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

Iga Narendra Pramawijaya1\*, Ariyadi1, Aroli Marcellinus1 , Ali Ikhsanul Qauli1 , Ki Moo Lim1

1: Department of IT Convergence Engineering, Kumoh National Institute of Technology, Korea

\*: [iga@kumoh.ac.kr](mailto:iga@kumoh.ac.kr)

**Abstract:**

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. 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. 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 (up to 49 times in fastest configuration) in GPU-based simulations. 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

**Keywords:** In silico, cardiotoxicity, CUDA, GPU Parallel Programming

**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 ethical 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 accelerate 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.

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. 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. Viewing at previous researches, enhancing *in silico* simulation performance in cardiotoxicity prediction remain untouched. Hence, 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.

**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.

Once cell model file is acquired in C or C++ format, conversion in terms of GPU memory adjustment and offsetting is needed. 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. 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 × 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.

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.

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.

(solving ODE) 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.

(simulation protocol) 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 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.

**3 Results and Discussion**