast, the computation time for the CPU system with 10 cores grows by the number of samples being simulated. Simulating 8000 samples would take the CPU system around 45,600 seconds. From this analysis, the GPU becomes the more efficient option when simulating 163 samples or more, as its fixed runtime of 928 seconds is significantly faster than the increasing runtime of the CPU.

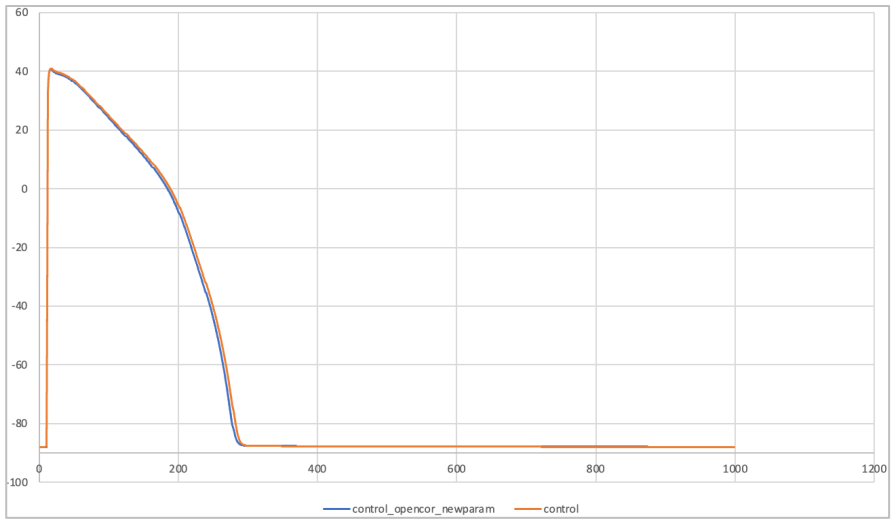
3.2 GPU Simulation Result Using ORd 2017 Cell Model

This section explores the outcomes of the GPU-based simulation using the ORd 2017 cell model. Unlike the ORd 2011 model, which leverages the efficiency of the Rush-Larsen method, the ORd 2017 model uses the forward Euler method for solving ordinary differential equations. While the forward Euler method is straightforward to implement and numerically stable for this model, it results in slower computational times compared to the Rush-Larsen method. This trade-off is necessary due to the instability observed when using the Rush-Larsen method with the ORd 2017 equations.

3.2.1 Result Validation

Similar to previous, validation of GPU simulation results for the ORd 2017 cell model was conducted by comparing outputs with the reference

solutions generated from OpenCOR. Visual comparisons of action potential time-series data confirmed a close alignment between the two simulation platforms.

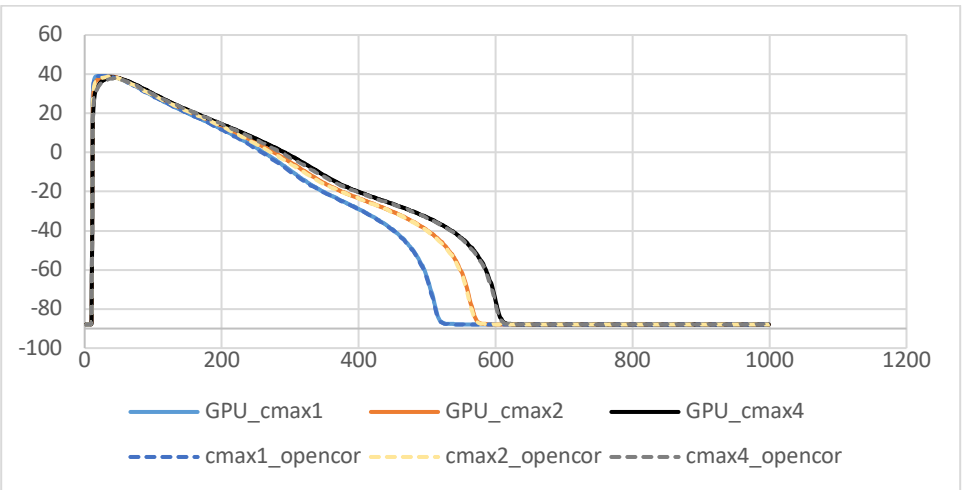


[Figure 3. 4] Action Potential (mV) Shape of both CPU (blue) and GPU (orange) Result Using ORd 2017

Key biomarker such as action potential duration, were analysed under identical no-drug conditions to ensure the physiological fidelity of the GPU-based results. As with the ORd 2011 model, the findings demonstrated that the GPU simulation similarly replicated the results from the CPU-based OpenCOR simulations. The numerical outputs showed no significant differences, confirming the reliability and accuracy of the GPU implementation for the ORd 2017 model. Figure 3.4 provides a visual comparison of the action potentials produced by the GPU and CPU simulations, illustrating their near-identical behaviour.

3.2.2 Result Validation Under Drug

In this section, the accuracy of the GPU-based simulation for the ORd 2017 cell model was evaluated under drug conditions. The simulation incorporated drug-induced effects by modifying ionic current parameters based on IC50 and Hill coefficient values. These adjustments were applied uniformly across both the GPU and CPU (OpenCOR) simulations to ensure consistency in the drug response modelling. The validation process involved comparing action potential traces and key electrophysiological biomarker, such as action potential, between the GPU and CPU simulations.



[Figure 3. 5] Action Potential Shape (mV) of both CPU (dashed) and GPU under drug effect Using ORd 2017

Despite the added complexity of drug effects, the GPU simulation produced outputs that were nearly identical to those of the CPU-based OpenCOR simulations, as shown in figure 3.5. These findings confirm that the GPU implementation of the ORd 2017 model accurately captures the physiological and pharmacological responses of the cell model under drug conditions. This

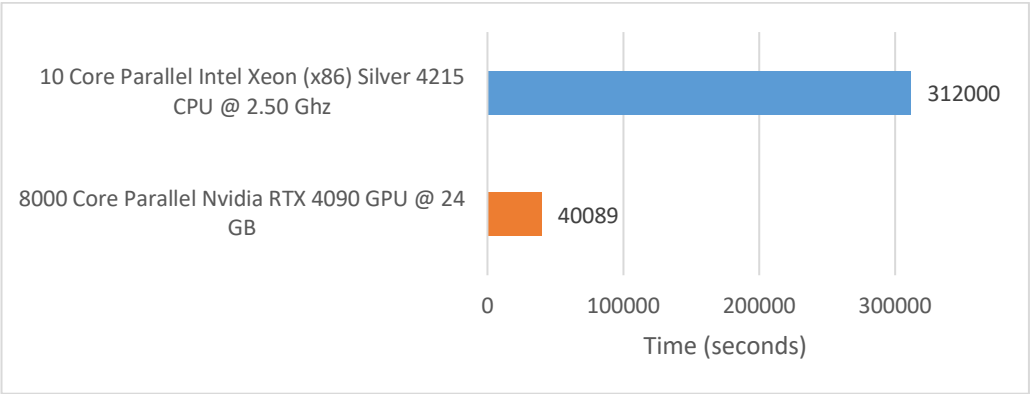
validation further establishes the robustness and reliability of the GPU-based simulation for scenarios involving drug effects.

3.2.3 Computational Time and Efficiency Analysis

This section examines the computational performance of the GPU-based simulation for the ORd 2017 cell model, compared against a 10-core Intel Xeon (x86) Silver 4215 CPU @ 2.50 GHz. The GPU simulations were executed using an NVIDIA RTX 4090 with 24 GB of memory. Both hardware setups processed 8000 samples (2000 samples per drug at four different concentrations). The computation time on CPUs scales linearly with the number of samples due to sequential processing limitations, even when multiple cores are utilised. However, the GPU's parallel processing architecture allows it to maintain consistent computational times regardless of sample size, effectively minimising linear growth in execution time. While GPUs generally operate at lower clock speeds compared to CPUs, their ability to handle large-scale parallel tasks offers a significant advantage. The GPU's simulation time is nearly constant regardless of the number of samples due to its parallel processing capability.

For this simulation, the GPU takes approximately 40,089 seconds to simulate any number of samples, whether it's 1 or 8,000. On the other hand, the CPU system, using 10 CPUs in parallel, takes about 390 seconds to simulate one sample on each CPU. This means that the 10-core CPU system can complete 10 samples in 390 seconds. Therefore, to simulate all 8,000 samples, the CPU system would require a total of 312,000 seconds. Based on this information, it is calculated that for fewer than 1,028 samples, the CPU system with 10 cores is more efficient, as its total computation time

(less than 40,089 seconds) would be faster than the GPU's fixed simulation time.



[Figure 3.6] Simulation time comparison between GPU and CPU in ORd 2017

For the ORd 2017 cell model, the GPU-based simulation achieved a speedup of up to 7.78 times compared to the 10-core CPU implementation. As shown in figure 3.6, GPU parallelisation is not as dominant compared to GPU parallelisation in ORd 2011. This lower speedup compared to the ORd 2011 cell model is attributed to the use of the Forward Euler method, which is computationally more demanding than the Rush-Larsen method. Nevertheless, the GPU remains significantly more efficient for large-scale simulations.

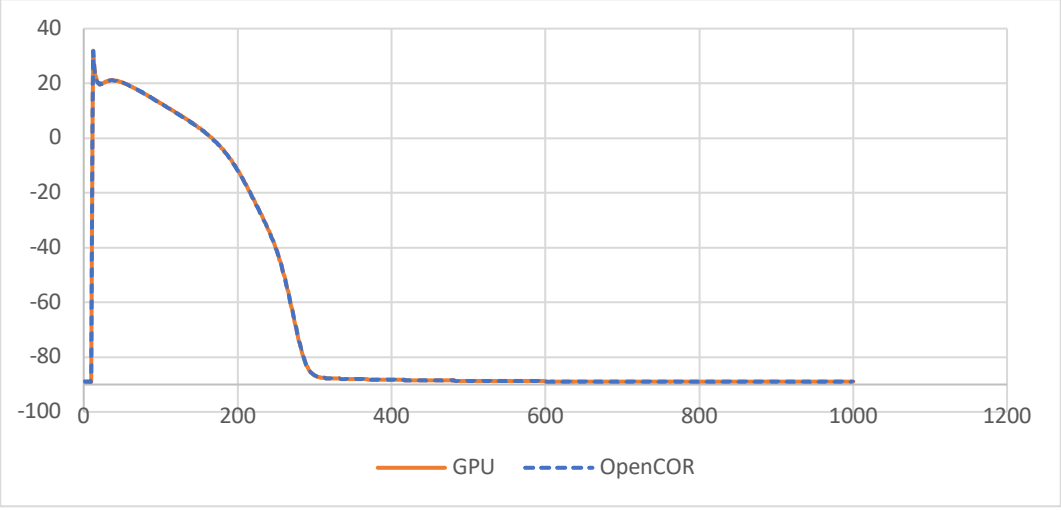
3.3 GPU Simulation Result Using ToR-ORd Cell Model

This section highlights the results obtained from GPU simulations using the ToR-ORd cell model. Similar to the ORd 2017 model, the forward Euler method was employed as the ODE solver. While this method provides adequate numerical stability and robustness for the ToR-ORD model, it results in longer computational times compared to the more efficient Rush-

Larsen method used in the ORd 2011 simulations. Regardless of the computational time drawback, forward Euler method ensure the simulation generates a reliable and usable result.

3.3.1 Result Validation

The validation process for the ToR-ORd cell model involved comparing GPU simulation outputs with the benchmark results obtained from OpenCOR. Time-series plots of action potentials were evaluated for qualitative consistency, and key biomarker, such as action potential duration, were quantitatively assessed under no-drug conditions.



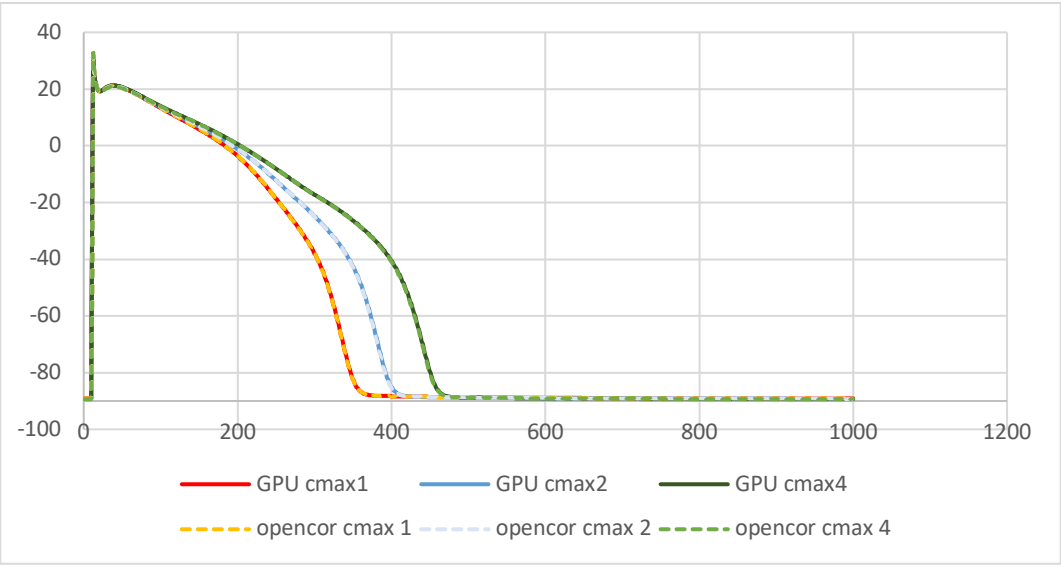
[Figure 3. 7] Action Potential (mV) Shape of both CPU (dashed blue) and GPU (orange) Result Using ToR-ORd

The results from figure 3.7 confirmed the accuracy of the GPU simulation, as it produced outputs that matched the CPU-based OpenCOR simulations without any discernible differences. This validation underscores the reliability of the GPU implementation for the ToR-ORd cell model, even

when employing the forward Euler method. Figure 3.6 illustrates the action potential traces from both GPU and CPU simulations, demonstrating their near-identical nature.

3.3.2 Result Validation Under Drug

For the validation in ToR-ORd cell model under drug, drug effects were incorporated into the simulation by altering ionic currents based on predefined IC50 and Hill coefficient parameters. Both GPU and CPU (OpenCOR) simulations were configured identically to ensure fair comparison.



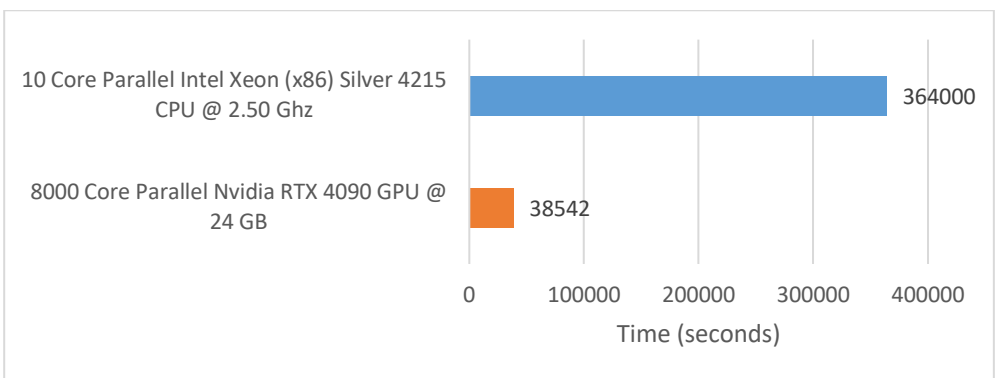
[Figure 3. 8] Action Potential (mV) Shape of both CPU (dashed) and GPU under drug effect Using ToR-ORd cell model

The results in figure 3. 8 showed that the GPU simulation accurately replicated the outputs of the CPU-based simulations, with no significant differences observed in action potential traces or in biomarkers such as APD, ionic current dynamics, and calcium handling. This consistency

demonstrates the validity of the GPU-based simulation for the ToR-ORd cell model, even when subjected to drug-induced perturbations. The successful validation of drug effects in the GPU simulations highlights the method's capability to simulate complex pharmacological scenarios reliably. These findings reinforce the utility of GPU-based simulations as a powerful tool for investigating drug-induced cellular behaviours.

3.3.3 Computational Time and Efficiency Analysis

The ToR-ORd cell model was also analysed using the same hardware: an NVIDIA RTX 4090 GPU and a 10-core Intel Xeon (x86) Silver 4215 CPU @ 2.50 GHz. The GPU executed the simulations for 8000 samples (2000 samples per drug at four concentrations), matching the experimental conditions of the CPU. Similar to the ORd 2017 cell model, the computational time for the ToR-ORd model on CPUs increased linearly with the number of samples and pacing due to sequential processing. In contrast, the GPU's architecture showed consistent computational performance, demonstrating parallelisation for large-scale simulations.



[Figure 3. 9] Simulation time comparison between GPU and CPU in ToR-ORd cell model

As shown in figure 3.9, the ToR-ORd model GPU-based simulation achieved a speedup of up to 9.44 times compared to the 10-core CPU implementation. It is calculated that for simulations with fewer than 847 samples, the 10-core CPU system is faster, as its total computation time is less than the GPU's fixed time of 38,542 seconds. Beyond this point, the GPU becomes more efficient due to its ability to process a large number of samples simultaneously without increasing computation time. While not as high as the ORd 2011 model, this speedup highlights the GPU's capability to handle complex simulations efficiently, even with computationally intensive solvers like Forward Euler. Overall, the GPU provides a substantial time-saving advantage across all tested cell models.

Chapter 4. Conclusion and Limitation

4.1 Conclusion

This study effectively addressed the computational challenges for *in silico* simulations for predicting cardiovascular drug toxicity by leveraging CUDA-based parallel processing. The optimized GPU approach achieved up to 40.91 times faster simulation speeds compared to traditional 10-core CPU methods, enabling the handling of larger datasets without significant performance loss. Validation results confirmed the accuracy of GPU simulations, even under suboptimal conditions, demonstrating their reliability for efficient and precise toxicity predictions. This advancement positions GPU-based methods as a cost-effective and practical alternative to CPU-based simulations in large-scale drug discovery research.

4.2 Suggestions

To further enhance the utility and impact of this research, several suggestions can be made. Expanding the model complexity by incorporating additional biological variables could improve the accuracy of simulations. Testing the methodology on diverse GPU hardware models would provide insights into performance variations across systems, ensuring broader applicability. The method's potential for industrial application is also notable, particularly in pharmaceutical pipelines where it could support large-scale drug development.

Furthermore, assessing the economic viability of the GPU-based approach, including long-term operational costs and energy efficiency, could ensure its sustainability. Standardising protocols for simulation workflows would help maintain consistency and reproducibility, which are critical for scientific and industrial adoption. Finally, fostering multidisciplinary collaboration with experts in pharmacology, bioinformatics, and hardware engineering could enhance the methodology's robustness and broaden its applicability. By addressing these areas, this CUDA-based approach could further solidify its role as a transformative tool in *in silico* drug discovery, reducing reliance on animal testing and accelerating the pace of research.

[References]

1. Jason Sanders and Edward Kandrot. 2010. CUDA by Example: An Introduction to General-Purpose GPU Programming (1st. ed.). Addison-Wesley Professional.
2. “What is OpenMP?” Accessed: Nov. 10, 2024. [Online]. Available: https://cvw.cac.cornell.edu/openmp/intro/what-is-openmp.
3. R. L. Graham, G. M. Shipman, B. W. Barrett, R. H. Castain, G. Bosilca and A. Lumsdaine, "Open MPI: A High-Performance, Heterogeneous MPI," 2006 IEEE International Conference on Cluster Computing, Barcelona, Spain, 2006, pp. 1-9, doi: 10.1109/CLUSTR.2006.311904.
4. A. Garny et al., “CellML and associated tools and techniques,” Sep. 13, 2008, Royal Society. doi: 10.1098/rsta.2008.0094.
5. M. Gómez, J. Carro, E. Pueyo, A. Pérez, A. Oliván, and V. Monasterio, “In Silico Modeling and Validation of the Effect of Calcium-Activated Potassium Current on Ventricular Repolarization in Failing Myocytes,” *IEEE J Biomed Health Inform*, pp. 1–9, 2024, doi: 10.1109/JBHI.2024.3495027.
6. C. M. Lloyd, J. R. Lawson, P. J. Hunter, and P. F. Nielsen, “The CellML Model Repository,” *Bioinformatics*, vol. 24, no. 18, pp. 2122–2123, 2008, doi: 10.1093/bioinformatics/btn390.
7. N. Le Novère et al., “BioModels Database: a free, centralized database of curated, published, quantitative kinetic models of biochemical and cellular systems.,” Nucleic Acids Res, vol. 34, no. Database issue, 2006, doi: 10.1093/nar/gkj092.

8. M. Berghoff, J. Rosenbauer, F. Hoffmann, and A. Schug, “Cells in Silico-introducing a high-performance framework for large-scale tissue modeling,” *BMC Bioinformatics*, vol. 21, no. 1, Oct. 2020, doi: 10.1186/s12859-020-03728-7.
9. M. Martínez-del-Amor, I. Pérez-Hurtado, D. Orellana-Martín, and M. J. Pérez-Jiménez, “Adaptative parallel simulators for bioinspired computing models,” *Future Generation Computer Systems*, vol. 107, pp. 469–484, Jun. 2020, doi: 10.1016/j.future.2020.02.012.
10. S. McIntosh-Smith, J. Price, R. B. Sessions, and A. A. Ibarra, “High performance in silico virtual drug screening on many-core processors,” *International Journal of High Performance Computing Applications*, vol. 29, no. 2, pp. 119–134, May 2015, doi: 10.1177/1094342014528252.
11. P. Amar, M. Baillieul, D. Barth, B. LeCun, F. Quessette, and S. Vial, “Parallel Biological In Silico Simulation,” Nov. 2014, doi: 10.1007/978-3-319-09465-6_40.
12. D. G. Whittaker, J. C. Hancox, and H. Zhang, “*In Silico* Assestment of Pharmacotherapy for Human Atrial Patho-Electrophysiology Associated With hERG-Linked Short QT Syndrome” Jan. 2019, doi: 10.3389/fphys.2018.01888.
13. T. O'Hara, L. Virág, A. Varró, and Y. Rudy, “Simulation of the Undiseased Human Cardiac Ventricular Action Potential: Model Formulation and Experimental Validation.” 2011, PLOS Computational Biology 7(5): e1002061. Available: https://doi.org/10.1371/journal.pcbi.1002061.
14. S. Dutta, K.C. Chang, K.A. Beattie, J. Sheng , P.N. Tran, W.W. Wu, M. Wu, D.G. Strauss, T. Colatsky, and Z. Li. "Optimization of an In silico

Cardiac Cell Model for Proarrhythmia Risk Assessment." Front Physiol. Aug 2017 doi: 10.3389/fphys.2017.00616.

15. Tomek, Jakub et al. “Development, calibration, and validation of a novel human ventricular myocyte model in health, disease, and drug block.” eLife vol. 8 e48890. Dec 2019, doi:10.7554/eLife.48890.
16. “Parallel Thread Execution ISA Version 8.5” Accessed: Nov. 24, 2024. [Online]. Available: https://docs.nvidia.com/cuda/parallel-thread-execution/index.html.
17. C., Yves, C. D. Lontsi, and C. Pierre. "Rush-Larsen time-stepping methods of high order for stiff problems in cardiac electrophysiology." 2017, arXiv preprint arXiv:1712.02260.
18. G.R. Mirams, Y. Cui, A. Sher, M. Fink, J. Cooper, B. M. Heath, et al. "Simulation of multiple ion channel block provides improved early prediction of compounds' clinical torsadogenic risk." 2011, Cardiovasc. Res. 91 (1), 53-61. doi:10.1093/cvr/cvr044.

Appendix

A. Project Structure

Codes for this research is available on Github repository of Computational Medicine Laboratory, Kumoh National Institute of Technology (https://github.com/kit-cml). There are some versions of the code in our Github profile, depending on the cell models and research scenario. This section will discuss deeply the code's version for ORd 2011 cell model (https://github.com/kit-cml/MultiConcGPU). Codes explained in this appendix is in same condition as the time this thesis being written, including some files that might be deleted due to redundancy in the future. More recent updates are available in the Github repository. Structure of the repository can be written as such:

```
MultiConcGPU
├── .gitignore
├── Makefile
├── Makefile_commercial
├── bin
│   ├── CVAR
│   │   ├── HF_male.csv
│   │   └── sens_healthy_male.csv
│   ├── autorun.sh
│   ├── control
│   │   ├── IC50_samples.csv
│   │   └── init_state.csv
│   ├── drug
│   │   ├── IC50_terfenadine.csv
│   │   ├── IC50_verapamil.csv
│   │   └── testing
│   │       └── IC50_Mexiletine.csv
```

```
|   ├── drug_sim
|   ├── ic50_sens
|   |   └── mitoxantrone_sens.csv
|   ├── input_deck.txt
|   ├── mitoxantrone
|   |   └── mitoxantrone_100_samples_50_conc.csv
|   └── result
|       ├── do_not_delete_this_folder
|       └── state_only.zip
├── cellmodels
|   ├── Ohara_Rudy_2011.cpp
|   ├── Ohara_Rudy_2011.hpp
|   ├── cellmodel.hpp
|   └── enums
|       ├── enum_Ohara_Rudy_2011.hpp
|       └── enum_ord2011.hpp
├── main.cu
├── modules
|   ├── cipa_t.cu
|   ├── cipa_t.cuh
|   ├── drug_conc.cpp
|   ├── drug_conc.hpp
|   ├── glob_funct.cpp
|   ├── glob_funct.hpp
|   ├── glob_type.cpp
|   ├── glob_type.hpp
|   ├── gpu.cu
|   ├── gpu.cuh
|   ├── gpu_cu_.backup
|   ├── gpu_cuh.backup
|   ├── gpu_glob_type.cu
|   ├── gpu_glob_type.cuh
|   ├── param.cpp
|   └── param.hpp
└── test_compile.bat
```

The main structure, such as folder names, will less likely to be revised. Scripts inside the folder, especially in the 'modules' folder, more likely to be changed by removing some redundant functions. Next section will deeply discuss each file and their possibility of redundancy.

B. Root folder

Root folder is the main folder that contains makefile, gitignore file, main code, and test_compile.bat. The next sub section will discuss each of them deeply.

a. Makefile

This script is used for compiling the whole project in correct order, and enables easy clean-up and re-compilation. Binary files will be cleaned, and the simulator will be re-compiled with 'make clean all' command. Compilation configuration and steps described as follow:

```
# Some notes:
# - Using ':=' instead of '=' assign the value at Makefile parsing time,
#   others are evaluated at usage time. This discards
# - Use ':set list' in Vi/Vim to show tabs (Ctrl-v-i force tab insertion)
#

# List to '.PHONY' all fake targets, those that are neither files nor folders.
# "all" and "clean" are good candidates.
.PHONY: all, clean

# Define the final program name
PROGNAME := drug_sim

# Pre-processor flags to be used for includes (-I) and defines (-D)
CPPFLAGS := -I/usr/local/cuda/include
# CPPFLAGS :=
# CXX to set the compiler
# CXX := g++
CXX := nvcc
CXXLINK := nvlink

# CXXFLAGS is used for C++ compilation options.
```

```
        #CXXFLAGS += -Wall -O0 -fpermissive -std=c++11
        #CXXFLAGS += -Wall -O2 -fno-alias -fpermissive
        # CXXFLAGS += -Wall
        # Use this if you want to use ToR-ORd 2019 cell model.
        # Otherwise, comment it
        #CXXFLAGS += -DTOR-ORD_2019

        # LDFLAGS is used for linker (-g enables debug symbols)
        # LDFLAGS  += -g -L/usr/local/cuda/lib64
        LDFLAGS  += -g -L/usr/local/cuda/lib64 -arch=sm_86 -rdc=true

        # List the project' sources to compile or let the Makefile recognize
        # them for you using 'wildcard' function.
        #
        SOURCES  = $(wildcard *.cpp) $(wildcard **/*.cpp) $(wildcard *.c) $(wildcard **/*.c)
$(wildcard **/*.cu) $(wildcard *.cu)

        # List the project' headers or let the Makefile recognize
        # them for you using 'wildcard' function.
        #
        HEADERS  = $(wildcard *.hpp) $(wildcard **/*.hpp) $(wildcard *.h) $(wildcard **/*.h)
$(wildcard **/*.cuh) $(wildcard *.cuh)

        # Construct the list of object files based on source files using
        # simple extension substitution.
        OBJECTS := $(SOURCES:%.cpp=%.o)
        LIBS= -lopenblas -lpthread -lcudart -lcublas

        #
        # Now declare the dependencies rules and targets
        #
        # Starting with 'all' make it  becomes the default target when none
        # is specified on 'make' command line.
        all : $(PROGNAME)

        # Declare that the final program depends on all objects and the Makfile
        $(PROGNAME) : $(OBJECTS) Makefile
                $(CXX) -o bin/$@ $(OBJECTS) $(LDFLAGS)

        # Now the choice of using implicit rules or not (my choice)...
        #
        # Choice 1: use implicit rules and then we only need to add some dependencies
        #           to each object.
        #
        ## Tells make that each object file depends on all headers and this Makefile.
        #$(OBJECTS) : $(HEADERS) Makefile
        #
        # Choice 2: don't use implicit rules and specify our will
        %.o: %.cpp $(HEADERS) Makefile
```

```
	$(CXX) -x cu $(CXXFLAGS) $(CPPFLAGS) -dc -arch=sm_86 $(OUTPUT_OPTION) $<
# -dc -rdc=true

# Simple clean-up target
# notes:
# - the '@' before 'echo' informs make to hide command invocation.
# - the '-' before 'rm' command to informs make to ignore errors.
clean :
	@echo "Clean."
	rm -rf *.o bin/$(PROGNAME)
	rm -rf **/*.o
```

b. .gitignore

.gitignore file specifies files and folders to be ignored by git, the version control used in this research. The git will ignore simulation results, binary files, CUDA related libraries, log files, and a jupyter notebook used in plotting. Below is the list of files and folder I ignore in the research:

```
*.i
*.ii
*.gpu
*.ptx
*.cubin
*.fatbin
.DS_Store
bin/drug_sim
*.o
*.plt
*.out
bin/result/*/*.csv
bin/result/*/*/*.csv
bin/result/parse.ipynb
output.*
*.old
bin/result/*
```

c. main.cu

The main.cu file serves as the primary entry point for executing the drug simulation program. This file orchestrates the interaction between the core modules, manages the GPU-based computations, and handles input and output processes. Every CPU related orchestration for the simulation happen in main.cu. Its primary responsibilities are initialisation, loading input data, initialise GPU environment, core number calculation, executing simulation, output handling, memory clean-up, logging, and making sure all input and output are correct. I found that CUDA debugger does not really help the debugging process due to unique parallelisation in this research. Self-created debugging points were also introduced in the main.cu. The main.cu coded as below:

```
#include <cuda.h>
#include <cuda_runtime.h>

// #include "modules/drug_sim.hpp"
#include "modules/glob_funct.hpp"
#include "modules/glob_type.hpp"
#include "modules/gpu.cuh"
#include "modules/cipa_t.cuh"

#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <ctime>
#include <iostream>
#include <math.h>
#include <vector>
#include <sys/stat.h>

#define ENOUGH ((CHAR_BIT * sizeof(int) - 1) / 3 + 2)
char buffer[255];


// unsigned int datapoint_size = 7000;
const unsigned int sample_limit = 10000;
```

```
clock_t START_TIMER;

clock_t tic();
void toc(clock_t start = START_TIMER);

clock_t tic()
{
    return START_TIMER = clock();
}

void toc(clock_t start)
{
    std::cout
        << "Elapsed time: "
        << (clock() - start) / (double)CLOCKS_PER_SEC << "s"
        << std::endl;
}

int gpu_check(unsigned int datasize){
    int num_gpus;
    float percent;
    int id;
    size_t free, total;
    cudaGetDeviceCount( &num_gpus );
    for ( int gpu_id = 0; gpu_id < num_gpus; gpu_id++ ) {
        cudaSetDevice( gpu_id );
        cudaGetDevice( &id );
        cudaMemGetInfo( &free, &total );
        percent = (free/(float)total);
        printf("GPU No %d\nFree Memory: %ld, Total Memory: %ld (%f percent free)\n",
id,free,total,percent*100.0);
    }
    percent = 1.0-(datasize/(float)total);
    //// this code strangely gave out too small value, so i disable the safety switch for now

    // printf("The program uses GPU No %d and %f percent of its memory\n", id,percent*100.0);
    // printf("\n");
    // if (datasize<=free) {
    //   return 0;
    // }
    // else {
    //   return 1;
    // }


    return 0;

}
```

```
// get the IC50 data from file
drug_t get_IC50_data_from_file(const char* file_name);
// return error and message based on the IC50 data
int check_IC50_content(const drug_t* ic50, const param_t* p_param);


int get_IC50_data_from_file(const char* file_name, double *ic50)
{
    /*
    a host function to take all samples from the file, assuming each sample has 14 features.

    it takes the file name, and an ic50 (already declared in 1D, everything become 1D)
    as a note, the data will be stored in 1D array, means this functions applies flatten.

    it returns 'how many samples were detected?' in integer.
    */
  FILE *fp_drugs;
//   drug_t ic50;
  char *token;
  char buffer_ic50[255];
  unsigned int idx;

  if( (fp_drugs = fopen(file_name, "r")) == NULL){
    printf("Cannot open file %s\n",
      file_name);
    return 0;
  }
  idx = 0;
  int sample_size = 0;
  fgets(buffer_ic50, sizeof(buffer_ic50), fp_drugs); // skip header
  while( fgets(buffer_ic50, sizeof(buffer_ic50), fp_drugs) != NULL )
  { // begin line reading
    token = strtok( buffer_ic50, "," );
    while( token != NULL )
    { // begin data tokenizing
      ic50[idx++] = strtod(token, NULL);
      token = strtok(NULL, ",");
    } // end data tokenizing
    sample_size++;
  } // end line reading

  fclose(fp_drugs);
  return sample_size;
}

int get_cvar_data_from_file(const char* file_name, unsigned int limit, double *cvar)
{
  // buffer for writing in snprintf() function
```

```
  char buffer_cvar[255];
  FILE *fp_cvar;
  // cvar_t cvar;
  char *token;
  // std::array<double,18> temp_array;
  unsigned int idx;

  if( (fp_cvar = fopen(file_name, "r")) == NULL){
    printf("Cannot open file %s\n",
      file_name);
  }
  idx = 0;
  int sample_size = 0;
  fgets(buffer_cvar, sizeof(buffer_cvar), fp_cvar); // skip header
  while( (fgets(buffer_cvar, sizeof(buffer_cvar), fp_cvar) != NULL) && (sample_size<limit))
  { // begin line reading
    token = strtok( buffer_cvar, "," );
    while( token != NULL )
    { // begin data tokenizing
      cvar[idx++] = strtod(token, NULL);
      token = strtok(NULL, ",");
    } // end data tokenizing
    // printf("\n");
    sample_size++;
    // cvar.push_back(temp_array);
  } // end line reading

  fclose(fp_cvar);
  return sample_size;
}


int get_init_data_from_file(const char* file_name, double *init_states)
{
  // buffer for writing in snprintf() function
  char buffer_cache[1023];
  FILE *fp_cache;
  // cvar_t cvar;
  char *token;
  // std::array<double,18> temp_array;
  unsigned long idx;

  if( (fp_cache = fopen(file_name, "r")) == NULL){
    printf("Cannot open file %s\n",
      file_name);
  }
  idx = 0;
  unsigned int sample_size = 0;
  // fgets(buffer_cvar, sizeof(buffer_cvar), fp_cvar); // skip header
```

```
  while( (fgets(buffer_cache, sizeof(buffer_cache), fp_cache) != NULL) )
  { // begin line reading
    token = strtok( buffer_cache, "," );
    while( token != NULL )
    { // begin data tokenizing
      init_states[idx++] = strtod(token, NULL);
      // if(idx < 82){
      //     printf("%d: %lf\n",idx-1,init_states[idx-1]);
      // }
      token = strtok(NULL, ",");
    } // end data tokenizing
    // printf("\n");
    sample_size++;
    // cvar.push_back(temp_array);
  } // end line reading

  fclose(fp_cache);
  return sample_size;
}
int exists(const char *fname)
{
    FILE *file;
    if ((file = fopen(fname, "r")))
    {
        fclose(file);
        return 1;
    }
    // fclose(file);
    return 0;
}

int check_IC50_content(const drug_t* ic50, const param_t* p_param)
{
        if(ic50->size() == 0){
                printf("Something problem with the IC50 file!\n");
                return 1;
        }
        else if(ic50->size() > 2000){
                printf( "Too much input! Maximum sample data is 2000!\n");
                return 2;
        }
        else if(p_param->pace_max < 750 && p_param->pace_max > 1000){
                printf("Make sure the maximum pace is around 750 to 1000!\n");
                return 3;
        }
        // else if(mympi::size > ic50->size()){
        //       printf("%s\n%s\n",
  //              "Overflow of MPI Process!",
  //              "Make sure MPI Size is less than or equal the number of sample");
```

```
        //      return 4;
        // }
        else{
                return 0;
        }
}

int main(int argc, char **argv)
{
        // enable real-time output in stdout
        //setvbuf( stdout, NULL, _IONBF, 0 );

// NEW CODE STARTS HERE //
    // mycuda *thread_id;
    // cudaMalloc(&thread_id, sizeof(mycuda));
```

This is how the main.cu manages multi-concentration in the simulation. Since ‘kernel_DoDrugSim’ uses less memory, I used them to put more samples, then I can put more concentration in the modified IC50 file.

```
  // TODO: Automation 3. check file inside folder
    for (const auto &entry : fs::directory_iterator(drug_dir)) {
        param_t *p_param, *d_p_param;
        p_param = new param_t();
        p_param->init();
        edison_assign_params(argc, argv, p_param);

        std::filesystem::directory_entry dir_entry = entry;
        std::string entry_str = dir_entry.path().string();
        std::cout << entry_str << std::endl;
        std::regex pattern("/([a-zA-Z0-9_\.]+)\.csv");
        std::smatch match;
        std::regex_search(entry_str, match, pattern);

        // TODO: Automation 2. create drug_name and conc

        // TODO: NewFile 2. disable drug name for now since the file name is inside it
        // strcpy(p_param->drug_name, match[1].str().c_str());
        strcpy(p_param->hill_file, entry_str.c_str());
        // strcat(p_param->hill_file, ".csv");
        // strcat(p_param->hill_file, "/IC50_samples.csv");

        // TODO: NewFile 3. getvalue from source is unnecessary
        // p_param->conc = getValue(drugConcentration, match[1].str()) * cmax;
```

```
        // p_param->show_val();

    // for qinwards calculation
    double inal_auc_control = -90.547322;    // AUC of INaL under control model
    double ical_auc_control = -105.935067;   // AUC of ICaL under control model

    // input variables for cell simulation
    param_t *p_param = new param_t();  // input data for CPU
    param_t *d_p_param;  // input data for GPU parsing

         p_param->init();
    edison_assign_params(argc,argv,p_param);
    p_param->show_val();

    double* ic50 = (double *)malloc(14 * sample_limit * sizeof(double));
    // if (p_param->is_cvar == true) cvar = (double *)malloc(18 * sample_limit *
sizeof(double));
    double* cvar = (double *)malloc(18 * sample_limit * sizeof(double));  // conductance
variability

    const int num_of_constants = 146;
    const int num_of_states = 41;
    const int num_of_algebraic = 199;
    const int num_of_rates = 41;
    const double CONC = p_param->conc;
```

Below is how the main.cu manages memory and data output for ‘kernel_DoDrugSim_single’. This kernel function requires more memory due to its more detailed output, hence requires a slightly adapted way to ensure all temporary memory wiped after simulation. In practice, this function also takes up to 60% more GPU memory compared to ‘kernel_DoDrugSim’.

```
    ////////// if we are in write time series mode (post processing) //////////
    if(p_param->is_time_series == 1 /*&& exists(p_param->cache_file) == 1 <- still
unstable*/){

    printf("Using cached initial state from previous result!!!! \n\n");

    const unsigned int datapoint_size = p_param->sampling_limit; // sampling_limit: limit of
num of data points in one sample
```

```
    double* cache = (double *)malloc((num_of_states+2) * sample_limit * sizeof(double)); //
array for in silico results


    static const int CALCIUM_SCALING = 1000000;
    static const int CURRENT_SCALING = 1000;

    // snprintf(buffer, sizeof(buffer),
    //   "./drugs/bepridil/IC50_samples.csv"
    //   // "./drugs/bepridil/IC50_optimal.csv"
    //   // "./IC50_samples.csv"
    //   );

    int sample_size = get_IC50_data_from_file(p_param->hill_file, ic50);
    if(sample_size == 0)
        printf("Something problem with the IC50 file!\n");
    // else if(sample_size > 2000)
    //     printf("Too much input! Maximum sample data is 2000!\n");
    printf("Sample size: %d\n",sample_size);
    printf("Set GPU Number: %d\n",p_param->gpu_index);

    cudaSetDevice(p_param->gpu_index);  // select a specific GPU

    if(p_param->is_cvar == true){
      int cvar_sample = get_cvar_data_from_file(p_param->cvar_file,sample_size,cvar);
      printf("Reading: %d Conductance Variability samples\n",cvar_sample);
    }

    printf("preparing GPU memory space \n");

      // char buffer_cvar[255];
      // snprintf(buffer_cvar, sizeof(buffer_cvar),
      // "./result/66_00.csv"
      // // "./drugs/optimized_pop_10k.csv"
      // );
      int cache_num = get_init_data_from_file(p_param->cache_file,cache);  //

      printf("Found cache for %d samples\n",cache_num);
      // note to self:
      // num of states+2 gave you at the very end of the file (pace number)
      // the very beginning -> the core number
    //   for (int z = 0; z <  num_of_states; z++) {printf("%lf\n", cache[z+1]);}
    //   printf("\n");
    //   for (int z = 0; z <  num_of_states; z++) {printf("%lf\n", cache[ 1*(num_of_states+2)
+ (z+2)]);}
    //   printf("\n");
    //   for (int z = 0; z <  num_of_states; z++) {printf("%lf\n", cache[ 2*(num_of_states+2)
+ (z+3)]);}
    // return 0 ;
```

```
    double *d_ic50;
    double *d_cvar;
    double *d_ALGEBRAIC;
    double *d_CONSTANTS;
    double *d_RATES;
    double *d_STATES;
    double *d_STATES_cache;
    // actually not used but for now, this is only for satisfiying the GPU regulator
parameters
    double *d_STATES_RESULT;
    double *d_all_states;
    cudaMalloc(&d_ALGEBRAIC, num_of_algebraic * sample_size * sizeof(double));
    cudaMalloc(&d_CONSTANTS, num_of_constants * sample_size * sizeof(double));
    cudaMalloc(&d_RATES, num_of_rates * sample_size * sizeof(double));
    cudaMalloc(&d_STATES, num_of_states * sample_size * sizeof(double));
    cudaMalloc(&d_STATES_cache, (num_of_states+2) * sample_size * sizeof(double));
    cudaMalloc(&d_p_param,  sizeof(param_t));

    double *time;
    double *dt;
    double *states;
    double *ical;
    double *inal;
    double *cai_result;
    double *ina;
    double *ito;
    double *ikr;
    double *iks;
    double *ik1;
    cipa_t *temp_result, *cipa_result;
    // prep for 1 cycle plus a bit (7000 * sample_size)
    cudaMalloc(&temp_result, sample_size * sizeof(cipa_t));  // for temporal ??
    cudaMalloc(&cipa_result, sample_size * sizeof(cipa_t));  // output of postprocessing

    cudaMalloc(&time, sample_size * datapoint_size * sizeof(double));
    cudaMalloc(&dt, sample_size * datapoint_size * sizeof(double));

    cudaMalloc(&states, sample_size * datapoint_size * sizeof(double));
    cudaMalloc(&ical, sample_size * datapoint_size * sizeof(double));
    cudaMalloc(&inal, sample_size * datapoint_size * sizeof(double));
    cudaMalloc(&cai_result, sample_size * datapoint_size * sizeof(double));
    cudaMalloc(&ina, sample_size * datapoint_size * sizeof(double));
    cudaMalloc(&ito, sample_size * datapoint_size * sizeof(double));
    cudaMalloc(&ikr, sample_size * datapoint_size * sizeof(double));
    cudaMalloc(&iks, sample_size * datapoint_size * sizeof(double));
    cudaMalloc(&ik1, sample_size * datapoint_size * sizeof(double));
    // cudaMalloc(&d_STATES_RESULT, (num_of_states+1) * sample_size * sizeof(double));
    // cudaMalloc(&d_all_states, num_of_states * sample_size * p_param->find_steepest_start *
sizeof(double));
```

```
    cudaMalloc(&d_ic50, sample_size * 14 * sizeof(double));  // ic50s of 7 channels
    cudaMalloc(&d_cvar, sample_size * 18 * sizeof(double));  // conductances of 18

    printf("Copying sample files to GPU memory space \n");
    cudaMemcpy(d_STATES_cache, cache, (num_of_states+2) * sample_size * sizeof(double),
cudaMemcpyHostToDevice);
    cudaMemcpy(d_ic50, ic50, sample_size * 14 * sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(d_cvar, cvar, sample_size * 18 * sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(d_p_param, p_param, sizeof(param_t), cudaMemcpyHostToDevice);

    // // Get the maximum number of active blocks per multiprocessor
    // cudaOccupancyMaxActiveBlocksPerMultiprocessor(&numBlocks, do_drug_sim_analytical,
threadsPerBlock);

    // // Calculate the total number of blocks
    // int numTotalBlocks = numBlocks * cudaDeviceGetMultiprocessorCount();
    tic();
    printf("Timer started, doing simulation.... \n\n\nGPU Usage at this moment: \n");
    int thread;
    if (sample_size>=16) thread = 16; // change this according to hardware
// optimal number of thread by experience -> might be different for each GPU, can be 16, can
be 32
    else thread = sample_size;
    // int block = int(ceil(sample_size*1.0/thread)+1);
    int block = (sample_size + thread - 1) / thread;
    // int block = (sample_size + thread - 1) / thread;
    if(gpu_check(15 * sample_size * sizeof(double) + sizeof(param_t)) == 1){
      printf("GPU memory insufficient!\n");
      return 0;
    }
    printf("Sample size: %d\n",sample_size);
    cudaSetDevice(p_param->gpu_index);
    printf("\n   Configuration: \n\n\tblock\t||\tthread\n-----------------------------------
--\n  \t%d\t||\t%d\n\n\n", block,thread);
    // initscr();
    //
printf("[_____________________________________________________________________________________
_____________]  0.00 %% \n");


    kernel_DrugSimulation<<<block,thread>>>(d_ic50, d_cvar, d_CONSTANTS, d_STATES,
d_STATES_cache, d_RATES, d_ALGEBRAIC,
                                            d_STATES_RESULT, d_all_states,
                                            time, states, dt, cai_result,
                                            ina, inal,
                                            ical, ito,
                                            ikr, iks,
                                            ik1,
                                            sample_size,
```

```
                                        temp_result, cipa_result,
                                        d_p_param
                                        );
                                    //block per grid, threads per block
    // endwin();

    cudaDeviceSynchronize();


    printf("allocating memory for computation result in the CPU, malloc style \n");
    double
*h_states,*h_time,*h_dt,*h_ical,*h_inal,*h_cai_result,*h_ina,*h_ito,*h_ikr,*h_iks,*h_ik1;
    cipa_t *h_cipa_result;

    h_states = (double *)malloc(datapoint_size * sample_size * sizeof(double));
    printf("...allocated for STATES, \n");
    h_time = (double *)malloc(datapoint_size * sample_size * sizeof(double));
    printf("...allocated for time, \n");
    h_dt = (double *)malloc(datapoint_size * sample_size * sizeof(double));
    printf("...allocated for dt, \n");
    h_cai_result= (double *)malloc(datapoint_size * sample_size * sizeof(double));
    printf("...allocated for Cai, \n");
     h_ina= (double *)malloc(datapoint_size * sample_size * sizeof(double));
    printf("...allocated for iNa, \n");
     h_ito= (double *)malloc(datapoint_size * sample_size * sizeof(double));
    printf("...allocated for ito, \n");
     h_ikr= (double *)malloc(datapoint_size * sample_size * sizeof(double));
    printf("...allocated for ikr, \n");
     h_iks= (double *)malloc(datapoint_size * sample_size * sizeof(double));
    printf("...allocated for iks, \n");
     h_ik1= (double *)malloc(datapoint_size * sample_size * sizeof(double));
    printf("...allocated for ik1, \n");
     h_ical= (double *)malloc(datapoint_size * sample_size * sizeof(double));
    printf("...allocated for ICaL, \n");
    h_inal = (double *)malloc(datapoint_size * sample_size * sizeof(double));

    h_cipa_result = (cipa_t *)malloc( sample_size * sizeof(cipa_t));
    printf("...allocating for INaL and postprocessing, all set!\n");

    ////// copy the data back to CPU, and write them into file ////////
    printf("copying the data back to the CPU \n");

    cudaMemcpy(h_states, states, sample_size * datapoint_size * sizeof(double),
cudaMemcpyDeviceToHost);
    cudaMemcpy(h_time, time, sample_size * datapoint_size * sizeof(double),
cudaMemcpyDeviceToHost);
    cudaMemcpy(h_dt, dt, sample_size * datapoint_size * sizeof(double),
cudaMemcpyDeviceToHost);
```

```
    cudaMemcpy(h_ical, ical, sample_size * datapoint_size * sizeof(double),
cudaMemcpyDeviceToHost);
    cudaMemcpy(h_inal, inal, sample_size * datapoint_size * sizeof(double),
cudaMemcpyDeviceToHost);
    cudaMemcpy(h_cai_result, cai_result, sample_size * datapoint_size * sizeof(double),
cudaMemcpyDeviceToHost);
    cudaMemcpy(h_ina, ina, sample_size * datapoint_size * sizeof(double),
cudaMemcpyDeviceToHost);
    cudaMemcpy(h_ito, ito, sample_size * datapoint_size * sizeof(double),
cudaMemcpyDeviceToHost);
    cudaMemcpy(h_ikr, ikr, sample_size * datapoint_size * sizeof(double),
cudaMemcpyDeviceToHost);
    cudaMemcpy(h_iks, iks, sample_size * datapoint_size * sizeof(double),
cudaMemcpyDeviceToHost);
    cudaMemcpy(h_ik1, ik1, sample_size * datapoint_size * sizeof(double),
cudaMemcpyDeviceToHost);

    cudaMemcpy(h_cipa_result, cipa_result, sample_size * sizeof(cipa_t),
cudaMemcpyDeviceToHost);

    FILE *writer;
    int check;
    bool folder_created = false;

    printf("writing to file... \n");
    // sample loop
    for (int sample_id = 0; sample_id<sample_size; sample_id++){
      // printf("writing sample %d... \n",sample_id);
      char sample_str[ENOUGH];
      char conc_str[ENOUGH];
      char filename[500] = "./result/";
      sprintf(sample_str, "%d", sample_id);
      sprintf(conc_str, "%.2f", CONC);
      strcat(filename,conc_str);
      strcat(filename,"/");
      if (folder_created == false){
        check = mkdir(filename,0777);
        // check if directory is created or not
        if (!check){
          printf("Directory created\n");
          }
        else {
          printf("Unable to create directory, or the folder is already created, relax
mate...\n");
      }
      folder_created = true;
      }

      strcat(filename,sample_str);
```

```
      strcat(filename,"_timeseries.csv");

      writer = fopen(filename,"w");
      fprintf(writer, "Time,Vm,dVm/dt,Cai,INa,INaL,ICaL,IKs,IKr,IK1,Ito\n");
      for (int datapoint = 1; datapoint<datapoint_size; datapoint++){
       if (h_time[ sample_id + (datapoint * sample_size)] == 0.0) {break;}
        fprintf(writer,"%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf\n", // change this into
string, or limit the decimal accuracy, so we can decrease filesize
        h_time[ sample_id + (datapoint * sample_size)],
        h_states[ sample_id + (datapoint * sample_size)],
        h_dt[ sample_id + (datapoint * sample_size)],
        h_cai_result[ sample_id + (datapoint * sample_size)],

        h_ina[ sample_id + (datapoint * sample_size)],
        h_inal[ sample_id + (datapoint * sample_size)],

        h_ical[ sample_id + (datapoint * sample_size)],
        h_iks[ sample_id + (datapoint * sample_size)],

        h_ikr[ sample_id + (datapoint * sample_size)],
        h_ik1[ sample_id + (datapoint * sample_size)],

        h_ito[ sample_id + (datapoint * sample_size)]
        );
      }
      fclose(writer);
    }

    printf("writing each biomarkers value... \n");
    // sample loop
      char conc_str[ENOUGH];
      char filename[500] = "./result/";
      // sprintf(sample_str, "%d", sample_id);
      sprintf(conc_str, "%.2f", CONC);
      strcat(filename,conc_str);
      strcat(filename,"/");
      // printf("creating %s... \n", filename);
      if (folder_created == false){
        check = mkdir(filename,0777);
        // check if directory is created or not
        if (!check){
          printf("Directory created\n");
          }
        else {
          printf("Unable to create directory, or the folder is already created, relax
mate...\n");
      }
      folder_created = true;
      }
```

```
      // strcat(filename,sample_str);
    strcat(filename,"_biomarkers.csv");

    writer = fopen(filename,"a");

    fprintf(writer,
"sample,qnet,qInward,inal_auc,ical_auc,apd90,apd50,apd_tri,cad90,cad50,cad_tri,dvmdt_repol,vm_
peak,vm_valley,vm_dia,ca_peak,ca_valley,ca_dia\n");
    for (int sample_id = 0; sample_id<sample_size; sample_id++){
      // printf("writing sample %d... \n",sample_id);


fprintf(writer,"%d,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf\n", //
change this into string, or limit the decimal accuracy, so we can decrease filesize
        sample_id,
        h_cipa_result[sample_id].qnet,
        0.5*((h_cipa_result[sample_id].ical_auc /
ical_auc_control)+(h_cipa_result[sample_id].inal_auc / inal_auc_control)),
        h_cipa_result[sample_id].inal_auc,
        h_cipa_result[sample_id].ical_auc,

        h_cipa_result[sample_id].apd90,
        h_cipa_result[sample_id].apd50,
        h_cipa_result[sample_id].apd90 - h_cipa_result[sample_id].apd50,

        h_cipa_result[sample_id].cad90,
        h_cipa_result[sample_id].cad50,
        h_cipa_result[sample_id].cad90 - h_cipa_result[sample_id].cad50,

        h_cipa_result[sample_id].dvmdt_repol,
        h_cipa_result[sample_id].vm_peak,
        h_cipa_result[sample_id].vm_valley,
        h_cipa_result[sample_id].vm_dia,

        h_cipa_result[sample_id].ca_peak,
        h_cipa_result[sample_id].ca_valley,
        h_cipa_result[sample_id].ca_dia

    //      temp_result[sample_id].qnet = 0.;
    // temp_result[sample_id].inal_auc = 0.;
    // temp_result[sample_id].ical_auc = 0.;

    // temp_result[sample_id].dvmdt_repol = -999;
    // temp_result[sample_id].dvmdt_max = -999;
    // temp_result[sample_id].vm_peak = -999;
    // temp_result[sample_id].vm_valley = d_STATES[(sample_id * num_of_states) +V];
    // temp_result[sample_id].vm_dia = -999;
```

```
    // temp_result[sample_id].apd90 = 0.;
    // temp_result[sample_id].apd50 = 0.;
    // temp_result[sample_id].ca_peak = -999;
    // temp_result[sample_id].ca_valley = d_STATES[(sample_id * num_of_states) +cai];
    // temp_result[sample_id].ca_dia = -999;
    // temp_result[sample_id].cad90 = 0.;
    // temp_result[sample_id].cad50 = 0.;
        );

    }
     fclose(writer);

    toc();

    return 0;
  }
```

The following is the *in silico* mode. This part of the main.cu explains how differ the handling for 'kernel_DoDrugSim' compared to 'kernel_DoDrugSim_single'. 'kernel_DoDrugSim' requires less memory, so this following section of the code removes some memory assignments compared to 'kernel_DoDrugSim_single:

```
    ////////// find cache mode (in silico code) //////////
    else{
    printf("in silico mode, creating cache file because we don't have that yet, or
is_time_series is intentionally false \n\n");
    double *d_ic50;
    double *d_cvar;
    double *d_ALGEBRAIC;
    double *d_CONSTANTS;
    double *d_RATES;
    double *d_STATES;

    // not used, only to satisfy the parameters of the GPU regulator's function
    double *d_STATES_cache;
    double *time;
    double *dt;
    double *states;
    double *cai_result;
    double *ical;
    double *inal;
    double *ina;
    double *ito;
```

```
    double *ikr;
    double *iks;
    double *ik1;

    double *d_STATES_RESULT;
    double *d_all_states;

    cipa_t *temp_result, *cipa_result;

    int sample_size = get_IC50_data_from_file(p_param->hill_file, ic50);
    if(sample_size == 0)
        printf("Something problem with the IC50 file!\n");
    // else if(sample_size > 2000)
    //     printf("Too much input! Maximum sample data is 2000!\n");
    printf("Sample size: %d\n",sample_size);
    cudaSetDevice(p_param->gpu_index);
    printf("preparing GPU memory space \n");

    if(p_param->is_cvar == true){
      int cvar_sample = get_cvar_data_from_file(p_param->cvar_file,sample_size,cvar);
      printf("Reading: %d Conductance Variability samples\n",cvar_sample);
    }

    cudaMalloc(&d_ALGEBRAIC, num_of_algebraic * sample_size * sizeof(double));
    cudaMalloc(&d_CONSTANTS, num_of_constants * sample_size * sizeof(double));
    cudaMalloc(&d_RATES, num_of_rates * sample_size * sizeof(double));
    cudaMalloc(&d_STATES, num_of_states * sample_size * sizeof(double));

    cudaMalloc(&d_p_param,  sizeof(param_t));

    // prep for 1 cycle plus a bit (7000 * sample_size)
    cudaMalloc(&temp_result, sample_size * sizeof(cipa_t));
    cudaMalloc(&cipa_result, sample_size * sizeof(cipa_t));

    cudaMalloc(&d_STATES_RESULT, (num_of_states+1) * sample_size * sizeof(double)); // for
cache file
    cudaMalloc(&d_all_states, num_of_states * sample_size * p_param->find_steepest_start *
sizeof(double)); // for each sample

    printf("Copying sample files to GPU memory space \n");
    cudaMalloc(&d_ic50, sample_size * 14 * sizeof(double));
    // if(p_param->is_cvar == true) cudaMalloc(&d_cvar, sample_size * 18 * sizeof(double));
    cudaMalloc(&d_cvar, sample_size * 18 * sizeof(double));

    cudaMemcpy(d_ic50, ic50, sample_size * 14 * sizeof(double), cudaMemcpyHostToDevice);
    // if(p_param->is_cvar == true) cudaMemcpy(d_cvar, cvar, sample_size * 18 *
sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(d_cvar, cvar, sample_size * 18 * sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(d_p_param, p_param, sizeof(param_t), cudaMemcpyHostToDevice);
```

```
    // // Get the maximum number of active blocks per multiprocessor
    // cudaOccupancyMaxActiveBlocksPerMultiprocessor(&numBlocks, do_drug_sim_analytical,
threadsPerBlock);

    // // Calculate the total number of blocks
    // int numTotalBlocks = numBlocks * cudaDeviceGetMultiprocessorCount();

    tic();
    printf("Timer started, doing simulation.... \n GPU Usage at this moment: \n");
    int thread;
    if (sample_size>=16){
      thread = 16; // change this according to hardware
    }
    else thread = sample_size;
    // int block = int(ceil(sample_size*1.0/thread)+1);
    int block = (sample_size + thread - 1) / thread;
    // int block = (sample_size + thread - 1) / thread;
    if(gpu_check(15 * sample_size * sizeof(double) + sizeof(param_t)) == 1){
      printf("GPU memory insufficient!\n");
      return 0;
    }
    printf("Sample size: %d\n",sample_size);
    cudaSetDevice(p_param->gpu_index);
    printf("\n   Configuration: \n\n\tblock\t||\tthread\n-----------------------------------
--\n  \t%d\t||\t%d\n\n\n", block,thread);
    // initscr();
    //
printf("[________________________________________________________________________________________
_____________]  0.00 %% \n");

    kernel_DrugSimulation<<<block,thread>>>(d_ic50, d_cvar, d_CONSTANTS, d_STATES,
d_STATES_cache, d_RATES, d_ALGEBRAIC,
                                            d_STATES_RESULT, d_all_states,
                                            time, states, dt, cai_result,
                                            ina, inal,
                                            ical, ito,
                                            ikr, iks,
                                            ik1,
                                            sample_size,
                                            temp_result, cipa_result,
                                            d_p_param
                                            );
                                      //block per grid, threads per block
    // endwin();

    cudaDeviceSynchronize();
```

```
    printf("allocating memory for computation result in the CPU, malloc style \n");
    double *h_states, *h_all_states;
    cipa_t *h_cipa_result;

    h_states = (double *)malloc((num_of_states+1) * sample_size * sizeof(double)); //cache
file
    h_all_states = (double *)malloc( (num_of_states) * sample_size * p_param-
>find_steepest_start * sizeof(double)); //all core
    h_cipa_result = (cipa_t *)malloc(sample_size * sizeof(cipa_t));
    printf("...allocating for all states, all set!\n");

    ////// copy the data back to CPU, and write them into file ////////
    printf("copying the data back to the CPU \n");

    cudaMemcpy(h_cipa_result, cipa_result, sample_size * sizeof(cipa_t),
cudaMemcpyDeviceToHost);
    cudaMemcpy(h_states, d_STATES_RESULT, sample_size * (num_of_states+1) *  sizeof(double),
cudaMemcpyDeviceToHost);
    cudaMemcpy(h_all_states, d_all_states, (num_of_states) * sample_size  * p_param-
>find_steepest_start  *  sizeof(double), cudaMemcpyDeviceToHost);

    FILE *writer;
    int check;
    bool folder_created = false;


    // char sample_str[ENOUGH];
    char conc_str[ENOUGH];
    char filename[500] = "./result/";
    sprintf(conc_str, "%.2f", CONC);
    strcat(filename,conc_str);
    // strcat(filename,"_steepest");
      if (folder_created == false){
        check = mkdir(filename,0777);
        // check if directory is created or not
        if (!check){
          printf("Directory created\n");
          }
        else {
          printf("Unable to create directory, or the folder is already created, relax
mate...\n");
      }
      folder_created = true;
      }

    // strcat(filename,sample_str);
    strcat(filename,".csv");
    printf("writing to %s ... \n", filename);
    writer = fopen(filename,"w");
```

```
    // sample loop
    for (int sample_id = 0; sample_id<sample_size; sample_id++){
      // writer = fopen(filename,"a"); // because we have multiple fwrites
      fprintf(writer,"%d,",sample_id); // write core number at the front
      for (int datapoint = 0; datapoint<num_of_states; datapoint++){
       // if (h_time[ sample_id + (datapoint * sample_size)] == 0.0) {continue;}
        fprintf(writer,"%.5f,", // change this into string, or limit the decimal accuracy, so
we can decrease filesize
        h_states[(sample_id * (num_of_states+1)) + datapoint]
        );
      }
        // fprintf(writer,"%lf,%lf\n", // write last data
        // h_states[(sample_id * num_of_states+1) + num_of_states],
        // h_states[(sample_id * num_of_states+1) + num_of_states+1]
        // );
        fprintf(writer,"%.5f\n", h_states[(sample_id * (num_of_states+1))+num_of_states] );
        // fprintf(writer, "\n");

      // fclose(writer);
    }
     fclose(writer);

    // // FILE *writer;
    // // int check;
    // // bool folder_created = false;

    // printf("writing each core value... \n");
    // // sample loop
    // for (int sample_id = 0; sample_id<sample_size; sample_id++){
    //   // printf("writing sample %d... \n",sample_id);
    //   char sample_str[ENOUGH];
    //   char conc_str[ENOUGH];
    //   char filename[500] = "./result/";
    //   sprintf(sample_str, "%d", sample_id);
    //   sprintf(conc_str, "%.2f", CONC);
    //   strcat(filename,conc_str);
    //   strcat(filename,"/");
    //   // printf("creating %s... \n", filename);
    //   if (folder_created == false){
    //     check = mkdir(filename,0777);
    //     // check if directory is created or not
    //     if (!check){
    //       printf("Directory created\n");
    //       }
    //     else {
    //       printf("Unable to create directory, or the folder is already created, relax
mate...\n");
    //   }
    //   folder_created = true;
```

```
    //   }

    //   strcat(filename,sample_str);
    //   strcat(filename,".csv");

    //   writer = fopen(filename,"w");
    //   for (int pacing = 0; pacing < p_param->find_steepest_start; pacing++){ //pace loop
    //    // if (h_time[ sample_id + (datapoint * sample_size)] == 0.0) {continue;}
    //     for(int datapoint = 0; datapoint < num_of_states; datapoint++){ // each data loop
    //     fprintf(writer,"%lf,",h_all_states[((sample_id * num_of_states)) + ((sample_size) *
pacing) + datapoint]);
    //     // fprintf(writer,"%lf,",h_all_states[((sample_id * num_of_states))+ datapoint]);
    //     }
    //     // fprintf(writer,"%d",p_param->find_steepest_start + pacing);
    //     fprintf(writer,"%d\n",pacing + (p_param->pace_max - p_param-
>find_steepest_start)+1 );

    //   }
    //   fclose(writer);
    // }
    toc();
    return 0;

    }

        }
```

d. test_compile.bat

This file is a result from previous iteration. There was a moment that I wanted to develop the simulator for Windows Operating System outside of Linux. It is finally decided this repository will be containerised using Docker instead of separately develop Windows version of this simulator. This script will more likely to be deleted in the next update.

C. ‘bin’ Folder

The bin folder serves as the central directory for storing various input files, intermediate data, and simulation outputs. This folder is organised into subdirectories that categorise data for ease of access and maintainability.

a. CVAR

This folder will be used to store inter-individual conductance variability file for future development of this research.

b. Control

The control subfolder contains files that are fundamental to running simulations under control conditions (without drug effects). It typically includes:

- IC50_samples.csv: This file provides a baseline reference for the IC50 values of various ionic currents, which are used for comparison in simulations involving drug-induced conditions. IC50 file formatted as:

```
drug_name,conc,ICaL_IC50,ICaL_h,IK1_IC50,IK1_h,IKs_IC50,IKs_h,INa_IC50,INa_h,IN
aL_IC50,INaL_h,Ito_IC50,Ito_h,hERG_IC50,hERG_h
bepridil,0,2704,0.6954,NA,NA,NA,NA,2371,1.984,1947,1.473,NA,NA,139.1,3.199
bepridil,0,2818,0.6409,NA,NA,NA,NA,2734,1.225,1802,1.212,NA,NA,181.4,2.77
bepridil,0,3939,0.718,NA,NA,NA,NA,3064,1.108,1921,1.421,NA,NA,194.8,0.8339
```

- init_state.csv: This file contains the initial states of all variables in the cell model, such as membrane voltage and ion concentrations. These states serve as the starting point for the simulations, ensuring consistent and reproducible results.

By isolating control data in its dedicated subfolder, the simulation framework ensures clarity when comparing control and drug-altered conditions.

c. drug

The drug subfolder stores input files related to simulations involving specific drugs. These files typically include the IC50 values and Hill coefficients for the drugs under study, which define their effects on various

ionic currents. For example, IC50_terfenadine.csv and IC50_verapamil.csv: These files describe the pharmacological properties of terfenadine and verapamil, respectively.

d. result

The result subfolder is used to store simulation outputs, ensuring that data generated during the computational runs is systematically archived. Simulation output classified as two types, the init file, and post-processing folder. Init file is the output from the first phase of the simulation. It contains the simulation's state when gradient of action potential is at its steepest. Named _state_only.csv in a folder named from the IC50 file, this initial state usually shaped like:

0,-89.14848,0.01218,0.00007,12.14017,12.14051,144.11277,144.11273,1.56242,1.55949,0.00008,0.00074,0.83611,0.83590,0.68309,0.83534,0.00015,0.53126,0.28205,0.00093,0.99964,0.56097,0.00047,0.99964,0.61777,0.00000,1.00000,0.92678,1.00000,0.99980,0.99996,1.00000,1.00000,0.00051,0.00087,0.00070,0.00083,0.99790,0.00002,0.00060,0.27721,0.00017,0.00000,0.00000,999.00000

1,-89.14444,0.01211,0.00007,12.12804,12.12838,144.10139,144.10135,1.55818,1.55521,0.00008,0.00074,0.83604,0.83582,0.68296,0.83526,0.00015,0.53107,0.28182,0.00093,0.99963,0.56062,0.00047,0.99963,0.61738,0.00000,1.00000,0.92684,1.00000,0.99980,0.99996,1.00000,1.00000,0.00051,0.00086,0.00070,0.00083,0.99789,0.00002,0.00060,0.27735,0.00017,0.00000,0.00000,999.00000

With the first column as sample ID number, and the last column acts as number of pace recorded. This file will become the input of the second phase. Post-processing folder contains a biomarker file and time-series data per sample. The biomarker file will be used for next researches that requires data analysis. In the research, I validate the result, and manually analyse action potential and ion channels with the time-series files.

This subfolder's structure allows for easy retrieval of outputs for validation and analysis, ensuring that results from different runs are preserved without overwriting.

D. 'cellmodels' Folder

This folder contains mathematical models of the cell. The folder is created to make cell models easier to change and modify. I put the converted C file in here, with an enum file as an addition.

a. Ohara_Rudy_2011.hpp

This file declares all the function in ORd 2011 cell model. This header is required because the main parallelisation will be handled by gpu.cu, hence requiring an object-oriented programming approach. Notice that every function here is a kernel function, since each cell model will be simulated in parallel using each computing thread of the GPU. Below is the format of header file I implement, and this should be changed with any function changes in Ohara_Rudy_2011.cpp file:

```
#ifndef OHARA_RUDY_2011_HPP
#define OHARA_RUDY_2011_HPP

#include "enums/enum_Ohara_Rudy_2011.hpp"
#include <cuda_runtime.h>

        // void initConsts();
        // void initConsts(double type);
        // void initConsts(bool is_dutta);
        __device__ void initConsts(double *CONSTANTS, double *STATES, double type,
double conc, double *ic50, double *cvar, bool is_dutta, bool is_cvar,  int offset);
        __device__ void computeRates(double TIME, double* CONSTANTS, double* RATES,
double* STATES, double* ALGEBRAIC, int offset);
        __device__ void solveAnalytical(double *CONSTANTS, double *STATES, double
*ALGEBRAIC, double *RATES, double dt, int offset);
```

```
                __device__ void solveEuler( double *STATES, double *RATES, double dt, int
offset);
                __device__   double   set_time_step(double   TIME,double   time_point,double
max_time_step,
          double* CONSTANTS,
          double* RATES,
          double* STATES,
          double* ALGEBRAIC,
          int offset);
            __device__ void applyDrugEffect(double *CONSTANTS, double conc, double *ic50, double
epsilon, int offset);
                __device__ void ___applyDutta(double *CONSTANTS, int offset);
                __device__ void ___applyCvar(double *CONSTANTS, double *cvar, int offset);
                // void ___initConsts(double *CONSTANTS, double *STATES, double type, int
offset);

        #endif
```

b. Ohara_Rudy_2011.cpp

This file contains the mathematical cell model and its solver. I added the solver and drug effect simulation function in this script. The rest of the script follows the conversion from CellML. Below is how I implement the modification and the whole script:

```
        /*
           There are a total of 198 entries in the algebraic variable array.
           There are a total of 41 entries in each of the rate and state variable arrays.
           There are a total of 139+2 entries in the constant variable array.
         */

        #include "Ohara_Rudy_2011.hpp"
        #include <cmath>
        #include <cstdlib>
        #include <cstdio>
        #include "../modules/glob_funct.hpp"
        #include <cuda_runtime.h>
        #include <cuda.h>

        __device__ void ___initConsts(double *CONSTANTS, double *STATES, double type, int offset)
        {

        int num_of_constants = 145;
        int num_of_states = 41;
        // printf("%d\n", offset);
        CONSTANTS[(offset * num_of_constants) + celltype] = type;
```

```
CONSTANTS[(offset * num_of_constants) + nao] = 140;
CONSTANTS[(offset * num_of_constants) + cao] = 1.8;
CONSTANTS[(offset * num_of_constants) + ko] = 5.4;
CONSTANTS[(offset * num_of_constants) + R] = 8314;
CONSTANTS[(offset * num_of_constants) + T] = 310;
CONSTANTS[(offset * num_of_constants) + F] = 96485;
CONSTANTS[(offset * num_of_constants) + zna] = 1;
CONSTANTS[(offset * num_of_constants) + zca] = 2;
CONSTANTS[(offset * num_of_constants) + zk] = 1;
CONSTANTS[(offset * num_of_constants) + L] = 0.01;
CONSTANTS[(offset * num_of_constants) + rad] = 0.0011;
CONSTANTS[(offset * num_of_constants) + stim_start] = 10.0;
// bcl edited in the gpu.cu
CONSTANTS[(offset * num_of_constants) + BCL] = 1000.0;
// cvar starts here
CONSTANTS[(offset * num_of_constants) + Jrel_scale] = 1.0;
CONSTANTS[(offset * num_of_constants) + Jup_scale] = 1.0;
CONSTANTS[(offset * num_of_constants) + Jtr_scale] = 1.0;
CONSTANTS[(offset * num_of_constants) + Jleak_scale] = 1.0;
//CONSTANTS[(offset * num_of_constants) + KCaMK_scale] = 1.0;
// cvar ends here
STATES[(offset * num_of_states) + V] = -87;
CONSTANTS[(offset * num_of_constants) + amp] = -80;
CONSTANTS[(offset * num_of_constants) + duration] = 0.5;
CONSTANTS[(offset * num_of_constants) + KmCaMK] = 0.15;
CONSTANTS[(offset * num_of_constants) + aCaMK] = 0.05;
CONSTANTS[(offset * num_of_constants) + bCaMK] = 0.00068;
CONSTANTS[(offset * num_of_constants) + CaMKo] = 0.05;
CONSTANTS[(offset * num_of_constants) + KmCaM] = 0.0015;
STATES[(offset * num_of_states) + CaMKt] = 0;
STATES[(offset * num_of_states) + cass] = 1e-4;
CONSTANTS[(offset * num_of_constants) + cmdnmax_b] = 0.05;
CONSTANTS[(offset * num_of_constants) + kmcmdn] = 0.00238;
CONSTANTS[(offset * num_of_constants) + trpnmax] = 0.07;
CONSTANTS[(offset * num_of_constants) + kmtrpn] = 0.0005;
CONSTANTS[(offset * num_of_constants) + BSRmax] = 0.047;
CONSTANTS[(offset * num_of_constants) + KmBSR] = 0.00087;
CONSTANTS[(offset * num_of_constants) + BSLmax] = 1.124;
CONSTANTS[(offset * num_of_constants) + KmBSL] = 0.0087;
CONSTANTS[(offset * num_of_constants) + csqnmax] = 10;
CONSTANTS[(offset * num_of_constants) + kmcsqn] = 0.8;
STATES[(offset * num_of_states) + nai] = 7;
STATES[(offset * num_of_states) + nass] = 7;
STATES[(offset * num_of_states) + ki] = 145;
STATES[(offset * num_of_states) + kss] = 145;
STATES[(offset * num_of_states) + cansr] = 1.2;
STATES[(offset * num_of_states) + cajsr] = 1.2;
STATES[(offset * num_of_states) + cai] = 1e-4;
CONSTANTS[(offset * num_of_constants) + cm] = 1;
```

```
CONSTANTS[(offset * num_of_constants) + PKNa] = 0.01833;
CONSTANTS[(offset * num_of_constants) + mssV1] = 39.57;
CONSTANTS[(offset * num_of_constants) + mssV2] = 9.871;
CONSTANTS[(offset * num_of_constants) + mtV1] = 11.64;
CONSTANTS[(offset * num_of_constants) + mtV2] = 34.77;
CONSTANTS[(offset * num_of_constants) + mtD1] = 6.765;
CONSTANTS[(offset * num_of_constants) + mtD2] = 8.552;
CONSTANTS[(offset * num_of_constants) + mtV3] = 77.42;
CONSTANTS[(offset * num_of_constants) + mtV4] = 5.955;
STATES[(offset * num_of_states) + m] = 0;
CONSTANTS[(offset * num_of_constants) + hssV1] = 82.9;
CONSTANTS[(offset * num_of_constants) + hssV2] = 6.086;
CONSTANTS[(offset * num_of_constants) + Ahf] = 0.99;
STATES[(offset * num_of_states) + hf] = 1;
STATES[(offset * num_of_states) + hs] = 1;
CONSTANTS[(offset * num_of_constants) + GNa] = 75;
STATES[(offset * num_of_states) + j] = 1;
STATES[(offset * num_of_states) + hsp] = 1;
STATES[(offset * num_of_states) + jp] = 1;
STATES[(offset * num_of_states) + mL] = 0;
CONSTANTS[(offset * num_of_constants) + thL] = 200;
STATES[(offset * num_of_states) + hL] = 1;
STATES[(offset * num_of_states) + hLp] = 1;
CONSTANTS[(offset * num_of_constants) + GNaL_b] = 0.0075;
CONSTANTS[(offset * num_of_constants) + Gto_b] = 0.02;
STATES[(offset * num_of_states) + a] = 0;
STATES[(offset * num_of_states) + iF] = 1;
STATES[(offset * num_of_states) + iS] = 1;
STATES[(offset * num_of_states) + ap] = 0;
STATES[(offset * num_of_states) + iFp] = 1;
STATES[(offset * num_of_states) + iSp] = 1;
CONSTANTS[(offset * num_of_constants) + Kmn] = 0.002;
CONSTANTS[(offset * num_of_constants) + k2n] = 1000;
CONSTANTS[(offset * num_of_constants) + PCa_b] = 0.0001;
STATES[(offset * num_of_states) + d] = 0;
STATES[(offset * num_of_states) + ff] = 1;
STATES[(offset * num_of_states) + fs] = 1;
STATES[(offset * num_of_states) + fcaf] = 1;
STATES[(offset * num_of_states) + fcas] = 1;
STATES[(offset * num_of_states) + jca] = 1;
STATES[(offset * num_of_states) + ffp] = 1;
STATES[(offset * num_of_states) + fcafp] = 1;
STATES[(offset * num_of_states) + nca] = 0;
CONSTANTS[(offset * num_of_constants) + GKr_b] = 0.046;
STATES[(offset * num_of_states) + xrf] = 0;
STATES[(offset * num_of_states) + xrs] = 0;
CONSTANTS[(offset * num_of_constants) + GKs_b] = 0.0034;
STATES[(offset * num_of_states) + xs1] = 0;
STATES[(offset * num_of_states) + xs2] = 0;
```

```
        CONSTANTS[(offset * num_of_constants) + GK1_b] = 0.1908;
        STATES[(offset * num_of_states) + xk1] = 1;
        CONSTANTS[(offset * num_of_constants) + kna1] = 15;
        CONSTANTS[(offset * num_of_constants) + kna2] = 5;
        CONSTANTS[(offset * num_of_constants) + kna3] = 88.12;
        CONSTANTS[(offset * num_of_constants) + kasymm] = 12.5;
        CONSTANTS[(offset * num_of_constants) + wna] = 6e4;
        CONSTANTS[(offset * num_of_constants) + wca] = 6e4;
        CONSTANTS[(offset * num_of_constants) + wnaca] = 5e3;
        CONSTANTS[(offset * num_of_constants) + kcaon] = 1.5e6;
        CONSTANTS[(offset * num_of_constants) + kcaoff] = 5e3;
        CONSTANTS[(offset * num_of_constants) + qna] = 0.5224;
        CONSTANTS[(offset * num_of_constants) + qca] = 0.167;
        CONSTANTS[(offset * num_of_constants) + KmCaAct] = 150e-6;
        CONSTANTS[(offset * num_of_constants) + Gncx_b] = 0.0008;
        CONSTANTS[(offset * num_of_constants) + k1p] = 949.5;
        CONSTANTS[(offset * num_of_constants) + k1m] = 182.4;
        CONSTANTS[(offset * num_of_constants) + k2p] = 687.2;
        CONSTANTS[(offset * num_of_constants) + k2m] = 39.4;
        CONSTANTS[(offset * num_of_constants) + k3p] = 1899;
        CONSTANTS[(offset * num_of_constants) + k3m] = 79300;
        CONSTANTS[(offset * num_of_constants) + k4p] = 639;
        CONSTANTS[(offset * num_of_constants) + k4m] = 40;
        CONSTANTS[(offset * num_of_constants) + Knai0] = 9.073;
        CONSTANTS[(offset * num_of_constants) + Knao0] = 27.78;
        CONSTANTS[(offset * num_of_constants) + delta] = -0.155;
        CONSTANTS[(offset * num_of_constants) + Kki] = 0.5;
        CONSTANTS[(offset * num_of_constants) + Kko] = 0.3582;
        CONSTANTS[(offset * num_of_constants) + MgADP] = 0.05;
        CONSTANTS[(offset * num_of_constants) + MgATP] = 9.8;
        CONSTANTS[(offset * num_of_constants) + Kmgatp] = 1.698e-7;
        CONSTANTS[(offset * num_of_constants) + H] = 1e-7;
        CONSTANTS[(offset * num_of_constants) + eP] = 4.2;
        CONSTANTS[(offset * num_of_constants) + Khp] = 1.698e-7;
        CONSTANTS[(offset * num_of_constants) + Knap] = 224;
        CONSTANTS[(offset * num_of_constants) + Kxkur] = 292;
        CONSTANTS[(offset * num_of_constants) + Pnak_b] = 30;
        CONSTANTS[(offset * num_of_constants) + GKb_b] = 0.003;
        CONSTANTS[(offset * num_of_constants) + PNab] = 3.75e-10;
        CONSTANTS[(offset * num_of_constants) + PCab] = 2.5e-8;
        CONSTANTS[(offset * num_of_constants) + GpCa] = 0.0005;
        CONSTANTS[(offset * num_of_constants) + KmCap] = 0.0005;
        CONSTANTS[(offset * num_of_constants) + bt] = 4.75;
        STATES[(offset * num_of_states) + Jrelnp] = 0;
        STATES[(offset * num_of_states) + Jrelp] = 0;
        CONSTANTS[(offset * num_of_constants) + cmdnmax] = (CONSTANTS[(offset * num_of_constants)
+  celltype]==1.00000  ?   CONSTANTS[(offset  *  num_of_constants)  +  cmdnmax_b]*1.30000  :
CONSTANTS[(offset * num_of_constants) + cmdnmax_b]);
```

```
        CONSTANTS[(offset * num_of_constants) + Ahs] = 1.00000 - CONSTANTS[(offset *
num_of_constants) + Ahf];
        CONSTANTS[(offset * num_of_constants) + thLp] = 3.00000 * CONSTANTS[(offset *
num_of_constants) + thL];
        CONSTANTS[(offset * num_of_constants) + GNaL] = (CONSTANTS[(offset * num_of_constants)
+ celltype]==1.00000 ? CONSTANTS[(offset * num_of_constants) + GNaL_b]*0.600000 :
CONSTANTS[(offset * num_of_constants) + GNaL_b]);
        CONSTANTS[(offset * num_of_constants) + Gto] = (CONSTANTS[(offset * num_of_constants)
+ celltype]==1.00000 ? CONSTANTS[(offset * num_of_constants) + Gto_b]*4.00000 :
CONSTANTS[(offset * num_of_constants) + celltype]==2.00000 ? CONSTANTS[(offset *
num_of_constants) + Gto_b]*4.00000 : CONSTANTS[(offset * num_of_constants) + Gto_b]);
        CONSTANTS[(offset * num_of_constants) + Aff] = 0.600000;
        CONSTANTS[(offset * num_of_constants) + PCa] = (CONSTANTS[(offset * num_of_constants)
+ celltype]==1.00000 ? CONSTANTS[(offset * num_of_constants) + PCa_b]*1.20000 :
CONSTANTS[(offset * num_of_constants) + celltype]==2.00000 ? CONSTANTS[(offset *
num_of_constants) + PCa_b]*2.50000 : CONSTANTS[(offset * num_of_constants) + PCa_b]);
        CONSTANTS[(offset * num_of_constants) + tjca] = 75.0000;
        CONSTANTS[(offset * num_of_constants) + GKr] = (CONSTANTS[(offset * num_of_constants)
+ celltype]==1.00000 ? CONSTANTS[(offset * num_of_constants) + GKr_b]*1.30000 :
CONSTANTS[(offset * num_of_constants) + celltype]==2.00000 ? CONSTANTS[(offset *
num_of_constants) + GKr_b]*0.800000 : CONSTANTS[(offset * num_of_constants) + GKr_b]);
        CONSTANTS[(offset * num_of_constants) + GKs] = (CONSTANTS[(offset * num_of_constants)
+ celltype]==1.00000 ? CONSTANTS[(offset * num_of_constants) + GKs_b]*1.40000 :
CONSTANTS[(offset * num_of_constants) + GKs_b]);
        CONSTANTS[(offset * num_of_constants) + GK1] = (CONSTANTS[(offset * num_of_constants)
+ celltype]==1.00000 ? CONSTANTS[(offset * num_of_constants) + GK1_b]*1.20000 :
CONSTANTS[(offset * num_of_constants) + celltype]==2.00000 ? CONSTANTS[(offset *
num_of_constants) + GK1_b]*1.30000 : CONSTANTS[(offset * num_of_constants) + GK1_b]);
        CONSTANTS[(offset * num_of_constants) + vcell] = 1000.00*3.14000*CONSTANTS[(offset *
num_of_constants) + rad]*CONSTANTS[(offset * num_of_constants) + rad]*CONSTANTS[(offset *
num_of_constants) + L];
        CONSTANTS[(offset * num_of_constants) + GKb] = (CONSTANTS[(offset * num_of_constants)
+ celltype]==1.00000 ? CONSTANTS[(offset * num_of_constants) + GKb_b]*0.600000 :
CONSTANTS[(offset * num_of_constants) + GKb_b]);
        CONSTANTS[(offset * num_of_constants) + a_rel] = 0.500000*CONSTANTS[(offset *
num_of_constants) + bt];
        CONSTANTS[(offset * num_of_constants) + btp] = 1.25000*CONSTANTS[(offset *
num_of_constants) + bt];
        CONSTANTS[(offset * num_of_constants) + upScale] = (CONSTANTS[(offset * num_of_constants)
+ celltype]==1.0000 ? 1.30000 : 1.00000);
        CONSTANTS[(offset * num_of_constants) + Afs] = 1.00000 - CONSTANTS[(offset *
num_of_constants) + Aff];
        CONSTANTS[(offset * num_of_constants) + PCap] = 1.10000*CONSTANTS[(offset *
num_of_constants) + PCa];
        CONSTANTS[(offset * num_of_constants) + PCaNa] = 0.00125000*CONSTANTS[(offset *
num_of_constants) + PCa];
        CONSTANTS[(offset * num_of_constants) + PCaK] = 0.000357400*CONSTANTS[(offset *
num_of_constants) + PCa];
```

```
        CONSTANTS[(offset * num_of_constants) + Ageo] =  2.00000*3.14000*CONSTANTS[(offset *
num_of_constants)     +     rad]*CONSTANTS[(offset     *     num_of_constants)     +     rad]+
2.00000*3.14000*CONSTANTS[(offset    *    num_of_constants)    +    rad]*CONSTANTS[(offset    *
num_of_constants) + L];
        CONSTANTS[(offset  *  num_of_constants)  +  a_relp]  =   0.500000*CONSTANTS[(offset  *
num_of_constants) + btp];
        CONSTANTS[(offset  *  num_of_constants)  +  PCaNap]  =   0.00125000*CONSTANTS[(offset  *
num_of_constants) + PCap];
        CONSTANTS[(offset  *  num_of_constants)  +  PCaKp]  =   0.000357400*CONSTANTS[(offset  *
num_of_constants) + PCap];
        CONSTANTS[(offset  *  num_of_constants)  +  Acap]  =    2.00000*CONSTANTS[(offset  *
num_of_constants) + Ageo];
        CONSTANTS[(offset  *  num_of_constants)  +  vmyo]  =    0.680000*CONSTANTS[(offset  *
num_of_constants) + vcell];
        CONSTANTS[(offset  *  num_of_constants)  +  vnsr]  =    0.0552000*CONSTANTS[(offset  *
num_of_constants) + vcell];
        CONSTANTS[(offset  *  num_of_constants)  +  vjsr]  =    0.00480000*CONSTANTS[(offset  *
num_of_constants) + vcell];
        CONSTANTS[(offset  *  num_of_constants)  +  vss]  =    0.0200000*CONSTANTS[(offset  *
num_of_constants) + vcell];
        CONSTANTS[(offset * num_of_constants) + h10_i] = CONSTANTS[(offset * num_of_constants)
+  kasymm]+1.00000+  (CONSTANTS[(offset  *  num_of_constants)  +  nao]/CONSTANTS[(offset  *
num_of_constants)     +     kna1])*(1.00000+CONSTANTS[(offset     *     num_of_constants)     +
nao]/CONSTANTS[(offset * num_of_constants) + kna2]);
        CONSTANTS[(offset * num_of_constants) + h11_i] = ( CONSTANTS[(offset * num_of_constants)
+ nao]*CONSTANTS[(offset * num_of_constants) + nao])/( CONSTANTS[(offset * num_of_constants) +
h10_i]*CONSTANTS[(offset  *  num_of_constants)  +  kna1]*CONSTANTS[(offset  *  num_of_constants)  +
kna2]);
        CONSTANTS[(offset  *  num_of_constants)  +  h12_i]  =  1.00000/CONSTANTS[(offset  *
num_of_constants) + h10_i];
        CONSTANTS[(offset * num_of_constants) + k1_i] =  CONSTANTS[(offset * num_of_constants)
+ h12_i]*CONSTANTS[(offset * num_of_constants) + cao]*CONSTANTS[(offset * num_of_constants) +
kcaon];
        CONSTANTS[(offset * num_of_constants) + k2_i] = CONSTANTS[(offset * num_of_constants)
+ kcaoff];
        CONSTANTS[(offset * num_of_constants) + k5_i] = CONSTANTS[(offset * num_of_constants)
+ kcaoff];
        CONSTANTS[(offset * num_of_constants) + Gncx] = (CONSTANTS[(offset * num_of_constants)
+  celltype]==1.00000  ?  CONSTANTS[(offset  *  num_of_constants)  +  Gncx_b]*1.10000  :
CONSTANTS[(offset  *  num_of_constants)  +  celltype]==2.00000  ?  CONSTANTS[(offset  *
num_of_constants) + Gncx_b]*1.40000 : CONSTANTS[(offset * num_of_constants) + Gncx_b]);
        CONSTANTS[(offset * num_of_constants) + h10_ss] = CONSTANTS[(offset * num_of_constants)
+  kasymm]+1.00000+  (CONSTANTS[(offset  *  num_of_constants)  +  nao]/CONSTANTS[(offset  *
num_of_constants)     +     kna1])*(1.00000+CONSTANTS[(offset     *     num_of_constants)     +
nao]/CONSTANTS[(offset * num_of_constants) + kna2]);
        CONSTANTS[(offset * num_of_constants) + h11_ss] = ( CONSTANTS[(offset * num_of_constants)
+ nao]*CONSTANTS[(offset * num_of_constants) + nao])/( CONSTANTS[(offset * num_of_constants) +
h10_ss]*CONSTANTS[(offset  *  num_of_constants)  +  kna1]*CONSTANTS[(offset  *  num_of_constants)  +
kna2]);
```

```
        CONSTANTS[(offset * num_of_constants) + h12_ss] = 1.00000/CONSTANTS[(offset *
num_of_constants) + h10_ss];
        CONSTANTS[(offset * num_of_constants) + k1_ss] = CONSTANTS[(offset * num_of_constants)
+ h12_ss]*CONSTANTS[(offset * num_of_constants) + cao]*CONSTANTS[(offset * num_of_constants) +
kcaon];
        CONSTANTS[(offset * num_of_constants) + k2_ss] = CONSTANTS[(offset * num_of_constants)
+ kcaoff];
        CONSTANTS[(offset * num_of_constants) + k5_ss] = CONSTANTS[(offset * num_of_constants)
+ kcaoff];
        CONSTANTS[(offset * num_of_constants) + b1] = CONSTANTS[(offset * num_of_constants) +
k1m]*CONSTANTS[(offset * num_of_constants) + MgADP];
        CONSTANTS[(offset * num_of_constants) + a2] = CONSTANTS[(offset * num_of_constants) +
k2p];
        CONSTANTS[(offset * num_of_constants) + a4] = (( CONSTANTS[(offset * num_of_constants)
+ k4p]*CONSTANTS[(offset * num_of_constants) + MgATP])/CONSTANTS[(offset * num_of_constants) +
Kmgatp])/(1.00000+CONSTANTS[(offset * num_of_constants) + MgATP]/CONSTANTS[(offset *
num_of_constants) + Kmgatp]);
        CONSTANTS[(offset * num_of_constants) + Pnak] = (CONSTANTS[(offset * num_of_constants)
+ celltype]==1.00000 ? CONSTANTS[(offset * num_of_constants) + Pnak_b]*0.900000 :
CONSTANTS[(offset * num_of_constants) + celltype]==2.00000 ? CONSTANTS[(offset *
num_of_constants) + Pnak_b]*0.700000 : CONSTANTS[(offset * num_of_constants) + Pnak_b]);
        }

        __device__ void ___applyDutta(double *CONSTANTS, int offset)
        {
        int num_of_constants = 145;
        //sisanya ganti jadi G (GKs for example)
        CONSTANTS[GKs + (offset * num_of_constants)] *= 1.870;
        CONSTANTS[GKr + (offset * num_of_constants)] *= 1.013;
        CONSTANTS[GK1 + (offset * num_of_constants)] *= 1.698;
        CONSTANTS[PCa + (offset * num_of_constants)] *= 1.007; //pca
        CONSTANTS[GNaL + (offset * num_of_constants)] *= 2.661;
        }

        /*===============*/
        /* Added by ALI */
        /*===============*/
        __device__ void ___applyCvar(double *CONSTANTS, double *cvar, int offset)
        {
          int num_of_constants = 145;

          CONSTANTS[(offset * num_of_constants) +GNa] *= cvar[0 + (offset*18)];
        // GNa
          CONSTANTS[(offset * num_of_constants) +GNaL] *= cvar[1 + (offset*18)];
        // GNaL
          CONSTANTS[(offset * num_of_constants) +Gto] *= cvar[2 + (offset*18)];
        // Gto
          CONSTANTS[(offset * num_of_constants) +GKr] *= cvar[3 + (offset*18)];
        // GKr
```

```
  CONSTANTS[(offset * num_of_constants) +GKs] *= cvar[4 + (offset*18)];
// GKs
  CONSTANTS[(offset * num_of_constants) +GK1] *= cvar[5 + (offset*18)];
// GK1
  CONSTANTS[(offset * num_of_constants) +Gncx] *= cvar[6 + (offset*18)];
// GNaCa
  CONSTANTS[(offset * num_of_constants) +GKb] *= cvar[7 + (offset*18)];
// GKb
  CONSTANTS[(offset * num_of_constants) +PCa] *= cvar[8 + (offset*18)];
// PCa
  CONSTANTS[(offset * num_of_constants) +Pnak] *= cvar[9 + (offset*18)];
// INaK
  CONSTANTS[(offset * num_of_constants) +PNab] *= cvar[10 + (offset*18)];
// PNab
  CONSTANTS[(offset * num_of_constants) +PCab] *= cvar[11 + (offset*18)];
// PCab
  CONSTANTS[(offset * num_of_constants) +GpCa] *= cvar[12 + (offset*18)];
// GpCa
  CONSTANTS[(offset * num_of_constants) +KmCaMK] *= cvar[17 + (offset*18)];  // KCaMK

  // Additional constants
  CONSTANTS[(offset  *  num_of_constants)  +Jrel_scale]  *=  cvar[13  +  (offset*18)];
// SERCA_Total (release)
  CONSTANTS[(offset  *  num_of_constants)  +Jup_scale]  *=  cvar[14  +  (offset*18)];
// RyR_Total (uptake)
  CONSTANTS[(offset  *  num_of_constants)  +Jtr_scale]  *=  cvar[15  +  (offset*18)];
// Trans_Total (NSR to JSR translocation)
  CONSTANTS[(offset  *  num_of_constants)  +Jleak_scale]  *=  cvar[16  +  (offset*18)];
// Leak_Total (Ca leak from NSR)
  // CONSTANTS[(offset * num_of_constants) +KCaMK_scale] *= cvar[17 + (offset*18)];
// KCaMK
}

__device__ void applyDrugEffect(double *CONSTANTS, double conc, double *ic50, double
epsilon, int offset)
{
int num_of_constants = 145;

CONSTANTS[PCa_b+(offset    *    num_of_constants)]    =    CONSTANTS[PCa_b+(offset    *
num_of_constants)] * ( (ic50[0 + (offset*14)] > epsilon && ic50[1+ (offset*14)] > epsilon) ?
1./(1.+pow(conc/ic50[0+ (offset*14)],ic50[1+ (offset*14)])) : 1.);
CONSTANTS[GK1_b+(offset    *    num_of_constants)]    =    CONSTANTS[GK1_b+(offset    *
num_of_constants)] * ((ic50[2 + (offset*14)] > epsilon && ic50[3+ (offset*14)] > epsilon) ?
1./(1.+pow(conc/ic50[2+ (offset*14)],ic50[3+ (offset*14)])) : 1.);
CONSTANTS[GKs_b+(offset    *    num_of_constants)]    =    CONSTANTS[GKs_b+(offset    *
num_of_constants)] * ((ic50[4 + (offset*14)] > epsilon && ic50[5+ (offset*14)] > epsilon) ?
1./(1.+pow(conc/ic50[4+ (offset*14)],ic50[5+ (offset*14)])) : 1.);
```

```
        CONSTANTS[GNa+(offset * num_of_constants)] = CONSTANTS[GNa+(offset * num_of_constants)]
* ((ic50[6 + (offset*14)] > epsilon && ic50[7+ (offset*14)] > epsilon) ? 1./(1.+pow(conc/ic50[6+
(offset*14)],ic50[7+ (offset*14)])) : 1.);
        CONSTANTS[GNaL_b+(offset    *    num_of_constants)]    =    CONSTANTS[GNaL_b+(offset    *
num_of_constants)] * ((ic50[8+ (offset*14)] > epsilon && ic50[9+ (offset*14)] > epsilon) ?
1./(1.+pow(conc/ic50[8+ (offset*14)],ic50[9+ (offset*14)])) : 1.);
        CONSTANTS[Gto_b+(offset    *    num_of_constants)]    =    CONSTANTS[Gto_b+(offset    *
num_of_constants)] * ((ic50[10 + (offset*14)] > epsilon && ic50[11+ (offset*14)] > epsilon) ?
1./(1.+pow(conc/ic50[10+ (offset*14)],ic50[11+ (offset*14)])) : 1.);
        CONSTANTS[GKr_b+(offset    *    num_of_constants)]    =    CONSTANTS[GKr_b+(offset    *
num_of_constants)] * ((ic50[12+ (offset*14)] > epsilon && ic50[13+ (offset*14)] > epsilon) ?
1./(1.+pow(conc/ic50[12+ (offset*14)],ic50[13+ (offset*14)])) : 1.);

        }

        // void initConsts(int offset)
        // {
        //      ___initConsts(0.,offset);
        // }

        // void initConsts(double type)
        // {
        //      ___initConsts(type, offset);
        // }

        __device__ void initConsts(double *CONSTANTS, double *STATES, double type, double conc,
double *ic50, double *cvar, bool is_dutta, bool is_cvar,  int offset)
        {
          // int num_of_constants = 145;
          //  printf("ic50:%d  %lf,  %lf,  %lf\n",offset,ic50[0  +  (offset*14)],ic50[1  +
(offset*14)],ic50[2 + (offset*14)]);

                ___initConsts(CONSTANTS, STATES, type, offset); // initconst kan minta
                // // mpi_printf(0,"Celltype: %lf\n", CONSTANTS[celltype]);
                // #ifndef COMPONENT_PATCH // for patch clamp component based research
                //  //  mpi_printf(0,"Control  %lf  %lf  %lf  %lf  %lf\n",  CONSTANTS[PCa],
CONSTANTS[GK1], CONSTANTS[GKs], CONSTANTS[GNaL], CONSTANTS[GKr]);
                // #endif
                if(is_dutta == true){
                        ___applyDutta(CONSTANTS, offset);
                }

          if(is_cvar == true){
                        ___applyCvar(CONSTANTS, cvar, offset);
                }
                // #ifndef COMPONENT_PATCH
                //  //  mpi_printf(0,"After  Dutta  %lf  %lf  %lf  %lf  %lf\n",  CONSTANTS[PCa],
CONSTANTS[GK1], CONSTANTS[GKs], CONSTANTS[GNaL], CONSTANTS[GKr]);
                // #endif
```

```
                // ___applyDrugEffect(CONSTANTS, conc, ic50, 10E-14, offset);
                // #ifndef COMPONENT_PATCH
                //  //  mpi_printf(0,"After  drug  %lf  %lf  %lf  %lf  %lf\n",  CONSTANTS[PCa],
CONSTANTS[GK1], CONSTANTS[GKs], CONSTANTS[GNaL], CONSTANTS[GKr]);
                // #endif


        }

        __device__ void computeRates( double TIME, double *CONSTANTS, double *RATES, double
*STATES, double *ALGEBRAIC, int offset )
        {
        int num_of_constants = 145; //done
        int num_of_states = 41; //done
        int num_of_algebraic = 199; //done
        int num_of_rates = 41; //done

        //new part
        CONSTANTS[(offset * num_of_constants) + cmdnmax] = (CONSTANTS[(offset * num_of_constants)
+  celltype]==1.00000  ?    CONSTANTS[(offset  *  num_of_constants)  +  cmdnmax_b]*1.30000  :
CONSTANTS[(offset * num_of_constants) + cmdnmax_b]);
        CONSTANTS[(offset * num_of_constants) + GNaL] = (CONSTANTS[(offset * num_of_constants)
+  celltype]==1.00000  ?    CONSTANTS[(offset  *  num_of_constants)  +  GNaL_b]*0.600000  :
CONSTANTS[(offset * num_of_constants) + GNaL_b]);
        CONSTANTS[(offset * num_of_constants) + Gto] = (CONSTANTS[(offset * num_of_constants)
+  celltype]==1.00000  ?    CONSTANTS[(offset  *  num_of_constants)  +  Gto_b]*4.00000  :
CONSTANTS[(offset  *  num_of_constants)  +  celltype]==2.00000  ?    CONSTANTS[(offset  *
num_of_constants) + Gto_b]*4.00000 : CONSTANTS[(offset * num_of_constants) + Gto_b]);
        CONSTANTS[(offset * num_of_constants) + PCa] = (CONSTANTS[(offset * num_of_constants)
+  celltype]==1.00000  ?    CONSTANTS[(offset  *  num_of_constants)  +  PCa_b]*1.20000  :
CONSTANTS[(offset  *  num_of_constants)  +  celltype]==2.00000  ?    CONSTANTS[(offset  *
num_of_constants) + PCa_b]*2.50000 : CONSTANTS[(offset * num_of_constants) + PCa_b]);
        CONSTANTS[(offset * num_of_constants) + GKr] = (CONSTANTS[(offset * num_of_constants)
+  celltype]==1.00000  ?    CONSTANTS[(offset  *  num_of_constants)  +  GKr_b]*1.30000  :
CONSTANTS[(offset  *  num_of_constants)  +  celltype]==2.00000  ?    CONSTANTS[(offset  *
num_of_constants) + GKr_b]*0.800000 : CONSTANTS[(offset * num_of_constants) + GKr_b]);
        CONSTANTS[(offset * num_of_constants) + GKs] = (CONSTANTS[(offset * num_of_constants)
+  celltype]==1.00000  ?    CONSTANTS[(offset  *  num_of_constants)  +  GKs_b]*1.40000  :
CONSTANTS[(offset * num_of_constants) + GKs_b]);
        CONSTANTS[(offset * num_of_constants) + GK1] = (CONSTANTS[(offset * num_of_constants)
+  celltype]==1.00000  ?    CONSTANTS[(offset  *  num_of_constants)  +  GK1_b]*1.20000  :
CONSTANTS[(offset  *  num_of_constants)  +  celltype]==2.00000  ?    CONSTANTS[(offset  *
num_of_constants) + GK1_b]*1.30000 : CONSTANTS[(offset * num_of_constants) + GK1_b]);
        CONSTANTS[(offset * num_of_constants) + GKb] = (CONSTANTS[(offset * num_of_constants)
+  celltype]==1.00000  ?    CONSTANTS[(offset  *  num_of_constants)  +  GKb_b]*0.600000  :
CONSTANTS[(offset * num_of_constants) + GKb_b]);
        CONSTANTS[(offset * num_of_constants) + upScale] = (CONSTANTS[(offset * num_of_constants)
+ celltype]==1.0000 ? 1.30000 : 1.00000);
```

```
        CONSTANTS[(offset * num_of_constants) + Gncx] = (CONSTANTS[(offset * num_of_constants)
+ celltype]==1.00000 ? CONSTANTS[(offset * num_of_constants) + Gncx_b]*1.10000 :
CONSTANTS[(offset * num_of_constants) + celltype]==2.00000 ? CONSTANTS[(offset *
num_of_constants) + Gncx_b]*1.40000 : CONSTANTS[(offset * num_of_constants) + Gncx_b]);
        CONSTANTS[(offset * num_of_constants) + Pnak] = (CONSTANTS[(offset * num_of_constants)
+ celltype]==1.00000 ? CONSTANTS[(offset * num_of_constants) + Pnak_b]*0.900000 :
CONSTANTS[(offset * num_of_constants) + celltype]==2.00000 ? CONSTANTS[(offset *
num_of_constants) + Pnak_b]*0.700000 : CONSTANTS[(offset * num_of_constants) + Pnak_b]);
        // new part ends
        ALGEBRAIC[(offset * num_of_algebraic) +Istim] = (TIME>=CONSTANTS[(offset *
num_of_constants) + stim_start] && (TIME - CONSTANTS[(offset * num_of_constants) + stim_start])
- floor((TIME - CONSTANTS[(offset * num_of_constants) + stim_start])/CONSTANTS[(offset *
num_of_constants) + BCL])*CONSTANTS[(offset * num_of_constants) + BCL]<=CONSTANTS[(offset *
num_of_constants) + duration] ? CONSTANTS[(offset * num_of_constants) + amp] : 0.000000);
        // in libcml there is ifdef TISSUE, ask further

        ALGEBRAIC[(offset * num_of_algebraic) +hLss] = 1.00000/(1.00000+exp((STATES[(offset *
num_of_states) + V]+87.6100)/7.48800));
        ALGEBRAIC[(offset * num_of_algebraic) +hLssp] = 1.00000/(1.00000+exp((STATES[(offset *
num_of_states) + V]+93.8100)/7.48800));
        ALGEBRAIC[(offset * num_of_algebraic) +mss] = 1.00000/(1.00000+exp(- (STATES[(offset *
num_of_states) + V]+CONSTANTS[(offset * num_of_constants) + mssV1])/CONSTANTS[(offset *
num_of_constants) + mssV2]));
        ALGEBRAIC[(offset * num_of_algebraic) +tm] = 1.00000/( CONSTANTS[(offset *
num_of_constants) + mtD1]*exp((STATES[(offset * num_of_states) + V]+CONSTANTS[(offset *
num_of_constants) + mtV1])/CONSTANTS[(offset * num_of_constants) + mtV2])+ CONSTANTS[(offset *
num_of_constants) + mtD2]*exp(- (STATES[(offset * num_of_states) + V]+CONSTANTS[(offset *
num_of_constants) + mtV3])/CONSTANTS[(offset * num_of_constants) + mtV4]));
        ALGEBRAIC[(offset * num_of_algebraic) +hss] = 1.00000/(1.00000+exp((STATES[(offset *
num_of_states) + V]+CONSTANTS[(offset * num_of_constants) + hssV1])/CONSTANTS[(offset *
num_of_constants) + hssV2]));
        ALGEBRAIC[(offset * num_of_algebraic) +thf] = 1.00000/( 1.43200e-05*exp(-
(STATES[(offset * num_of_states) + V]+1.19600)/6.28500)+ 6.14900*exp((STATES[(offset *
num_of_states) + V]+0.509600)/20.2700));
        ALGEBRAIC[(offset * num_of_algebraic) +ths] = 1.00000/( 0.00979400*exp(-
(STATES[(offset * num_of_states) + V]+17.9500)/28.0500)+ 0.334300*exp((STATES[(offset *
num_of_states) + V]+5.73000)/56.6600));
        ALGEBRAIC[(offset * num_of_algebraic) +ass] = 1.00000/(1.00000+exp(- (STATES[(offset *
num_of_states) + V] - 14.3400)/14.8200));
        ALGEBRAIC[(offset * num_of_algebraic) +ta] = 1.05150/(1.00000/( 1.20890*(1.00000+exp(-
(STATES[(offset * num_of_states) + V] - 18.4099)/29.3814)))+3.50000/(1.00000+exp((STATES[(offset
* num_of_states) + V]+100.000)/29.3814)));
        ALGEBRAIC[(offset * num_of_algebraic) +dss] = 1.00000/(1.00000+exp(- (STATES[(offset *
num_of_states) + V]+3.94000)/4.23000));
        ALGEBRAIC[(offset * num_of_algebraic) +td] = 0.600000+1.00000/(exp( -
0.0500000*(STATES[(offset * num_of_states) + V]+6.00000))+exp( 0.0900000*(STATES[(offset *
num_of_states) + V]+14.0000)));
        ALGEBRAIC[(offset * num_of_algebraic) + fss] = 1.00000/(1.00000+exp((STATES[(offset *
num_of_states) + V]+19.5800)/3.69600));
```

```
        ALGEBRAIC[(offset * num_of_algebraic) + tff] = 7.00000+1.00000/( 0.00450000*exp(-
(STATES[(offset * num_of_states) + V]+20.0000)/10.0000)+ 0.00450000*exp((STATES[(offset *
num_of_states) + V]+20.0000)/10.0000));
        ALGEBRAIC[(offset * num_of_algebraic) + tfs] = 1000.00+1.00000/( 3.50000e-05*exp(-
(STATES[(offset * num_of_states) + V]+5.00000)/4.00000)+ 3.50000e-05*exp((STATES[(offset *
num_of_states) + V]+5.00000)/6.00000));
        ALGEBRAIC[(offset * num_of_algebraic) + fcass] = ALGEBRAIC[(offset * num_of_algebraic)
+ fss];
        ALGEBRAIC[(offset * num_of_algebraic) + km2n] =  STATES[(offset * num_of_states) +
jca]*1.00000;
        ALGEBRAIC[(offset * num_of_algebraic) + anca] = 1.00000/(CONSTANTS[(offset *
num_of_constants) + k2n]/ALGEBRAIC[(offset * num_of_algebraic) +
km2n]+pow(1.00000+CONSTANTS[(offset * num_of_constants) + Kmn]/STATES[(offset * num_of_states)
+ cass], 4.00000));
        ALGEBRAIC[(offset * num_of_algebraic) + xrss] = 1.00000/(1.00000+exp(- (STATES[(offset
* num_of_states) + V]+8.33700)/6.78900));
        ALGEBRAIC[(offset * num_of_algebraic) + txrf] =
12.9800+1.00000/( 0.365200*exp((STATES[(offset * num_of_states) + V] - 31.6600)/3.86900)+
4.12300e-05*exp(- (STATES[(offset * num_of_states) + V] - 47.7800)/20.3800));
        ALGEBRAIC[(offset * num_of_algebraic) + txrs] =
1.86500+1.00000/( 0.0662900*exp((STATES[(offset * num_of_states) + V] - 34.7000)/7.35500)+
1.12800e-05*exp(- (STATES[(offset * num_of_states) + V] - 29.7400)/25.9400));
        ALGEBRAIC[(offset * num_of_algebraic) + xs1ss] = 1.00000/(1.00000+exp(- (STATES[(offset
* num_of_states) + V]+11.6000)/8.93200));
        ALGEBRAIC[(offset * num_of_algebraic) + txs1] =
817.300+1.00000/( 0.000232600*exp((STATES[(offset * num_of_states) + V]+48.2800)/17.8000)+
0.00129200*exp(- (STATES[(offset * num_of_states) + V]+210.000)/230.000));
        ALGEBRAIC[(offset * num_of_algebraic) + xk1ss] = 1.00000/(1.00000+exp(- (STATES[(offset
* num_of_states) + V]+ 2.55380*CONSTANTS[(offset * num_of_constants) +
ko]+144.590)/( 1.56920*CONSTANTS[(offset * num_of_constants) + ko]+3.81150)));
        ALGEBRAIC[(offset * num_of_algebraic) + txk1] = 122.200/(exp(- (STATES[(offset *
num_of_states) + V]+127.200)/20.3600)+exp((STATES[(offset * num_of_states) +
V]+236.800)/69.3300));
        ALGEBRAIC[(offset * num_of_algebraic) + jss] = ALGEBRAIC[(offset * num_of_algebraic) +
hss];
        ALGEBRAIC[(offset * num_of_algebraic) + tj] = 2.03800+1.00000/( 0.0213600*exp(-
(STATES[(offset * num_of_states) + V]+100.600)/8.28100)+ 0.305200*exp((STATES[(offset *
num_of_states) + V]+0.994100)/38.4500));
        ALGEBRAIC[(offset * num_of_algebraic) + assp] = 1.00000/(1.00000+exp(- (STATES[(offset
* num_of_states) + V] - 24.3400)/14.8200));
        ALGEBRAIC[(offset * num_of_algebraic) + tfcaf] = 7.00000+1.00000/( 0.0400000*exp(-
(STATES[(offset * num_of_states) + V] - 4.00000)/7.00000)+ 0.0400000*exp((STATES[(offset *
num_of_states) + V] - 4.00000)/7.00000));
        ALGEBRAIC[(offset * num_of_algebraic) + tfcas] = 100.000+1.00000/( 0.000120000*exp(-
STATES[(offset * num_of_states) + V]/3.00000)+ 0.000120000*exp(STATES[(offset * num_of_states)
+ V]/7.00000));
        ALGEBRAIC[(offset * num_of_algebraic) + tffp] =  2.50000*ALGEBRAIC[(offset *
num_of_algebraic) + tff];
```

```
        ALGEBRAIC[(offset * num_of_algebraic) + xs2ss] = ALGEBRAIC[(offset * num_of_algebraic)
+ xs1ss];
        ALGEBRAIC[(offset * num_of_algebraic) + txs2] = 1.00000/( 0.0100000*exp((STATES[(offset
* num_of_states) + V] - 50.0000)/20.0000)+ 0.0193000*exp(- (STATES[(offset * num_of_states) +
V]+66.5400)/31.0000));
        ALGEBRAIC[(offset * num_of_algebraic) + CaMKb] = ( CONSTANTS[(offset * num_of_constants)
+ CaMKo]*(1.00000 - STATES[(offset * num_of_states) + CaMKt]))/(1.00000+CONSTANTS[(offset *
num_of_constants) + KmCaM]/STATES[(offset * num_of_states) + cass]);
        ALGEBRAIC[(offset * num_of_algebraic) + hssp] = 1.00000/(1.00000+exp((STATES[(offset *
num_of_states) + V]+89.1000)/6.08600));
        ALGEBRAIC[(offset * num_of_algebraic) + thsp] = 3.00000*ALGEBRAIC[(offset *
num_of_algebraic) + ths];
        ALGEBRAIC[(offset * num_of_algebraic) + tjp] = 1.46000*ALGEBRAIC[(offset *
num_of_algebraic) + tj];
        ALGEBRAIC[(offset * num_of_algebraic) + mLss] = 1.00000/(1.00000+exp(- (STATES[(offset
* num_of_states) + V]+42.8500)/5.26400));
        ALGEBRAIC[(offset * num_of_algebraic) + tmL] = ALGEBRAIC[(offset * num_of_algebraic) +
tm];
        ALGEBRAIC[(offset * num_of_algebraic) + tfcafp] = 2.50000*ALGEBRAIC[(offset *
num_of_algebraic) + tfcaf];
        ALGEBRAIC[(offset * num_of_algebraic) + iss] = 1.00000/(1.00000+exp((STATES[(offset *
num_of_states) + V]+43.9400)/5.71100));
        ALGEBRAIC[(offset * num_of_algebraic) + delta_epi] = (CONSTANTS[(offset *
num_of_constants) + celltype]==1.00000 ? 1.00000 - 0.950000/(1.00000+exp((STATES[(offset *
num_of_states) + V]+70.0000)/5.00000)) : 1.00000);
        ALGEBRAIC[(offset * num_of_algebraic) + tiF_b] = 4.56200+1.00000/( 0.393300*exp(-
(STATES[(offset * num_of_states) + V]+100.000)/100.000)+ 0.0800400*exp((STATES[(offset *
num_of_states) + V]+50.0000)/16.5900));
        ALGEBRAIC[(offset * num_of_algebraic) + tiF] = ALGEBRAIC[(offset * num_of_algebraic)
+ tiF_b]*ALGEBRAIC[(offset * num_of_algebraic) + delta_epi];
        ALGEBRAIC[(offset * num_of_algebraic) + tiS_b] = 23.6200+1.00000/( 0.00141600*exp(-
(STATES[(offset * num_of_states) + V]+96.5200)/59.0500)+ 1.78000e-08*exp((STATES[(offset *
num_of_states) + V]+114.100)/8.07900));
        ALGEBRAIC[(offset * num_of_algebraic) + tiS] = ALGEBRAIC[(offset * num_of_algebraic)
+ tiS_b]*ALGEBRAIC[(offset * num_of_algebraic) + delta_epi];
        ALGEBRAIC[(offset * num_of_algebraic) + dti_develop] =
1.35400+0.000100000/(exp((STATES[(offset * num_of_states) + V] - 167.400)/15.8900)+exp(-
(STATES[(offset * num_of_states) + V] - 12.2300)/0.215400));
        ALGEBRAIC[(offset * num_of_algebraic) + dti_recover] = 1.00000 -
0.500000/(1.00000+exp((STATES[(offset * num_of_states) + V]+70.0000)/20.0000));
        ALGEBRAIC[(offset * num_of_algebraic) + tiFp] = ALGEBRAIC[(offset * num_of_algebraic)
+ dti_develop]*ALGEBRAIC[(offset * num_of_algebraic) + dti_recover]*ALGEBRAIC[(offset *
num_of_algebraic) + tiF];
        ALGEBRAIC[(offset * num_of_algebraic) + tiSp] = ALGEBRAIC[(offset * num_of_algebraic)
+ dti_develop]*ALGEBRAIC[(offset * num_of_algebraic) + dti_recover]*ALGEBRAIC[(offset *
num_of_algebraic) + tiS];
        ALGEBRAIC[(offset * num_of_algebraic) + f] = CONSTANTS[(offset * num_of_constants) +
Aff]*STATES[(offset * num_of_states) + ff]+ CONSTANTS[(offset * num_of_constants) +
Afs]*STATES[(offset * num_of_states) + fs];
```

```
        ALGEBRAIC[(offset * num_of_algebraic) + Afcaf] = 0.300000+0.600000/(1.00000+exp((STATES[(offset * num_of_states) + V] - 10.0000)/10.0000));
        ALGEBRAIC[(offset * num_of_algebraic) + Afcas] = 1.00000 - ALGEBRAIC[(offset * num_of_algebraic) + Afcaf];
        ALGEBRAIC[(offset * num_of_algebraic) + fca] = ALGEBRAIC[(offset * num_of_algebraic) + Afcaf]*STATES[(offset * num_of_states) + fcaf]+ ALGEBRAIC[(offset * num_of_algebraic) + Afcas]*STATES[(offset * num_of_states) + fcas];
        ALGEBRAIC[(offset * num_of_algebraic) + fp] = CONSTANTS[(offset * num_of_constants) + Aff]*STATES[(offset * num_of_states) + ffp]+ CONSTANTS[(offset * num_of_constants) + Afs]*STATES[(offset * num_of_states) + fs];
        ALGEBRAIC[(offset * num_of_algebraic) + fcap] = ALGEBRAIC[(offset * num_of_algebraic) + Afcaf]*STATES[(offset * num_of_states) + fcafp]+ ALGEBRAIC[(offset * num_of_algebraic) + Afcas]*STATES[(offset * num_of_states) + fcas];
        ALGEBRAIC[(offset * num_of_algebraic) + vffrt] = ( STATES[(offset * num_of_states) + V]*CONSTANTS[(offset * num_of_constants) + F]*CONSTANTS[(offset * num_of_constants) + F])/( CONSTANTS[(offset * num_of_constants) + R]*CONSTANTS[(offset * num_of_constants) + T]);
        ALGEBRAIC[(offset * num_of_algebraic) + vfrt] = ( STATES[(offset * num_of_states) + V]*CONSTANTS[(offset * num_of_constants) + F])/( CONSTANTS[(offset * num_of_constants) + R]*CONSTANTS[(offset * num_of_constants) + T]);
        ALGEBRAIC[(offset * num_of_algebraic) + PhiCaL] = ( 4.00000*ALGEBRAIC[(offset * num_of_algebraic) + vffrt]*( STATES[(offset * num_of_states) + cass]*exp( 2.00000*ALGEBRAIC[(offset * num_of_algebraic) + vfrt]) - 0.341000*CONSTANTS[(offset * num_of_constants) + cao]))/(exp( 2.00000*ALGEBRAIC[(offset * num_of_algebraic) + vfrt]) - 1.00000);
        ALGEBRAIC[(offset * num_of_algebraic) + CaMKa] = ALGEBRAIC[(offset * num_of_algebraic) + CaMKb]+STATES[(offset * num_of_states) + CaMKt];
        ALGEBRAIC[(offset * num_of_algebraic) + fICaLp] = 1.00000/(1.00000+CONSTANTS[(offset * num_of_constants) + KmCaMK]/ALGEBRAIC[(offset * num_of_algebraic) + CaMKa]);
        ALGEBRAIC[(offset * num_of_algebraic) + ICaL] = (1.00000 - ALGEBRAIC[(offset * num_of_algebraic) + fICaLp])*CONSTANTS[(offset * num_of_constants) + PCa]*ALGEBRAIC[(offset * num_of_algebraic) + PhiCaL]*STATES[(offset * num_of_states) + d]*( ALGEBRAIC[(offset * num_of_algebraic) + f]*(1.00000 - STATES[(offset * num_of_states) + nca])+ STATES[(offset * num_of_states) + jca]*ALGEBRAIC[(offset * num_of_algebraic) + fca]*STATES[(offset * num_of_states) + nca])+ ALGEBRAIC[(offset * num_of_algebraic) + fICaLp]*CONSTANTS[(offset * num_of_constants) + PCap]*ALGEBRAIC[(offset * num_of_algebraic) + PhiCaL]*STATES[(offset * num_of_states) + d]*( ALGEBRAIC[(offset * num_of_algebraic) + fp]*(1.00000 - STATES[(offset * num_of_states) + nca])+ STATES[(offset * num_of_states) + jca]*ALGEBRAIC[(offset * num_of_algebraic) + fcap]*STATES[(offset * num_of_states) + nca]);
        ALGEBRAIC[(offset * num_of_algebraic) + Jrel_inf_temp] = ( CONSTANTS[(offset * num_of_constants) + a_rel]*- ALGEBRAIC[(offset * num_of_algebraic) + ICaL])/(1.00000+1.00000*pow(1.50000/STATES[(offset * num_of_states) + cajsr], 8.00000));
        ALGEBRAIC[(offset * num_of_algebraic) + Jrel_inf] = (CONSTANTS[(offset * num_of_constants) + celltype]==2.00000 ? ALGEBRAIC[(offset * num_of_algebraic) + Jrel_inf_temp]*1.70000 : ALGEBRAIC[(offset * num_of_algebraic) + Jrel_inf_temp]);
        ALGEBRAIC[(offset * num_of_algebraic) + tau_rel_temp] = CONSTANTS[(offset * num_of_constants) + bt]/(1.00000+0.0123000/STATES[(offset * num_of_states) + cajsr]);
        ALGEBRAIC[(offset * num_of_algebraic) + tau_rel] = (ALGEBRAIC[(offset * num_of_algebraic) + tau_rel_temp]<0.00100000 ? 0.00100000 : ALGEBRAIC[(offset * num_of_algebraic) + tau_rel_temp]);
```

```
        ALGEBRAIC[(offset * num_of_algebraic) + Jrel_temp] = ( CONSTANTS[(offset *
num_of_constants) + a_relp]*- ALGEBRAIC[(offset * num_of_algebraic) +
ICaL])/(1.00000+pow(1.50000/STATES[(offset * num_of_states) + cajsr], 8.00000));
        ALGEBRAIC[(offset * num_of_algebraic) + Jrel_infp] = (CONSTANTS[(offset *
num_of_constants) + celltype]==2.00000 ? ALGEBRAIC[(offset * num_of_algebraic) +
Jrel_temp]*1.70000 : ALGEBRAIC[(offset * num_of_algebraic) + Jrel_temp]);
        ALGEBRAIC[(offset * num_of_algebraic) + tau_relp_temp] = CONSTANTS[(offset *
num_of_constants) + btp]/(1.00000+0.0123000/STATES[(offset * num_of_states) + cajsr]);
        ALGEBRAIC[(offset * num_of_algebraic) + tau_relp] = (ALGEBRAIC[(offset *
num_of_algebraic) + tau_relp_temp]<0.00100000 ? 0.00100000 : ALGEBRAIC[(offset *
num_of_algebraic) + tau_relp_temp]);
        ALGEBRAIC[(offset * num_of_algebraic) + EK] = (( CONSTANTS[(offset * num_of_constants)
+ R]*CONSTANTS[(offset * num_of_constants) + T])/CONSTANTS[(offset * num_of_constants) +
F])*log(CONSTANTS[(offset * num_of_constants) + ko]/STATES[(offset * num_of_states) + ki]);
        ALGEBRAIC[(offset * num_of_algebraic) + AiF] = 1.00000/(1.00000+exp((STATES[(offset *
num_of_states) + V] - 213.600)/151.200));
        ALGEBRAIC[(offset * num_of_algebraic) + AiS] = 1.00000 - ALGEBRAIC[(offset *
num_of_algebraic) + AiF];
        ALGEBRAIC[(offset * num_of_algebraic) + i] = ALGEBRAIC[(offset * num_of_algebraic) +
AiF]*STATES[(offset * num_of_states) + iF]+ ALGEBRAIC[(offset * num_of_algebraic) +
AiS]*STATES[(offset * num_of_states) + iS];
        ALGEBRAIC[(offset * num_of_algebraic) + ip] = ALGEBRAIC[(offset * num_of_algebraic) +
AiF]*STATES[(offset * num_of_states) + iFp]+ ALGEBRAIC[(offset * num_of_algebraic) +
AiS]*STATES[(offset * num_of_states) + iSp];
        ALGEBRAIC[(offset * num_of_algebraic) + fItop] = 1.00000/(1.00000+CONSTANTS[(offset *
num_of_constants) + KmCaMK]/ALGEBRAIC[(offset * num_of_algebraic) + CaMKa]);
        ALGEBRAIC[(offset * num_of_algebraic) + Ito] = CONSTANTS[(offset * num_of_constants)
+ Gto]*(STATES[(offset * num_of_states) + V] - ALGEBRAIC[(offset * num_of_algebraic) +
EK])*( (1.00000 - ALGEBRAIC[(offset * num_of_algebraic) + fItop])*STATES[(offset * num_of_states)
+ a]*ALGEBRAIC[(offset * num_of_algebraic) + i]+ ALGEBRAIC[(offset * num_of_algebraic) +
fItop]*STATES[(offset * num_of_states) + ap]*ALGEBRAIC[(offset * num_of_algebraic) + ip]);
        ALGEBRAIC[(offset * num_of_algebraic) + Axrf] = 1.00000/(1.00000+exp((STATES[(offset *
num_of_states) + V]+54.8100)/38.2100));
        ALGEBRAIC[(offset * num_of_algebraic) + Axrs] = 1.00000 - ALGEBRAIC[(offset *
num_of_algebraic) + Axrf];
        ALGEBRAIC[(offset * num_of_algebraic) + xr] = ALGEBRAIC[(offset * num_of_algebraic) +
Axrf]*STATES[(offset * num_of_states) + xrf]+ ALGEBRAIC[(offset * num_of_algebraic) +
Axrs]*STATES[(offset * num_of_states) + xrs];
        ALGEBRAIC[(offset * num_of_algebraic) + rkr] = ( (1.00000/(1.00000+exp((STATES[(offset
* num_of_states) + V]+55.0000)/75.0000)))*1.00000)/(1.00000+exp((STATES[(offset * num_of_states)
+ V] - 10.0000)/30.0000));
        ALGEBRAIC[(offset * num_of_algebraic) + IKr] = CONSTANTS[(offset * num_of_constants)
+ GKr]* pow((CONSTANTS[(offset * num_of_constants) + ko]/5.40000), 1.0 / 2)*ALGEBRAIC[(offset *
num_of_algebraic) + xr]*ALGEBRAIC[(offset * num_of_algebraic) + rkr]*(STATES[(offset *
num_of_states) + V] - ALGEBRAIC[(offset * num_of_algebraic) + EK]);
        ALGEBRAIC[(offset * num_of_algebraic) + EKs] = (( CONSTANTS[(offset * num_of_constants)
+ R]*CONSTANTS[(offset * num_of_constants) + T])/CONSTANTS[(offset * num_of_constants) +
F])*log((CONSTANTS[(offset * num_of_constants) + ko]+ CONSTANTS[(offset * num_of_constants) +
```

```
PKNa]*CONSTANTS[(offset * num_of_constants) + nao])/(STATES[(offset * num_of_states) + ki]+
CONSTANTS[(offset * num_of_constants) + PKNa]*STATES[(offset * num_of_states) + nai]));
        ALGEBRAIC[(offset * num_of_algebraic) + KsCa] = 1.00000+0.600000/(1.00000+pow(3.80000e-
05/STATES[(offset * num_of_states) + cai], 1.40000));
        ALGEBRAIC[(offset * num_of_algebraic) + IKs] =  CONSTANTS[(offset * num_of_constants)
+ GKs]*ALGEBRAIC[(offset * num_of_algebraic) + KsCa]*STATES[(offset * num_of_states) +
xs1]*STATES[(offset * num_of_states) + xs2]*(STATES[(offset * num_of_states) + V] -
ALGEBRAIC[(offset * num_of_algebraic) + EKs]);
        ALGEBRAIC[(offset * num_of_algebraic) + rk1] = 1.00000/(1.00000+exp(((STATES[(offset *
num_of_states) + V]+105.800) -  2.60000*CONSTANTS[(offset * num_of_constants) + ko])/9.49300));
        ALGEBRAIC[(offset * num_of_algebraic) + IK1] =  CONSTANTS[(offset * num_of_constants)
+ GK1]*  pow(CONSTANTS[(offset * num_of_constants) + ko], 1.0 / 2)*ALGEBRAIC[(offset *
num_of_algebraic) + rk1]*STATES[(offset * num_of_states) + xk1]*(STATES[(offset * num_of_states)
+ V] - ALGEBRAIC[(offset * num_of_algebraic) + EK]);
        ALGEBRAIC[(offset * num_of_algebraic) + Knao] =  CONSTANTS[(offset * num_of_constants)
+ Knao0]*exp((  (1.00000 - CONSTANTS[(offset * num_of_constants) + delta])*STATES[(offset *
num_of_states) + V]*CONSTANTS[(offset * num_of_constants) + F])/( 3.00000*CONSTANTS[(offset *
num_of_constants) + R]*CONSTANTS[(offset * num_of_constants) + T]));
        ALGEBRAIC[(offset * num_of_algebraic) + a3] = ( CONSTANTS[(offset * num_of_constants)
+ k3p]*pow(CONSTANTS[(offset * num_of_constants) + ko]/CONSTANTS[(offset * num_of_constants) +
Kko], 2.00000))/((pow(1.00000+CONSTANTS[(offset * num_of_constants) + nao]/ALGEBRAIC[(offset *
num_of_algebraic) + Knao], 3.00000)+pow(1.00000+CONSTANTS[(offset * num_of_constants) +
ko]/CONSTANTS[(offset * num_of_constants) + Kko], 2.00000)) - 1.00000);
        ALGEBRAIC[(offset * num_of_algebraic) + P] = CONSTANTS[(offset * num_of_constants) +
eP]/(1.00000+CONSTANTS[(offset * num_of_constants) + H]/CONSTANTS[(offset * num_of_constants) +
Khp]+STATES[(offset * num_of_states) + nai]/CONSTANTS[(offset * num_of_constants) +
Knap]+STATES[(offset * num_of_states) + ki]/CONSTANTS[(offset * num_of_constants) + Kxkur]);
        ALGEBRAIC[(offset * num_of_algebraic) + b3] = ( CONSTANTS[(offset * num_of_constants)
+ k3m]*ALGEBRAIC[(offset * num_of_algebraic) + P]*CONSTANTS[(offset * num_of_constants) +
H])/(1.00000+CONSTANTS[(offset * num_of_constants) + MgATP]/CONSTANTS[(offset * num_of_constants)
+ Kmgatp]);
        ALGEBRAIC[(offset * num_of_algebraic) + Knai] =  CONSTANTS[(offset * num_of_constants)
+ Knai0]*exp(( CONSTANTS[(offset * num_of_constants) + delta]*STATES[(offset * num_of_states) +
V]*CONSTANTS[(offset * num_of_constants) + F])/( 3.00000*CONSTANTS[(offset * num_of_constants)
+ R]*CONSTANTS[(offset * num_of_constants) + T]));
        ALGEBRAIC[(offset * num_of_algebraic) + a1] = ( CONSTANTS[(offset * num_of_constants)
+ k1p]*pow(STATES[(offset * num_of_states) + nai]/ALGEBRAIC[(offset * num_of_algebraic) + Knai],
3.00000))/((pow(1.00000+STATES[(offset * num_of_states) + nai]/ALGEBRAIC[(offset *
num_of_algebraic) + Knai], 3.00000)+pow(1.00000+STATES[(offset * num_of_states) +
ki]/CONSTANTS[(offset * num_of_constants) + Kki], 2.00000)) - 1.00000);
        ALGEBRAIC[(offset * num_of_algebraic) + b2] = ( CONSTANTS[(offset * num_of_constants)
+ k2m]*pow(CONSTANTS[(offset * num_of_constants) + nao]/ALGEBRAIC[(offset * num_of_algebraic) +
Knao], 3.00000))/((pow(1.00000+CONSTANTS[(offset * num_of_constants) + nao]/ALGEBRAIC[(offset *
num_of_algebraic) + Knao], 3.00000)+pow(1.00000+CONSTANTS[(offset * num_of_constants) +
ko]/CONSTANTS[(offset * num_of_constants) + Kko], 2.00000)) - 1.00000);
        ALGEBRAIC[(offset * num_of_algebraic) + b4] = ( CONSTANTS[(offset * num_of_constants)
+ k4m]*pow(STATES[(offset * num_of_states) + ki]/CONSTANTS[(offset * num_of_constants) + Kki],
2.00000))/((pow(1.00000+STATES[(offset * num_of_states) + nai]/ALGEBRAIC[(offset *
```

```
num_of_algebraic)  +  Knai],  3.00000)+pow(1.00000+STATES[(offset  *  num_of_states)  +
ki]/CONSTANTS[(offset * num_of_constants) + Kki], 2.00000)) - 1.00000);
        ALGEBRAIC[(offset * num_of_algebraic) + x1] =  CONSTANTS[(offset * num_of_constants) +
a4]*ALGEBRAIC[(offset * num_of_algebraic) + a1]*CONSTANTS[(offset * num_of_constants) + a2]+
ALGEBRAIC[(offset  *  num_of_algebraic)  +  b2]*ALGEBRAIC[(offset  *  num_of_algebraic)  +
b4]*ALGEBRAIC[(offset  *  num_of_algebraic)  +  b3]+  CONSTANTS[(offset  *  num_of_constants)  +
a2]*ALGEBRAIC[(offset * num_of_algebraic) + b4]*ALGEBRAIC[(offset * num_of_algebraic) + b3]+
ALGEBRAIC[(offset  *  num_of_algebraic)  +  b3]*ALGEBRAIC[(offset  *  num_of_algebraic)  +
a1]*CONSTANTS[(offset * num_of_constants) + a2];
        ALGEBRAIC[(offset * num_of_algebraic) + x2] =  ALGEBRAIC[(offset * num_of_algebraic) +
b2]*CONSTANTS[(offset * num_of_constants) + b1]*ALGEBRAIC[(offset * num_of_algebraic) + b4]+
ALGEBRAIC[(offset  *  num_of_algebraic)  +  a1]*CONSTANTS[(offset  *  num_of_constants)  +
a2]*ALGEBRAIC[(offset  *  num_of_algebraic)  +  a3]+  ALGEBRAIC[(offset  *  num_of_algebraic)  +
a3]*CONSTANTS[(offset * num_of_constants) + b1]*ALGEBRAIC[(offset * num_of_algebraic) + b4]+
CONSTANTS[(offset  *  num_of_constants)  +  a2]*ALGEBRAIC[(offset  *  num_of_algebraic)  +
a3]*ALGEBRAIC[(offset * num_of_algebraic) + b4];
        ALGEBRAIC[(offset * num_of_algebraic) + x3] =  CONSTANTS[(offset * num_of_constants) +
a2]*ALGEBRAIC[(offset * num_of_algebraic) + a3]*CONSTANTS[(offset * num_of_constants) + a4]+
ALGEBRAIC[(offset  *  num_of_algebraic)  +  b3]*ALGEBRAIC[(offset  *  num_of_algebraic)  +
b2]*CONSTANTS[(offset  *  num_of_constants)  +  b1]+  ALGEBRAIC[(offset  *  num_of_algebraic)  +
b2]*CONSTANTS[(offset * num_of_constants) + b1]*CONSTANTS[(offset * num_of_constants) + a4]+
ALGEBRAIC[(offset  *  num_of_algebraic)  +  a3]*CONSTANTS[(offset  *  num_of_constants)  +
a4]*CONSTANTS[(offset * num_of_constants) + b1];
        ALGEBRAIC[(offset * num_of_algebraic) + x4] =  ALGEBRAIC[(offset * num_of_algebraic) +
b4]*ALGEBRAIC[(offset * num_of_algebraic) + b3]*ALGEBRAIC[(offset * num_of_algebraic) + b2]+
ALGEBRAIC[(offset  *  num_of_algebraic)  +  a3]*CONSTANTS[(offset  *  num_of_constants)  +
a4]*ALGEBRAIC[(offset  *  num_of_algebraic)  +  a1]+  ALGEBRAIC[(offset  *  num_of_algebraic)  +
b2]*CONSTANTS[(offset * num_of_constants) + a4]*ALGEBRAIC[(offset * num_of_algebraic) + a1]+
ALGEBRAIC[(offset  *  num_of_algebraic)  +  b3]*ALGEBRAIC[(offset  *  num_of_algebraic)  +
b2]*ALGEBRAIC[(offset * num_of_algebraic) + a1];
        ALGEBRAIC[(offset * num_of_algebraic) + E1] = ALGEBRAIC[(offset * num_of_algebraic) +
x1]/(ALGEBRAIC[(offset  *  num_of_algebraic)  +  x1]+ALGEBRAIC[(offset  *  num_of_algebraic)  +
x2]+ALGEBRAIC[(offset * num_of_algebraic) + x3]+ALGEBRAIC[(offset * num_of_algebraic) + x4]);
        ALGEBRAIC[(offset * num_of_algebraic) + E2] = ALGEBRAIC[(offset * num_of_algebraic) +
x2]/(ALGEBRAIC[(offset  *  num_of_algebraic)  +  x1]+ALGEBRAIC[(offset  *  num_of_algebraic)  +
x2]+ALGEBRAIC[(offset * num_of_algebraic) + x3]+ALGEBRAIC[(offset * num_of_algebraic) + x4]);
        ALGEBRAIC[(offset  *  num_of_algebraic)  +  JnakNa]  =  3.00000*(  ALGEBRAIC[(offset  *
num_of_algebraic) + E1]*ALGEBRAIC[(offset * num_of_algebraic) + a3] -  ALGEBRAIC[(offset *
num_of_algebraic) + E2]*ALGEBRAIC[(offset * num_of_algebraic) + b3]);
        ALGEBRAIC[(offset * num_of_algebraic) + E3] = ALGEBRAIC[(offset * num_of_algebraic) +
x3]/(ALGEBRAIC[(offset  *  num_of_algebraic)  +  x1]+ALGEBRAIC[(offset  *  num_of_algebraic)  +
x2]+ALGEBRAIC[(offset * num_of_algebraic) + x3]+ALGEBRAIC[(offset * num_of_algebraic) + x4]);
        ALGEBRAIC[(offset * num_of_algebraic) + E4] = ALGEBRAIC[(offset * num_of_algebraic) +
x4]/(ALGEBRAIC[(offset  *  num_of_algebraic)  +  x1]+ALGEBRAIC[(offset  *  num_of_algebraic)  +
x2]+ALGEBRAIC[(offset * num_of_algebraic) + x3]+ALGEBRAIC[(offset * num_of_algebraic) + x4]);
        ALGEBRAIC[(offset  *  num_of_algebraic)  +  JnakK]  =  2.00000*(  ALGEBRAIC[(offset  *
num_of_algebraic) + E4]*CONSTANTS[(offset * num_of_constants) + b1] -  ALGEBRAIC[(offset *
num_of_algebraic) + E3]*ALGEBRAIC[(offset * num_of_algebraic) + a1]);
```

```
        ALGEBRAIC[(offset * num_of_algebraic) + INaK] =  CONSTANTS[(offset * num_of_constants)
+ Pnak]*( CONSTANTS[(offset * num_of_constants) + zna]*ALGEBRAIC[(offset * num_of_algebraic) +
JnakNa]+ CONSTANTS[(offset * num_of_constants) + zk]*ALGEBRAIC[(offset * num_of_algebraic) +
JnakK]);
        ALGEBRAIC[(offset * num_of_algebraic) + xkb] = 1.00000/(1.00000+exp(- (STATES[(offset
* num_of_states) + V] - 14.4800)/18.3400));
        ALGEBRAIC[(offset * num_of_algebraic) + IKb] =  CONSTANTS[(offset * num_of_constants)
+ GKb]*ALGEBRAIC[(offset * num_of_algebraic) + xkb]*(STATES[(offset * num_of_states) + V] -
ALGEBRAIC[(offset * num_of_algebraic) + EK]);
        ALGEBRAIC[(offset * num_of_algebraic) + JdiffK] = (STATES[(offset * num_of_states) +
kss] - STATES[(offset * num_of_states) + ki])/2.00000;
        ALGEBRAIC[(offset * num_of_algebraic) + PhiCaK] = ( 1.00000*ALGEBRAIC[(offset *
num_of_algebraic)     +     vffrt]*(     0.750000*STATES[(offset     *     num_of_states)     +
kss]*exp( 1.00000*ALGEBRAIC[(offset * num_of_algebraic) + vfrt]) -  0.750000*CONSTANTS[(offset
* num_of_constants) + ko]))/(exp( 1.00000*ALGEBRAIC[(offset * num_of_algebraic) + vfrt]) -
1.00000);
        ALGEBRAIC[(offset * num_of_algebraic) + ICaK] =   (1.00000 - ALGEBRAIC[(offset *
num_of_algebraic) + fICaLp])*CONSTANTS[(offset * num_of_constants) + PCaK]*ALGEBRAIC[(offset *
num_of_algebraic) + PhiCaK]*STATES[(offset * num_of_states) + d]*( ALGEBRAIC[(offset *
num_of_algebraic) + f]*(1.00000 - STATES[(offset * num_of_states) + nca])+ STATES[(offset *
num_of_states) + jca]*ALGEBRAIC[(offset * num_of_algebraic) + fca]*STATES[(offset *
num_of_states) + nca])+ ALGEBRAIC[(offset * num_of_algebraic) + fICaLp]*CONSTANTS[(offset *
num_of_constants) + PCaKp]*ALGEBRAIC[(offset * num_of_algebraic) + PhiCaK]*STATES[(offset *
num_of_states) + d]*( ALGEBRAIC[(offset * num_of_algebraic) + fp]*(1.00000 - STATES[(offset *
num_of_states) + nca])+ STATES[(offset * num_of_states) + jca]*ALGEBRAIC[(offset *
num_of_algebraic) + fcap]*STATES[(offset * num_of_states) + nca]);
        ALGEBRAIC[(offset * num_of_algebraic) + ENa] =  (( CONSTANTS[(offset * num_of_constants)
+ R]*CONSTANTS[(offset * num_of_constants) + T])/CONSTANTS[(offset * num_of_constants) +
F])*log(CONSTANTS[(offset * num_of_constants) + nao]/STATES[(offset * num_of_states) + nai]);
        ALGEBRAIC[(offset * num_of_algebraic) + h] =  CONSTANTS[(offset * num_of_constants) +
Ahf]*STATES[(offset * num_of_states) + hf]+ CONSTANTS[(offset * num_of_constants) +
Ahs]*STATES[(offset * num_of_states) + hs];
        ALGEBRAIC[(offset * num_of_algebraic) + hp] =  CONSTANTS[(offset * num_of_constants) +
Ahf]*STATES[(offset * num_of_states) + hf]+ CONSTANTS[(offset * num_of_constants) +
Ahs]*STATES[(offset * num_of_states) + hsp];
        ALGEBRAIC[(offset * num_of_algebraic) + fINap] = 1.00000/(1.00000+CONSTANTS[(offset *
num_of_constants) + KmCaMK]/ALGEBRAIC[(offset * num_of_algebraic) + CaMKa]);
        ALGEBRAIC[(offset * num_of_algebraic) + INa] =  CONSTANTS[(offset * num_of_constants)
+ GNa]*(STATES[(offset * num_of_states) + V] - ALGEBRAIC[(offset * num_of_algebraic) +
ENa])*pow(STATES[(offset * num_of_states) + m], 3.00000)*( (1.00000 - ALGEBRAIC[(offset *
num_of_algebraic) + fINap])*ALGEBRAIC[(offset * num_of_algebraic) + h]*STATES[(offset *
num_of_states) + j]+ ALGEBRAIC[(offset * num_of_algebraic) + fINap]*ALGEBRAIC[(offset *
num_of_algebraic) + hp]*STATES[(offset * num_of_states) + jp]);
        ALGEBRAIC[(offset * num_of_algebraic) + fINaLp] = 1.00000/(1.00000+CONSTANTS[(offset *
num_of_constants) + KmCaMK]/ALGEBRAIC[(offset * num_of_algebraic) + CaMKa]);
        ALGEBRAIC[(offset * num_of_algebraic) + INaL] =  CONSTANTS[(offset * num_of_constants)
+ GNaL]*(STATES[(offset * num_of_states) + V] - ALGEBRAIC[(offset * num_of_algebraic) +
ENa])*STATES[(offset * num_of_states) + mL]*( (1.00000 - ALGEBRAIC[(offset * num_of_algebraic)
```

```
+ fINaLp])*STATES[(offset * num_of_states) + hL]+ ALGEBRAIC[(offset * num_of_algebraic) +
fINaLp]*STATES[(offset * num_of_states) + hLp]);
        ALGEBRAIC[(offset        *        num_of_algebraic)        +        allo_i]        =
1.00000/(1.00000+pow(CONSTANTS[(offset   *   num_of_constants)   +   KmCaAct]/STATES[(offset   *
num_of_states) + cai], 2.00000));
        ALGEBRAIC[(offset   *   num_of_algebraic)   +   hna]   =   exp((   CONSTANTS[(offset   *
num_of_constants)   +   qna]*STATES[(offset   *   num_of_states)   +   V]*CONSTANTS[(offset   *
num_of_constants)  +  F])/(  CONSTANTS[(offset  *  num_of_constants)  +  R]*CONSTANTS[(offset  *
num_of_constants) + T]));
        ALGEBRAIC[(offset  *  num_of_algebraic)  +  h7_i]  =  1.00000+  (CONSTANTS[(offset  *
num_of_constants)       +       nao]/CONSTANTS[(offset       *       num_of_constants)       +
kna3])*(1.00000+1.00000/ALGEBRAIC[(offset * num_of_algebraic) + hna]);
        ALGEBRAIC[(offset * num_of_algebraic) + h8_i] = CONSTANTS[(offset * num_of_constants)
+ nao]/( CONSTANTS[(offset * num_of_constants) + kna3]*ALGEBRAIC[(offset * num_of_algebraic) +
hna]*ALGEBRAIC[(offset * num_of_algebraic) + h7_i]);
        ALGEBRAIC[(offset * num_of_algebraic) + k3pp_i] =  ALGEBRAIC[(offset * num_of_algebraic)
+ h8_i]*CONSTANTS[(offset * num_of_constants) + wnaca];
        ALGEBRAIC[(offset * num_of_algebraic) + h1_i] = 1.00000+ (STATES[(offset * num_of_states)
+   nai]/CONSTANTS[(offset   *   num_of_constants)   +   kna3])*(1.00000+ALGEBRAIC[(offset   *
num_of_algebraic) + hna]);
        ALGEBRAIC[(offset * num_of_algebraic) + h2_i] = ( STATES[(offset * num_of_states) +
nai]*ALGEBRAIC[(offset * num_of_algebraic) + hna])/( CONSTANTS[(offset * num_of_constants) +
kna3]*ALGEBRAIC[(offset * num_of_algebraic) + h1_i]);
        ALGEBRAIC[(offset * num_of_algebraic) + k4pp_i] =  ALGEBRAIC[(offset * num_of_algebraic)
+ h2_i]*CONSTANTS[(offset * num_of_constants) + wnaca];
        ALGEBRAIC[(offset * num_of_algebraic) + h4_i] = 1.00000+ (STATES[(offset * num_of_states)
+ nai]/CONSTANTS[(offset * num_of_constants) + kna1])*(1.00000+STATES[(offset * num_of_states)
+ nai]/CONSTANTS[(offset * num_of_constants) + kna2]);
        ALGEBRAIC[(offset * num_of_algebraic) + h5_i] = ( STATES[(offset * num_of_states) +
nai]*STATES[(offset  *  num_of_states)  +  nai])/(  ALGEBRAIC[(offset  *  num_of_algebraic)  +
h4_i]*CONSTANTS[(offset  *  num_of_constants)  +  kna1]*CONSTANTS[(offset  *  num_of_constants)  +
kna2]);
        ALGEBRAIC[(offset * num_of_algebraic) + k7_i] =  ALGEBRAIC[(offset * num_of_algebraic)
+ h5_i]*ALGEBRAIC[(offset * num_of_algebraic) + h2_i]*CONSTANTS[(offset * num_of_constants) +
wna];
        ALGEBRAIC[(offset * num_of_algebraic) + k8_i] =  ALGEBRAIC[(offset * num_of_algebraic)
+ h8_i]*CONSTANTS[(offset * num_of_constants) + h11_i]*CONSTANTS[(offset * num_of_constants) +
wna];
        ALGEBRAIC[(offset   *   num_of_algebraic)   +   h9_i]   =   1.00000/ALGEBRAIC[(offset   *
num_of_algebraic) + h7_i];
        ALGEBRAIC[(offset * num_of_algebraic) + k3p_i] =  ALGEBRAIC[(offset * num_of_algebraic)
+ h9_i]*CONSTANTS[(offset * num_of_constants) + wca];
        ALGEBRAIC[(offset * num_of_algebraic) + k3_i] = ALGEBRAIC[(offset * num_of_algebraic)
+ k3p_i]+ALGEBRAIC[(offset * num_of_algebraic) + k3pp_i];
        ALGEBRAIC[(offset   *   num_of_algebraic)   +   hca]   =   exp((   CONSTANTS[(offset   *
num_of_constants)   +   qca]*STATES[(offset   *   num_of_states)   +   V]*CONSTANTS[(offset   *
num_of_constants)  +  F])/(  CONSTANTS[(offset  *  num_of_constants)  +  R]*CONSTANTS[(offset  *
num_of_constants) + T]));
```

```
        ALGEBRAIC[(offset * num_of_algebraic) + h3_i] = 1.00000/ALGEBRAIC[(offset *
num_of_algebraic) + h1_i];
        ALGEBRAIC[(offset * num_of_algebraic) + k4p_i] = ( ALGEBRAIC[(offset * num_of_algebraic)
+ h3_i]*CONSTANTS[(offset * num_of_constants) + wca])/ALGEBRAIC[(offset * num_of_algebraic) +
hca];
        ALGEBRAIC[(offset * num_of_algebraic) + k4_i] = ALGEBRAIC[(offset * num_of_algebraic)
+ k4p_i]+ALGEBRAIC[(offset * num_of_algebraic) + k4pp_i];
        ALGEBRAIC[(offset * num_of_algebraic) + h6_i] = 1.00000/ALGEBRAIC[(offset *
num_of_algebraic) + h4_i];
        ALGEBRAIC[(offset * num_of_algebraic) + k6_i] = ALGEBRAIC[(offset * num_of_algebraic)
+ h6_i]*STATES[(offset * num_of_states) + cai]*CONSTANTS[(offset * num_of_constants) + kcaon];
        ALGEBRAIC[(offset * num_of_algebraic) + x1_i] = CONSTANTS[(offset * num_of_constants)
+ k2_i]*ALGEBRAIC[(offset * num_of_algebraic) + k4_i]*(ALGEBRAIC[(offset * num_of_algebraic) +
k7_i]+ALGEBRAIC[(offset * num_of_algebraic) + k6_i])+ CONSTANTS[(offset * num_of_constants) +
k5_i]*ALGEBRAIC[(offset * num_of_algebraic) + k7_i]*(CONSTANTS[(offset * num_of_constants) +
k2_i]+ALGEBRAIC[(offset * num_of_algebraic) + k3_i]);
        ALGEBRAIC[(offset * num_of_algebraic) + x2_i] = CONSTANTS[(offset * num_of_constants)
+ k1_i]*ALGEBRAIC[(offset * num_of_algebraic) + k7_i]*(ALGEBRAIC[(offset * num_of_algebraic) +
k4_i]+CONSTANTS[(offset * num_of_constants) + k5_i])+ ALGEBRAIC[(offset * num_of_algebraic) +
k4_i]*ALGEBRAIC[(offset * num_of_algebraic) + k6_i]*(CONSTANTS[(offset * num_of_constants) +
k1_i]+ALGEBRAIC[(offset * num_of_algebraic) + k8_i]);
        ALGEBRAIC[(offset * num_of_algebraic) + x3_i] = CONSTANTS[(offset * num_of_constants)
+ k1_i]*ALGEBRAIC[(offset * num_of_algebraic) + k3_i]*(ALGEBRAIC[(offset * num_of_algebraic) +
k7_i]+ALGEBRAIC[(offset * num_of_algebraic) + k6_i])+ ALGEBRAIC[(offset * num_of_algebraic) +
k8_i]*ALGEBRAIC[(offset * num_of_algebraic) + k6_i]*(CONSTANTS[(offset * num_of_constants) +
k2_i]+ALGEBRAIC[(offset * num_of_algebraic) + k3_i]);
        ALGEBRAIC[(offset * num_of_algebraic) + x4_i] = CONSTANTS[(offset * num_of_constants)
+ k2_i]*ALGEBRAIC[(offset * num_of_algebraic) + k8_i]*(ALGEBRAIC[(offset * num_of_algebraic) +
k4_i]+CONSTANTS[(offset * num_of_constants) + k5_i])+ ALGEBRAIC[(offset * num_of_algebraic) +
k3_i]*CONSTANTS[(offset * num_of_constants) + k5_i]*(CONSTANTS[(offset * num_of_constants) +
k1_i]+ALGEBRAIC[(offset * num_of_algebraic) + k8_i]);
        ALGEBRAIC[(offset * num_of_algebraic) + E1_i] = ALGEBRAIC[(offset * num_of_algebraic)
+ x1_i]/(ALGEBRAIC[(offset * num_of_algebraic) + x1_i]+ALGEBRAIC[(offset * num_of_algebraic)
+ x2_i]+ALGEBRAIC[(offset * num_of_algebraic) + x3_i]+ALGEBRAIC[(offset * num_of_algebraic) +
x4_i]);
        ALGEBRAIC[(offset * num_of_algebraic) + E2_i] = ALGEBRAIC[(offset * num_of_algebraic)
+ x2_i]/(ALGEBRAIC[(offset * num_of_algebraic) + x1_i]+ALGEBRAIC[(offset * num_of_algebraic) +
x2_i]+ALGEBRAIC[(offset * num_of_algebraic) + x3_i]+ALGEBRAIC[(offset * num_of_algebraic) +
x4_i]);
        ALGEBRAIC[(offset * num_of_algebraic) + E3_i] = ALGEBRAIC[(offset * num_of_algebraic)
+ x3_i]/(ALGEBRAIC[(offset * num_of_algebraic) + x1_i]+ALGEBRAIC[(offset * num_of_algebraic) +
x2_i]+ALGEBRAIC[(offset * num_of_algebraic) + x3_i]+ALGEBRAIC[(offset * num_of_algebraic) +
x4_i]);
        ALGEBRAIC[(offset * num_of_algebraic) + E4_i] = ALGEBRAIC[(offset * num_of_algebraic)
+ x4_i]/(ALGEBRAIC[(offset * num_of_algebraic) + x1_i]+ALGEBRAIC[(offset * num_of_algebraic) +
x2_i]+ALGEBRAIC[(offset * num_of_algebraic) + x3_i]+ALGEBRAIC[(offset * num_of_algebraic) +
x4_i]);
        ALGEBRAIC[(offset * num_of_algebraic) + JncxNa_i] = ( 3.00000*( ALGEBRAIC[(offset *
num_of_algebraic) + E4_i]*ALGEBRAIC[(offset * num_of_algebraic) + k7_i] - ALGEBRAIC[(offset *
```

```
num_of_algebraic) + E1_i]*ALGEBRAIC[(offset * num_of_algebraic) + k8_i])+ ALGEBRAIC[(offset *
num_of_algebraic) + E3_i]*ALGEBRAIC[(offset * num_of_algebraic) + k4pp_i]) - ALGEBRAIC[(offset
* num_of_algebraic) + E2_i]*ALGEBRAIC[(offset * num_of_algebraic) + k3pp_i];
        ALGEBRAIC[(offset * num_of_algebraic) + JncxCa_i] = ALGEBRAIC[(offset *
num_of_algebraic) + E2_i]*CONSTANTS[(offset * num_of_constants) + k2_i] - ALGEBRAIC[(offset *
num_of_algebraic) + E1_i]*CONSTANTS[(offset * num_of_constants) + k1_i];
        ALGEBRAIC[(offset * num_of_algebraic) + INaCa_i] = 0.800000*CONSTANTS[(offset *
num_of_constants) + Gncx]*ALGEBRAIC[(offset * num_of_algebraic) + allo_i]*( CONSTANTS[(offset *
num_of_constants) + zna]*ALGEBRAIC[(offset * num_of_algebraic) + JncxNa_i]+ CONSTANTS[(offset *
num_of_constants) + zca]*ALGEBRAIC[(offset * num_of_algebraic) + JncxCa_i]);
        ALGEBRAIC[(offset * num_of_algebraic) + INab] = ( CONSTANTS[(offset * num_of_constants)
+ PNab]*ALGEBRAIC[(offset * num_of_algebraic) + vffrt]*( STATES[(offset * num_of_states) +
nai]*exp(ALGEBRAIC[(offset * num_of_algebraic) + vfrt]) - CONSTANTS[(offset * num_of_constants)
+ nao]))/(exp(ALGEBRAIC[(offset * num_of_algebraic) + vfrt]) - 1.00000);
        ALGEBRAIC[(offset * num_of_algebraic) + JdiffNa] = (STATES[(offset * num_of_states) +
nass] - STATES[(offset * num_of_states) + nai])/2.00000;
        ALGEBRAIC[(offset * num_of_algebraic) + PhiCaNa] = ( 1.00000*ALGEBRAIC[(offset *
num_of_algebraic) + vffrt]*( 0.750000*STATES[(offset * num_of_states) +
nass]*exp( 1.00000*ALGEBRAIC[(offset * num_of_algebraic) + vfrt]) - 0.750000*CONSTANTS[(offset
* num_of_constants) + nao]))/(exp( 1.00000*ALGEBRAIC[(offset * num_of_algebraic) + vfrt]) -
1.00000);
        ALGEBRAIC[(offset * num_of_algebraic) + ICaNa] = (1.00000 - ALGEBRAIC[(offset *
num_of_algebraic) + fICaLp])*CONSTANTS[(offset * num_of_constants) + PCaNa]*ALGEBRAIC[(offset *
num_of_algebraic) + PhiCaNa]*STATES[(offset * num_of_states) + d]*( ALGEBRAIC[(offset *
num_of_algebraic) + f]*(1.00000 - STATES[(offset * num_of_states) + nca])+ STATES[(offset *
num_of_states) + jca]*ALGEBRAIC[(offset * num_of_algebraic) + fca]*STATES[(offset *
num_of_states) + nca])+ ALGEBRAIC[(offset * num_of_algebraic) + fICaLp]*CONSTANTS[(offset *
num_of_constants) + PCaNap]*ALGEBRAIC[(offset * num_of_algebraic) + PhiCaNa]*STATES[(offset *
num_of_states) + d]*( ALGEBRAIC[(offset * num_of_algebraic) + fp]*(1.00000 - STATES[(offset *
num_of_states) + nca])+ STATES[(offset * num_of_states) + jca]*ALGEBRAIC[(offset *
num_of_algebraic) + fcap]*STATES[(offset * num_of_states) + nca]);
        ALGEBRAIC[(offset * num_of_algebraic) + allo_ss] =
1.00000/(1.00000+pow(CONSTANTS[(offset * num_of_constants) + KmCaAct]/STATES[(offset *
num_of_states) + cass], 2.00000));
        ALGEBRAIC[(offset * num_of_algebraic) + h7_ss] = 1.00000+ (CONSTANTS[(offset *
num_of_constants) + nao]/CONSTANTS[(offset * num_of_constants) +
kna3])*(1.00000+1.00000/ALGEBRAIC[(offset * num_of_algebraic) + hna]);
        ALGEBRAIC[(offset * num_of_algebraic) + h8_ss] = CONSTANTS[(offset * num_of_constants)
+ nao]/( CONSTANTS[(offset * num_of_constants) + kna3]*ALGEBRAIC[(offset * num_of_algebraic) +
hna]*ALGEBRAIC[(offset * num_of_algebraic) + h7_ss]);
        ALGEBRAIC[(offset * num_of_algebraic) + k3pp_ss] = ALGEBRAIC[(offset * num_of_algebraic)
+ h8_ss]*CONSTANTS[(offset * num_of_constants) + wnaca];
        ALGEBRAIC[(offset * num_of_algebraic) + h1_ss] = 1.00000+ (STATES[(offset *
num_of_states) + nass]/CONSTANTS[(offset * num_of_constants) + kna3])*(1.00000+ALGEBRAIC[(offset
* num_of_algebraic) + hna]);
        ALGEBRAIC[(offset * num_of_algebraic) + h2_ss] = ( STATES[(offset * num_of_states) +
nass]*ALGEBRAIC[(offset * num_of_algebraic) + hna])/( CONSTANTS[(offset * num_of_constants) +
kna3]*ALGEBRAIC[(offset * num_of_algebraic) + h1_ss]);
```

```
        ALGEBRAIC[(offset * num_of_algebraic) + k4pp_ss] =  ALGEBRAIC[(offset * num_of_algebraic)
+ h2_ss]*CONSTANTS[(offset * num_of_constants) + wnaca];
        ALGEBRAIC[(offset  *  num_of_algebraic)  +  h4_ss]  =  1.00000+  (STATES[(offset  *
num_of_states) + nass]/CONSTANTS[(offset * num_of_constants) + kna1])*(1.00000+STATES[(offset *
num_of_states) + nass]/CONSTANTS[(offset * num_of_constants) + kna2]);
        ALGEBRAIC[(offset * num_of_algebraic) + h5_ss] = ( STATES[(offset * num_of_states) +
nass]*STATES[(offset  *  num_of_states)  +  nass])/(  ALGEBRAIC[(offset  *  num_of_algebraic)  +
h4_ss]*CONSTANTS[(offset  *  num_of_constants)  +  kna1]*CONSTANTS[(offset  *  num_of_constants)  +
kna2]);
        ALGEBRAIC[(offset * num_of_algebraic) + k7_ss] =  ALGEBRAIC[(offset * num_of_algebraic)
+ h5_ss]*ALGEBRAIC[(offset * num_of_algebraic) + h2_ss]*CONSTANTS[(offset * num_of_constants) +
wna];
        ALGEBRAIC[(offset * num_of_algebraic) + k8_ss] =  ALGEBRAIC[(offset * num_of_algebraic)
+ h8_ss]*CONSTANTS[(offset * num_of_constants) + h11_ss]*CONSTANTS[(offset * num_of_constants)
+ wna];
        ALGEBRAIC[(offset  *  num_of_algebraic)  +  h9_ss]  =  1.00000/ALGEBRAIC[(offset  *
num_of_algebraic) + h7_ss];
        ALGEBRAIC[(offset * num_of_algebraic) + k3p_ss] =  ALGEBRAIC[(offset * num_of_algebraic)
+ h9_ss]*CONSTANTS[(offset * num_of_constants) + wca];
        ALGEBRAIC[(offset * num_of_algebraic) + k3_ss] = ALGEBRAIC[(offset * num_of_algebraic)
+ k3p_ss]+ALGEBRAIC[(offset * num_of_algebraic) + k3pp_ss];
        ALGEBRAIC[(offset  *  num_of_algebraic)  +  h3_ss]  =  1.00000/ALGEBRAIC[(offset  *
num_of_algebraic) + h1_ss];
        ALGEBRAIC[(offset * num_of_algebraic) + k4p_ss] = ( ALGEBRAIC[(offset * num_of_algebraic)
+ h3_ss]*CONSTANTS[(offset * num_of_constants) + wca])/ALGEBRAIC[(offset * num_of_algebraic) +
hca];
        ALGEBRAIC[(offset * num_of_algebraic) + k4_ss] = ALGEBRAIC[(offset * num_of_algebraic)
+ k4p_ss]+ALGEBRAIC[(offset * num_of_algebraic) + k4pp_ss];
        ALGEBRAIC[(offset  *  num_of_algebraic)  +  h6_ss]  =  1.00000/ALGEBRAIC[(offset  *
num_of_algebraic) + h4_ss];
        ALGEBRAIC[(offset * num_of_algebraic) + k6_ss] =  ALGEBRAIC[(offset * num_of_algebraic)
+ h6_ss]*STATES[(offset * num_of_states) + cass]*CONSTANTS[(offset * num_of_constants) + kcaon];
        ALGEBRAIC[(offset * num_of_algebraic) + x1_ss] =  CONSTANTS[(offset * num_of_constants)
+ k2_ss]*ALGEBRAIC[(offset * num_of_algebraic) + k4_ss]*(ALGEBRAIC[(offset * num_of_algebraic)
+ k7_ss]+ALGEBRAIC[(offset * num_of_algebraic) + k6_ss])+ CONSTANTS[(offset * num_of_constants)
+ k5_ss]*ALGEBRAIC[(offset * num_of_algebraic) + k7_ss]*(CONSTANTS[(offset * num_of_constants)
+ k2_ss]+ALGEBRAIC[(offset * num_of_algebraic) + k3_ss]);
        ALGEBRAIC[(offset * num_of_algebraic) + x2_ss] =  CONSTANTS[(offset * num_of_constants)
+ k1_ss]*ALGEBRAIC[(offset * num_of_algebraic) + k7_ss]*(ALGEBRAIC[(offset * num_of_algebraic)
+ k4_ss]+CONSTANTS[(offset * num_of_constants) + k5_ss])+ ALGEBRAIC[(offset * num_of_algebraic)
+ k4_ss]*ALGEBRAIC[(offset * num_of_algebraic) + k6_ss]*(CONSTANTS[(offset * num_of_constants)
+ k1_ss]+ALGEBRAIC[(offset * num_of_algebraic) + k8_ss]);
        ALGEBRAIC[(offset * num_of_algebraic) + x3_ss] =  CONSTANTS[(offset * num_of_constants)
+ k1_ss]*ALGEBRAIC[(offset * num_of_algebraic) + k3_ss]*(ALGEBRAIC[(offset * num_of_algebraic)
+ k7_ss]+ALGEBRAIC[(offset * num_of_algebraic) + k6_ss])+ ALGEBRAIC[(offset * num_of_algebraic)
+ k8_ss]*ALGEBRAIC[(offset * num_of_algebraic) + k6_ss]*(CONSTANTS[(offset * num_of_constants)
+ k2_ss]+ALGEBRAIC[(offset * num_of_algebraic) + k3_ss]);
        ALGEBRAIC[(offset * num_of_algebraic) + x4_ss] =  CONSTANTS[(offset * num_of_constants)
+ k2_ss]*ALGEBRAIC[(offset * num_of_algebraic) + k8_ss]*(ALGEBRAIC[(offset * num_of_algebraic)
```

```
+ k4_ss]+CONSTANTS[(offset * num_of_constants) + k5_ss])+ ALGEBRAIC[(offset * num_of_algebraic)
+ k3_ss]*CONSTANTS[(offset * num_of_constants) + k5_ss]*(CONSTANTS[(offset * num_of_constants)
+ k1_ss]+ALGEBRAIC[(offset * num_of_algebraic) + k8_ss]);
        ALGEBRAIC[(offset * num_of_algebraic) + E1_ss] = ALGEBRAIC[(offset * num_of_algebraic)
+ x1_ss]/(ALGEBRAIC[(offset * num_of_algebraic) + x1_ss]+ALGEBRAIC[(offset * num_of_algebraic)
+ x2_ss]+ALGEBRAIC[(offset * num_of_algebraic) + x3_ss]+ALGEBRAIC[(offset * num_of_algebraic) +
x4_ss]);
        ALGEBRAIC[(offset * num_of_algebraic) + E2_ss] = ALGEBRAIC[(offset * num_of_algebraic)
+ x2_ss]/(ALGEBRAIC[(offset * num_of_algebraic) + x1_ss]+ALGEBRAIC[(offset * num_of_algebraic)
+ x2_ss]+ALGEBRAIC[(offset * num_of_algebraic) + x3_ss]+ALGEBRAIC[(offset * num_of_algebraic) +
x4_ss]);
        ALGEBRAIC[(offset * num_of_algebraic) + E3_ss] = ALGEBRAIC[(offset * num_of_algebraic)
+ x3_ss]/(ALGEBRAIC[(offset * num_of_algebraic) + x1_ss]+ALGEBRAIC[(offset * num_of_algebraic)
+ x2_ss]+ALGEBRAIC[(offset * num_of_algebraic) + x3_ss]+ALGEBRAIC[(offset * num_of_algebraic) +
x4_ss]);
        ALGEBRAIC[(offset * num_of_algebraic) + E4_ss] = ALGEBRAIC[(offset * num_of_algebraic)
+ x4_ss]/(ALGEBRAIC[(offset * num_of_algebraic) + x1_ss]+ALGEBRAIC[(offset * num_of_algebraic)
+ x2_ss]+ALGEBRAIC[(offset * num_of_algebraic) + x3_ss]+ALGEBRAIC[(offset * num_of_algebraic) +
x4_ss]);
        ALGEBRAIC[(offset * num_of_algebraic) + JncxNa_ss] = ( 3.00000*( ALGEBRAIC[(offset *
num_of_algebraic) + E4_ss]*ALGEBRAIC[(offset * num_of_algebraic) + k7_ss] - ALGEBRAIC[(offset
* num_of_algebraic) + E1_ss]*ALGEBRAIC[(offset * num_of_algebraic) + k8_ss])+ ALGEBRAIC[(offset
* num_of_algebraic) + E3_ss]*ALGEBRAIC[(offset * num_of_algebraic) + k4pp_ss]) -
ALGEBRAIC[(offset * num_of_algebraic) + E2_ss]*ALGEBRAIC[(offset * num_of_algebraic) + k3pp_ss];
        ALGEBRAIC[(offset * num_of_algebraic) + JncxCa_ss] = ALGEBRAIC[(offset *
num_of_algebraic) + E2_ss]*CONSTANTS[(offset * num_of_constants) + k2_ss] - ALGEBRAIC[(offset
* num_of_algebraic) + E1_ss]*CONSTANTS[(offset * num_of_constants) + k1_ss];
        ALGEBRAIC[(offset * num_of_algebraic) + INaCa_ss] = 0.200000*CONSTANTS[(offset *
num_of_constants) + Gncx]*ALGEBRAIC[(offset * num_of_algebraic) + allo_ss]*( CONSTANTS[(offset
* num_of_constants) + zna]*ALGEBRAIC[(offset * num_of_algebraic) + JncxNa_ss]+ CONSTANTS[(offset
* num_of_constants) + zca]*ALGEBRAIC[(offset * num_of_algebraic) + JncxCa_ss]);
        ALGEBRAIC[(offset * num_of_algebraic) + IpCa] = ( CONSTANTS[(offset * num_of_constants)
+ GpCa]*STATES[(offset * num_of_states) + cai])/(CONSTANTS[(offset * num_of_constants) +
KmCap]+STATES[(offset * num_of_states) + cai]);
        ALGEBRAIC[(offset * num_of_algebraic) + ICab] = ( CONSTANTS[(offset * num_of_constants)
+ PCab]*4.00000*ALGEBRAIC[(offset * num_of_algebraic) + vffrt]*( STATES[(offset * num_of_states)
+ cai]*exp( 2.00000*ALGEBRAIC[(offset * num_of_algebraic) + vfrt]) - 0.341000*CONSTANTS[(offset
* num_of_constants) + cao]))/(exp( 2.00000*ALGEBRAIC[(offset * num_of_algebraic) + vfrt]) -
1.00000);
        ALGEBRAIC[(offset * num_of_algebraic) + Jdiff] = (STATES[(offset * num_of_states) +
cass] - STATES[(offset * num_of_states) + cai])/0.200000;
        ALGEBRAIC[(offset * num_of_algebraic) + fJrelp] = 1.00000/(1.00000+CONSTANTS[(offset *
num_of_constants) + KmCaMK]/ALGEBRAIC[(offset * num_of_algebraic) + CaMKa]);

        //cvar starts here
        ALGEBRAIC[(offset * num_of_algebraic) + Jrel] = CONSTANTS[(offset * num_of_constants)
+ Jrel_scale] * ( (1.00000 - ALGEBRAIC[(offset * num_of_algebraic) + fJrelp])*STATES[(offset *
num_of_states) + Jrelnp]+ ALGEBRAIC[(offset * num_of_algebraic) + fJrelp]*STATES[(offset *
num_of_states) + Jrelp]);
```

```
        ALGEBRAIC[(offset * num_of_algebraic) + Bcass] = 1.00000/(1.00000+( CONSTANTS[(offset
*     num_of_constants)     +     BSRmax]*CONSTANTS[(offset     *     num_of_constants)     +
KmBSR])/pow(CONSTANTS[(offset * num_of_constants) + KmBSR]+STATES[(offset * num_of_states) +
cass],  2.00000)+(  CONSTANTS[(offset  *  num_of_constants)  +  BSLmax]*CONSTANTS[(offset  *
num_of_constants) + KmBSL])/pow(CONSTANTS[(offset * num_of_constants) + KmBSL]+STATES[(offset *
num_of_states) + cass], 2.00000));
        ALGEBRAIC[(offset * num_of_algebraic) + Jupnp] = ( CONSTANTS[(offset * num_of_constants)
+ upScale]*0.00437500*STATES[(offset * num_of_states) + cai])/(STATES[(offset * num_of_states)
+ cai]+0.000920000);
        ALGEBRAIC[(offset * num_of_algebraic) + Jupp] = ( CONSTANTS[(offset * num_of_constants)
+  upScale]*2.75000*0.00437500*STATES[(offset  *  num_of_states)  +  cai])/((STATES[(offset  *
num_of_states) + cai]+0.000920000) - 0.000170000);
        ALGEBRAIC[(offset * num_of_algebraic) + fJupp] = 1.00000/(1.00000+CONSTANTS[(offset *
num_of_constants) + KmCaMK]/ALGEBRAIC[(offset * num_of_algebraic) + CaMKa]);

        ALGEBRAIC[(offset * num_of_algebraic) + Jleak] = CONSTANTS[(offset * num_of_constants)
+ Jleak_scale] * ( 0.00393750*STATES[(offset * num_of_states) + cansr])/15.0000;
        ALGEBRAIC[(offset * num_of_algebraic) + Jup] = CONSTANTS[(offset * num_of_constants) +
Jup_scale] * ( ( (1.00000 - ALGEBRAIC[(offset * num_of_algebraic) + fJupp])*ALGEBRAIC[(offset *
num_of_algebraic) + Jupnp]+ ALGEBRAIC[(offset * num_of_algebraic) + fJupp]*ALGEBRAIC[(offset *
num_of_algebraic) + Jupp]) - ALGEBRAIC[(offset * num_of_algebraic) + Jleak]);

        ALGEBRAIC[(offset * num_of_algebraic) + Bcai] = 1.00000/(1.00000+( CONSTANTS[(offset *
num_of_constants)      +      cmdnmax]*CONSTANTS[(offset      *      num_of_constants)      +
kmcmdn])/pow(CONSTANTS[(offset * num_of_constants) + kmcmdn]+STATES[(offset * num_of_states) +
cai],  2.00000)+(  CONSTANTS[(offset  *  num_of_constants)  +  trpnmax]*CONSTANTS[(offset  *
num_of_constants) + kmtrpn])/pow(CONSTANTS[(offset * num_of_constants) + kmtrpn]+STATES[(offset
* num_of_states) + cai], 2.00000));

        ALGEBRAIC[(offset * num_of_algebraic) + Jtr] = CONSTANTS[(offset * num_of_constants) +
Jtr_scale] * (STATES[(offset * num_of_states) + cansr] - STATES[(offset * num_of_states) +
cajsr])/100.000;
        //cvar ends here
        ALGEBRAIC[(offset * num_of_algebraic) + Bcajsr] = 1.00000/(1.00000+( CONSTANTS[(offset
*     num_of_constants)     +     csqnmax]*CONSTANTS[(offset     *     num_of_constants)     +
kmcsqn])/pow(CONSTANTS[(offset * num_of_constants) + kmcsqn]+STATES[(offset * num_of_states) +
cajsr], 2.00000));


        RATES[(offset * num_of_rates) + hL] = (ALGEBRAIC[(offset * num_of_algebraic) + hLss] -
STATES[(offset * num_of_states) + hL])/CONSTANTS[(offset * num_of_constants) + thL];
        RATES[(offset * num_of_rates) + hLp] = (ALGEBRAIC[(offset * num_of_algebraic) + hLssp]
- STATES[(offset * num_of_states) + hLp])/CONSTANTS[(offset * num_of_constants) + thLp];
        RATES[(offset * num_of_rates) + m] = (ALGEBRAIC[(offset * num_of_algebraic) + mss] -
STATES[(offset * num_of_states) + m])/ALGEBRAIC[(offset * num_of_algebraic) + tm];
        RATES[(offset * num_of_rates) + hf] = (ALGEBRAIC[(offset * num_of_algebraic) + hss] -
STATES[(offset * num_of_states) + hf])/ALGEBRAIC[(offset * num_of_algebraic) + thf];
```

```
        RATES[(offset * num_of_rates) + hs] = (ALGEBRAIC[(offset * num_of_algebraic) + hss] -
STATES[(offset * num_of_states) + hs])/ALGEBRAIC[(offset * num_of_algebraic) + ths];
        RATES[(offset * num_of_rates) + a] = (ALGEBRAIC[(offset * num_of_algebraic) + ass] -
STATES[(offset * num_of_states) + a])/ALGEBRAIC[(offset * num_of_algebraic) + ta];
        RATES[(offset * num_of_rates) + d] = (ALGEBRAIC[(offset * num_of_algebraic) + dss] -
STATES[(offset * num_of_states) + d])/ALGEBRAIC[(offset * num_of_algebraic) + td];
        RATES[(offset * num_of_rates) + ff] = (ALGEBRAIC[(offset * num_of_algebraic) + fss] -
STATES[(offset * num_of_states) + ff])/ALGEBRAIC[(offset * num_of_algebraic) + tff];
        RATES[(offset * num_of_rates) + fs] = (ALGEBRAIC[(offset * num_of_algebraic) + fss] -
STATES[(offset * num_of_states) + fs])/ALGEBRAIC[(offset * num_of_algebraic) + tfs];
        RATES[(offset * num_of_rates) + jca] = (ALGEBRAIC[(offset * num_of_algebraic) + fcass]
- STATES[(offset * num_of_states) + jca])/CONSTANTS[(offset * num_of_constants) + tjca];
        RATES[(offset * num_of_rates) + nca] =  ALGEBRAIC[(offset * num_of_algebraic) +
anca]*CONSTANTS[(offset * num_of_constants) + k2n] - STATES[(offset * num_of_states) +
nca]*ALGEBRAIC[(offset * num_of_algebraic) + km2n];
        RATES[(offset * num_of_rates) + xrf] = (ALGEBRAIC[(offset * num_of_algebraic) + xrss]
- STATES[(offset * num_of_states) + xrf])/ALGEBRAIC[(offset * num_of_algebraic) + txrf];
        RATES[(offset * num_of_rates) + xrs] = (ALGEBRAIC[(offset * num_of_algebraic) + xrss]
- STATES[(offset * num_of_states) + xrs])/ALGEBRAIC[(offset * num_of_algebraic) + txrs];
        RATES[(offset * num_of_rates) + xs1] = (ALGEBRAIC[(offset * num_of_algebraic) + xs1ss]
- STATES[(offset * num_of_states) + xs1])/ALGEBRAIC[(offset * num_of_algebraic) + txs1];
        RATES[(offset * num_of_rates) + xk1] = (ALGEBRAIC[(offset * num_of_algebraic) + xk1ss]
- STATES[(offset * num_of_states) + xk1])/ALGEBRAIC[(offset * num_of_algebraic) + txk1];
        RATES[(offset * num_of_rates) + j] = (ALGEBRAIC[(offset * num_of_algebraic) + jss] -
STATES[(offset * num_of_states) + j])/ALGEBRAIC[(offset * num_of_algebraic) + tj];
        RATES[(offset * num_of_rates) + ap] = (ALGEBRAIC[(offset * num_of_algebraic) + assp] -
STATES[(offset * num_of_states) + ap])/ALGEBRAIC[(offset * num_of_algebraic) + ta];
        RATES[(offset * num_of_rates) + fcaf] = (ALGEBRAIC[(offset * num_of_algebraic) + fcass]
- STATES[(offset * num_of_states) + fcaf])/ALGEBRAIC[(offset * num_of_algebraic) + tfcaf];
        RATES[(offset * num_of_rates) + fcas] = (ALGEBRAIC[(offset * num_of_algebraic) + fcass]
- STATES[(offset * num_of_states) + fcas])/ALGEBRAIC[(offset * num_of_algebraic) + tfcas];
        RATES[(offset * num_of_rates) + ffp] = (ALGEBRAIC[(offset * num_of_algebraic) + fss] -
STATES[(offset * num_of_states) + ffp])/ALGEBRAIC[(offset * num_of_algebraic) + tffp];
        RATES[(offset * num_of_rates) + xs2] = (ALGEBRAIC[(offset * num_of_algebraic) + xs2ss]
- STATES[(offset * num_of_states) + xs2])/ALGEBRAIC[(offset * num_of_algebraic) + txs2];
        RATES[(offset * num_of_rates) + CaMKt] =  CONSTANTS[(offset * num_of_constants) +
aCaMK]*ALGEBRAIC[(offset * num_of_algebraic) + CaMKb]*(ALGEBRAIC[(offset * num_of_algebraic) +
CaMKb]+STATES[(offset * num_of_states) + CaMKt]) -  CONSTANTS[(offset * num_of_constants) +
bCaMK]*STATES[(offset * num_of_states) + CaMKt];
        RATES[(offset * num_of_rates) + hsp] = (ALGEBRAIC[(offset * num_of_algebraic) + hssp]
- STATES[(offset * num_of_states) + hsp])/ALGEBRAIC[(offset * num_of_algebraic) + thsp];
        RATES[(offset * num_of_rates) + jp] = (ALGEBRAIC[(offset * num_of_algebraic) + jss] -
STATES[(offset * num_of_states) + jp])/ALGEBRAIC[(offset * num_of_algebraic) + tjp];
        RATES[(offset * num_of_rates) + mL] = (ALGEBRAIC[(offset * num_of_algebraic) + mLss] -
STATES[(offset * num_of_states) + mL])/ALGEBRAIC[(offset * num_of_algebraic) + tmL];
        RATES[(offset * num_of_rates) + fcafp] = (ALGEBRAIC[(offset * num_of_algebraic) + fcass]
- STATES[(offset * num_of_states) + fcafp])/ALGEBRAIC[(offset * num_of_algebraic) + tfcafp];
        RATES[(offset * num_of_rates) + iF] = (ALGEBRAIC[(offset * num_of_algebraic) + iss] -
STATES[(offset * num_of_states) + iF])/ALGEBRAIC[(offset * num_of_algebraic) + tiF];
```

```
        RATES[(offset * num_of_rates) + iS] = (ALGEBRAIC[(offset * num_of_algebraic) + iss] -
STATES[(offset * num_of_states) + iS])/ALGEBRAIC[(offset * num_of_algebraic) + tiS];
        RATES[(offset * num_of_rates) + iFp] = (ALGEBRAIC[(offset * num_of_algebraic) + iss] -
STATES[(offset * num_of_states) + iFp])/ALGEBRAIC[(offset * num_of_algebraic) + tiFp];
        RATES[(offset * num_of_rates) + iSp] = (ALGEBRAIC[(offset * num_of_algebraic) + iss] -
STATES[(offset * num_of_states) + iSp])/ALGEBRAIC[(offset * num_of_algebraic) + tiSp];
        RATES[(offset * num_of_rates) + Jrelnp] = (ALGEBRAIC[(offset * num_of_algebraic) +
Jrel_inf] - STATES[(offset * num_of_states) + Jrelnp])/ALGEBRAIC[(offset * num_of_algebraic) +
tau_rel];
        RATES[(offset * num_of_rates) + Jrelp] = (ALGEBRAIC[(offset * num_of_algebraic) +
Jrel_infp] - STATES[(offset * num_of_states) + Jrelp])/ALGEBRAIC[(offset * num_of_algebraic) +
tau_relp];
        RATES[(offset * num_of_rates) + ki] = ( - ((ALGEBRAIC[(offset * num_of_algebraic) +
Ito]+ALGEBRAIC[(offset * num_of_algebraic) + IKr]+ALGEBRAIC[(offset * num_of_algebraic) +
IKs]+ALGEBRAIC[(offset * num_of_algebraic) + IK1]+ALGEBRAIC[(offset * num_of_algebraic) +
IKb]+ALGEBRAIC[(offset * num_of_algebraic) + Istim]) - 2.00000*ALGEBRAIC[(offset *
num_of_algebraic) + INaK])*CONSTANTS[(offset * num_of_constants) + cm]*CONSTANTS[(offset *
num_of_constants) + Acap])/( CONSTANTS[(offset * num_of_constants) + F]*CONSTANTS[(offset *
num_of_constants) + vmyo])+( ALGEBRAIC[(offset * num_of_algebraic) + JdiffK]*CONSTANTS[(offset
* num_of_constants) + vss])/CONSTANTS[(offset * num_of_constants) + vmyo];
        RATES[(offset * num_of_rates) + kss] = ( - ALGEBRAIC[(offset * num_of_algebraic) +
ICaK]*CONSTANTS[(offset * num_of_constants) + cm]*CONSTANTS[(offset * num_of_constants) +
Acap])/( CONSTANTS[(offset * num_of_constants) + F]*CONSTANTS[(offset * num_of_constants) + vss])
- ALGEBRAIC[(offset * num_of_algebraic) + JdiffK];
        RATES[(offset * num_of_rates) + nai] = ( - (ALGEBRAIC[(offset * num_of_algebraic) +
INa]+ALGEBRAIC[(offset * num_of_algebraic) + INaL]+ 3.00000*ALGEBRAIC[(offset * num_of_algebraic)
+ INaCa_i]+ 3.00000*ALGEBRAIC[(offset * num_of_algebraic) + INaK]+ALGEBRAIC[(offset *
num_of_algebraic) + INab])*CONSTANTS[(offset * num_of_constants) + Acap]*CONSTANTS[(offset *
num_of_constants) + cm])/( CONSTANTS[(offset * num_of_constants) + F]*CONSTANTS[(offset *
num_of_constants) + vmyo])+( ALGEBRAIC[(offset * num_of_algebraic) + JdiffNa]*CONSTANTS[(offset
* num_of_constants) + vss])/CONSTANTS[(offset * num_of_constants) + vmyo];
        RATES[(offset * num_of_rates) + nass] = ( - (ALGEBRAIC[(offset * num_of_algebraic) +
ICaNa]+ 3.00000*ALGEBRAIC[(offset * num_of_algebraic) + INaCa_ss])*CONSTANTS[(offset *
num_of_constants) + cm]*CONSTANTS[(offset * num_of_constants) + Acap])/( CONSTANTS[(offset *
num_of_constants) + F]*CONSTANTS[(offset * num_of_constants) + vss]) - ALGEBRAIC[(offset *
num_of_algebraic) + JdiffNa];
        RATES[(offset * num_of_rates) + V] = - (ALGEBRAIC[(offset * num_of_algebraic) +
INa]+ALGEBRAIC[(offset * num_of_algebraic) + INaL]+ALGEBRAIC[(offset * num_of_algebraic) +
Ito]+ALGEBRAIC[(offset * num_of_algebraic) + ICaL]+ALGEBRAIC[(offset * num_of_algebraic) +
ICaNa]+ALGEBRAIC[(offset * num_of_algebraic) + ICaK]+ALGEBRAIC[(offset * num_of_algebraic) +
IKr]+ALGEBRAIC[(offset * num_of_algebraic) + IKs]+ALGEBRAIC[(offset * num_of_algebraic) +
IK1]+ALGEBRAIC[(offset * num_of_algebraic) + INaCa_i]+ALGEBRAIC[(offset * num_of_algebraic) +
INaCa_ss]+ALGEBRAIC[(offset * num_of_algebraic) + INaK]+ALGEBRAIC[(offset * num_of_algebraic) +
INab]+ALGEBRAIC[(offset * num_of_algebraic) + IKb]+ALGEBRAIC[(offset * num_of_algebraic) +
IpCa]+ALGEBRAIC[(offset * num_of_algebraic) + ICab]+ALGEBRAIC[(offset * num_of_algebraic) +
Istim]);
        RATES[(offset * num_of_rates) + cass] = ALGEBRAIC[(offset * num_of_algebraic) +
Bcass]*((( - (ALGEBRAIC[(offset * num_of_algebraic) + ICaL] - 2.00000*ALGEBRAIC[(offset *
num_of_algebraic) + INaCa_ss])*CONSTANTS[(offset * num_of_constants) + cm]*CONSTANTS[(offset *
```

```
num_of_constants)    +    Acap])/(    2.00000*CONSTANTS[(offset    *    num_of_constants)    +
F]*CONSTANTS[(offset  *  num_of_constants)  +  vss])+(  ALGEBRAIC[(offset  *  num_of_algebraic)  +
Jrel]*CONSTANTS[(offset  *  num_of_constants)  +  vjsr])/CONSTANTS[(offset  *  num_of_constants)  +
vss]) - ALGEBRAIC[(offset * num_of_algebraic) + Jdiff]);
        RATES[(offset  *  num_of_rates)  +  cai]  =   ALGEBRAIC[(offset  *  num_of_algebraic)  +
Bcai]*((( - ((ALGEBRAIC[(offset * num_of_algebraic) + IpCa]+ALGEBRAIC[(offset * num_of_algebraic)
+  ICab])  -   2.00000*ALGEBRAIC[(offset  *  num_of_algebraic)  +  INaCa_i])*CONSTANTS[(offset  *
num_of_constants)       +       cm]*CONSTANTS[(offset       *       num_of_constants)       +
Acap])/( 2.00000*CONSTANTS[(offset * num_of_constants) + F]*CONSTANTS[(offset * num_of_constants)
+ vmyo]) - ( ALGEBRAIC[(offset * num_of_algebraic) + Jup]*CONSTANTS[(offset * num_of_constants)
+ vnsr])/CONSTANTS[(offset * num_of_constants) + vmyo])+( ALGEBRAIC[(offset * num_of_algebraic)
+ Jdiff]*CONSTANTS[(offset * num_of_constants) + vss])/CONSTANTS[(offset * num_of_constants) +
vmyo]);
        RATES[(offset * num_of_rates) + cansr] = ALGEBRAIC[(offset * num_of_algebraic) + Jup]
-  (  ALGEBRAIC[(offset  *  num_of_algebraic)  +  Jtr]*CONSTANTS[(offset  *  num_of_constants)  +
vjsr])/CONSTANTS[(offset * num_of_constants) + vnsr];
        RATES[(offset  *  num_of_rates)  +  cajsr]  =   ALGEBRAIC[(offset  *  num_of_algebraic)  +
Bcajsr]*(ALGEBRAIC[(offset * num_of_algebraic) + Jtr] - ALGEBRAIC[(offset * num_of_algebraic) +
Jrel]);
        }

        __device__ void solveEuler( double *STATES, double *RATES, double dt, int offset){

          int num_of_states = 41;
          int num_of_rates = 41;

          STATES[(offset  *  num_of_states)  +V]  =  STATES[(offset  *  num_of_states)  +  V]  +
RATES[(offset * num_of_rates) + V] * dt;
          STATES[(offset * num_of_states) + CaMKt] = STATES[(offset * num_of_states) + CaMKt]
+ RATES[(offset * num_of_rates) + CaMKt] * dt;
          STATES[(offset * num_of_states) + cass] = STATES[(offset * num_of_states) + cass] +
RATES[(offset * num_of_rates) + cass] * dt;
          STATES[(offset * num_of_states) + nai] = STATES[(offset * num_of_states) + nai] +
RATES[(offset * num_of_rates) + nai] * dt;
          STATES[(offset * num_of_states) + nass] = STATES[(offset * num_of_states) + nass] +
RATES[(offset * num_of_rates) + nass] * dt;
          STATES[(offset  *  num_of_states)  +  ki]  =  STATES[(offset  *  num_of_states)  +  ki]  +
RATES[(offset * num_of_rates) + ki] * dt;
          STATES[(offset * num_of_states) + kss] = STATES[(offset * num_of_states) + kss] +
RATES[(offset * num_of_rates) + kss] * dt;
          STATES[(offset * num_of_states) + cansr] = STATES[(offset * num_of_states) + cansr]
+ RATES[(offset * num_of_rates) + cansr] * dt;
          STATES[(offset * num_of_states) + cajsr] = STATES[(offset * num_of_states) + cajsr]
+ RATES[(offset * num_of_rates) + cajsr] * dt;
          STATES[(offset * num_of_states) + cai] = STATES[(offset * num_of_states) + cai] +
RATES[(offset * num_of_rates) + cai] * dt;
          STATES[(offset  *  num_of_states)  +  m]  =  STATES[(offset  *  num_of_states)  +  m]  +
RATES[(offset * num_of_rates) + m] * dt;
```

```
          STATES[(offset * num_of_states) + hf] = STATES[(offset * num_of_states) + hf] +
RATES[(offset * num_of_rates) + hf] * dt;
          STATES[(offset * num_of_states) + hs] = STATES[(offset * num_of_states) + hs] +
RATES[(offset * num_of_rates) + hs] * dt;
          STATES[(offset * num_of_states) + j] = STATES[(offset * num_of_states) + j] +
RATES[(offset * num_of_rates) + j] * dt;
          STATES[(offset * num_of_states) + hsp] = STATES[(offset * num_of_states) + hsp] +
RATES[(offset * num_of_rates) + hsp] * dt;
          STATES[(offset * num_of_states) + jp] = STATES[(offset * num_of_states) + jp] +
RATES[(offset * num_of_rates) + jp] * dt;
          STATES[(offset * num_of_states) + mL] = STATES[(offset * num_of_states) + mL] +
RATES[(offset * num_of_rates) + mL] * dt;
          STATES[(offset * num_of_states) + hL] = STATES[(offset * num_of_states) + hL] +
RATES[(offset * num_of_rates) + hL] * dt;
          STATES[(offset * num_of_states) + hLp] = STATES[(offset * num_of_states) + hLp] +
RATES[(offset * num_of_rates) + hLp] * dt;
          STATES[(offset * num_of_states) + a] = STATES[(offset * num_of_states) + a] +
RATES[(offset * num_of_rates) + a] * dt;
          STATES[(offset * num_of_states) + iF] = STATES[(offset * num_of_states) + iF] +
RATES[(offset * num_of_rates) + iF] * dt;
          STATES[(offset * num_of_states) + iS] = STATES[(offset * num_of_states) + iS] +
RATES[(offset * num_of_rates) + iS] * dt;
          STATES[(offset * num_of_states) + ap] = STATES[(offset * num_of_states) + ap] +
RATES[(offset * num_of_rates) + ap] * dt;
          STATES[(offset * num_of_states) + iFp] = STATES[(offset * num_of_states) + iFp] +
RATES[(offset * num_of_rates) + iFp] * dt;
          STATES[(offset * num_of_states) + iSp] = STATES[(offset * num_of_states) + iSp] +
RATES[(offset * num_of_rates) + iSp] * dt;
          STATES[(offset * num_of_states) + d] = STATES[(offset * num_of_states) + d] +
RATES[(offset * num_of_rates) + d] * dt;
          STATES[(offset * num_of_states) + ff] = STATES[(offset * num_of_states) + ff] +
RATES[(offset * num_of_rates) + ff] * dt;
          STATES[(offset * num_of_states) + fs] = STATES[(offset * num_of_states) + fs] +
RATES[(offset * num_of_rates) + fs] * dt;
          STATES[(offset * num_of_states) + fcaf] = STATES[(offset * num_of_states) + fcaf] +
RATES[(offset * num_of_rates) + fcaf] * dt;
          STATES[(offset * num_of_states) + fcas] = STATES[(offset * num_of_states) + fcas] +
RATES[(offset * num_of_rates) + fcas] * dt;
          STATES[(offset * num_of_states) + jca] = STATES[(offset * num_of_states) + jca] +
RATES[(offset * num_of_rates) + jca] * dt;
          STATES[(offset * num_of_states) + ffp] = STATES[(offset * num_of_states) + ffp] +
RATES[(offset * num_of_rates) + ffp] * dt;
          STATES[(offset * num_of_states) + fcafp] = STATES[(offset * num_of_states) + fcafp]
+ RATES[(offset * num_of_rates) + fcafp] * dt;
          STATES[(offset * num_of_states) + nca] = STATES[(offset * num_of_states) + nca] +
RATES[(offset * num_of_rates) + nca] * dt;
          STATES[(offset * num_of_states) + xrf] = STATES[(offset * num_of_states) + xrf] +
RATES[(offset * num_of_rates) + xrf] * dt;
```

```
          STATES[(offset * num_of_states) + xrs] = STATES[(offset * num_of_states) + xrs] +
RATES[(offset * num_of_rates) + xrs] * dt;
          STATES[(offset * num_of_states) + xs1] = STATES[(offset * num_of_states) + xs1] +
RATES[(offset * num_of_rates) + xs1] * dt;
          STATES[(offset * num_of_states) + xs2] = STATES[(offset * num_of_states) + xs2] +
RATES[(offset * num_of_rates) + xs2] * dt;
          STATES[(offset * num_of_states) + xk1] = STATES[(offset * num_of_states) + xk1] +
RATES[(offset * num_of_rates) + xk1] * dt;
          STATES[(offset * num_of_states) + Jrelnp] = STATES[(offset * num_of_states) + Jrelnp]
+ RATES[(offset * num_of_rates) + Jrelnp] * dt;
          STATES[(offset * num_of_states) + Jrelp] = STATES[(offset * num_of_states) + Jrelp]
+ RATES[(offset * num_of_rates) + Jrelp] * dt;
        }

        __device__ void solveAnalytical(double *CONSTANTS, double *STATES, double *ALGEBRAIC,
double *RATES, double dt, int offset)
        {
          int num_of_constants = 145;
          int num_of_states = 41;
          int num_of_algebraic = 199;
          int num_of_rates = 41;

        // #ifdef EULER // moved as its own function

        // #else
        ////==============
          ////Exact solution
          ////==============
          ////INa
          STATES[(offset * num_of_states) + m] = ALGEBRAIC[(offset * num_of_algebraic) + mss]
- (ALGEBRAIC[(offset * num_of_algebraic) + mss] - STATES[(offset * num_of_states) + m]) * exp(-
dt / ALGEBRAIC[(offset * num_of_algebraic) + tm]);
          STATES[(offset * num_of_states) + hf] = ALGEBRAIC[(offset * num_of_algebraic) + hss]
- (ALGEBRAIC[(offset * num_of_algebraic) + hss] - STATES[(offset * num_of_states) + hf]) * exp(-
dt / ALGEBRAIC[(offset * num_of_algebraic) + thf]);
          STATES[(offset * num_of_states) + hs] = ALGEBRAIC[(offset * num_of_algebraic) + hss]
- (ALGEBRAIC[(offset * num_of_algebraic) + hss] - STATES[(offset * num_of_states) + hs]) * exp(-
dt / ALGEBRAIC[(offset * num_of_algebraic) + ths]);
          STATES[(offset * num_of_states) + j] = ALGEBRAIC[(offset * num_of_algebraic) + jss]
- (ALGEBRAIC[(offset * num_of_algebraic) + jss] - STATES[(offset * num_of_states) + j]) * exp(-
dt / ALGEBRAIC[(offset * num_of_algebraic) + tj]);
          STATES[(offset * num_of_states) + hsp] = ALGEBRAIC[(offset * num_of_algebraic) + hssp]
- (ALGEBRAIC[(offset * num_of_algebraic) + hssp] - STATES[(offset * num_of_states) + hsp]) *
exp(-dt / ALGEBRAIC[(offset * num_of_algebraic) + thsp]);
          STATES[(offset * num_of_states) + jp] = ALGEBRAIC[(offset * num_of_algebraic) + jss]
- (ALGEBRAIC[(offset * num_of_algebraic) + jss] - STATES[(offset * num_of_states) + jp]) * exp(-
dt / ALGEBRAIC[(offset * num_of_algebraic) + tjp]);
```

```
        STATES[(offset * num_of_states) + mL] = ALGEBRAIC[(offset * num_of_algebraic) + mLss]
- (ALGEBRAIC[(offset * num_of_algebraic) + mLss] - STATES[(offset * num_of_states) + mL]) *
exp(-dt / ALGEBRAIC[(offset * num_of_algebraic) + tmL]);
        STATES[(offset * num_of_states) + hL] = ALGEBRAIC[(offset * num_of_algebraic) + hLss]
- (ALGEBRAIC[(offset * num_of_algebraic) + hLss] - STATES[(offset * num_of_states) + hL]) *
exp(-dt / CONSTANTS[(offset * num_of_constants) + thL]);
        STATES[(offset * num_of_states) + hLp] = ALGEBRAIC[(offset * num_of_algebraic) + hLssp]
- (ALGEBRAIC[(offset * num_of_algebraic) + hLssp] - STATES[(offset * num_of_states) + hLp]) *
exp(-dt / CONSTANTS[(offset * num_of_constants) + thLp]);
        ////Ito
        STATES[(offset * num_of_states) + a] = ALGEBRAIC[(offset * num_of_algebraic) + ass]
- (ALGEBRAIC[(offset * num_of_algebraic) + ass] - STATES[(offset * num_of_states) + a]) * exp(-
dt / ALGEBRAIC[(offset * num_of_algebraic) + ta]);
        STATES[(offset * num_of_states) + iF] = ALGEBRAIC[(offset * num_of_algebraic) + iss]
- (ALGEBRAIC[(offset * num_of_algebraic) + iss] - STATES[(offset * num_of_states) + iF]) * exp(-
dt / ALGEBRAIC[(offset * num_of_algebraic) + tiF]);
        STATES[(offset * num_of_states) + iS] = ALGEBRAIC[(offset * num_of_algebraic) + iss]
- (ALGEBRAIC[(offset * num_of_algebraic) + iss] - STATES[(offset * num_of_states) + iS]) * exp(-
dt / ALGEBRAIC[(offset * num_of_algebraic) + tiS]);
        STATES[(offset * num_of_states) + ap] = ALGEBRAIC[(offset * num_of_algebraic) + assp]
- (ALGEBRAIC[(offset * num_of_algebraic) + assp] - STATES[(offset * num_of_states) + ap]) *
exp(-dt / ALGEBRAIC[(offset * num_of_algebraic) + ta]);
        STATES[(offset * num_of_states) + iFp] = ALGEBRAIC[(offset * num_of_algebraic) + iss]
- (ALGEBRAIC[(offset * num_of_algebraic) + iss] - STATES[(offset * num_of_states) + iFp]) *
exp(-dt / ALGEBRAIC[(offset * num_of_algebraic) + tiFp]);
        STATES[(offset * num_of_states) + iSp] = ALGEBRAIC[(offset * num_of_algebraic) + iss]
- (ALGEBRAIC[(offset * num_of_algebraic) + iss] - STATES[(offset * num_of_states) + iSp]) *
exp(-dt / ALGEBRAIC[(offset * num_of_algebraic) + tiSp]);
        ////ICaL
        STATES[(offset * num_of_states) + d] = ALGEBRAIC[(offset * num_of_algebraic) + dss]
- (ALGEBRAIC[(offset * num_of_algebraic) + dss] - STATES[(offset * num_of_states) + d]) * exp(-
dt / ALGEBRAIC[(offset * num_of_algebraic) + td]);
        STATES[(offset * num_of_states) + ff] = ALGEBRAIC[(offset * num_of_algebraic) + fss]
- (ALGEBRAIC[(offset * num_of_algebraic) + fss] - STATES[(offset * num_of_states) + ff]) * exp(-
dt / ALGEBRAIC[(offset * num_of_algebraic) + tff]);
        STATES[(offset * num_of_states) + fs] = ALGEBRAIC[(offset * num_of_algebraic) + fss]
- (ALGEBRAIC[(offset * num_of_algebraic) + fss] - STATES[(offset * num_of_states) + fs]) * exp(-
dt / ALGEBRAIC[(offset * num_of_algebraic) + tfs]);
        STATES[(offset * num_of_states) + fcaf] = ALGEBRAIC[(offset * num_of_algebraic) +
fcass] - (ALGEBRAIC[(offset * num_of_algebraic) + fcass] - STATES[(offset * num_of_states) +
fcaf]) * exp(-dt / ALGEBRAIC[(offset * num_of_algebraic) + tfcaf]);
        STATES[(offset * num_of_states) + fcas] = ALGEBRAIC[(offset * num_of_algebraic) +
fcass] - (ALGEBRAIC[(offset * num_of_algebraic) + fcass] - STATES[(offset * num_of_states) +
fcas]) * exp(-dt / ALGEBRAIC[(offset * num_of_algebraic) + tfcas]);
        STATES[(offset * num_of_states) + jca] = ALGEBRAIC[(offset * num_of_algebraic) + fcass]
- (ALGEBRAIC[(offset * num_of_algebraic) + fcass] - STATES[(offset * num_of_states) + jca]) *
exp(- dt / CONSTANTS[(offset * num_of_constants) + tjca]);
```

```
        STATES[(offset * num_of_states) + ffp] = ALGEBRAIC[(offset * num_of_algebraic) + fss]
- (ALGEBRAIC[(offset * num_of_algebraic) + fss] - STATES[(offset * num_of_states) + ffp]) *
exp(-dt / ALGEBRAIC[(offset * num_of_algebraic) + tffp]);
        STATES[(offset * num_of_states) + fcafp] = ALGEBRAIC[(offset * num_of_algebraic) +
fcass] - (ALGEBRAIC[(offset * num_of_algebraic) + fcass] - STATES[(offset * num_of_states) +
fcafp]) * exp(-d / ALGEBRAIC[(offset * num_of_algebraic) + tfcafp]);
        STATES[(offset * num_of_states) + nca] = ALGEBRAIC[(offset * num_of_algebraic) + anca]
* CONSTANTS[(offset * num_of_constants) + k2n] / ALGEBRAIC[(offset * num_of_algebraic) + km2n]
- (ALGEBRAIC[(offset * num_of_algebraic) + anca] * CONSTANTS[(offset * num_of_constants) + k2n]
/ ALGEBRAIC[(offset * num_of_algebraic) + km2n] - STATES[(offset * num_of_states) + nca]) *
exp(-ALGEBRAIC[(offset * num_of_algebraic) + km2n] * dt);
        ////IKr
        STATES[(offset * num_of_states) + xrf] = ALGEBRAIC[(offset * num_of_algebraic) + xrss]
- (ALGEBRAIC[(offset * num_of_algebraic) + xrss] - STATES[(offset * num_of_states) + xrf]) *
exp(-dt / ALGEBRAIC[(offset * num_of_algebraic) + txrf]);
        STATES[(offset * num_of_states) + xrs] = ALGEBRAIC[(offset * num_of_algebraic) + xrss]
- (ALGEBRAIC[(offset * num_of_algebraic) + xrss] - STATES[(offset * num_of_states) + xrs]) *
exp(-dt / ALGEBRAIC[(offset * num_of_algebraic) + txrs]);
        ////IKs
        STATES[(offset * num_of_states) + xs1] = ALGEBRAIC[(offset * num_of_algebraic) + xs1ss]
- (ALGEBRAIC[(offset * num_of_algebraic) + xs1ss] - STATES[(offset * num_of_states) + xs1]) *
exp(-dt / ALGEBRAIC[(offset * num_of_algebraic) + txs1]);
        STATES[(offset * num_of_states) + xs2] = ALGEBRAIC[(offset * num_of_algebraic) + xs2ss]
- (ALGEBRAIC[(offset * num_of_algebraic) + xs2ss] - STATES[(offset * num_of_states) + xs2]) *
exp(-dt / ALGEBRAIC[(offset * num_of_algebraic) + txs2]);
        ////IK1
        STATES[(offset * num_of_states) + xk1] = ALGEBRAIC[(offset * num_of_algebraic) + xk1ss]
- (ALGEBRAIC[(offset * num_of_algebraic) + xk1ss] - STATES[(offset * num_of_states) + xk1]) *
exp(-dt / ALGEBRAIC[(offset * num_of_algebraic) + txk1]);
        ////INaCa
        ////INaK
        ////IKb
        ////INab
        ////ICab
        ///IpCa
        ////Diffusion fluxes
        ////RyR receptors
        STATES[(offset * num_of_states) + Jrelnp] = ALGEBRAIC[(offset * num_of_algebraic) +
Jrel_inf] - (ALGEBRAIC[(offset * num_of_algebraic) + Jrel_inf] - STATES[(offset * num_of_states)
+ Jrelnp]) * exp(-dt / ALGEBRAIC[(offset * num_of_algebraic) + tau_rel]);
        STATES[(offset * num_of_states) + Jrelp] = ALGEBRAIC[(offset * num_of_algebraic) +
Jrel_infp] - (ALGEBRAIC[(offset * num_of_algebraic) + Jrel_infp] - STATES[(offset * num_of_states)
+ Jrelp]) * exp(-dt / ALGEBRAIC[(offset * num_of_algebraic) + tau_relp]);
        ////SERCA Pump
        ////Calcium translocation
        //
        ////============================
        ////Approximated solution (Euler)
        ////============================
```

```
          ////ICaL
          //STATES[jca] = STATES[jca] + RATES[jca] * dt;
          ////CaMK
          STATES[(offset * num_of_states) + CaMKt] = STATES[(offset * num_of_states) + CaMKt]
+ RATES[(offset * num_of_rates) + CaMKt] * dt;
          ////Membrane potential
          STATES[(offset * num_of_states) + V] = STATES[(offset * num_of_states) + V] +
RATES[(offset * num_of_rates) + V] * dt;
          ////Ion Concentrations and Buffers
          STATES[(offset * num_of_states) + nai] = STATES[(offset * num_of_states) + nai] +
RATES[(offset * num_of_rates) + nai] * dt;
          STATES[(offset * num_of_states) + nass] = STATES[(offset * num_of_states) + nass] +
RATES[(offset * num_of_rates) + nass] * dt;
          STATES[(offset * num_of_states) + ki] = STATES[(offset * num_of_states) + ki] +
RATES[(offset * num_of_rates) + ki] * dt;
          STATES[(offset * num_of_states) + kss] = STATES[(offset * num_of_states) + kss] +
RATES[(offset * num_of_rates) + kss] * dt;
          STATES[(offset * num_of_states) + cai] = STATES[(offset * num_of_states) + cai] +
RATES[(offset * num_of_rates) + cai] * dt;
          STATES[(offset * num_of_states) + cass] = STATES[(offset * num_of_states) + cass] +
RATES[(offset * num_of_rates) + cass] * dt;
          STATES[(offset * num_of_states) + cansr] = STATES[(offset * num_of_states) + cansr]
+ RATES[(offset * num_of_rates) + cansr] * dt;
          STATES[(offset * num_of_states) + cajsr] = STATES[(offset * num_of_states) + cajsr]
+ RATES[(offset * num_of_rates) + cajsr] * dt;
        // #endif
        }

        __device__ double set_time_step(double TIME,
          double time_point,
          double max_time_step,
          double *CONSTANTS,
          double *RATES,
          double *STATES,
          double *ALGEBRAIC,
          int offset) {
          double time_step = 0.005;
          int num_of_constants = 145;
          int num_of_rates = 41;

          if (TIME <= time_point || (TIME - floor(TIME / CONSTANTS[BCL + (offset *
num_of_constants)]) * CONSTANTS[BCL + (offset * num_of_constants)]) <= time_point) {
            //printf("TIME <= time_point ms\n");
            return time_step;
            //printf("dV = %lf, time_step = %lf\n",RATES[V] * time_step, time_step);
          }
          else {
            //printf("TIME > time_point ms\n");
```

```
            if (std::abs(RATES[V + (offset * num_of_rates)] * time_step) <= 0.2) {//Slow changes
in V
                // printf("dV/dt <= 0.2\n");
                time_step = std::abs(0.8 / RATES[V + (offset * num_of_rates)]);
                //Make sure time_step is between 0.005 and max_time_step
                if (time_step < 0.005) {
                    time_step = 0.005;
                }
                else if (time_step > max_time_step) {
                    time_step = max_time_step;
                }
                //printf("dV = %lf, time_step = %lf\n",std::abs(RATES[V] * time_step),
time_step);
            }
            else if (std::abs(RATES[V + (offset * num_of_rates)] * time_step) >= 0.8) {//Fast
changes in V
                // printf("dV/dt >= 0.8\n");
                time_step = std::abs(0.2 / RATES[V + (offset * num_of_rates)]);
                while (std::abs(RATES[V + (offset * num_of_rates)] * time_step) >= 0.8 &&
                       0.005 < time_step &&
                       time_step < max_time_step) {
                    time_step = time_step / 10.0;
                    // printf("dV = %lf, time_step = %lf\n",std::abs(RATES[V] * time_step),
time_step);
                }
            }
            // __syncthreads();
            return time_step;
          }
        }
```

c. Cellmodel.hpp

This file contains the general interface for the cell models. It declares common functions that all cellmodel should have. Cellmodel.hpp implemented as below:

```
        #ifndef CELL_HPP
        #define CELL_HPP

        class Cellmodel
        {
        protected:
          Cellmodel(){}
        public:
          unsigned short algebraic_size;
          unsigned short constants_size;
          unsigned short states_size;
          unsigned short gates_size;
```

```
        unsigned short current_size;
        unsigned short concs_size;
        double ALGEBRAIC[255];
        double CONSTANTS[255];
        double RATES[255];
        double STATES[255];
        char gates_header[255];
        unsigned short gates_indices[255];
        char current_header[255];
        unsigned short current_indices[255];
        char concs_header[255];
        unsigned short concs_indices[255];
        virtual ~Cellmodel() {}
        virtual void initConsts() = 0;
        virtual void initConsts(double type){}
        virtual void initConsts(double type, double conc, double *hill){}
        virtual void initConsts(double type, double conc, double *hill, bool is_dutta){}
        virtual  void  computeRates(double  TIME,  double  *CONSTANTS,  double  *RATES,  double
*STATES, double *ALGEBRAIC) = 0;
        virtual void solveAnalytical(double dt) {};
      };
      #endif
```

d. enums/enum_Ohara_rudy_2011.hpp

This header act as a 'translation dictionary' for each variable in the cell model file. Enumeration required to easily track each variable, so instead of looking at numbers, I looked at pre-defined variable names corresponds to each correct values in the cell model. The enumeration follows CellML's description of each value and their function, as in the list below:

```
    #ifndef
EN_OHARA_RUDY_2011_HPP
    #define
EN_OHARA_RUDY_2011_HPP

    enum E_ALGEBRAIC_T{
      vffrt = 29,
      vfrt = 39,
      INa = 62,
      INaL = 64,
      Ito = 70,
      ICaL = 81,
      ICaNa = 82,
      ICaK = 85,
      IKr = 96,
      IKs = 98,
      IK1 = 100,
      INaCa_i = 132,
      INaCa_ss = 162,
      INaK = 181,
      tjp = 41,
      fINap = 61,
      mLss = 32,
      tmL = 42,
      hLss = 2,
      hLssp = 3,
      fINaLp = 63,
      ass = 4,
      ta = 17,
      iss = 5,
      delta_epi = 18,
      tiF_b = 33,
      tiS_b = 43,
      tiF = 46,
      tiS = 48,
      AiF = 65,
      AiS = 66,
      i = 67,
      assp = 34,
      dti_develop = 50,
      Axrf = 88,
      Axrs = 91,
      xr = 94,
      rkr = 95,
      xs1ss = 10,
      xs2ss = 26,
      txs1 = 27,
      KsCa = 97,
      txs2 = 38,
      xk1ss = 11,
      txk1 = 28,
      rk1 = 99,
      hna = 102,
      hca = 101,
      h1_i = 103,
      h2_i = 104,
      h3_i = 105,
      h4_i = 106,
      h5_i = 107,
      h6_i = 108,
```

```
INab = 184,
IKb = 183,
IpCa = 188,
ICab = 186,
Istim = 12,
CaMKb = 45,
CaMKa = 47,
JdiffNa = 187,
Jdiff = 189,
Jup = 196,
JdiffK = 185,
Jrel = 191,
Jtr = 197,
Bcai = 49,
Bcajsr = 53,
Bcass = 51,
ENa = 56,
EK = 57,
EKs = 58,
mss = 0,
tm = 13,
hss = 1,
thf = 14,
ths = 15,
h = 59,
jss = 16,
tj = 30,
hssp = 31,
thsp = 40,
hp = 60,
h8_ss = 140,
h9_ss = 141,
k3p_ss = 142,
k3pp_ss = 143,
k3_ss = 144,
k4_ss = 147,
k4p_ss = 145,
k4pp_ss = 146,
k6_ss = 148,
k7_ss = 149,
k8_ss = 150,
x1_ss = 151,
x2_ss = 152,
x3_ss = 153,
x4_ss = 154,
E1_ss = 155,
E2_ss = 156,
E3_ss = 157,
E4_ss = 158,
allo_ss = 159,
JncxNa_ss = 160,
JncxCa_ss = 161,
Knai = 163,
Knao = 164,
P = 165,
a1 = 166,
b2 = 167,
a3 = 168,
b3 = 169,
b4 = 170,
x1 = 171,
x2 = 172,
x3 = 173,
x4 = 174,
E1 = 175,
E2 = 176,
E3 = 177,
E4 = 178,
JnakNa = 179,
JnakK = 180,
xkb = 182,
Jrel_inf = 86,
tau_rel = 92,
dti_recover = 52,
tiFp = 54,
tiSp = 55,
ip = 68,
fItop = 69,
dss = 6,
fss = 7,
f = 71,
fcass = 19,
Afcaf = 72,
Afcas = 73,
fca = 74,
fp = 75,
fcap = 76,
km2n = 8,
anca = 20,
PhiCaL = 77,
PhiCaNa = 78,
PhiCaK = 79,
fICaLp = 80,
td = 21,
tff = 22,
tfs = 23,
tfcaf = 35,
tfcas = 36,
tffp = 37,
tfcafp = 44,
xrss = 9,
txrf = 24,
txrs = 25,
    vmyo = 118,
    vnsr = 119,
    vjsr = 120,
    vss = 121,
    amp = 12,
    duration = 13,
    KmCaMK = 14,
    aCaMK = 15,
    bCaMK = 16,
    CaMKo = 17,
    KmCaM = 18,
    cmdnmax_b = 19,
    cmdnmax = 93,
    kmcmdn = 20,
    trpnmax = 21,
    kmtrpn = 22,
    BSRmax = 23,
    KmBSR = 24,
    BSLmax = 25,
    KmBSL = 26,
    csqnmax = 27,
    kmcsqn = 28,
    cm = 29,
    PKNa = 30,
    mssV1 = 31,
    mssV2 = 32,
    mtV1 = 33,
    mtV2 = 34,
    mtD1 = 35,
    mtD2 = 36,
    mtV3 = 37,
    mtV4 = 38,
    hssV1 = 39,
    hssV2 = 40,
    Ahs = 94,
    Ahf = 41,
    GNa = 42,
    thL = 43,
    thLp = 95,
    GNaL_b = 44,
    GNaL = 96,
    Gto_b = 45,
    Gto = 97,
h7_i = 109,
h8_i = 110,
h9_i = 111,
k3p_i = 112,
k3pp_i = 113,
k3_i = 114,
k4_i = 117,
k4p_i = 115,
k4pp_i = 116,
k6_i = 118,
k7_i = 119,
k8_i = 120,
x1_i = 121,
x2_i = 122,
x3_i = 123,
x4_i = 124,
E1_i = 125,
E2_i = 126,
E3_i = 127,
E4_i = 128,
allo_i = 129,
JncxNa_i = 130,
JncxCa_i = 131,
h1_ss = 133,
h2_ss = 134,
h3_ss = 135,
h4_ss = 136,
h5_ss = 137,
h6_ss = 138,
h7_ss = 139,
    Gncx_b = 64,
    Gncx = 128,
    h10_i = 122,
    h11_i = 123,
    h12_i = 124,
    k1_i = 125,
    k2_i = 126,
    k5_i = 127,
    h10_ss = 129,
    h11_ss = 130,
    h12_ss = 131,
    k1_ss = 132,
    k2_ss = 133,
    k5_ss = 134,
    k1p = 65,
    k1m = 66,
    k2p = 67,
    k2m = 68,
    k3p = 69,
    k3m = 70,
    k4p = 71,
    k4m = 72,
    Knai0 = 73,
    Knao0 = 74,
    delta = 75,
    Kki = 76,
    Kko = 77,
    MgADP = 78,
    MgATP = 79,
    Kmgatp = 80,
    H = 81,
    eP = 82,
    Khp = 83,
    Knap = 84,
    Kxkur = 85,
    Pnak_b = 86,
    Pnak = 138,
    b1 = 135,
    a2 = 136,
    a4 = 137,
    GKb_b = 87,
    GKb = 105,
    PNab = 88,
```

```
  Jrel_infp = 87,
  Jrel_temp = 84,
  tau_relp = 93,
  Jrel_inf_temp = 83,
  fJrelp = 190,
  tau_rel_temp = 89,
  tau_relp_temp = 90,
  Jupnp = 192,
  Jupp = 193,
  fJupp = 194,
  Jleak = 195,
  step_low = 196,
  step_high = 197,
};
enum E_CONSTANTS_T{
  celltype = 0,
  nao = 1,
  cao = 2,
  ko = 3,
  R = 4,
  T = 5,
  F = 6,
  zna = 7,
  zca = 8,
  zk = 9,
  L = 10,
  rad = 11,
  vcell = 104,
  Ageo = 113,
  Acap = 117,

  cai = 9,
  m = 10,
  hf = 11,
  hs = 12,
  j = 13,
  hsp = 14,
  jp = 15,
  mL = 16,
  hL = 17,
  hLp = 18,
  a = 19,
  Kmn = 46,
  k2n = 47,
  PCa_b = 48,
  Aff = 98,
  Afs = 109,
  PCa = 99,
  PCap = 110,
  PCaNa = 111,
  PCaK = 112,
  PCaNap = 115,
  PCaKp = 116,
  tjca = 100,
  GKr_b = 49,
  GKr = 101,
  GKs_b = 50,
  GKs = 102,
  GK1 = 103,
  GK1_b = 51,
  kna1 = 52,
  kna2 = 53,
  kna3 = 54,
  kasymm = 55,
  wna = 56,
  wca = 57,
  wnaca = 58,
  kcaon = 59,
  kcaoff = 60,
  qna = 61,
  qca = 62,
  KmCaAct = 63,
iF = 20,
  iS = 21,
  ap = 22,
  iFp = 23,
  iSp = 24,
  d = 25,
  ff = 26,
  fs = 27,
  fcaf = 28,
  fcas = 29,
  jca = 30,
      PCab = 89,
      GpCa = 90,
      KmCap = 91,
      bt = 92,
      a_rel = 106,
      btp = 107,
      a_relp = 114,
      upScale = 108,
      stim_start = 139,
      BCL = 140,
      //stim_end  = 141,

      // Additional constants
for cvar
      Jrel_scale = 141,
      Jup_scale = 142,
      Jtr_scale = 143,
      Jleak_scale = 144,
      //KCaMK_scale = 144,
    };
    enum E_STATES_T{
      V = 0,
      CaMKt = 1,
      cass = 2,
      nai = 3,
      nass = 4,
      ki = 5,
      kss = 6,
      cansr = 7,
      cajsr = 8,
    ffp = 31,
      fcafp = 32,
      nca = 33,
      xrf = 34,
      xrs = 35,
      xs1 = 36,
      xs2 = 37,
      xk1 = 38,
      Jrelnp = 39,
      Jrelp = 40,
    };

    #endif
```

E. ‘modules’ Folder

The modules folder is a vital component of the project, containing some essential files that implement the core functionalities of the simulation. These files are modularised into source (.cpp or .cu) and header (.hpp or .cuh) formats, ensuring that the codebase is both organised and extensible. Each module serves a specific purpose, contributing to the overall framework of the simulation pipeline. As in the writing of this thesis, I found that some scripts were redundant and will be removed in the next updates.

a. cipa_t.cu and cipa_t.cuh

These files handle data type related to the formatting of CiPA (Comprehensive in vitro Proarrhythmia Assay) metrics, which are critical and become standard for assessing drug-induced proarrhythmia risks. All implementation in cipa_t.cu has been migrated to its header file (cipa_t.cuh) and will be removed in the next updates. Header file of cipa_t.cuh contains commented old vectors and naturalised 1D arrays as such:

```
#ifndef CIPA_T_HPP
#define CIPA_T_HPP

#include <map>
#include <string>

#include <cuda_runtime.h>

// using std::multimap;
// using std::string;
__global__ struct cipa_t{
  double qnet_ap;
  double qnet4_ap;
  double inal_auc_ap;
  double ical_auc_ap;
  double qnet_cl;
  double qnet4_cl;
  double inal_auc_cl;
  double ical_auc_cl;

  double dvmdt_repol;
  double vm_peak;
  double vm_valley;
  // multimap<double, double> vm_data;
  // multimap<double, double> dvmdt_data;
  // multimap<double, double> cai_data;
  // multimap<double, string> ires_data;

  // multimap<double, string> inet_data;
  // multimap<double, string> qnet_data;
  // multimap<double, string> inet4_data;
  // multimap<double, string> qnet4_data;

  // multimap<double, string> time_series_data;
```

```
  // temporary fix for this
  double vm_data[7000];
  double vm_time[7000];

  double dvmdt_data[7000];
  double dvmdt_time[7000];

  double cai_data[7000];
  double cai_time[7000];

  double ires_data[7000];
  double ires_time[7000];

  double inet_data[7000];
  double inet_time[7000];

  double qnet_data[7000];
  double qnet_time[7000];

  double inet4_data[7000];
  double inet4_time[7000];

  double qnet4_data[7000];
  double qnet4_time[7000];

  // double time_series_data[7000];
  // double time_series_time[7000];



  // __device__ cipa_t();
  // __device__ cipa_t( const cipa_t &source );
  // cipa_t& operator=(const cipa_t & source);
  // __device__ void copy(const cipa_t &source);
  // __device__ void init(const double vm_val);
  // __device__ void clear_time_result();


};


#endif
```

b. drug_conc.cpp and drug_conc.hpp

These files manage the drug concentration data used in the simulations. These act as utility file to lookup if user did not declare the

concentration value and drugs used is in the scope of CiPA. Lookup process is happening in the CPU domain. Below is the list of known drug by this utility function and its implementation using map:

```
#include "drug_conc.hpp"

float getValue(const std::unordered_map<std::string, float>& drugConc, const std::string& key, float defaultValue) {
    auto it = drugConc.find(key);
    if (it != drugConc.end()) {
        return it->second;
    }
    return defaultValue;
}

// Instantiate the dictionary with keys and values
std::unordered_map<std::string, float> drugConcentration = {
    {"azimilide", 70.0f},
    {"bepridil", 33.0f},
    {"disopyramide", 742.0f},
    {"dofetilide", 2.0f},
    {"ibutilide", 100.0f},
    {"quinidine", 3237.0f},
    {"sotalol", 14690.0f},
    {"vandetanib", 255.0f},
    {"astemizole", 0.26f},
    {"chlorpromazine", 38.0f},
    {"cisapride", 2.6f},
    {"clarithromycin", 1206.0f},
    {"clozapine", 71.0f},
    {"domperidone", 19.0f},
    {"droperidol", 6.3f},
    {"ondansetron", 139.0f},
    {"pimozide", 0.431f},
    {"risperidone", 1.81f},
    {"terfenadine", 4.0f},
    {"diltiazem", 122.0f},
    {"loratadine", 0.45f},
    {"metoprolol", 1800.0f},
    {"mexiletine", 4129.0f},
    {"nifedipine", 7.7f},
    {"nitrendipine", 3.02f},
    {"ranolazine", 1948.2f},
    {"tamoxifen", 21.0f},
```

```
    {"verapamil", 81.0f}
};
```

And header file:

```
#ifndef DRUG_CONC_HPP
#define DRUG_CONC_HPP

#include <unordered_map>
#include <string>

// Declare the dictionary function template
float getValue(const std::unordered_map<std::string, float>& drugConc, const std::string& key, float defaultValue = 0.0);

// Declare the dictionary extern
extern std::unordered_map<std::string, float> drugConcentration;
#endif  // DRUG_CONC_HPP
```

c. glob_funct.cpp and glob_funct.hpp

Global function or glob_funct encapsulate global utility functions that are used across various parts of the codebase. At the time this thesis was written, this script regulates the reading of flags and input deck. There are some unused functions that will be removed in the future. The global function script is implemented as below:

```
#include "glob_funct.hpp"
// #include "../libs/zip.h"

#include <cstdarg>
#include <cstdio>
#include <cstdlib>
#include <cstring>

// to make it more "portable" between OSes.
#if defined _WIN32
  #include <direct.h>
  #define snprintf _snprintf
  #define vsnprintf _vsnprintf
  #define strcasecmp _stricmp
  #define strncasecmp _strnicmp
```

```
#else
  #include <dirent.h>
#endif
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>


void mpi_printf(unsigned short node_id, const char *fmt, ...)
{

  va_list args;
  va_start(args, fmt);
  vprintf(fmt, args);
  va_end(args);
}

void mpi_fprintf(unsigned short node_id, FILE *stream, const char *fmt, ...)
{
#ifndef _WIN32
  if(mympi::rank == node_id){
    va_list args;
    va_start(args, fmt);
    vfprintf(stream, fmt, args);
    va_end(args);
  }
#else
  va_list args;
  va_start(args, fmt);
  vfprintf(stream, fmt, args);
  va_end(args);
#endif
}

void edison_assign_params(int argc, char *argv[], param_t *p_param)
{
  bool is_default;
  char buffer[100];
  char key[100];
  char value[100];
  char file_name[150];
  FILE *fp_inputdeck;

  // parameters from arguments
  for (int idx = 1; idx < argc; idx += 2) {
    if (!strcasecmp(argv[idx], "-input_deck"))
      strncpy(file_name, argv[idx + 1], sizeof(file_name));
    else if (!strcasecmp(argv[idx], "-hill_file"))
      strncpy(p_param->hill_file, argv[idx + 1], sizeof(p_param->hill_file));
```

```
    else if (!strcasecmp(argv[idx], "-cvar_file"))
      strncpy(p_param->cvar_file, argv[idx + 1], sizeof(p_param->cvar_file));
  }

  is_default = false;
  fp_inputdeck = fopen( file_name, "r");
  if(fp_inputdeck == NULL){
    fprintf(stderr, "Cannot open input deck file %s!!!\nUse default value as the
failsafe.\n", file_name);
    is_default = true;
  }

  // read input_deck line by line
  // and store each line to the buffer
  while ( is_default == false && fgets( buffer, 100, fp_inputdeck ) != NULL ) {
    sscanf( buffer, "%s %*s %s", key, value );
    if (strcasecmp(key, "Simulation_Mode") == 0) {
      p_param->simulation_mode = strtod( value, NULL );
    }
    else if (strcasecmp(key, "Celltype") == 0) {
      p_param->celltype = strtod( value, NULL );
    }
    else if (strcasecmp(key, "Is_Dutta") == 0) {
      p_param->is_dutta = strtol( value, NULL, 10 );
    }
    else if (strcasecmp(key, "Use_Conductance_Variability") == 0) {
      p_param->is_cvar = strtol( value, NULL, 10 );
    }
    else if (strcasecmp(key, "Pace_Find_Steepest") == 0) {
      p_param->find_steepest_start = strtod( value, NULL);
    }
    else if (strcasecmp(key, "GPU_Index") == 0) {
      p_param->gpu_index = strtod( value, NULL);
    }
    else if (strcasecmp(key, "Basic_Cycle_Length") == 0) {
      p_param->bcl = strtod( value, NULL );
    }
    else if (strcasecmp(key, "Number_of_Pacing") == 0) {
      p_param->pace_max = strtod( value, NULL );
    }
    else if (strcasecmp(key, "Time_Step") == 0) {
      p_param->dt = strtod( value, NULL );
    }
    //TODO: #Automation 1. eliminate drug_name and concentration on drug_name
    // else if (strcasecmp(key, "Drug_Name") == 0) {
    //   strncpy( p_param->drug_name, value, sizeof(p_param->concs) );
    // }
    // else if (strcasecmp(key, "Concentrations") == 0) {
    //   p_param->conc = strtod( value, NULL );
```

```
    // }

  }

  if( is_default == false ) fclose( fp_inputdeck );
}


int make_directory(const char* dirname )
{
#if defined _WIN32
  return _mkdir(dirname);
#else
  return mkdir(dirname, 0775);
#endif
}

int is_file_existed(const char* pathname)
{
#if defined _WIN32
  struct _stat buf;
  return _stat( pathname, &buf );
#else
  struct stat st = {0};
  return stat(pathname, &st);
#endif
}
```

d. glob_type.cpp and glob_type.hpp

The glob_type files define and manage the global data types and structures used throughout the project. For now, this script is being used to store IC50 data. The glob_type.hpp is found unused and will be removed in the next update. As glob_type.cpp is still being used, here is our implementation into it:

```
#ifndef GLOB_TYPE_HPP
#define GLOB_TYPE_HPP

#include <vector>

// global variable for MPI.
struct mympi
{
  static char host_name[255];
```

```
  static int host_name_len;
  static int rank;
  static int size;
};

// data structure for IC50
typedef struct row_data { double data[14]; } row_data;
typedef std::vector< row_data > drug_t;

// data structure to store
// ICaL/INaL control value
// for calculating qinward
// control means 0 concentration
// otherwise, drugs
typedef struct{
  double ical_auc_control;
  double inal_auc_control;
  double ical_auc_drug;
  double inal_auc_drug;
} qinward_t;

#endif
```

e. gpu.cu and gpu.cuh

The gpu files represent the core computational engine of the simulation, leveraging the parallel processing capabilities of NVIDIA GPUs to achieve significant speedups in simulation tasks. These files are essential for the framework's computational efficiency, enabling the processing of thousands of samples in parallel.

gpu.cu is the implementation file that contains CUDA-specific kernels and device functions. CUDA kernels are at the heart of the GPU simulation, responsible for solving the differential equations of the cell models, processing drug interactions, and performing large-scale computations for multiple samples concurrently. The kernels in this file are designed to optimise memory usage and minimise latency, ensuring the efficient execution of simulations. Header gpu.cuh serves as the header file that defines the interface for the GPU-specific functionalities implemented

in gpu.cu. It declares the prototypes of CUDA kernels and device functions, providing a structured entry point for other parts of the code to interact with GPU-based operations. The main gpu.cu code will look as this:

```
#include "../cellmodels/Ohara_Rudy_2011.hpp"
#include <stdio.h>
#include <cuda_runtime.h>
#include <cuda.h>

#include "glob_funct.hpp"
#include "glob_type.hpp"
#include "gpu_glob_type.cuh"
#include "gpu.cuh"


/*
all kernel function has been moved. Unlike the previous GPU code, now we seperate everything
into each modules.
all modules here are optimised for GPU and slightly different than the original component based
code
differences are related to GPU offset calculations
*/

__device__  void  kernel_DoDrugSim(double  *d_ic50,  double  *d_cvar,  double  d_conc,  double
*d_CONSTANTS, double *d_STATES, double *d_RATES, double *d_ALGEBRAIC,
                                      double *d_STATES_RESULT,
                                    //  double *time, double *states, double *out_dt,  double
*cai_result,
                                    //  double *ina, double *inal,
                                    //  double *ical, double *ito,
                                    //  double *ikr, double *iks,
                                    //  double *ik1,
                                     double  *tcurr,  double  *dt,  unsigned  short  sample_id,
unsigned int sample_size,
                                     cipa_t *temp_result, cipa_t *cipa_result,
                                     param_t *p_param
                                     )
    {

    unsigned int input_counter = 0;

    int num_of_constants = 145;
    int num_of_states = 41;
    int num_of_algebraic = 199;
    int num_of_rates = 41;
```

```
// INIT STARTS
temp_result[sample_id].qnet_ap = 0.;
temp_result[sample_id].qnet4_ap = 0.;
temp_result[sample_id].inal_auc_ap = 0.;
temp_result[sample_id].ical_auc_ap = 0.;

temp_result[sample_id].qnet_cl = 0.;
temp_result[sample_id].qnet4_cl = 0.;
temp_result[sample_id].inal_auc_cl = 0.;
temp_result[sample_id].ical_auc_cl = 0.;

temp_result[sample_id].dvmdt_repol = -999;
temp_result[sample_id].vm_peak = -999;
temp_result[sample_id].vm_valley = d_STATES[(sample_id * num_of_states) +V];

cipa_result[sample_id].qnet_ap = 0.;
cipa_result[sample_id].qnet4_ap = 0.;
cipa_result[sample_id].inal_auc_ap = 0.;
cipa_result[sample_id].ical_auc_ap = 0.;

cipa_result[sample_id].qnet_cl = 0.;
cipa_result[sample_id].qnet4_cl = 0.;
cipa_result[sample_id].inal_auc_cl = 0.;
cipa_result[sample_id].ical_auc_cl = 0.;

cipa_result[sample_id].dvmdt_repol = -999;
cipa_result[sample_id].vm_peak = -999;
cipa_result[sample_id].vm_valley = d_STATES[(sample_id * num_of_states) +V];
// INIT ENDS
bool is_peak = false;
// to search max dvmdt repol

tcurr[sample_id] = 0.0;
dt[sample_id] = p_param->dt;
double tmax;
double max_time_step = 1.0, time_point = 25.0;
double dt_set;

int cipa_datapoint = 0;

const double bcl = p_param->bcl;

// const double inet_vm_threshold = p_param->inet_vm_threshold;
// const unsigned short pace_max = 300;
// const unsigned short pace_max = 1000;
const unsigned short pace_max = p_param->pace_max;
const unsigned short last_drug_check_pace = p_param->find_steepest_start;
double conc = d_conc; //mmol
```

```
    double type = p_param->celltype;
    double epsilon = 10E-14;

    // eligible AP shape means the Vm_peak > 0.
    bool is_eligible_AP;
    // Vm value at 30% repol, 50% repol, and 90% repol, respectively.
    double vm_repol30, vm_repol50, vm_repol90;
    double t_peak_capture = 0.0;
    unsigned short pace_steepest = 0;

    bool init_states_captured = false;

    // qnet_ap/inet_ap values
          double inet_ap, qnet_ap, inet4_ap, qnet4_ap, inet_cl, qnet_cl, inet4_cl, qnet4_cl;
          double inal_auc_ap, ical_auc_ap,inal_auc_cl, ical_auc_cl;
    // qinward_cl;

    // char buffer[255];

    // static const int CALCIUM_SCALING = 1000000;
          // static const int CURRENT_SCALING = 1000;

    // printf("Core %d:\n",sample_id);
    initConsts(d_CONSTANTS, d_STATES, type, conc, d_ic50, d_cvar, p_param->is_dutta, p_param-
>is_cvar, sample_id);


    applyDrugEffect(d_CONSTANTS, conc, d_ic50, epsilon, sample_id);
    d_CONSTANTS[BCL + (sample_id * num_of_constants)] = bcl;

    // generate file for time-series output

    tmax = pace_max * bcl;
    int pace_count = 0;


    // printf("%d,%lf,%lf,%lf,%lf\n", sample_id, dt[sample_id], tcurr[sample_id], d_STATES[V +
(sample_id * num_of_states)],d_RATES[V + (sample_id * num_of_rates)]);
    // printf("%lf,%lf,%lf,%lf,%lf\n", d_ic50[0 + (14*sample_id)], d_ic50[1+ (14*sample_id)],
d_ic50[2+ (14*sample_id)], d_ic50[3+ (14*sample_id)], d_ic50[4+ (14*sample_id)]);

    while (tcurr[sample_id]<tmax)
    {
        computeRates(tcurr[sample_id], d_CONSTANTS, d_RATES, d_STATES, d_ALGEBRAIC, sample_id);

        dt_set = set_time_step( tcurr[sample_id], time_point, max_time_step,
        d_CONSTANTS,
        d_RATES,
        d_STATES,
```

```
        d_ALGEBRAIC,
        sample_id);

        //euler only
        // dt_set = 0.005;

        // printf("tcurr at core %d: %lf\n",sample_id,tcurr[sample_id]);
        if (floor((tcurr[sample_id] + dt_set) / bcl) == floor(tcurr[sample_id] / bcl)) {
          dt[sample_id] = dt_set;
          // printf("dt : %lf\n",dt_set);
          // it goes in here, but it does not, you know, adds the pace,
        }
        else{
          dt[sample_id] = (floor(tcurr[sample_id] / bcl) + 1) * bcl - tcurr[sample_id];

          // new part starts
          if( is_eligible_AP && pace_count >= pace_max-last_drug_check_pace) {
            temp_result[sample_id].qnet_ap = qnet_ap;
            temp_result[sample_id].qnet4_ap = qnet4_ap;
            temp_result[sample_id].inal_auc_ap = inal_auc_ap;
            temp_result[sample_id].ical_auc_ap = ical_auc_ap;
            temp_result[sample_id].qnet_cl = qnet_cl;
            temp_result[sample_id].qnet4_cl = qnet4_cl;
            temp_result[sample_id].inal_auc_cl = inal_auc_cl;
            temp_result[sample_id].ical_auc_cl = ical_auc_cl;
            //         fprintf(fp_vmdebug,         "%hu,%.2lf,%.2lf,%.2lf,%.2lf,%.2lf,%.2lf\n",
pace_count,t_peak_capture,temp_result.vm_peak,vm_repol30,vm_repol50,vm_repol90,temp_result.dvm
dt_repol);
            // replace result with steeper repolarization AP or first pace from the last 250
paces
            // if( temp_result->dvmdt_repol > cipa_result.dvmdt_repol ) {
            //   pace_steepest = pace_count;
            //   cipa_result = temp_result;
            //   }
            if( temp_result[sample_id].dvmdt_repol > cipa_result[sample_id].dvmdt_repol ) {
              pace_steepest = pace_count;
              //            printf("Steepest            pace            updated:            %d
dvmdt_repol: %lf\n",pace_steepest,temp_result[sample_id].dvmdt_repol);
              // cipa_result = temp_result;
              cipa_result[sample_id].qnet_ap = temp_result[sample_id].qnet_ap;
              cipa_result[sample_id].qnet4_ap = temp_result[sample_id].qnet4_ap;
              cipa_result[sample_id].inal_auc_ap = temp_result[sample_id].inal_auc_ap;
              cipa_result[sample_id].ical_auc_ap = temp_result[sample_id].ical_auc_ap;

              cipa_result[sample_id].qnet_cl = temp_result[sample_id].qnet_cl;
              cipa_result[sample_id].qnet4_cl = temp_result[sample_id].qnet4_cl;
              cipa_result[sample_id].inal_auc_cl = temp_result[sample_id].inal_auc_cl;
              cipa_result[sample_id].ical_auc_cl = temp_result[sample_id].ical_auc_cl;
```

```
            cipa_result[sample_id].dvmdt_repol = temp_result[sample_id].dvmdt_repol;
            cipa_result[sample_id].vm_peak = temp_result[sample_id].vm_peak;
            cipa_result[sample_id].vm_valley = d_STATES[(sample_id * num_of_states) +V];
            is_peak = true;
            init_states_captured = false;
            }
          else{
            is_peak = false;
          }
        };
        inet_ap = 0.;
        qnet_ap = 0.;
        inet4_ap = 0.;
        qnet4_ap = 0.;
        inal_auc_ap = 0.;
        ical_auc_ap = 0.;
        inet_cl = 0.;
        qnet_cl = 0.;
        inet4_cl = 0.;
        qnet4_cl = 0.;
        inal_auc_cl = 0.;
        ical_auc_cl = 0.;
        t_peak_capture = 0.;

        // temp_result->init( p_cell->STATES[V]);
        temp_result[sample_id].qnet_ap = 0.;
        temp_result[sample_id].qnet4_ap = 0.;
        temp_result[sample_id].inal_auc_ap = 0.;
        temp_result[sample_id].ical_auc_ap = 0.;

        temp_result[sample_id].qnet_cl = 0.;
        temp_result[sample_id].qnet4_cl = 0.;
        temp_result[sample_id].inal_auc_cl = 0.;
        temp_result[sample_id].ical_auc_cl = 0.;

        temp_result[sample_id].dvmdt_repol = -999;
        temp_result[sample_id].vm_peak = -999;
        temp_result[sample_id].vm_valley = d_STATES[(sample_id * num_of_states) +V];
        // end of init

        pace_count++;
        input_counter = 0; // at first, we reset the input counter since we re gonna only take
one, but I remember we don't have this kind of thing previously, so do we need this still?
        cipa_datapoint = 0; // new pace? reset variables related to saving the values,

        is_eligible_AP = false;
        // new part ends
         if(sample_id == 0 || (sample_id % 1000 == 0 && sample_id>999)){
```

```
            printf("core: %d pace count: %d, steepest: %d, dvmdt_repol: %lf, conc: %lf,
celltype: %lf\n",sample_id,pace_count, pace_steepest, cipa_result[sample_id].dvmdt_repol, conc,
d_CONSTANTS[(sample_id * num_of_constants) + celltype]);
          }
          //  printf("core: %d pace count: %d t: %lf, steepest: %d, dvmdt_repol: %lf,
t_peak:          %lf\n",sample_id,pace_count,          tcurr[sample_id],          pace_steepest,
cipa_result[sample_id].dvmdt_repol,t_peak_capture);
          // writen = false;
        }

        solveAnalytical(d_CONSTANTS, d_STATES, d_ALGEBRAIC, d_RATES,  dt[sample_id], sample_id);
        // solveEuler(d_STATES, d_RATES, dt[sample_id], sample_id);

        if (pace_count >= pace_max-last_drug_check_pace)
        {

                            if( tcurr[sample_id] > ((d_CONSTANTS[(sample_id * num_of_constants)
+BCL]*pace_count)+(d_CONSTANTS[(sample_id * num_of_constants) +stim_start]+2)) &&
                                      tcurr[sample_id]    <    ((d_CONSTANTS[(sample_id    *
num_of_constants)      +BCL]*pace_count)+(d_CONSTANTS[(sample_id      *      num_of_constants)
+stim_start]+10)) &&
                                      abs(d_ALGEBRAIC[(sample_id * num_of_algebraic) +INa]) <
1)
          {
            // printf("check 1\n");
            if( d_STATES[(sample_id * num_of_states) +V] > temp_result[sample_id].vm_peak )
            {
              temp_result[sample_id].vm_peak = d_STATES[(sample_id * num_of_states) +V];
              if(temp_result[sample_id].vm_peak > 0)
              {
                vm_repol30      =      temp_result[sample_id].vm_peak      -      (0.3      *
(temp_result[sample_id].vm_peak - temp_result[sample_id].vm_valley));
                vm_repol50      =      temp_result[sample_id].vm_peak      -      (0.5      *
(temp_result[sample_id].vm_peak - temp_result[sample_id].vm_valley));
                vm_repol90      =      temp_result[sample_id].vm_peak      -      (0.9      *
(temp_result[sample_id].vm_peak - temp_result[sample_id].vm_valley));
                is_eligible_AP = true;
                t_peak_capture = tcurr[sample_id];
              }
              else is_eligible_AP = false;
            }
                            }
                            else   if(   tcurr[sample_id]   >   ((d_CONSTANTS[(sample_id   *
num_of_constants)      +BCL]*pace_count)+(d_CONSTANTS[(sample_id      *      num_of_constants)
+stim_start]+10)) && is_eligible_AP )
          {

                                     if(   d_RATES[(sample_id   *   num_of_rates)   +V]   >
temp_result[sample_id].dvmdt_repol &&
```

```
                                                d_STATES[(sample_id  *  num_of_states)  +V]  <=
vm_repol30 &&
                                                d_STATES[(sample_id  *  num_of_states)  +V]  >=
vm_repol90 )
              {
                                                temp_result[sample_id].dvmdt_repol          =
d_RATES[(sample_id * num_of_rates) +V];
                // printf("check 4\n");
                                        }
          }
                              // calculate AP shape
                              if(is_eligible_AP  &&  d_STATES[(sample_id  *  num_of_states)  +V]  >
vm_repol90)
          {

          inet_ap = (d_ALGEBRAIC[(sample_id * num_of_algebraic) +INaL]+d_ALGEBRAIC[(sample_id *
num_of_algebraic)          +ICaL]+d_ALGEBRAIC[(sample_id          *          num_of_algebraic)
+Ito]+d_ALGEBRAIC[(sample_id     *     num_of_algebraic)     +IKr]+d_ALGEBRAIC[(sample_id     *
num_of_algebraic) +IKs]+d_ALGEBRAIC[(sample_id * num_of_algebraic) +IK1]);
          inet4_ap = (d_ALGEBRAIC[(sample_id * num_of_algebraic) +INaL]+d_ALGEBRAIC[(sample_id
*        num_of_algebraic)        +ICaL]+d_ALGEBRAIC[(sample_id        *        num_of_algebraic)
+IKr]+d_ALGEBRAIC[(sample_id * num_of_algebraic) +INa]);
          qnet_ap += (inet_ap * dt[sample_id])/1000.;
          qnet4_ap += (inet4_ap * dt[sample_id])/1000.;
          inal_auc_ap += (d_ALGEBRAIC[(sample_id * num_of_algebraic) +INaL]*dt[sample_id]);
          ical_auc_ap += (d_ALGEBRAIC[(sample_id * num_of_algebraic) +ICaL]*dt[sample_id]);
                              }
          inet_cl = (d_ALGEBRAIC[(sample_id * num_of_algebraic) +INaL]+d_ALGEBRAIC[(sample_id *
num_of_algebraic)          +ICaL]+d_ALGEBRAIC[(sample_id          *          num_of_algebraic)
+Ito]+d_ALGEBRAIC[(sample_id     *     num_of_algebraic)     +IKr]+d_ALGEBRAIC[(sample_id     *
num_of_algebraic) +IKs]+d_ALGEBRAIC[(sample_id * num_of_algebraic) +IK1]);
          inet4_cl = (d_ALGEBRAIC[(sample_id * num_of_algebraic) +INaL]+d_ALGEBRAIC[(sample_id
*        num_of_algebraic)        +ICaL]+d_ALGEBRAIC[(sample_id        *        num_of_algebraic)
+IKr]+d_ALGEBRAIC[(sample_id * num_of_algebraic) +INa]);
          qnet_cl += (inet_cl * dt[sample_id])/1000.;
          qnet4_cl += (inet4_cl * dt[sample_id])/1000.;
          inal_auc_cl += (d_ALGEBRAIC[(sample_id * num_of_algebraic) +INaL]*dt[sample_id]);
          ical_auc_cl += (d_ALGEBRAIC[(sample_id * num_of_algebraic) +ICaL]*dt[sample_id]);

          // save temporary result -> ALL TEMP RESULTS IN, TEMP RESULT != WRITTEN RESULT
          if((pace_count   >=   pace_max-last_drug_check_pace)   &&   (is_peak   ==   true)   &&
(pace_count<pace_max) )
          {
            // printf("input_counter: %d\n",input_counter);
            // datapoint_at_this_moment = tcurr[sample_id] - (pace_count * bcl);
            temp_result[sample_id].cai_data[cipa_datapoint]    =       d_STATES[(sample_id    *
num_of_states) +cai] ;
            temp_result[sample_id].cai_time[cipa_datapoint] =  tcurr[sample_id];
```

```
            temp_result[sample_id].vm_data[cipa_datapoint]     =     d_STATES[(sample_id     *
num_of_states) +V];
            temp_result[sample_id].vm_time[cipa_datapoint] = tcurr[sample_id];

            temp_result[sample_id].dvmdt_data[cipa_datapoint]    =    d_RATES[(sample_id    *
num_of_rates) +V];
            temp_result[sample_id].dvmdt_time[cipa_datapoint] = tcurr[sample_id];

            if(init_states_captured == false){
              for(int counter=0; counter<num_of_states; counter++){
                d_STATES_RESULT[(sample_id * num_of_states) + counter] = d_STATES[(sample_id *
num_of_states) + counter];
              }
              init_states_captured = true;
            }

            // time series result

            // time[input_counter + sample_id] = tcurr[sample_id];
            // states[input_counter + sample_id] = d_STATES[V + (sample_id * num_of_states)];

            // out_dt[input_counter + sample_id] = d_RATES[V + (sample_id * num_of_states)];


            //  cai_result[input_counter  +  sample_id]  =  d_ALGEBRAIC[cai  +  (sample_id  *
num_of_algebraic)];

            //   ina[input_counter   +   sample_id]   =   d_ALGEBRAIC[INa   +   (sample_id   *
num_of_algebraic)] ;
            //   inal[input_counter   +   sample_id]   =   d_ALGEBRAIC[INaL   +   (sample_id   *
num_of_algebraic)] ;

            //   ical[input_counter   +   sample_id]   =   d_ALGEBRAIC[ICaL   +   (sample_id   *
num_of_algebraic)] ;
            //   ito[input_counter   +   sample_id]   =   d_ALGEBRAIC[Ito   +   (sample_id   *
num_of_algebraic)] ;

            //   ikr[input_counter   +   sample_id]   =   d_ALGEBRAIC[IKr   +   (sample_id   *
num_of_algebraic)] ;
            //   iks[input_counter   +   sample_id]   =   d_ALGEBRAIC[IKs   +   (sample_id   *
num_of_algebraic)] ;

            //   ik1[input_counter   +   sample_id]   =   d_ALGEBRAIC[IK1   +   (sample_id   *
num_of_algebraic)] ;

            input_counter = input_counter + sample_size;
            cipa_datapoint = cipa_datapoint + 1; // this causes the resource usage got so mega
and crashed in running
```

```
            } // temporary guard ends here

                    } // end the last 250 pace operations
        tcurr[sample_id] = tcurr[sample_id] + dt[sample_id];
        //printf("t after addition: %lf\n", tcurr[sample_id]);

    } // while loop ends here
    // __syncthreads();
}



__global__ void kernel_DrugSimulation(double *d_ic50, double *d_cvar, double *d_conc, double
*d_CONSTANTS, double *d_STATES, double *d_RATES, double *d_ALGEBRAIC,
                                      double *d_STATES_RESULT,
                                      // double *time, double *states, double *out_dt,  double
*cai_result,
                                      // double *ina, double *inal,
                                      // double *ical, double *ito,
                                      // double *ikr, double *iks,
                                      // double *ik1,
                                      unsigned int sample_size,
                                      cipa_t *temp_result, cipa_t *cipa_result,
                                      param_t *p_param
                                      )
  {
    unsigned short thread_id;
    thread_id = blockIdx.x * blockDim.x + threadIdx.x;
    if (thread_id >= sample_size) return;
    double time_for_each_sample[10000];
    double dt_for_each_sample[10000];
    // cipa_t temp_per_sample[2000];
    // cipa_t cipa_per_sample[2000];
    // printf("in\n");
    // printf("Calculating %d\n",thread_id);
    kernel_DoDrugSim(d_ic50,  d_cvar,  d_conc[thread_id],  d_CONSTANTS,  d_STATES,  d_RATES,
d_ALGEBRAIC,
                           d_STATES_RESULT,
                           // time, states, out_dt, cai_result,
                           // ina, inal,
                           // ical, ito,
                           // ikr, iks,
                           // ik1,
                           time_for_each_sample, dt_for_each_sample, thread_id, sample_size,
                           temp_result, cipa_result,
                           p_param
                           );
                           // __syncthreads();
```

```
    // printf("Calculation for core %d done\n",sample_id);
  }
```

The header file of gpu.hpp declared as follows:

```
#ifndef GPU_CUH
#define GPU_CUH
#include <cuda_runtime.h>
#include <cuda.h>
#include "cipa_t.cu"
__global__ void kernel_DrugSimulation(double *d_ic50, double *d_cvar, double *d_conc, double
*d_CONSTANTS, double *d_STATES, double *d_RATES, double *d_ALGEBRAIC,
                                      double *d_STATES_RESULT,
                                    //   double *time, double *states, double *out_dt,  double
*cai_result,
                                    //   double *ina, double *inal,
                                    //   double *ical, double *ito,
                                    //   double *ikr, double *iks,
                                    //   double *ik1,
                                      unsigned int sample_size,
                                      cipa_t *temp_result, cipa_t *cipa_result,
                                      param_t *p_param
                                      );


__device__ void kernel_DoDrugSim(double *d_ic50, double *d_cvar, double d_conc, double
*d_CONSTANTS, double *d_STATES, double *d_RATES, double *d_ALGEBRAIC,
                                      double *d_STATES_RESULT,
                                    //    double *time, double *states, double *out_dt,  double
*cai_result,
                                    //    double *ina, double *inal,
                                    //    double *ical, double *ito,
                                    //    double *ikr, double *iks,
                                    //    double *ik1,
                                       double *tcurr, double *dt, unsigned short sample_id,
unsigned int sample_size,
                                       cipa_t *temp_result, cipa_t *cipa_result,
                                       param_t *p_param
                                       );


#endif
```

In the post-processing mode (the mode that only runs the simulation once and take all biomarkers calculation), these comments were uncommented to let more data flow in and out of the GPU kernel.

f. param.cpp and param.hpp

These scripts manages the simulation parameters. It reads the input deck file and put them into the code as variables. The param.cpp implements functions to read the input file, load, parse, validate parameter files, and act as a failsafe if one or more parameters or parameter file is not readable. This script has the default values and path of each required numbers and files. This file ensures that the simulation runs with accurate and user-defined settings. Header of param.hpp declares variables used to store these parameter values and ensuring that they are well defined in the simulation. The header file goes as a following struct:

```
#ifndef PARAM_HPP
#define PARAM_HPP

struct param_t
{
  unsigned short simulation_mode; // toggle between sample-based or full-pace simulations
  bool is_dutta; // TRUE if using Dutta scaling
  unsigned short gpu_index;
  bool is_print_graph; // TRUE if we want to print graph
  bool is_using_output; // TRUE if using last output file
  bool is_cvar;
  double bcl; // basic cycle length
  // unsigned int max_samples;
  unsigned short pace_max; // maximum pace
  unsigned short find_steepest_start;
  unsigned short celltype;  // cell types
  double dt;        // time step
  double dt_write;  // writing step
  double inet_vm_threshold; // Vm threshold for calculating inet
  char hill_file[1024];
  char cvar_file[1024];
  char drug_name[100];
  char concs[100];
  float conc;
  void init();
```

```
  void show_val();
};

#endif
```

And the main file reads parameter and their default values to the whole project with following way:

```
#include "param.hpp"
#include <cstdio>
#include "glob_funct.hpp"
void param_t::init()
{
  simulation_mode = 0;
  // max_samples = 10000;
  is_dutta = true;
  gpu_index = 0;

  is_cvar = false;
  bcl = 2000.;
  pace_max = 10;

  find_steepest_start = 5;

  celltype = 0.;
  dt = 0.005;
  // dt = 0.1;

  conc = 99.0;
  // dt_write = 2.0;
  // inet_vm_threshold = -88.0;

  snprintf(hill_file, sizeof(hill_file), "%s", "./drugs/bepridil/IC50_samples.csv");
  snprintf(cvar_file, sizeof(cvar_file), "%s", "./drugs/10000_pop.csv");
  snprintf(drug_name, sizeof(drug_name), "%s", "bepridil");
  // snprintf(concs, sizeof(concs), "%s", "99.0");
}

void param_t::show_val()
{
  //change this to printf somehow
  mpi_printf( 0, "%s -- %s\n", "Simulation mode", simulation_mode ? "full-pace" :
"sample-based" );
  mpi_printf( 0, "%s -- %s\n", "Hill File", hill_file );
  mpi_printf( 0, "%s -- %hu\n", "Celltype", celltype);
  mpi_printf( 0, "%s -- %s\n", "Is_Dutta", is_dutta ? "true" : "false" );
```

```
    mpi_printf( 0, "%s -- %s\n", "Is_Cvar", is_cvar ? "true" : "false" );
    mpi_printf( 0, "%s -- %lf\n", "Basic_Cycle_Length", bcl);
    mpi_printf( 0, "%s -- %d\n", "GPU_Index", gpu_index);
    mpi_printf( 0, "%s -- %hu\n", "Number_of_Pacing", pace_max);
    mpi_printf( 0, "%s -- %hu\n", "Pace_Find_Steepest", find_steepest_start);
    mpi_printf( 0, "%s -- %lf\n", "Time_Step", dt);
    mpi_printf( 0, "%s -- %s\n", "Drug_Name", drug_name);
    mpi_printf( 0, "%s -- %lf\n\n\n", "Concentrations", conc);
}
```

F. Critical Scripts

This section highlights the key scripts within the framework, focusing on their roles and significance in executing and customising simulations. These critical scripts form the backbone of the codebase, enabling efficient management of computations, GPU operations, and parameter configurations. If there is any issue with the code, 80% of the issue can be solved by looking at these three script first. Most of debugging, adjustement, and modification happen in these scripts. Below is a detailed breakdown:

a. Main script

The main script (main.cu) serves as the central controller of the simulation framework. It is the entry point of the program, responsible for managing the simulation's overall workflow, from initialisation to the final output. Below are its primary responsibilities and modifications related to main.cu:

- Simulation Parameter Parsing
    - Reads user-provided inputs, such as input_deck.txt, initial state files, and IC50/Hill coefficient files.
    - Validates the correctness and presence of required input files.
    - Flags any missing or improperly formatted files, preventing the simulation from starting.
- Simulation Setup

    - Loads and processes input parameters from the provided files.
    - Sets up the pacing protocol, drug concentrations, and simulation-specific configurations.
    - Handles initialisation for multiple samples and conditions.
- Memory Management
    - Allocates and deallocates memory for the CPU and GPU, ensuring efficient resource utilisation using core (thread) per block calculation.
    - Transfers data such as parameters, initial states, and configurations between host (CPU) and device (GPU).
- GPU Kernel Launch
    - Initiates GPU computations by invoking the kernel functions for parallel execution.
    - Ensures proper thread and block configurations to optimise performance for the given workload.
- Result Handling
    - Collects output data from the GPU and processes it for storage.
    - Saves simulation results to the designated output directory.
    - Generates logs for debugging and validation purposes.
- Error Handling and Debugging
    - Includes checks to identify and report common errors, such as missing inputs, memory allocation failures, or kernel launch issues. If something is wrong with the kernel function in the GPU, the code always goes back to this script, assuming the kernel function has done even with no output.
    - Provides detailed logs to help trace and resolve errors during simulation execution.

b. GPU control script

The GPU control script (`gpu.cu` and `gpu.cuh`) is the core component that drives the high-performance computations. It includes all GPU-specific operations, such as kernel implementations, and parallel execution logic. The kernel simulates the electrical behaviour of cardiomyocyte cells over

time, under the influence of a specific drug at a given concentration. This script manages the distribution of workloads across GPU threads and blocks, ensuring that simulations are processed efficiently.

Key responsibilities of this script include:

- Calling the numerical solvers (e.g., Forward Euler, Rush-Larsen) for simulating cardiac cell dynamics.
- Managing steps and loops of simulation, such as calling drug effect applying function.
- Calculating metrics for biomarkers, time series data, and such related to simulation, or calculated during simulation loops.

Most performance issues or GPU errors, such as kernel launch failures or memory overflows, can be traced back to this script. It is also the primary location for implementing modifications to the numerical methods or adding new solvers.

c. Parameters

The parameter script (param.cpp and param.hpp) defines and manages the input parameters required for the simulation. It acts as a configuration layer, ensuring that all essential variables are correctly initialised and passed to the simulation workflow. This script is also responsible for providing default values for parameters when they are not explicitly defined in the input files. It ensures that the simulation remains robust against incomplete or inconsistent configurations.

Adjustments to this script are necessary when:

- Adding new features or variables to the simulation.
- Customising the simulation for different biological models or experimental setups.

- Debugging errors related to undefined or mismatched parameters.

G. Commands and Flag Usages

This section outlines the essential commands and flags used to execute, customise, and manage the simulation framework. This part tend to face human error as in wrong path. The error usually shown as the file being read as 0, or file not found. Also the code being run without proper compilation previously. The first phase of the simulation's command and flags run as below:

```
./drug_sim -input_deck input_deck.txt -hill_file drug/IC50_file.csv
```

Second phase of the simulation can be run as below:

```
./drug_sim -input_deck input_deck.txt -hill_file drug/IC50_file.csv -init_file
initfile.csv
```

The commands above represent two primary phases of the simulation process. In the first phase, the simulation is initiated with the essential input deck and Hill coefficient file. The -input_deck flag specifies the configuration file (input_deck.txt), which contains simulation parameters such as pacing details, cell models to be used, and output settings. The -hill_file flag points to a CSV file (drug/IC50_file.csv) containing IC50 and Hill coefficient values for the drug being simulated. This step generates initial conditions and baseline data required for further simulations.

In the second phase, the simulation proceeds with additional parameters specified by the -init_file flag. This flag allows the user to include an initial state file (initfile.csv), which represents the system's state at the end of the first phase. By leveraging this initial state, the second phase can bypass reinitialisation and focus on computing drug effects or specific pacing

protocols. This phase is particularly useful for testing drug interactions or extended pacing simulations without repeating initial computations.

In other cell models, more input is required and uses the same method to be registered in the simulation. ORd 2017 requires additional -herg_file as input, since this newer cell model needs herg fitting value from CiPA. This update increases the simulation accuracy. A common ORd 2017 simulation will be called as such:

```
./drug_sim -input_deck input_deck.txt -hill_file drug/IC50_file.csv -herg_file drug/herg.csv
```

While herg.csv is structured as:

```
Kmax,Ku,n,halfmax,Vhalf,slope
5594000,0.0001719,0.9374,147200000,-61.34,NA
```

Each drug will have their own hERG file.

Also in three of the cell models, future update related to inter-individual variability will be applied. In the future, these cell models will require additional -cvar_file (conductance variability) as input. This update will enrich the simulation capability to simulate drug effect in different population segments such as healthy people compared with people living with heart failure history. This update will increase the simulation accuracy and variability. A common ORd 2017 simulation with inter-individual variability will be called as such:

```
./drug_sim -input_deck input_deck.txt -hill_file drug/IC50_file.csv -herg_file drug/herg.csv -cvar_file cvar.csv
```

H. Troubleshooting

This section provides guidance on addressing frequent issues encountered when using the simulation framework. These troubleshooting steps are designed to assist users in diagnosing and resolving problems effectively. Especially in the situation where CUDA debugger were not informative enough due to the uniqueness of the parallelisation.

- File Not Found Errors

Problem: The simulation fails with an error indicating that a required file is missing or cannot be accessed.

Cause:

- Incorrect file path provided in the command-line arguments.
- The file does not exist in the specified directory.

Solution:

Verify the file path in the command. Ensure the paths match the directory structure (e.g., drug/IC50_file.csv). Check if the file exists in the specified folder. If not, create or move the file to the correct location. Ensure file permissions allow read access.

- Compilation Errors

Problem: Compilation fails, showing errors related to missing libraries or incompatible flags.

Cause:

- Required CUDA or GCC/G++ versions are not installed.
- Incorrect flags in the Makefile.

Solution:

Verify that the system has the correct version of CUDA (e.g., version 11.x or later).Check that nvcc and gcc are installed and added to the system's PATH. Review and update the Makefile to match the system configuration. For instance, ensure paths to CUDA libraries and headers are correct.

- Zeroes or Incomplete Output

Problem: The output files contain no data or are incomplete.

Cause:

- Incorrect input parameters or missing initial state files.
- Errors in the numerical solver or GPU execution.

Solution:

Ensure the input deck (input_deck.txt) is correctly formatted and contains all required parameters.Verify that the initial state file (initfile.csv) exists and is properly formatted. Check the GPU kernel execution logs (printed in the terminal) for errors (e.g., memory overflows or segmentation faults).

- GPU Kernel Launch Failures,

Problem: The simulation crashes during execution, or right just before interacting with GPU kernel function.

Cause:

- Insufficient GPU memory to handle the workload.

- Incompatible GPU architecture flags in the Makefile.

Solution:

Reduce the number of samples or concentrations in the input parameters to fit within available GPU memory. For information, in average it takes 2 MB of GPU memory to simulate one sample. Verify the GPU's compute capability and ensure the Makefile includes the correct -arch flag (e.g., -arch=sm_86 for NVIDIA RTX 4090).

- Runtime Errors Due to Command-Line Arguments

Problem: The program crashes or produces unexpected results when running with specific arguments.

Cause:

- Missing or incorrectly formatted command-line arguments.

Solution:

Ensure all required arguments (-input_deck, -hill_file, -init_file, etc.) are included and point to valid files. Refer to the section on "Commands and Flag Usages" for examples of valid commands.

- Mismatched Results

Problem: The GPU simulation results do not match the expected outputs from the CPU simulation.

Cause:

- Numerical instability or precision issues in the GPU solver.
- Issue with drug effect implementation

Solution:

Double-check the numerical solver configurations to ensure consistency between CPU and GPU implementations. Compare the input data files to confirm they are identical for both simulations. Start comparison from non-drug, control data first. If control from both CPU and GPU shows no noticeable difference in action potential shape, it is confirmed that the issue comes from drug effect implementation.

Check the IC50 input, current development applies these values to the ratio of the gate instead of the gate. Previous implementation may implement directly to the gate. These two kinds of IC50 files can be distinguished by looking at the orde of the numbers. Old implementation method usually uses thousands or hundred of thousands, while more recent version usually stays at most around the thousands.

If the IC50 uses a more recent version, ensure in the drug effect implementation function, it does not change the gate, but instead the '_b' of the gate (GNaL_b for example).

- Slow Performance on GPU

Problem: The GPU simulation runs slower than expected, or it runs more than 24 even 48 hours for 1000 pacing or less.

Cause:

  - Inappropriate workload size or insufficient parallelisation.
  - Suboptimal GPU memory management.
  - Less compatible GPU clock speed.

Solution:

Increase the workload size to fully utilise GPU cores (e.g., simulate more samples or concentrations). Profile the simulation using tools like nvprof or Nsight to identify bottlenecks in memory transfers or kernel executions. Ensure the simulation runs on a more recent GPU generation from NVIDIA, specifically the RTX series (RTX 3080Ti, RTX 4090, etc.). Also for information, this simulation tested and optimised for a gaming grade PC GPU with clock speed around 900-1200 MHz. Some of server-grade GPU has a lot of GPU memory but lacking of clockspeed. I conducted a test on a server grade GPU, NVIDIA T4 with 81 GB of GPU memory, but it was considered too slow. It finished the job simulating 100 pacing of ORd 2011 with drug effect in around 36-40 hours.

- Memory Overflows or Leaks

Problem: The simulation crashes with strange, unreadable error message, makes us cannot access the GPU, or hangs due to excessive memory usage.

Cause:

  - Improper memory allocation or deallocation in GPU kernels.

Solution:

Inspect the GPU control script (gpu.cu) for memory management issues.
Use tools like cuda-memcheck to debug memory leaks or access violations.

- Compatibility Issues

Problem: The code does not work on certain hardware or software configurations. Signed by a fail in running the kernel script with no error at all.

Cause:

- Dependency mismatches, such as outdated libraries or drivers.

Solution:

Update the GPU drivers to the latest version recommended by NVIDIA. Ensure the CUDA version matches the code's requirements (CUDA 11.x). Also ensure the Makefile includes the correct -arch flag (e.g., -arch=sm_86 for NVIDIA RTX 4090). The code is not tested on other than GTX or RTX family. The oldest GPU I have ever tested the code was GTX 1660 Ti.

Enhancing the efficiency of animal-alternative *in silico* drug cardiotoxicity prediction through CUDA-based parallel processing		Thesis for Master of Engineering	December 2024	Iga Narendra Pramawijaya