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

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.

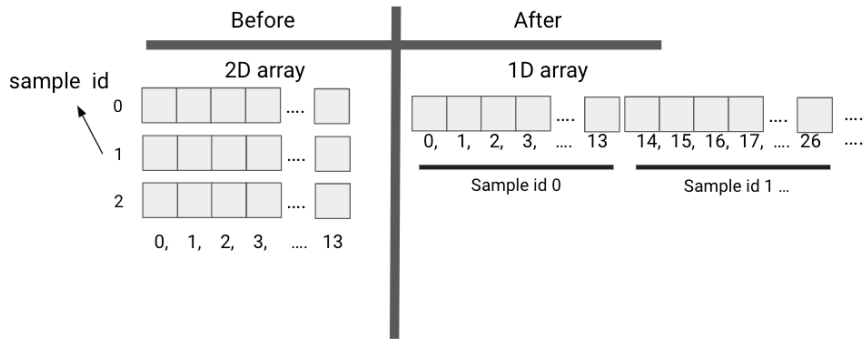
Offsetting is a technique used to manage data indexing efficiently, ensuring that thread indices correspond to the correct memory addresses.

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.

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

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

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.