Simulation of Large Sets of Metabolic Reactions on Graphics Processing Units (GPUs)

Guilherme Ryu Odaguiri Kobori

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisor: Prof. Paulo Alexandre Crisóstomo Lopes

Examination Committee

Chairperson: Prof. Teresa Maria Sá Ferreira Vazão Vasques

Supervisor: Prof. Paulo Alexandre Crisóstomo Lopes

Members of the Committee: Prof. José Carlos Alves Pereira Monteiro

July 2019

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Declaração

Declaro que o presente documento é um trabalho original da minha autoria e que cumpre todos os requisitos do Código de Conduta e Boas Práticas da Universidade de Lisboa.

Acknowledgements

I would like to thank Professor Paulo Alexandre Crisóstomo Lopes for all the help in the process of developing this dissertation. For giving me the opportunity to work on this subject as well as for providing the computer equipped with the GPU needed for this work. Also, for the weekly guidance and help with better understanding concepts and guidance on the steps necessary for the project progression.

My thanks to Instituto Superior Técnico, for the opportunity given to me to obtain a master's degree on their exchange students programme, and for all the support provided in these two years of studying in this institution.

I would also like to express my gratitude towards Escola Politécnica da Universidade de São Paulo, also involved in this exchange opportunity, for providing me with the bases of knowledge that were of great help on this journey.

As important as the other institutions, I would like to thanks INESC-ID and all its staff for providing a physical space where I could work on this dissertation on remarkably great conditions.

At last I would like to thank my pets, my friends, my family, for all the support throughout the years.

Abstract

Simulating large sets of metabolic reactions is a very time-consuming process. Here CUDA enabled GPUs are utilized to simulate such metabolic reactions in an attempt to reduce simulation execution times. By using metaprogramming and libSBML biochemical models can be extracted from SBML files, and a CUDA program can be generated to simulate this model in a GPU. Two approaches were tested, one based on the Euler Method and another based on the Gillespie Algorithm, the latter being the one used in the final iteration of the program. The simulation results and execution times were compared to another simulation software, COPASI, which does not have parallel characteristics. The CUDA application presented mixed results, for a well-behaved model it presented a better execution time, when extrapolating, for a number of simulations of 220 and higher; for a highly volatile model it did not present a speed up when compared to a linear approach. This study showcases challenges when trying to implement the Euler Method approach in GPUs and the reasons why the Gillespie Algorithm was chosen over it. The final application represents a first step in simulating biochemical models in GPUs, and showcases main development points when creating similar applications.

Keywords

biochemical models, CUDA, Euler method, Gillespie algorithm, GPU, libSBML, metabolic reactions, SBML

Resumo

O processo de simulação de grandes conjuntos de reações metabólicas é um processo que consome tempo. Neste documento GPUs com suporte a CUDA são utilizadas para simular estas reações em uma tentativa de reduzir o tempo de execução das simulações. Metaprogramação e libSBML são usados para extrair modelos bioquímicos de ficheiros SBML, gerando um programa em CUDA capaz de simular o modelo em GPU. Dois métodos foram testados, um baseado no Método de Euler e outro baseado no Algoritmo Gillespie, o último é o utilizado na versão final do programa. Os resultados das simulações e os tempos de execução foram comparados com outro software de simulação, COPASI, que não possui características de paralelização. A aplicação em CUDA obteve resultados mistos, para um modelo relativamente estável a aplicação obteve tempos de execução melhores, por extrapolação, a partir de 220 simulações; para um modelo volátil não há melhorias no tempo de execução. Este estudo apresenta os desafios encontrados ao implementar o Método de Euler em GPUs e as razões pelas quais o Algoritmo de Gillespie é o utilizado na versão final. A aplicação desenvolvida representa um primeiro passo para simular modelos bioquímicos em GPUs, e demonstra elementos de desenvolvimento principais envolvidos na criação de aplicações similares.

Palavras-chave

algoritmo de Gillespie, CUDA, GPU, libSBML, método de Euler, modelos bioquímicos, reações metabólicas, SBML

Index

List of figures

List of tables

List of abbreviations

SBML	Systems Biology Markup Language
CPU	Central Processing Unit
GPU	Graphics Processing Unit
XML	Extensible Markup Language
CUDA	Compute Unified Device Architecture
ID	Identification
CSV	Comma Separated Values
ODE	Ordinary Differential Equation

1. Introduction

Simulating sets of metabolic reactions can give us a better understanding on how complex biochemical processes work. By being able to completely monitor the progression of all the chemical elements involved, it enables the process of making educated guesses on their effects on each other and the analysis on how the entire biological model functions on a deeply fundamental level, the chemical reactions. Furthermore, it enables simulating a large number of different situations, enabling observations on how the behaviour of the biochemical model alters with changes in initial parameters, and beyond these initial parameters, it is also possible to simulate external interferences to the model by modifying parameters when certain conditions are met (e.g., simulating a drug administration on a certain disease development stage).

This work attempts to develop an application capable of executing such simulations using a GPU. It tries to achieve feasible results in an attempt to show that the generated program could be used in future investigations, or at least serve as a support for future GPU centered simulation software. It is also a study of the challenges, and possible solutions to these challenges, encountered when developing such programs, and how it can help future developers of similar applications.

The kinetic equations that model a chemical system can describe deterministic or stochastic systems or even both. The deterministic systems are described by differential equations which can be solved by a wide range of methods, for example the Euler Method. On stochastic systems one example is the Gillespie's Direct Method.

At first the Euler Method was considered but, after encountering some complications with properly fitting the method with the GPU, the stochastic method was elected as the next attempt. Given that the Gillespie Algorithm [3] was a better fit for the GPU implementation it was adopted in the final application developed.

1.1. Deterministic simulations

In deterministic simulations there is no presence of random elements during the execution, a given set of inputs always produces the same set of outputs. One advantage over stochastic approaches is a faster execution time, since there is no need to repeat the simulations, and in order to validate them (assuming that the model is correct), a single run provides the expected results.

The biochemical model can be taken as a series of ODEs and solved using the Euler Method. The reaction rates described in the model are the first derivatives that we need in order to apply the

method, they represent the rates on which the chemical species change. After a simulation step all the species values are updated.

The parallelization technique envisioned when using this method in a GPU implementation was a scenario where each thread would be responsible for one reaction, with all reactions in parallel.

1.2. Stochastic simulations

Stochastic simulations are often used to represent physical phenomena, using the randomness present on its core to generate different results for a simulation, and using statistics to obtain the expected behaviour using these distinct results. Multiple simulations need to be executed in order to guarantee statistical significance.

The Gillespie Algorithm approaches the simulation at the molecular level, as a container with a number of molecules, where the randomness comes from the propensity of the molecules to collide with each other which may then trigger a single chemical reaction. After a simulation step only one chemical reaction occurs, therefore only a few species values are updated.

When integrating this algorithm with a GPU the main concept is a method to simulate many biochemical model identical instances in parallel, obtaining several simulation outputs from various executions of the same initial state of a model in one run of the GPU.

1.3. Contents of the document

In the second chapter of this document all the technologies involved in this project are introduced, serving as an introduction to reach and focusing on the most fundamental parts employed in the resulting application.

The third chapter discusses the deterministic method, why it was the first step taken towards simulating, how it was implemented and the problems with this implementation.

The fourth chapter examines the stochastic method. It goes into development details and directly addresses the resulting program.

In the fifth and sixth chapters the developed application is validated and its performance is analyzed. This is done with the support of the COPASI [5] software, an application for simulation and analysis of biochemical models, and utilizes models obtained from the BioModels repository.

The seventh chapter presents the conclusion regarding the results obtained and the feasibility of the developed application.

2. Technologies involved

Multiple technologies were involved in this project, biological models representation for computer applications (SBML [1]), simulation algorithms (Euler, Gillespie [3]), parallel computing platforms (CUDA [4]), existing model simulation software (COPASI [5]). All of which were of great help in developing a solid application.

In the following subtopics these technologies and their contributions are further exemplified.

2.1. Systems Biology Markup Language - SBML

SBML [1] is a format of representing biological models based in XML. The format encapsulates all the necessary components for describing a model and simulating it, having a high quantity of metadata in order to further organize and relate all the information. Its main objective is to create a base for machine-readable models, promoting the integration between computational modelling and biological systems.

Interesting SBML files formatting characteristics include full description of reactions, including reaction rates, which represent the rate that molecules vary dictated by the stoichiometric values, which are the numbers of molecules involved in the reaction. It also contains all values used in the reaction rates calculations, which are the chemical species amounts and chemical constants.

All SBML files used in the development of this dissertation were obtained from the BioModels website, which contains a vast array of these models with associated articles which are great for validating the developed program and for assisting the debug process by giving a better insight on how the model described functions.

2.2. libSBML

libSBML [2] is an open-source library available in multiple programming languages that enables reading, writing, modifying, translating and validating SBML files. It is of great help when accessing the information contained in an SBML file, organizing the model in an object oriented fashion.

libSBML can evaluate mathematical formulas using its eval function, it makes use of the mathML notation used in SBML files and substitutes the variables values in the calculations in order to obtain the mathematical formulas values.

2.3. Euler Method

The Euler Method is a deterministic simulation method. It is an iterative method that solves ODEs. It works by approximating the next values of the functions at a future time by using the first derivative and a time step which influence the approximation result values. The method is described in the next paragraphs.

$$let\ y'(t)\ be\ the\ reaction\ rate\ equations$$

$$and\ the\ initial\ values\ be\ y(t_0) = y_0$$

$$given\ a\ step\ size\ h$$

$$we\ can\ calculate\ y_{n+1} = y_n + h \cdot y'(t_n)$$

$$and\ y_n \approx y(t_n)$$

In order to start the algorithm the first derivative values of all the functions are needed. In this case these formulas are given by the reaction rate equations of each reaction. Also, the initial values of the functions are necessary. In our model this is represented by the initial chemical species concentration values.

Then, for each reaction, given a time step *h*, we can calculate the values of the species after a step is made. This is done by adding to the value previous to the step the derivative value multiplied by the time step. By repeating this calculations one can advance the simulation and obtain species values for any desired time.

This method must be used carefully because it can have stability problems. Stable ODEs may become unstable when simulated with the Euler Method. On these occasions other methods are needed.

2.4. Gillespie Algorithm

The Gillespie Algorithm [3] is a stochastic simulation method, a variant of the Monte Carlo Method, that was first used to simulate chemical reactions. Its main concept is using the number of molecules instead of the concentrations to describe the components in a reaction, simulating the process at the level of collisions between the molecules that are responsible for chemical reactions.

The algorithm works as follows:

Step 1, system initialization, the number of molecules, and reaction constants are initialized. The propensities values are calculated using the reaction formulas.

Step 2, is the Monte Carlo Step: Using a random number generators two values are defined. First, using a uniform distribution based on the propensities values, which reaction is going to occur is determined. Second, using an exponential distribution using the sum of all propensities, the time step associated with such reaction is determined.

Step 3, the simulation time is increased by the time step calculated previously and the numbers of the molecules are updated based on which reaction was chosen, considering a singular occurrence of such reaction.

Step 4, go back to step 2 and repeat the process, unless a stop condition is met, which could be a certain amount of iterations or a final simulation time, for example.

2.5. CUDA

CUDA [4] is a parallel computing platform and programming model developed by NVIDIA that enables the use of compatible GPUs for general purpose applications.

The atomic unit of a CUDA process is a thread, where a single instance of a program runs. Each thread has its own local memory, accessible only by itself, which is the fastest accessible memory. These threads are organized in groups that are called blocks, these blocks share some memory exclusive only to the threads on the same block. This memory is the best one in terms of access times if memory must be shared between threads, but has higher latency than local memory. A grid is composed by a set of blocks, and a single GPU may have more than one grid. In order to exchange information between blocks and grids there is a global memory. It is the most costly to access but it has the advantage of being accessible to all threads. Managing these memory aspects is crucial in order to optimize the execution of a program using a graphics chip, choosing the fastest available memory while still avoiding data hazards is essential for a GPU application.

The multiprocessor also partitions blocks into warps, which are groups of 32 threads. These warps are assigned threads in an ordered fashion, meaning that the warp with ID 0 contains threads

with IDs 0 to 31. A warp executes one single instruction per cycle, meaning that all the threads inside a warp execute the same instruction on a cycle.

Another main aspect of CUDA programming is the thread parallelism, the GPU processes data using steps that are shared between all threads in a warp, running the same instruction simultaneously. As a result of this architecture a GPU program execution must be as linear as possible, avoiding ramifications on the execution order that cause multiple ramifications to be taken, since when branching happens, while we execute one of the ramifications all other ramifications are idly waiting for their turn.

When programming in CUDA the functions are developed for execution in either the CPU (host) or the GPU (device). Kernels are specific functions that can be executed in a parallelized fashion in the GPU, these functions are called by the CPU which then launches the GPU accordingly. In a typical CUDA implementation the host is responsible for memory management while the device does the bulk of data processing, usually the CPU is responsible for the global memory initialization, allocating all the global memory used by the GPU, and then launching the kernel, beginning the parallel execution.

2.6. COPASI

COPASI [5] is a software application for simulation of biochemical models, it has multiple uses and applications but the main characteristic relevant to this project is the ability to read and simulate models using an SBML file. In its large array of functions COPASI can also simulate the models using the Gillespie Algorithm.

Another function that will be used in this study is the ability of running multiple simulations in sequence, which will be used as an estimate for how long it takes to simulate a number of iterations using only the CPU.

3. Deterministic method application

3.1. Motivation

During the first steps of development the objective was to experiment with simulation possibilities and to obtain a better understanding of all the tools needed. As a starting point the intent was to create a serial and a parallel version of the same algorithm and compare how their performance values correlate with different magnitudes of biochemical models.

The Euler Method was chosen due to its widespread usage when dealing with deterministic approaches, as an added benefit, it was also a comfortable method to work with, due to its familiarity.

3.2. The application developed

The steps in which the application progressed will focus on two main versions in order to better organize the process of development.

3.2.1. Serial implementation with libSBML support

The very first version of the application was a straightforward serial implementation of the Euler Method, using libSBML [2]. It contributed in being a firsthand experience on handling SBML [1] files with the aid of the libSBML resources. A great resource provided by libSBML is an evaluation function that could calculate the mathematical expressions making the development uncomplicated.

3.2.2. Parallel implementation

With a working serial version, the code was analyzed in order to identify possible parallelizable regions to be implemented using CUDA [4]. However, executing CUDA code and using libSBML in the same file is not possible, because CUDA programs require specific libraries created exclusively for GPUs, and libSBML does not support CUDA.

The solution found to circumvent this problem was the usage of metaprogramming, where a program generates a program code that can then be compiled and executed. This approach uses libSBML first, extracting all the data necessary from the SBML file, and then uses this information to generate another program, a CUDA program, the resulting code does not have any dependencies with libSBML functionalities. Metaprogramming is characterized by having a code generating another code.

The deterministic method was parallelized in this version by distributing each reaction increment values calculation to one thread, attempting to calculate all update values in parallel and therefore obtaining all updated species values in one instruction cycle.

3.2.3. Limitations of the model

When further analyzing the performance of the aforementioned implementation an issue was found due to the ramifications in processing done by that approach. In order to assign each reaction to each thread to execute their designated calculations a switch case statement was used; however, this technique creates the worst possible ramification tree for the GPU. Since each specific reaction calculation is done once and only once, during this step in the execution only one of these calculations are done at any given moment, implying that only one thread is executing while all the others are idle, waiting for their opportunities.

This happens due to the choice made during development to use only one block of the GPU, which grouped threads into warps and made these warps have a longer execution time due to the limitation of executing the same instruction per warp group per cycle. Having this same instruction per warp limitation with the ramifications in the code instructions caused the execution in the GPU to have only one thread executing its desired instruction while all the others in the warp group were idle waiting for their turns, where they would be running exclusively as well.

This represented a roadblock in development. While researching other approaches for simulating the model a suggestion was made to have a look on stochastic methods, and figure out if they were more fit for the results this endeavor aims to obtain.

3.3. Lessons learned

Although the program version developed during this pursue of the deterministic model approach is not used in the final application the experience provided by this ordeal was fundamental to reach the final version, which was inspired by lessons learned in this implementation.

The usage of metaprogramming is a characteristic that was kept in the stochastic version, fundamental for making the libSBML and CUDA work together. And the learning experience with the problems of ramification gave a hands-on lesson on how architectures that are suitable for CPU execution sometimes must be avoided when developing for GPU applications.

4. Stochastic method application

4.1. Motivation

Given the independent nature between simulations in a stochastic process the usage of a GPU and its capabilities is quite fitting, the only bottleneck that happens between simulations is the access to the global memory in order to register the results of the simulation, but that represents an unavoidable cost that is an integral part of the process. However, in any other instance of a simulation, only local memory is involved, which is the fastest memory available for the graphics chip.

With that in mind the development of the program was done, the resulting application being not very user friendly, but having in its core the functionalities envisioned, and bearing in mind the possibilities on future stochastic simulation software.

4.2. The application developed

In order to join functionalities of the libSBML [2] and CUDA [4], which do not support each other, a metaprogramming approach was used. A C program (*generateCUDAGillespie.c*) utilizes the libSBML functionalities to extract the information present in a given SBML [1] file in order to generate a CUDA program (*kernelGill.cu*) as the output. The resulting CUDA application then can be run on a CUDA enabled GPU, returning the results of the simulation of the biological model in a CSV file (results.csv).

In a regular use, the file *generateCUDAGillespie.c* is compiled, and the resulting executable is run, passing as parameters the name of an SBML file, the sample interval size in seconds, the final simulation time in seconds, and the number of different simulations. After the program finishes it will have produced several text files, as such, the folder where the executable was run from will now contain 16 code segments.

Afterwards, in this same folder, the script *concatGill.bat* must be run. This script executes a simple operation joining all the segments that were created and generating a new file *kernelGill.cu*, which is the CUDA program that can subsequently be compiled and run on the GPU.

4.3. Development and functionality

In this application two program files are involved, first, *generateCUDAGillespie.c* is a C program that uses libSBML and metaprogramming in order to read a given SBML file and generate the

second program, *kernelGill.cu*, which is a CUDA program that simulates the SBML model read by the first program in a GPU.

4.3.1. CUDA program description - kernelGill.cu

The execution starts at the CPU, where the arrays *dev_output* and *species_global* are allocated in the global memory, which is shared between the CPU and the GPU. The initial species values (the quantity of the molecules) are copied to the *species_global* array, while the *dev_output* array is initialized with all zeros (this is the array were the simulation results will be registered).

Another array *devStates* is allocated in the global memory, this array is responsible for saving the state of the pseudorandom generator for each thread. The kernel function *initCurand* is called, which simply initializes the random number generator with a different seed for each thread (default seed is 23).

The main simulation is now ready to start, the simulation is partitioned in fixed intervals (arbitrary default interval of 10 000s), this has the purpose of giving a feedback of the execution times to the user. In addition, it reduces the kernel run time avoiding problems with hardware drivers. A kernel run is performed for each of these intervals.

Finally, on the GPU a local array *species* is created and initialized with the values saved in the *species_global* array previously mentioned. These values are stored in local memory because they are unique for each thread. Also, they are extensively used which requires a fast access time. Next two local matrices are created *reactionSpecies* and *reactionValues*, they work as a key value pairs, enabling a quick access to the species involved in a reaction and their stoichiometric values (the number of molecules involved in a chemical reaction) and they are essential in order to avoid ramifications in the execution, guaranteeing a linear approach.

After initializing the local variables, the simulation loop begins, it goes on until the segmentation time or the final execution time is reached. In this loop, it is checked if the sampling interval was reached, if so the *species* array values are added atomically to the *output* array in the corresponding position and the next sampling interval is updated, this is done in order to calculate the average values of the species at each sampling time, and avoids data hazards of multiple threads updating the same memory position at the same time. Next, the propensity of each reaction is calculated cumulatively (which are represented by the reaction rates in the SBML file), in other words, the next reaction propensity is their own propensity formula plus the sum of all the previous calculated propensities values. Then the time step is randomized, it uses an exponential distribution with lambda

equal to the total sum of propensities. The reaction is chosen by using a uniform distribution with the interval between zero, included, and the total sum of propensities, excluded, and by finding the index where this random number is lower than the cumulative propensity of said index. In order to find such index a binary search is implemented, exploiting the fact that the cumulative propensities are ordenated. Unfortunately this method adds a corner case where if a reaction propensity is zero it has the same cumulative value as the previous one and therefore it can be selected by the binary search; this issue is resolved with an extra verification where it is required that the random number generated is also higher or equal to the accumulated propensity of the previous index. However, this generates another corner case where the reaction with index zero can never be validated, this is resolved by adding a verification step that checks if the selected reaction is the first one and if not then the binary search is made. This generates some level of ramification in the execution, but represents an unavoidable part of the algorithm. A search must be done in order to obtain the selected reaction, and the binary search is a great fit in an attempt to reduce ramification issues, in comparison to simpler, more naive search methods.

With the time step and the reaction index values at hand, by using the reaction index the matrices *reactionsSpecies* and *reactionsValues* are used to update the quantities of the species and then the simulation time is updated with the time step.

In the case where a reaction is selected and the number of reactants consumed is higher than one, it could be possible to cause a negative specie quantity. To avoid that the application verifies the resulting species quantities after the update and reverts their values if a negative quantity is found. The simulation then goes on as if, on that interaction, no reaction occurred.

At this stage it is also verified if any event was activated. Events represent external interferences to the system, and are declared in the SBML file with a condition and an update formula, when the condition is met the update formula is executed. For example, in one of the models presented, on the simulation results section, 50 molecules of the species antiAb are added when the simulation reaches the fourth day, simulating a drug administration.

At the end of one simulation loop (kernel run) the *species* array values are saved in the global array *species_global*, in order to be able to restart the simulation in the same condition that it was in the case of this simulation being one of the intermediate partitions. Each thread has enough space in the global memory in order to register all its species values, since all these values are unique for each thread.

After executing all the segments, the program continues on the CPU. The data registered in the global array *dev_output* are saved in the hosts local memory, these values are then divided by the number of threads that were run, in order to obtain the average values of each species in each time sample. Finally they are exported to an external file using the CSV format.

The application was developed using the SBML file provided together with the paper "Investigating Interventions in Alzheimer's Disease with Computer Simulation Models" [6]. Some of the constants used are optimal for this model and need to be changed to better fit other biological models. Namely, these constants are the sampling time interval, the final simulation time and the segmentation time size. Later other models [7] were used to validate the accuracy and precision of the application, and these constants had to be adjusted for each.

4.3.2. C program description - generateCUDAGillespie.c

This is the program responsible for the metaprogramming segment of the application. It generates 16 text files (listed on table 1) with code segments that are fused together by a batch script, resulting in the CUDA file *kernelGill.cu*.

Table I: List of all the files generated by the C program *generateCUDAGillespie.c*

Files generated by *generateCUDAGillespie.c*			
header	defineConstants	kernelFunction	kernelVariablesInit
defineSpeciesUpdates	updatePropensities	kernelWriteInGlobal	curandInit
initMain	initializeSpecies	curandCall	kernelCall
receiveData	exportResults	freeDevice	endMain

The application receives as execution parameters the SBML filename, the sampling time interval in seconds, the final simulation time in seconds and the number of simulations to be run in parallel. These parameters must be adjusted accordingly to each model characteristics. Another parameter that may be adjusted is the constant *SEGMENT_SIZE* that defines the segment size for the simulation partitioning, but this characteristic is only needed for the user feedback behaviour on the application and is not crucial for the simulation to run properly.

At the start of execution, the SBML file is read and a model is created by libSBML that enables manipulating all the information present in the SBML file in an object oriented fashion (in C this is done in the coding level by the use of Structs).

With the model in a format were the data can be easily extracted the code generating part begins. While iterating on some information in the model it is common for the application to edit multiple of the output text files at once in order to utilize the iterations as much as possible. This does not mean, however, that the code is completely optimized, since the main interest is the CUDA program generated by this one it was not a high priority to fully optimize this code, and since it is a quite simple data extraction application, its execution times are quick and not an issue.

Main metaprogramming advantages explored include the usage of the species IDs as variables in the CUDA code, making a quite legible product that if needed can be modified directly since navigating is done easily. Another advantage is the direct usage of the mathematical formulas obtained by using libSBML in the CUDA program, this avoids the need of an infix notation evaluator in order to calculate these formulas.

Some functions that exist in SBML models that were stumbled upon and were not implemented are:

- non-constant parameters, where the reaction parameters constants that are used can be changed;

- function definitions, where mathematical functions that can be called with some parameters to define a mathematical formula can be defined;

- initial assignments, where initial amount values can be overridden and species values can be changed by assigning them a formula to be calculated at instant zero;

- verification of measurement units, the application assumes that all values have coherent units, so that no unit conversions is required.

4.4. Implementation decisions

In this section decision choices are discussed., Some aspects of the stochastic simulation needed clever solutions in order to try to optimize the application.

4.4.1. Metaprogramming usage

Using the resources provided by the libSBML library were of great help manipulating SBML files, but integrating this library directly with the CUDA programming is a difficult task, CUDA requires specific libraries implemented specifically for GPUs, which is not the case for libSBML. By using metaprogramming these issues are avoided, and the resulting program is independent from any libSBML resource, being a pure CUDA program that can be modified on itself. Metaprogramming also benefited the processing of the formulas present in the files, removing the necessity of implementing an evaluator that interprets formulas given in strings.

4.4.2. Segmentation of the simulation

The CUDA program is executed in partitions (default 10 000s for the base model). At first this was done as an attempt to avoid a problem that caused a timeout it the communications between the device and the host. As this problem was resolved the functionality was kept as a convenience at first, but which later has been proven to be useful on debugging steps, and providing a more user friendly interface, sacrificing some small execution time in order to give some feedback to the user about the progress of the simulation.

If, for some reason, the user wishes to run the entire simulation in only one GPU call this can be easily done by setting the constant *SEGMENT_SIZE* time to a higher value than the final simulation time, effectively creating only one partition to be run on the GPU.

4.4.3. Usage of the reaction update matrices

In order to improve the linearization of the code in the GPU an implementation was developed using an approach with matrices, available in local memory for quick access, these matrices *reactionsSpecies* and *reactionsValues* act as a key value pair in such a way that given the index of a reaction, accessing the data containing which species are involved and how their quantities change can be done directly, avoiding the need of any kind of search algorithm to find the information.

This implementation has the benefit of being a more linearized approach and therefore a quicker approach for the GPU. However, it consumes more memory in order to store the update matrices, and most of this memory is redundant, since each thread has their own local copy of the matrices in order to improve memory access times (since these matrices are accessed every iteration of the simulation). One approach that was not tested was using the constant memory, by declaring a constant variable in the GPU memory, this could save local memory and lead to improvements.

If, by some reason, the local memory cannot be used the *reactionsSpecies* and the *rectionsValues* matrices could be stored in shared or global memory as well, but the program will have a higher runtime due to the slower memory access.

Other option, that does not require extra memory, is using different codes for each chosen reaction. However, this would result in code ramifications as in the Euler Method implementation and in a bad performance in the GPU.

4.4.4. Binary search

In the step of finding which reaction needs to be chosen for one iteration of the model simulation it is required to make a search in the values of the propensity of each reaction, finding in which index the value created by the random number generator falls. This propensity data is organized in a cumulative way, meaning that the value of the next propensity is its own value plus the sum of all the previous propensities that were calculated, creating a naturally organized data array, which is made in such a fashion with the purpose of facilitating finding the reaction.

As a naive implementation one could simply sweep the array from index zero until it finds the threshold value where the value at a given index is higher than the value searched, obtaining such value as the desired result. However, in a worst case scenario, for an array of size N at max N calculations will be made, and for large arrays this approach is quite time consuming and it does not fully utilizes the organization structure of the data array.

5. Validating the application

In order to validate the data outputted by the program, SBML [1] model files were used to compare the results between the COPASI [5] output and the developed application output.

The models used are:

- Proctor2013 - Effect of Aβ immunization in Alzheimer's disease [6];
- Proctor2005 - Actions of chaperones and their role in ageing [7].

The COPASI software allows to do another simulation of the SBML model and compare and validate the results.

5.1. Comparing the results

Here results from different available SBML models are compared between the output produced by the COPASI software and the output produced by the developed application. The analysis is done by visual inspection to verify the main behaviour of the system and also by checking the maximum error percentages for each species during the simulation.

The error average was calculated using the absolute difference between the COPASI and the CUDA [4] application results for each time in relation to the maximum species value reached by the COPASI simulation.

For each species and each time sample:

$$\text{AverageError} = \frac{\frac{1}{N}\sum_{i=1}^{N} |\text{COPASI}(i) - \text{APPLICATION}(i)|}{\max_{i} \text{COPASI}(i)}$$

With *i* representing each simulation executed and N representing the number of simulations executed.

5.1.1. Proctor2013 - Effect of Aβ immunization in Alzheimer's disease [6]

The article "Investigating Interventions in Alzheimer's Disease with Computer Simulation Models" [6] simulates some metabolic pathways related to Alzheimer's Disease using the Gillespie Algorithm, and has an SBML file attached that enables the reproduction of the results obtained in this study, making it a great fit for testing and validating the application. Its BioModels ID is MODEL1704060000.

Some highly sensitive species show a high percentage error average, this occurs due to the fact that these species are present in very low quantities, with some being unitary at some times. This model also has a high variance between simulations, for example, the species *damDNA* represents damaged DNA, and changes in its quantity imply drastic changes in the simulation behaviour.

Due to this high variance in the species values a better comparison can be done visually and, when inspecting the graphical representation, it can be clearly seen that the main behaviour of the model is present. The COPASI graph has less noise than the CUDA program graph, both were generated using 128 simulations.

Table II: Average percentage difference for species when comparing COPASI average values with the developed application average values

Species	Average difference	Species	Average difference	Species	Average Difference
Mdm2	8.1%	Mdm2_p53_Ub	22%	GSK3b_p53_P	9.1%
p53	7.4%	Mdm2_p53_Ub2	21%	AggAbeta_Proteasome	9.4%
Mdm2_p53	1.3%	Mdm2_p53_Ub3	19%	Abeta	6.3%
Mdm2_mRNA	7.2%	Mdm2_p53_Ub4	27%	AbetaPlaque	12%
p53_mRNA	2.4%	Mdm2_P1_p53_ Ub4	3.2%	Tau	17%
ATMA	7.5%	Mdm2_Ub	23%	Tau_P1	10%
ATMI	0.48%	Mdm2_Ub2	29%	Tau_P2	9.9%
p53_P	7.6%	Mdm2_Ub3	26%	MT_Tau	0.26%
Mdm2_P	8.6%	Mdm2_Ub4	7.2%	Proteassome_Tau	9.8%
ROS	9.6%	Mdm2_P_Ub	11%	Proteasome	0.18%
E1	0.98%	Mdm2_P_Ub2	10%	PP1	0%
E2	1.1%	Mdm2_P_Ub3	15%	NFT	4.0%
E1_Ub	0.98%	Mdm2_P_Ub4	8.6%	AbetaDimer	14%
E2_Ub	1.2%	Mdm2_Ub4_ Proteasome	11%	disaggPlaque1	13%
GliaI	0.055%	Mdm2_P_Ub4_ Proteasome	9.3%	Ub	0.49%
GliaM1	5.9%	GSK3b	0.25%	damDNA	7.2%
GliaM2	31%	GSK3b_p53	7.5%	p53_Ub4_Proteasome	3.6%

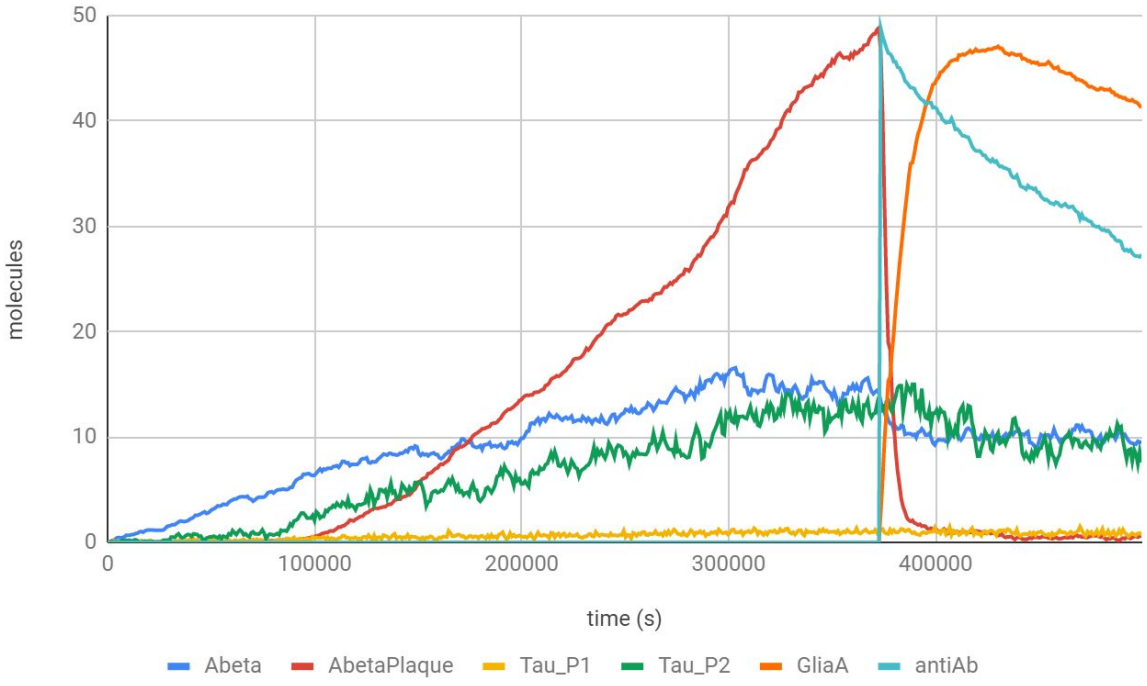

Figure 1: Graph generated by the CUDA program containing species of interest

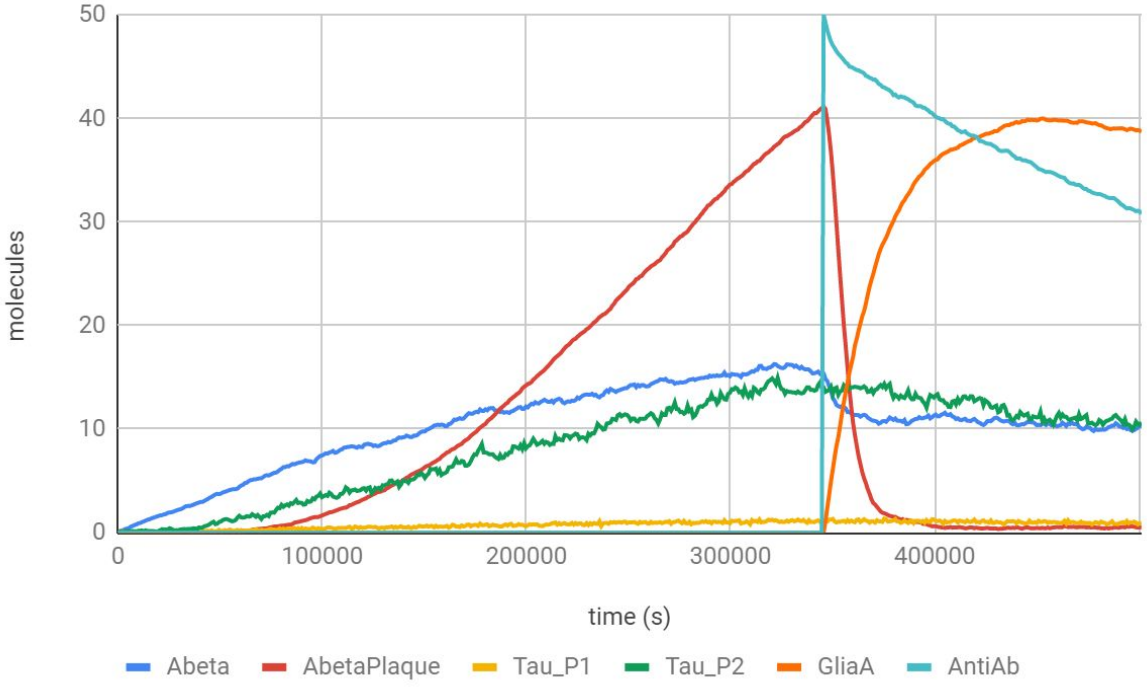

Figure 2: Graph generated by the COPASI program containing species of interest

5.1.2. Proctor2005 - Actions of chaperones and their role in ageing [7]

The article “Modelling the actions of chaperones and their role in ageing” [7] also has an attached SBML file. The main objective was to have a different model to validate the CUDA program, observing how it behaves with different models of the same type and showing that it is an adequate application for this type of model.

The average errors values are all good, with the exception of the MisP species, when analysing this particular species it is noticed that its values are quite close to zero, ranging from 0 to 0.5 at maximum. Given that the Gillespie Algorithm works with full molecule count it is expected to have a highly unstable average value when the averages are too close to zero and some misrepresentation of the model might occur.

The graphs inspection, on the other hand, shows that, on the interesting species where we can observe the main behaviours of the model, the data is very coherent. The other species not shown in the graphs have quite constant values, and those match between COPASI and the CUDA program as well.

Table III: Average percentage difference for species when comparing COPASI average values with the developed application average values

Species	Average difference	Species	Average difference	Species	Average difference
Hsp90	0.040%	MCom	0.13%	ATP	0.022%
HCom	0.0060%	NatP	0.0020%	ADP	0.24%
HSF1	0.36%	X	2.2%		
MisP	3300%	ROS	0.044%		

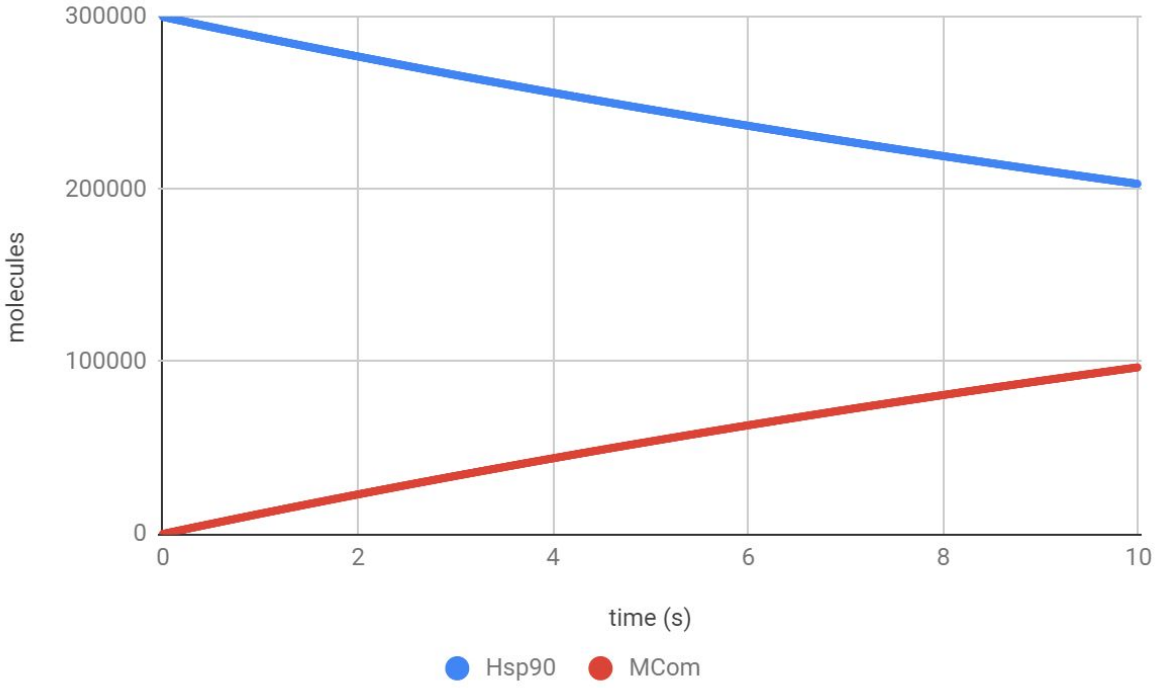

Figure 3: Graph generated by the CUDA program

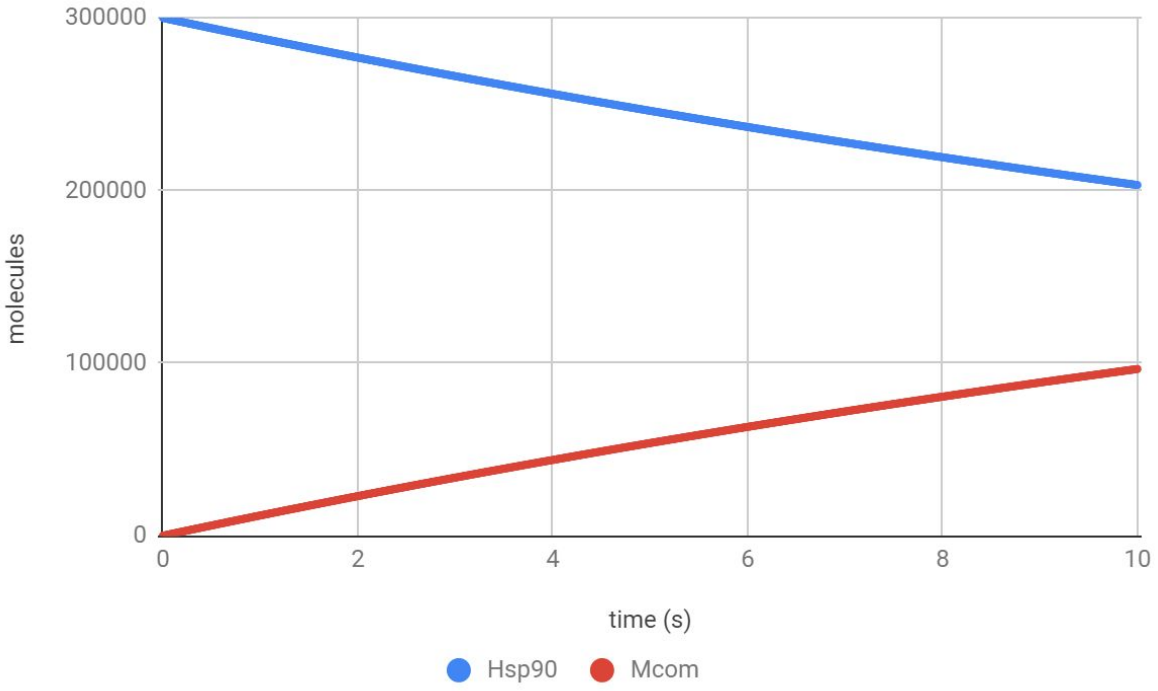

Figure 4: Graph generated by the COPASI program

6. Performance analysis

In order to evaluate the performance of the developed application we are going to compare its execution times with an application that runs only on a CPU. This will be done using COPASI [5] and its feature that enables multiple sequential executions of the simulation.

All simulations were run on the same computer, a Windows 8.1 64-bit, with an Intel Core i3-2100 CPU, a clock rate of 3.10 GHz and 4.00 GB of RAM. The GPU is a GeForce GTX 970.

6.1. Effect of Aβ immunization in Alzheimer's disease [6]

6.1.1. CPU only run times

These times were obtained by doing a manual measurement. Given the human error involved and the imperfections of this approach these time should be taken with care, and are only used as rough estimates to compare performances.

All simulations were done using the same base SBML [1] model and the Gillespie direct method for exact stochastic kinetics [3], the number of simulations used were 16, 32, 64 and 128 simulations, with an interval size of 900s and a duration of 100 000s. Each time was measured 10 times and the averages obtained are shown in table IV.

Table IV: CPU average execution times for different quantities of simulations, using an interval size of 900s and a duration of 100 000s for each simulation

Quantity of simulations executed	Average execution time (s)
16	23
32	46
64	104
128	253

As one would expect the execution times obtained roughly follow the increase in the quantity of simulations. Doubling the number of executions doubles the execution time.

6.1.2. GPU simulation run times

These times were obtained using the timing capabilities that are already part of the final application.

All simulations were executed in the same conditions as the ones used in the CPU run time measurement. The application was run for 16, 32, 64 and 128 simulations, with an interval size of 900s and a duration of 100 000s. Since the pseudo random number generator uses a seed value to initialize, this will be changed between measurements. Each time was measured 3 times and the averages are shown in table V.

Table V: GPU average execution times for different quantities of simulations, using an interval size of 900s and a duration of 100 000s for each simulation

Quantity of simulations executed	Average execution time (s)
16	1462
32	1589
64	2707
128	3368

These times are not what one would expect for a parallel program, similar values were expected for these execution times. This behaviour happens due to the model characteristics, it is highly volatile, meaning its executions are not very uniform. The model has a *damDNA* species that increases the number of molecules in the simulation proportionally to its value, making the simulation take longer to execute by decreasing the time step values. Due to this unstable behaviour the time each simulation takes varies considerably, in a warp we can observe threads that have already finished simulating but that are waiting for another thread that is taking longer.

In the case of the GPU simulation, the running time is at best the worst case time of all the simulations simulation times, while in the CPU is the sum of all simulation times. For this model the worst case time gets worse as the number of simulations increases.

6.1.3. Comparative analysis

By extrapolating the execution times obtained with a linear trend line, it shows that the CUDA [4] program always has a higher execution time than the COPASI. After a further analysis of the model it was found that this model in particular has an uneven execution, making a couple threads much

slower than the others and compromising the entire parallel execution, since all other threads must wait for the slower ones.

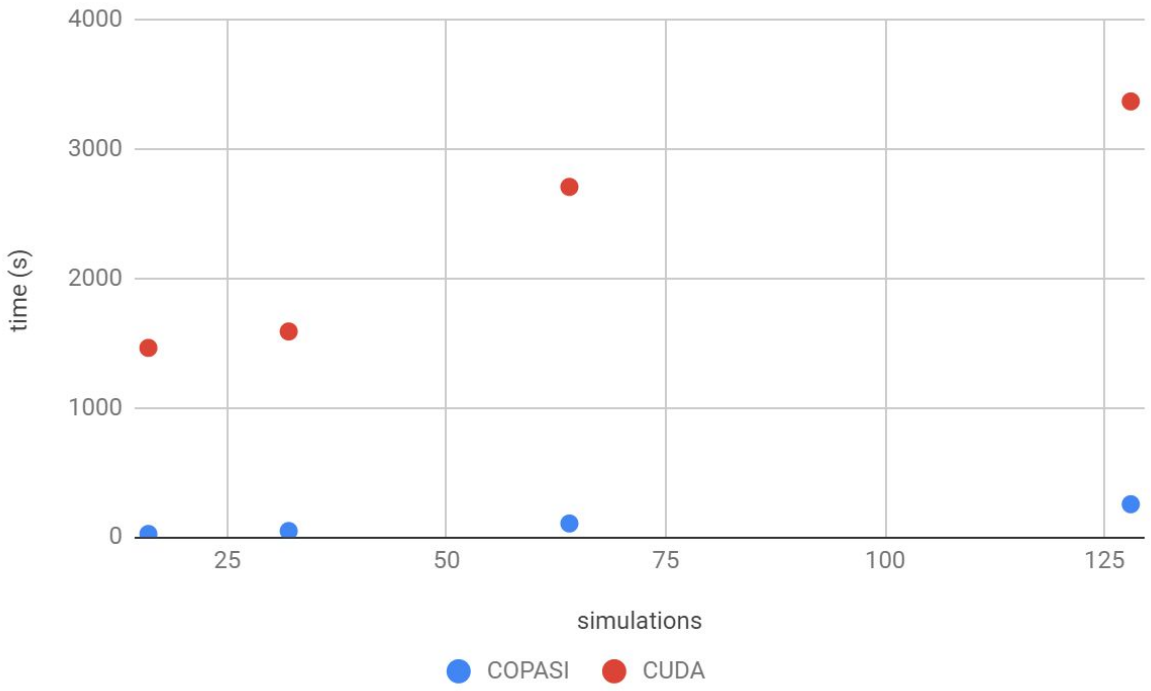

Figure 5: Trendline comparing the COPASI and the CUDA application times

6.2. Actions of chaperones and their role in ageing [7]

This model was chosen because it is more stable than the previous, it does not have behaviour changing random elements.

6.2.1. CPU only run times

An interval size of 0.1s and a duration of 100s was used. The times were measured for 16, 32, 64 and 128 simulations.

Table VI: CPU average execution times for different quantities of simulations, using an interval size of 0.1s and a duration of 100s for each simulation

Quantity of simulations executed	Average execution times (s)
16	43
32	85
64	175
128	344
220	576

6.2.2. GPU simulation run times

The same parameters were used in the GPU simulation, using a single block.

Table VII: GPU average execution times for different quantities of simulations, using an interval size of 0.1s and a duration of 100s for each simulation

Quantity of simulations executed	Average execution times (s)
16	501
32	512
64	541
128	547
220	555

6.2.3. Comparative analysis

Once more the data was extrapolated, but on this model the behaviour of the simulations is more uniform, since it does not have an unexpected change between simulations behaviour the extrapolation shows promising results, having a breakeven point around 220 simulations where after this point the CUDA application becomes faster than the COPASI.

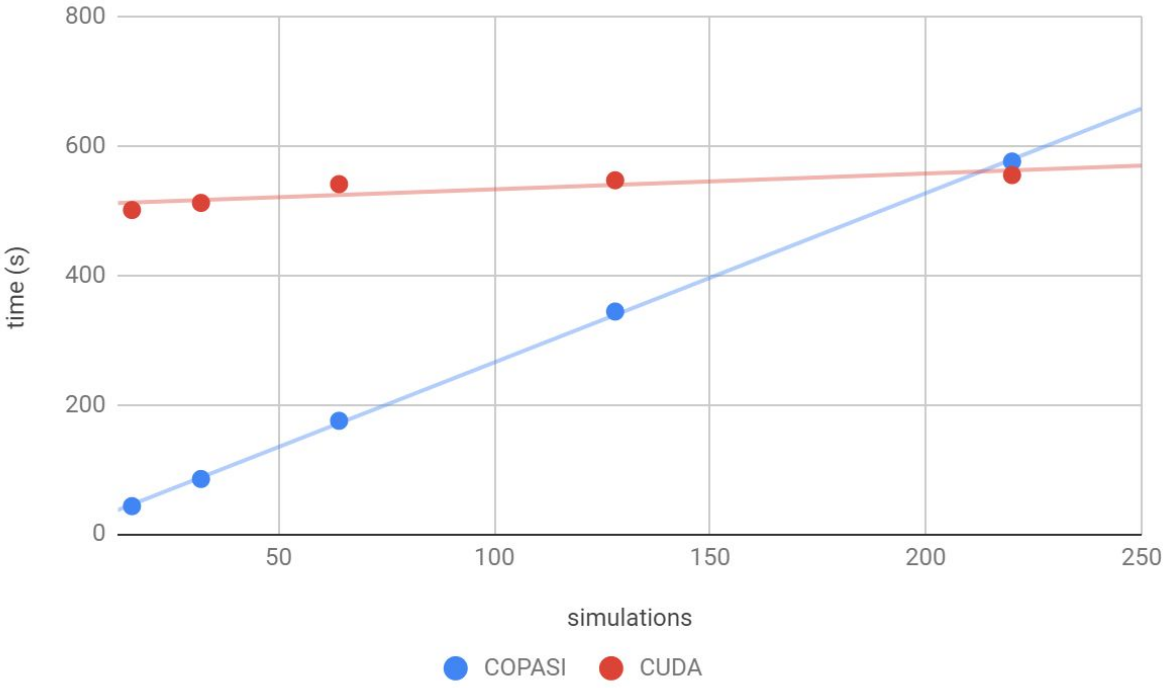

Figure 6: Trendline comparing COPASI and CUDA application execution times

7. Conclusions

The application developed with this document represents a first step in simulating large sets of metabolic reactions in a GPU. It successfully simulates SBML [1] models that are within its developed scope, but has some shortcomings when dealing with all kinds of SBML files.

The results obtained show that the CUDA [4] application deals well with well behaved models, where the randomness of the stochastic simulations do not change drastically the course of the model. On the same note, the results present an issue with the parallelization method implemented, it does not thrive with models with drastic, randomness related, changes, in which case the COPASI serial application showed better execution times.

The final version of this program has some points that could be improved in future versions, to create a more universal and robust version.

7.1. Improvement points

Some shortcomings of the application as a whole, that if implemented would increase the quantity of models supported and improve the GPU usage.

7.1.1. SBML related improvements

- support for non-constant parameters, the reaction parameters constants that are used can be changed;

- support for function definitions, mathematical functions that can be called with some parameters to define a mathematical formula can be defined;

- support for initial assignments, initial amount values can be changed by assigning a formula to be calculated at instant zero;

- support for verification of measurement units, the application assumes that all values have coherent units.

7.1.2. CUDA related improvements

- support for usage of more than one block and for more than one grid in the GPU, increasing the number of simulations that can be run at once;

- support for scientific format values, allowing very small constants to be declared. At the moment only constants until 10^{-10} are supported;

- support for a better distribution of threads between blocks, making use of the maximum capacity of the GPU.

8. References

[1] M. Hucka, et al. The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models, Bioinformatics, Volume 19, Issue 4, 1 March 2003, Pages 524–531, https://doi.org/10.1093/bioinformatics/btg015

[2] Bornstein BJ, Keating SM, Jouraku A, Hucka M. LibSBML: an API library for SBML. Bioinformatics. 2008;24(6):880–881. doi:10.1093/bioinformatics/btn051

[3] Gillespie, Daniel T.. “Exact Stochastic Simulation of Coupled Chemical Reactions.” The Journal of Physical Chemistry 1977 81 (25), 2340-2361 DOI: 10.1021/j100540a008

[4] NVIDIA CUDA, “NVIDIA CUDA C Programming Guide”, NVIDIA CUDA, May 2019. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/ [Accessed: July 17, 2019]

[5] Stefan Hoops, Sven Sahle, Ralph Gauges, Christine Lee, Jürgen Pahle, Natalia Simus, Mudita Singhal, Liang Xu, Pedro Mendes, Ursula Kummer, COPASI—a COmplex PAthway SImulator, Bioinformatics, Volume 22, Issue 24, 15 December 2006, Pages 3067–3074, https://doi.org/10.1093/bioinformatics/btl485

[6] Proctor CJ, Boche D, Gray DA, Nicoll JA. Investigating interventions in Alzheimer's disease with computer simulation models. *PLoS One*. 2013;8(9):e73631, 16 Sep. 2013. doi:10.1371/journal.pone.0073631

[7] Proctor CJ, Soti C, Boys RJ, Gillespie CS, Shanley DP, Wilkinson DJ, Kirkwood TB. Modelling the actions of chaperones and their role in ageing. Mech Ageing Dev. 2005 Jan;126(1):119-31. PubMed PMID: 15610770.

[8] Hatakeyama M. “Autocatalytic Reaction System by Gillespie's Algorithm (Direct Method)“ June 2010. [Online]. Available: http://masa.o.oo7.jp/dm/ [Accessed: July 17, 2019]

Appendix A

generateCUDAGillespie.c

```
#include <stdio.h>
#include <stdlib.h>

#include <sbml/SBMLTypes.h>

#include <math.h>
#include <stddef.h>
#include <string.h>

#define SQR(x) ((x)*(x))
#define SQRT(x) pow((x),(.5))

#define SEGMENT_SIZE 10000

void generateCUDA(Model_t* m, double step, int simulations, double endTime) {
        //get compartments and initial concentrations in key value pairs
        ListOf_t* species = Model_getListOfSpecies(m);
        ListOf_t* compartments = Model_getListOfCompartments(m);
        ListOf_t* parameters = Model_getListOfParameters(m);
        char* Key;
        double Value;
        FILE* updatePropencities = fopen("GILLupdatePropencities", "w");          //contains
metaprogramming for updating the propensities values
        FILE* defineConstants = fopen("GILLdefineConstants", "w");
//contains constants for the simulation, containers, local parameters and global paramenter
        FILE* initializeSpecies = fopen("GILLinitializeSpecies", "w");                    //declares
a variable and associates the value for each species, also creates and initializes devices variables
        FILE* kernelFunction = fopen("GILLkernelFunction", "w");                          //device
function, loops around a simple Runge-Kutta simulation
        FILE* kernelCall = fopen("GILLkernelCall", "w");
//declares the call for the kernel function
        FILE* kernelVariablesInit = fopen("GILLkernelVariablesInit", "w");
//initializes the variables for the kernel function
        FILE* kernelWriteInGlobal = fopen("GILLkernelWriteInGlobal", "w");                //writes
the variables in global memory for next kernel execution
        FILE* receiveData = fopen("GILLreceiveData", "w");
        //contains code that receives data from device to host
        FILE* freeDevice = fopen("GILLfreeDevice", "w");
//contains free declarations for the device variables
        FILE* exportResults = fopen("GILLexportResults", "w");
//prints each species with the values received from device
        FILE* curandInit = fopen("GILLcurandInit", "w");
//initializes random number generator
        FILE* curandCall = fopen("GILLcurandCall", "w");
//calls random number generator initializer
        FILE* defineSpeciesUpdates = fopen("GILLdefineSpeciesUpdates", "w");      //defines values
for each species that must be updated after a reaction occurs
        FILE* header = fopen("GILLheader", "w");
```

```
//contains header definitions                              --SAME FOR EVERY SBML FILE
FILE* initMain = fopen("GILLinitMain", "w");
//contains main function declaration             --SAME FOR EVERY SBML FILE
FILE* endMain = fopen("GILLendMain", "w");
        //contains main function ending section          --SAME FOR EVERY SBML FILE

//error handling
if (updatePropencities == NULL)
{
        printf("Error accessing updatePropencities!");
        exit(1);
}
if (defineConstants == NULL)
{
        printf("Error accessing defineConstants!");
        exit(1);
}
if (initializeSpecies == NULL)
{
        printf("Error accessing initializeSpecies!");
        exit(1);
}
if (kernelFunction == NULL)
{
        printf("Error accessing kernelFunction!");
        exit(1);
}
if (kernelCall == NULL)
{
        printf("Error accessing kernelCall!");
        exit(1);
}
if (kernelVariablesInit == NULL)
{
        printf("Error accessing kernelVariablesInit!");
        exit(1);
}
if (kernelWriteInGlobal == NULL)
{
        printf("Error accessing kernelWriteInGlobal!");
        exit(1);
}
if (receiveData == NULL)
{
        printf("Error accessing receiveData!");
        exit(1);
}
if (freeDevice == NULL)
{
        printf("Error accessing freeDevice!");
        exit(1);
}
if (exportResults == NULL)
{
        printf("Error accessing exportResults!");
        exit(1);
}
```

```
        if (curandInit == NULL)
        {
                printf("Error accessing curandInit!");
                exit(1);
        }
        if (curandCall == NULL)
        {
                printf("Error accessing curandCall!");
                exit(1);
        }
        if (defineSpeciesUpdates == NULL)
        {
                printf("Error accessing defineSpeciesUpdates!");
                exit(1);
        }
        if (header == NULL)
        {
                printf("Error accessing header!");
                exit(1);
        }
        if (initMain == NULL)
        {
                printf("Error accessing initMain!");
                exit(1);
        }
        if (endMain == NULL)
        {
                printf("Error accessing endMain!");
                exit(1);
        }

        fprintf(header, "#include \"cuda_runtime.h\"\n");
        fprintf(header, "#include \"device_launch_parameters.h\"\n");
        fprintf(header, "#include <cuda.h>\n");
        fprintf(header, "#include <curand_kernel.h>\n");
        fprintf(header, "#include <stdio.h>\n");
        fprintf(header, "#include <stdlib.h>\n");
        fprintf(header, "#include <time.h>\n");
        fprintf(header, "#define CUDA_CALL(x) do { if((x) != cudaSuccess) { \\\n");
        fprintf(header, "printf(\"Error at %%s:%%d\\n\",__FILE__,__LINE__); \\\n");
        fprintf(header, "return EXIT_FAILURE;}} while(0) \n");
        fprintf(header, "#define pow powf\n");
        fprintf(header, "#define SEED 23\n");

        fprintf(initMain, "int main()\n{\n");

        fprintf(endMain, "return 0;\n}\n");

        fprintf(curandInit, "\n__global__ \nvoid initCurand(curandState* state, unsigned long long
seed){\n");
        fprintf(curandInit, "curand_init(seed, threadIdx.x, 0, &state[threadIdx.x]);\n");
        fprintf(curandInit, "}\n");

        fprintf(curandCall, "curandState *devStates;\n");
        fprintf(curandCall, "CUDA_CALL(cudaMalloc((void **)&devStates, %d *
sizeof(curandState)));\n", simulations);
        fprintf(curandCall, "initCurand<<<1, %d>>>(devStates, SEED);\n", simulations);
```

```
	int maxReactionSpecies = 0;
	for (int i = 0; i < Model_getNumReactions(m); i++)
	{
		if (maxReactionSpecies < Reaction_getNumReactants(Model_getReaction(m, i)) +
Reaction_getNumProducts(Model_getReaction(m, i))) {
			maxReactionSpecies = Reaction_getNumReactants(Model_getReaction(m, i)) +
Reaction_getNumProducts(Model_getReaction(m, i));
		}
	}

	fprintf(defineSpeciesUpdates, "int reactionsSpecies[%d][%d];\nint
reactionsValues[%d][%d];\n", Model_getNumReactions(m), maxReactionSpecies, Model_getNumReactions(m),
maxReactionSpecies);
	fprintf(defineSpeciesUpdates, "for(int i = 0; i < %d; i++){\n", Model_getNumReactions(m));
	fprintf(defineSpeciesUpdates, "for(int j = 0; j < %d; j++){\n", maxReactionSpecies);
	fprintf(defineSpeciesUpdates, "reactionsSpecies[i][j] = -1;\n");
	fprintf(defineSpeciesUpdates, "reactionsValues[i][j] = 0;\n");
	fprintf(defineSpeciesUpdates, "}\n");
	fprintf(defineSpeciesUpdates, "}\n");

	//get list of reactions and formulas
	KineticLaw_t *kl;

	int index;
	for (int i = 0; i < Model_getNumReactions(m); i++)
	{
		index = 0;
		//save formulas processing time vs memory size
		if (Reaction_isSetKineticLaw(Model_getReaction(m, i)))
		{
			kl = Reaction_getKineticLaw(Model_getReaction(m, i));
			if (KineticLaw_isSetMath(kl))
			{
				if (i == 0) fprintf(updatePropencities, "cummulative_p[%d] = %s;
\n", i, SBML_formulaToString(KineticLaw_getMath(kl)));
				else fprintf(updatePropencities, "cummulative_p[%d] =
cummulative_p[%d] + %s; \n", i, i - 1, SBML_formulaToString(KineticLaw_getMath(kl)));

				//reactants
				ListOf_t* reactants =
Reaction_getListOfReactants(Model_getReaction(m, i));
				for (int k = 0; k < Reaction_getNumReactants(Model_getReaction(m,
i)); k++) {
					SpeciesReference_t* reactant = ListOf_get(reactants, k);
					if (Species_getConstant(ListOfSpecies_getById(species,
SpeciesReference_getSpecies(reactant))) == 0) {
						fprintf(defineSpeciesUpdates,
"reactionsSpecies[%d][%d] = %s_id; \n", i, index, SpeciesReference_getSpecies(reactant));
						fprintf(defineSpeciesUpdates,
"reactionsValues[%d][%d] = -%.10lf; \n", i, index, SpeciesReference_getStoichiometry(reactant));
						index++;
					}
				}

				//products
				ListOf_t* products =
```

```
Reaction_getListOfProducts(Model_getReaction(m, i));
                                for (int k = 0; k < Reaction_getNumProducts(Model_getReaction(m,
i)); k++) {
                                        SpeciesReference_t* product = ListOf_get(products, k);
                                        if (Species_getConstant(ListOfSpecies_getById(species,
SpeciesReference_getSpecies(product))) == 0) {
                                                fprintf(defineSpeciesUpdates,
"reactionsSpecies[%d][%d] = %s_id; \n", i, index, SpeciesReference_getSpecies(product));
                                                fprintf(defineSpeciesUpdates,
"reactionsValues[%d][%d] = %.10lf; \n", i, index, SpeciesReference_getStoichiometry(product));
                                                index++;
                                        }
                                }

                                for (int j = 0; j < KineticLaw_getNumParameters(kl); j++) {
                                        Parameter_t* p = KineticLaw_getParameter(kl, j);
                                        fprintf(defineConstants, "#define %s %.10lf\n",
Parameter_getId(p), Parameter_getValue(p));
                                }
                        }
                }
        }

        fprintf(defineSpeciesUpdates, "curandState localState = state[threadIdx.x];\n");
        fprintf(defineSpeciesUpdates, "while(time < endTime && time < (numberOfExecutions +
1)*segmentSize){\n");

        //get initial values and constants

        fprintf(kernelFunction, "\n__global__ \n");
        fprintf(kernelFunction, "void simulate (int numberOfExecutions, float* output, curandState
*state, float step, float endTime, float segmentSize, float* species_global");

        fprintf(initializeSpecies, "cudaError_t cudaStatus;\n");

        fprintf(initializeSpecies, "float* output;\n");
        fprintf(initializeSpecies, "float* dev_output;\n");

        fprintf(initializeSpecies, "output = (float*)malloc(%d*%d*sizeof(float));\n",
(int)ceil(endTime / step), Model_getNumSpecies(m));

        fprintf(initializeSpecies, "for(int i = 0; i < %d*%d; i++){\n", (int)ceil(endTime / step),
Model_getNumSpecies(m));
        fprintf(initializeSpecies, "output[i] = 0;\n");
        fprintf(initializeSpecies, "}\n");

        fprintf(initializeSpecies, "cudaStatus = cudaMalloc(&dev_output, %d*%d*sizeof(float));\n",
(int)ceil(endTime / step), Model_getNumSpecies(m));
        fprintf(initializeSpecies, "if (cudaStatus != cudaSuccess) {fprintf(stderr, \"cudaMalloc
failed!\");goto Error;}\n");
        fprintf(initializeSpecies, "cudaStatus = cudaMemcpy(dev_output, output, %d*%d*sizeof(float),
cudaMemcpyHostToDevice);\n", (int)ceil(endTime / step), Model_getNumSpecies(m));
        fprintf(initializeSpecies, "if (cudaStatus != cudaSuccess) {fprintf(stderr, \"cudaMemcpy
failed!\");goto Error;}\n");

        fprintf(initializeSpecies, "float* species_global;\n");
        fprintf(initializeSpecies, "cudaStatus = cudaMalloc(&species_global,
```

```
%d*%d*sizeof(float));\n", Model_getNumSpecies(m), simulations);
        fprintf(initializeSpecies, "if (cudaStatus != cudaSuccess) {fprintf(stderr, \"cudaMalloc
failed!\");goto Error;}\n");

        fprintf(initializeSpecies, "float init_species[%d];\n", Model_getNumSpecies(m));

        fprintf(kernelCall, "clock_t begin = clock();\n");
        fprintf(kernelCall, "cudaEvent_t start, stop;\n");
        fprintf(kernelCall, "float milliseconds;\n");
        fprintf(kernelCall, "cudaEventCreate(&start);\n");
        fprintf(kernelCall, "cudaEventCreate(&stop);\n");
        fprintf(kernelCall, "printf(\"Starting kernel\\n\");\n");
        fprintf(kernelCall, "for(int i = 0; i < %d; i++){\n", (int)ceil(endTime / SEGMENT_SIZE));
        fprintf(kernelCall, "cudaEventRecord(start);\n");

        //This feels weird TODO invenstigate better ways?
        int segmentSize = SEGMENT_SIZE;
        fprintf(kernelCall, "simulate<<<1, %d>>>(i, dev_output, devStates, %.10lf, %.10lf, %d,
species_global);", simulations, step, endTime, segmentSize);

        fprintf(kernelCall, "cudaStatus = cudaGetLastError(); if (cudaStatus != cudaSuccess)
{fprintf(stderr, \"addKernel launch failed: %%s\\n\", cudaGetErrorString(cudaStatus));goto
Error;}\n\n");
        fprintf(kernelCall, "cudaStatus = cudaDeviceSynchronize(); if (cudaStatus != cudaSuccess)
{fprintf(stderr, \"cudaDeviceSynchronize returned error code %%d after launching addKernel!\\n\",
cudaStatus);goto Error;}");
        fprintf(kernelCall, "cudaEventRecord(stop);\n");
        fprintf(kernelCall, "printf(\"SEGMENT %%d\\n\", i);\n");
        fprintf(kernelCall, "cudaEventSynchronize(stop);\n");
        fprintf(kernelCall, "milliseconds = 0;\n");
        fprintf(kernelCall, "cudaEventElapsedTime(&milliseconds, start, stop);\n");
        fprintf(kernelCall, "printf(\"TIME: %%lf\\n\\n\", milliseconds);\n");
        fprintf(kernelCall, "}\n");

        fprintf(receiveData, "\n\ncudaStatus = cudaMemcpy(output, dev_output, %d*%d*sizeof(float),
cudaMemcpyDeviceToHost);\n", (int)ceil(endTime / step), Model_getNumSpecies(m));
        fprintf(receiveData, "if (cudaStatus != cudaSuccess) {fprintf(stderr, \"cudaMemcpy
failed!\");goto Error;}\n");

        fprintf(freeDevice, "Error:\n");
        fprintf(freeDevice, "cudaFree(dev_output);\n");
        fprintf(freeDevice, "cudaFree(species_global);\n");


        fprintf(kernelVariablesInit, "float species[%d];\n", Model_getNumSpecies(m));
        fprintf(kernelVariablesInit, "float aux_species[%d];\n", maxReactionSpecies);
        fprintf(kernelVariablesInit, "int flag_negative;\n");

        for (int i = 0; i < Model_getNumSpecies(m); i++) {
                Species_t* s = ListOf_get(species, i);
                Key = Species_getId(s);
                Value = Species_getInitialAmount(s);
                fprintf(initializeSpecies, "init_species[%d] = %.10lf;\n", i, Value);

                fprintf(kernelVariablesInit, "if(numberOfExecutions == 0){\n");
                fprintf(kernelVariablesInit, "species[%d] = species_global[%d];\n", i, i);
                fprintf(kernelVariablesInit, "} else {\n");
```

```
                fprintf(kernelVariablesInit, "species[%d] = species_global[%s_id*%d +
threadIdx.x];\n", i, Key, simulations);
                fprintf(kernelVariablesInit, "}\n");

                fprintf(kernelWriteInGlobal, "species_global[%s_id*%d + threadIdx.x] =
species[%d];\n", Key, simulations, i);

                fprintf(defineConstants, "#define %s species[%d]\n", Key, i);
                fprintf(defineConstants, "#define %s_id %d\n", Key, i);
        }

        fprintf(initializeSpecies, "cudaStatus = cudaMemcpy(species_global, &init_species,
sizeof(float)*%d, cudaMemcpyHostToDevice);\n", Model_getNumSpecies(m));
        fprintf(initializeSpecies, "if (cudaStatus != cudaSuccess) {fprintf(stderr, \"cudaMemcpy
failed!\");goto Error;}\n");

        fprintf(kernelWriteInGlobal, "state[threadIdx.x] = localState;\n");
        fprintf(kernelWriteInGlobal, "}\n");

        fprintf(exportResults, "FILE* results = fopen(\"results.csv\", \"w\");\n");
        fprintf(exportResults, "if(results == NULL){\n");
        fprintf(exportResults, "printf(\"Error acesssing results!\");\n");
        fprintf(exportResults, "exit(1);\n");
        fprintf(exportResults, "}\n");

        fprintf(exportResults, "clock_t end = clock();\n");
        fprintf(exportResults, "double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;\n");
        fprintf(exportResults, "fprintf(results, \"TOTAL EXECUTION TIME : %%lf\\n\",
time_spent);\n");

        fprintf(exportResults, "fprintf(results, \"time\");\n");

        for (int i = 0; i < Model_getNumSpecies(m); i++) {
                fprintf(exportResults, "fprintf(results, \", %s\");\n",
Species_getId(ListOf_get(species, i)));
        }

        fprintf(exportResults, "fprintf(results, \"\\n\");\n");

        fprintf(exportResults, "for(int i = 0; i < %d; i++){\n", (int)ceil(endTime / step));
        fprintf(exportResults, "fprintf(results, \"%%.10lf\", %.10lf*i);\n", step);
        fprintf(exportResults, "for(int j = 0; j < %d; j++){\n", Model_getNumSpecies(m));
        fprintf(exportResults, "fprintf(results, \", %%.10lf\", output[%d*i+j]/%d);\n",
Model_getNumSpecies(m), simulations);
        fprintf(exportResults, "}\n");
        fprintf(exportResults, "fprintf(results, \"\\n\");\n");
        fprintf(exportResults, "}\n");
        fprintf(exportResults, "fprintf(results, \"\\n\");\n");

        fprintf(kernelFunction, ") {\n");
        fprintf(kernelFunction, "int reaction, stepCount = 0;\n");
        fprintf(kernelFunction, "int indexMin, indexMax;\n");
        fprintf(kernelFunction, "float time = numberOfExecutions*segmentSize;\n");
        fprintf(kernelFunction, "float sum_p, timeStep, random;\n");
        fprintf(kernelFunction, "float cummulative_p[%d];\n", Model_getNumReactions(m));

        fprintf(updatePropencities, "if(time >= segmentSize * numberOfExecutions + step *
```

```
stepCount){\n");

        for (int i = 0; i < Model_getNumSpecies(m); i++) {
                fprintf(updatePropencities, "atomicAdd(&output[%d*%d*numberOfExecutions +
stepCount*%d + %d], species[%d]);\n", Model_getNumSpecies(m), (int)ceil(segmentSize / step),
Model_getNumSpecies(m), i, i);
        }

        fprintf(updatePropencities, "stepCount++;\n");
        fprintf(updatePropencities, "}\n");

        fprintf(updatePropencities, "sum_p = cummulative_p[%d];\n", Model_getNumReactions(m) - 1);


        fprintf(updatePropencities, "random = curand_uniform(&localState);\n");

        fprintf(updatePropencities, "if(sum_p > 0) timeStep = -log(random)/sum_p;\n");
        fprintf(updatePropencities, "else break;\n");

        fprintf(updatePropencities, "random = 1 - curand_uniform(&localState);\n");

        fprintf(updatePropencities, "random *= sum_p;\n");
        fprintf(updatePropencities, "indexMin = 0;\n");
        fprintf(updatePropencities, "indexMax = %d;\n", Model_getNumReactions(m) - 1);
        fprintf(updatePropencities, "reaction = 0;\n");
        fprintf(updatePropencities, "if(random >= cummulative_p[0]){\n");
        fprintf(updatePropencities, "while(indexMax > indexMin){\n");
        fprintf(updatePropencities, "reaction = (indexMin + indexMax + 1)/2;\n");
        fprintf(updatePropencities, "if(cummulative_p[reaction - 1] <= random){\n");
        fprintf(updatePropencities, "if(cummulative_p[reaction] > random){\n");
        fprintf(updatePropencities, "break;\n");
        fprintf(updatePropencities, "}\n");
        fprintf(updatePropencities, "else{\n");
        fprintf(updatePropencities, "indexMin = reaction;\n");
        fprintf(updatePropencities, "}\n");
        fprintf(updatePropencities, "}\n");
        fprintf(updatePropencities, "else{\n");
        fprintf(updatePropencities, "indexMax = reaction;\n");
        fprintf(updatePropencities, "}\n");
        fprintf(updatePropencities, "}\n");
        fprintf(updatePropencities, "}\n");

        fprintf(updatePropencities, "flag_negative = 0;\n");
        fprintf(updatePropencities, "for(int i = 0; i < %d; i++){\n", maxReactionSpecies);
        fprintf(updatePropencities, "if(reactionsSpecies[reaction][i] == -1) {continue;}\n");
        fprintf(updatePropencities, "aux_species[i] = species[reactionsSpecies[reaction][i]];\n");
        fprintf(updatePropencities, "species[reactionsSpecies[reaction][i]] +=
reactionsValues[reaction][i];\n");
        fprintf(updatePropencities, "if(species[reactionsSpecies[reaction][i]] < 0) flag_negative =
1;\n");
        fprintf(updatePropencities, "}\n");

        fprintf(updatePropencities, "for(int i = 0; i < %d && flag_negative == 1; i++){\n",
maxReactionSpecies);
        fprintf(updatePropencities, "if(reactionsSpecies[reaction][i] == -1) {continue;}\n");
        fprintf(updatePropencities, "species[reactionsSpecies[reaction][i]] = aux_species[i];\n");
        fprintf(updatePropencities, "}\n");
```

```
        ListOf_t* events = Model_getListOfEvents(m);
        for (int i = 0; i < Model_getNumEvents(m); i++) {
                Event_t* event = ListOf_get(events, i);
                fprintf(kernelFunction, "int triggerEvent%d = 0;\n", i);
                fprintf(kernelFunction, "if(%s) {triggerEvent%d = 1;}\n",
SBML_formulaToL3String(Trigger_getMath(Event_getTrigger(event))), i);
                fprintf(updatePropencities, "if(triggerEvent%d == 0 && %s){\n", i,
SBML_formulaToL3String(Trigger_getMath(Event_getTrigger(event))));
                fprintf(updatePropencities, "triggerEvent%d = 1;\n", i);
                ListOf_t* eventAssignments = Event_getListOfEventAssignments(event);
                for (int j = 0; j < Event_getNumEventAssignments(event); j++) {
                        fprintf(updatePropencities, "%s += %s;\n",
EventAssignment_getVariable(ListOf_get(eventAssignments, j)),
SBML_formulaToL3String(EventAssignment_getMath(ListOf_get(eventAssignments, j))));
                }
                fprintf(updatePropencities, "}\n");
        }

        fprintf(updatePropencities, "time += timeStep;\n");


        fprintf(updatePropencities, "}\n");

        for (int i = 0; i < Model_getNumCompartments(m); i++) {
                Compartment_t* c = ListOf_get(compartments, i);
                Key = Compartment_getId(c);
                Value = Compartment_getSize(c);
                fprintf(defineConstants, "#define %s %.10lf\n", Key, Value);
        }

        for (int i = 0; i < Model_getNumParameters(m); i++) {
                Parameter_t* p = ListOf_get(parameters, i);
                Key = Parameter_getId(p);
                Value = Parameter_getValue(p);
                fprintf(defineConstants, "#define %s %.10lf\n", Key, Value);
        }

        int streams_closed = fcloseall();

        if (streams_closed == EOF) {
                printf("Error closing files!");
                exit(1);
        }
}


int
main(int argc, char *argv[])
{
        SBMLDocument_t *d;
        Model_t        *m;
        double step, endTime;
        int numSimulations;

        if (argc != 5)
        {
```

```
            printf("Usage: generateCUDA filename sampleTimeStep endTime numberKernels\n");
            return 1;
        }

        d = readSBML(argv[1]);
        m = SBMLDocument_getModel(d);

        step = atof(argv[2]);
        endTime = atof(argv[3]);
        numSimulations = atoi(argv[4]);

        SBMLDocument_printErrors(d, stdout);

        generateCUDA(m, step, numSimulations, endTime);

        SBMLDocument_free(d);
        return 0;
}
```

Appendix B

kernelGill.cu

A sample example code for the CUDA program generated when using the “Proctor2005 - Actions of chaperones and their role in ageing” model.

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <cuda.h>
#include <curand_kernel.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define CUDA_CALL(x) do { if((x) != cudaSuccess) { \
printf("Error at %s:%d\n",__FILE__,__LINE__); \
return EXIT_FAILURE;}} while(0)
#define pow powf
#define SEED 23
#define Hsp90 species[0]
#define Hsp90_id 0
#define HCom species[1]
#define HCom_id 1
#define HSF1 species[2]
#define HSF1_id 2
#define MisP species[3]
#define MisP_id 3
#define MCom species[4]
#define MCom_id 4
#define TriH species[5]
#define TriH_id 5
#define DiH species[6]
#define DiH_id 6
#define NatP species[7]
#define NatP_id 7
#define AggP species[8]
#define AggP_id 8
#define HSE species[9]
#define HSE_id 9
#define HSETriH species[10]
#define HSETriH_id 10
#define X species[11]
#define X_id 11
#define ROS species[12]
#define ROS_id 12
#define ATP species[13]
#define ATP_id 13
#define ADP species[14]
#define ADP_id 14
```

```
#define source species[15]
#define source_id 15
#define compartment 1.0000000000
#define k1 10.0000000000
#define k2 0.0000200000
#define k3 50.0000000000
#define k4 0.0000100000
#define k5 0.0000040000
#define k6 0.0000006000
#define k7 0.0000001000
#define k8 500.0000000000
#define k9 1.0000000000
#define k10 0.0100000000
#define k11 100.0000000000
#define k12 0.5000000000
#define k13 0.5000000000
#define k14 0.0500000000
#define k15 0.0800000000
#define k16 1000.0000000000
#define k17 0.0000000080
#define k18 12.0000000000
#define k19 0.0200000000
#define k20 0.1000000000
#define k21 0.0010000000

__global__
void simulate(int numberOfExecutions, float* output, curandState *state, float step,
float endTime, float segmentSize, float* species_global) {
	int reaction, stepCount = 0;
	int indexMin, indexMax;
	float time = numberOfExecutions * segmentSize;
	float sum_p, timeStep, random;
	float cumulative_p[23];
	float species[16];
	float aux_species[5];
	int flag_negative;
	if (numberOfExecutions == 0) {
		species[0] = species_global[0];
	}
	else {
		species[0] = species_global[Hsp90_id * 128 + threadIdx.x];
	}
	if (numberOfExecutions == 0) {
		species[1] = species_global[1];
	}
	else {
		species[1] = species_global[HCom_id * 128 + threadIdx.x];
	}
	if (numberOfExecutions == 0) {
		species[2] = species_global[2];
```

```
}
else {
	species[2] = species_global[HSF1_id * 128 + threadIdx.x];
}
if (numberOfExecutions == 0) {
	species[3] = species_global[3];
}
else {
	species[3] = species_global[MisP_id * 128 + threadIdx.x];
}
if (numberOfExecutions == 0) {
	species[4] = species_global[4];
}
else {
	species[4] = species_global[MCom_id * 128 + threadIdx.x];
}
if (numberOfExecutions == 0) {
	species[5] = species_global[5];
}
else {
	species[5] = species_global[TriH_id * 128 + threadIdx.x];
}
if (numberOfExecutions == 0) {
	species[6] = species_global[6];
}
else {
	species[6] = species_global[DiH_id * 128 + threadIdx.x];
}
if (numberOfExecutions == 0) {
	species[7] = species_global[7];
}
else {
	species[7] = species_global[NatP_id * 128 + threadIdx.x];
}
if (numberOfExecutions == 0) {
	species[8] = species_global[8];
}
else {
	species[8] = species_global[AggP_id * 128 + threadIdx.x];
}
if (numberOfExecutions == 0) {
	species[9] = species_global[9];
}
else {
	species[9] = species_global[HSE_id * 128 + threadIdx.x];
}
if (numberOfExecutions == 0) {
	species[10] = species_global[10];
}
else {
```

```
        species[10] = species_global[HSETriH_id * 128 + threadIdx.x];
}
if (numberOfExecutions == 0) {
        species[11] = species_global[11];
}
else {
        species[11] = species_global[X_id * 128 + threadIdx.x];
}
if (numberOfExecutions == 0) {
        species[12] = species_global[12];
}
else {
        species[12] = species_global[ROS_id * 128 + threadIdx.x];
}
if (numberOfExecutions == 0) {
        species[13] = species_global[13];
}
else {
        species[13] = species_global[ATP_id * 128 + threadIdx.x];
}
if (numberOfExecutions == 0) {
        species[14] = species_global[14];
}
else {
        species[14] = species_global[ADP_id * 128 + threadIdx.x];
}
if (numberOfExecutions == 0) {
        species[15] = species_global[15];
}
else {
        species[15] = species_global[source_id * 128 + threadIdx.x];
}
int reactionsSpecies[23][5];
int reactionsValues[23][5];
for (int i = 0; i < 23; i++) {
        for (int j = 0; j < 5; j++) {
                reactionsSpecies[i][j] = -1;
                reactionsValues[i][j] = 0;
        }
}
reactionsSpecies[0][0] = source_id;
reactionsValues[0][0] = -0.0000000000;
reactionsSpecies[0][1] = NatP_id;
reactionsValues[0][1] = 1.0000000000;
reactionsSpecies[1][0] = NatP_id;
reactionsValues[1][0] = -1.0000000000;
reactionsSpecies[1][1] = ROS_id;
reactionsValues[1][1] = -1.0000000000;
reactionsSpecies[1][2] = MisP_id;
reactionsValues[1][2] = 1.0000000000;
```

```
reactionsSpecies[1][3] = ROS_id;
reactionsValues[1][3] = 1.0000000000;
reactionsSpecies[2][0] = MisP_id;
reactionsValues[2][0] = -1.0000000000;
reactionsSpecies[2][1] = Hsp90_id;
reactionsValues[2][1] = -1.0000000000;
reactionsSpecies[2][2] = MCom_id;
reactionsValues[2][2] = 1.0000000000;
reactionsSpecies[3][0] = MCom_id;
reactionsValues[3][0] = -1.0000000000;
reactionsSpecies[3][1] = MisP_id;
reactionsValues[3][1] = 1.0000000000;
reactionsSpecies[3][2] = Hsp90_id;
reactionsValues[3][2] = 1.0000000000;
reactionsSpecies[4][0] = MCom_id;
reactionsValues[4][0] = -1.0000000000;
reactionsSpecies[4][1] = ATP_id;
reactionsValues[4][1] = -1.0000000000;
reactionsSpecies[4][2] = Hsp90_id;
reactionsValues[4][2] = 1.0000000000;
reactionsSpecies[4][3] = NatP_id;
reactionsValues[4][3] = 1.0000000000;
reactionsSpecies[4][4] = ADP_id;
reactionsValues[4][4] = 1.0000000000;
reactionsSpecies[5][0] = MisP_id;
reactionsValues[5][0] = -1.0000000000;
reactionsSpecies[5][1] = ATP_id;
reactionsValues[5][1] = -1.0000000000;
reactionsSpecies[5][2] = ADP_id;
reactionsValues[5][2] = 1.0000000000;
reactionsSpecies[6][0] = MisP_id;
reactionsValues[6][0] = -2.0000000000;
reactionsSpecies[6][1] = AggP_id;
reactionsValues[6][1] = 1.0000000000;
reactionsSpecies[7][0] = MisP_id;
reactionsValues[7][0] = -1.0000000000;
reactionsSpecies[7][1] = AggP_id;
reactionsValues[7][1] = -1.0000000000;
reactionsSpecies[7][2] = AggP_id;
reactionsValues[7][2] = 2.0000000000;
reactionsSpecies[8][0] = Hsp90_id;
reactionsValues[8][0] = -1.0000000000;
reactionsSpecies[8][1] = HSF1_id;
reactionsValues[8][1] = -1.0000000000;
reactionsSpecies[8][2] = HCom_id;
reactionsValues[8][2] = 1.0000000000;
reactionsSpecies[9][0] = HCom_id;
reactionsValues[9][0] = -1.0000000000;
reactionsSpecies[9][1] = Hsp90_id;
reactionsValues[9][1] = 1.0000000000;
```

```
reactionsSpecies[9][2] = HSF1_id;
reactionsValues[9][2] = 1.0000000000;
reactionsSpecies[10][0] = HSF1_id;
reactionsValues[10][0] = -2.0000000000;
reactionsSpecies[10][1] = DiH_id;
reactionsValues[10][1] = 1.0000000000;
reactionsSpecies[11][0] = HSF1_id;
reactionsValues[11][0] = -1.0000000000;
reactionsSpecies[11][1] = DiH_id;
reactionsValues[11][1] = -1.0000000000;
reactionsSpecies[11][2] = TriH_id;
reactionsValues[11][2] = 1.0000000000;
reactionsSpecies[12][0] = TriH_id;
reactionsValues[12][0] = -1.0000000000;
reactionsSpecies[12][1] = HSF1_id;
reactionsValues[12][1] = 1.0000000000;
reactionsSpecies[12][2] = DiH_id;
reactionsValues[12][2] = 1.0000000000;
reactionsSpecies[13][0] = DiH_id;
reactionsValues[13][0] = -1.0000000000;
reactionsSpecies[13][1] = HSF1_id;
reactionsValues[13][1] = 2.0000000000;
reactionsSpecies[14][0] = TriH_id;
reactionsValues[14][0] = -1.0000000000;
reactionsSpecies[14][1] = HSE_id;
reactionsValues[14][1] = -1.0000000000;
reactionsSpecies[14][2] = HSETriH_id;
reactionsValues[14][2] = 1.0000000000;
reactionsSpecies[15][0] = HSETriH_id;
reactionsValues[15][0] = -1.0000000000;
reactionsSpecies[15][1] = HSE_id;
reactionsValues[15][1] = 1.0000000000;
reactionsSpecies[15][2] = TriH_id;
reactionsValues[15][2] = 1.0000000000;
reactionsSpecies[16][0] = HSETriH_id;
reactionsValues[16][0] = -1.0000000000;
reactionsSpecies[16][1] = HSETriH_id;
reactionsValues[16][1] = 1.0000000000;
reactionsSpecies[16][2] = Hsp90_id;
reactionsValues[16][2] = 1.0000000000;
reactionsSpecies[17][0] = Hsp90_id;
reactionsValues[17][0] = -1.0000000000;
reactionsSpecies[17][1] = ATP_id;
reactionsValues[17][1] = -1.0000000000;
reactionsSpecies[17][2] = ADP_id;
reactionsValues[17][2] = 1.0000000000;
reactionsSpecies[18][0] = source_id;
reactionsValues[18][0] = -0.0000000000;
reactionsSpecies[18][1] = X_id;
reactionsValues[18][1] = 1.0000000000;
```

```
        reactionsSpecies[19][0] = ADP_id;
        reactionsValues[19][0] = -1.0000000000;
        reactionsSpecies[19][1] = ATP_id;
        reactionsValues[19][1] = 1.0000000000;
        reactionsSpecies[20][0] = ATP_id;
        reactionsValues[20][0] = -1.0000000000;
        reactionsSpecies[20][1] = ADP_id;
        reactionsValues[20][1] = 1.0000000000;
        reactionsSpecies[21][0] = source_id;
        reactionsValues[21][0] = -0.0000000000;
        reactionsSpecies[21][1] = ROS_id;
        reactionsValues[21][1] = 1.0000000000;
        reactionsSpecies[22][0] = ROS_id;
        reactionsValues[22][0] = -1.0000000000;
        curandState localState = state[threadIdx.x];
        while (time < endTime && time < (numberOfExecutions + 1)*segmentSize) {
                cumulative_p[0] = k1;
                cumulative_p[1] = cumulative_p[0] + k2 * NatP * ROS;
                cumulative_p[2] = cumulative_p[1] + k3 * MisP * Hsp90;
                cumulative_p[3] = cumulative_p[2] + k4 * MCom;
                cumulative_p[4] = cumulative_p[3] + k5 * MCom * ATP;
                cumulative_p[5] = cumulative_p[4] + k6 * MisP * ATP;
                cumulative_p[6] = cumulative_p[5] + (MisP - 1) * k7 * MisP / 2;
                cumulative_p[7] = cumulative_p[6] + k7 * MisP * AggP;
                cumulative_p[8] = cumulative_p[7] + k8 * Hsp90 * HSF1;
                cumulative_p[9] = cumulative_p[8] + k9 * HCom;
                cumulative_p[10] = cumulative_p[9] + (HSF1 - 1) * k10 * HSF1 / 2;
                cumulative_p[11] = cumulative_p[10] + k11 * HSF1 * DiH;
                cumulative_p[12] = cumulative_p[11] + k12 * TriH;
                cumulative_p[13] = cumulative_p[12] + k13 * DiH;
                cumulative_p[14] = cumulative_p[13] + k14 * HSE * TriH;
                cumulative_p[15] = cumulative_p[14] + k15 * HSETriH;
                cumulative_p[16] = cumulative_p[15] + k16 * HSETriH;
                cumulative_p[17] = cumulative_p[16] + k17 * Hsp90 * ATP;
                cumulative_p[18] = cumulative_p[17] + 1;
                cumulative_p[19] = cumulative_p[18] + k18 * ADP;
                cumulative_p[20] = cumulative_p[19] + k19 * ATP;
                cumulative_p[21] = cumulative_p[20] + k20;
                cumulative_p[22] = cumulative_p[21] + k21 * ROS;
                if (time >= segmentSize * numberOfExecutions + step * stepCount) {
                        atomicAdd(&output[16 * 100 * numberOfExecutions + stepCount * 16 +
0], species[0]);
                        atomicAdd(&output[16 * 100 * numberOfExecutions + stepCount * 16 +
1], species[1]);
                        atomicAdd(&output[16 * 100 * numberOfExecutions + stepCount * 16 +
2], species[2]);
                        atomicAdd(&output[16 * 100 * numberOfExecutions + stepCount * 16 +
3], species[3]);
                        atomicAdd(&output[16 * 100 * numberOfExecutions + stepCount * 16 +
4], species[4]);
```

```
                        atomicAdd(&output[16 * 100 * numberOfExecutions + stepCount * 16 +
5], species[5]);
                        atomicAdd(&output[16 * 100 * numberOfExecutions + stepCount * 16 +
6], species[6]);
                        atomicAdd(&output[16 * 100 * numberOfExecutions + stepCount * 16 +
7], species[7]);
                        atomicAdd(&output[16 * 100 * numberOfExecutions + stepCount * 16 +
8], species[8]);
                        atomicAdd(&output[16 * 100 * numberOfExecutions + stepCount * 16 +
9], species[9]);
                        atomicAdd(&output[16 * 100 * numberOfExecutions + stepCount * 16 +
10], species[10]);
                        atomicAdd(&output[16 * 100 * numberOfExecutions + stepCount * 16 +
11], species[11]);
                        atomicAdd(&output[16 * 100 * numberOfExecutions + stepCount * 16 +
12], species[12]);
                        atomicAdd(&output[16 * 100 * numberOfExecutions + stepCount * 16 +
13], species[13]);
                        atomicAdd(&output[16 * 100 * numberOfExecutions + stepCount * 16 +
14], species[14]);
                        atomicAdd(&output[16 * 100 * numberOfExecutions + stepCount * 16 +
15], species[15]);
                        stepCount++;
                }
                sum_p = cumulative_p[22];
                random = curand_uniform(&localState);
                if (sum_p > 0) timeStep = -log(random) / sum_p;
                else break;
                random = 1 - curand_uniform(&localState);
                random *= sum_p;
                indexMin = 0;
                indexMax = 22;
                reaction = 0;
                if (random >= cumulative_p[0]) {
                        while (indexMax > indexMin) {
                                reaction = (indexMin + indexMax + 1) / 2;
                                if (cumulative_p[reaction - 1] <= random) {
                                        if (cumulative_p[reaction] > random) {
                                                break;
                                        }
                                        else {
                                                indexMin = reaction;
                                        }
                                }
                                else {
                                        indexMax = reaction;
                                }
                        }
                }
                flag_negative = 0;
```

```
		for (int i = 0; i < 5; i++) {
			if (reactionsSpecies[reaction][i] == -1) { continue; }
			aux_species[i] = species[reactionsSpecies[reaction][i]];
			species[reactionsSpecies[reaction][i]] +=
reactionsValues[reaction][i];
			if (species[reactionsSpecies[reaction][i]] < 0) flag_negative = 1;
		}
		for (int i = 0; i < 5 && flag_negative == 1; i++) {
			if (reactionsSpecies[reaction][i] == -1) { continue; }
			species[reactionsSpecies[reaction][i]] = aux_species[i];
		}
		time += timeStep;
	}
	species_global[Hsp90_id * 128 + threadIdx.x] = species[0];
	species_global[HCom_id * 128 + threadIdx.x] = species[1];
	species_global[HSF1_id * 128 + threadIdx.x] = species[2];
	species_global[MisP_id * 128 + threadIdx.x] = species[3];
	species_global[MCom_id * 128 + threadIdx.x] = species[4];
	species_global[TriH_id * 128 + threadIdx.x] = species[5];
	species_global[DiH_id * 128 + threadIdx.x] = species[6];
	species_global[NatP_id * 128 + threadIdx.x] = species[7];
	species_global[AggP_id * 128 + threadIdx.x] = species[8];
	species_global[HSE_id * 128 + threadIdx.x] = species[9];
	species_global[HSETriH_id * 128 + threadIdx.x] = species[10];
	species_global[X_id * 128 + threadIdx.x] = species[11];
	species_global[ROS_id * 128 + threadIdx.x] = species[12];
	species_global[ATP_id * 128 + threadIdx.x] = species[13];
	species_global[ADP_id * 128 + threadIdx.x] = species[14];
	species_global[source_id * 128 + threadIdx.x] = species[15];
	state[threadIdx.x] = localState;
}

__global__
void initCurand(curandState* state, unsigned long long seed) {
	curand_init(seed, threadIdx.x, 0, &state[threadIdx.x]);
}
int main()
{
	cudaError_t cudaStatus;
	float* output;
	float* dev_output;
	output = (float*)malloc(1000 * 16 * sizeof(float));
	for (int i = 0; i < 1000 * 16; i++) {
		output[i] = 0;
	}
	cudaStatus = cudaMalloc(&dev_output, 1000 * 16 * sizeof(float));
	if (cudaStatus != cudaSuccess) { fprintf(stderr, "cudaMalloc failed!"); goto
Error; }
	cudaStatus = cudaMemcpy(dev_output, output, 1000 * 16 * sizeof(float),
cudaMemcpyHostToDevice);
```

```
        if (cudaStatus != cudaSuccess) { fprintf(stderr, "cudaMemcpy failed!"); goto
Error; }
        float* species_global;
        cudaStatus = cudaMalloc(&species_global, 16 * 128 * sizeof(float));
        if (cudaStatus != cudaSuccess) { fprintf(stderr, "cudaMalloc failed!"); goto
Error; }
        float init_species[16];
        init_species[0] = 300000.0000000000;
        init_species[1] = 5900.0000000000;
        init_species[2] = 100.0000000000;
        init_species[3] = 0.0000000000;
        init_species[4] = 0.0000000000;
        init_species[5] = 0.0000000000;
        init_species[6] = 0.0000000000;
        init_species[7] = 6000000.0000000000;
        init_species[8] = 0.0000000000;
        init_species[9] = 1.0000000000;
        init_species[10] = 0.0000000000;
        init_species[11] = 0.0000000000;
        init_species[12] = 100.0000000000;
        init_species[13] = 10000.0000000000;
        init_species[14] = 1000.0000000000;
        init_species[15] = 0.0000000000;
        cudaStatus = cudaMemcpy(species_global, &init_species, sizeof(float) * 16,
cudaMemcpyHostToDevice);
        if (cudaStatus != cudaSuccess) { fprintf(stderr, "cudaMemcpy failed!"); goto
Error; }
        curandState *devStates;
        CUDA_CALL(cudaMalloc((void **)&devStates, 128 * sizeof(curandState)));
        initCurand << <1, 128 >> > (devStates, SEED);
        clock_t begin = clock();
        cudaEvent_t start, stop;
        float milliseconds;
        cudaEventCreate(&start);
        cudaEventCreate(&stop);
        printf("Starting kernel\n");
        for (int i = 0; i < 10; i++) {
                cudaEventRecord(start);
                simulate << <1, 128 >> > (i, dev_output, devStates, 0.1000000000,
100.0000000000, 10, species_global); cudaStatus = cudaGetLastError(); if (cudaStatus !=
cudaSuccess) { fprintf(stderr, "addKernel launch failed: %s\n",
cudaGetErrorString(cudaStatus)); goto Error; }

                cudaStatus = cudaDeviceSynchronize(); if (cudaStatus != cudaSuccess) {
fprintf(stderr, "cudaDeviceSynchronize returned error code %d after launching
addKernel!\n", cudaStatus); goto Error; }cudaEventRecord(stop);
                printf("SEGMENT %d\n", i);
                cudaEventSynchronize(stop);
                milliseconds = 0;
                cudaEventElapsedTime(&milliseconds, start, stop);
```

```
                printf("TIME: %lf\n\n", milliseconds);
        }


        cudaStatus = cudaMemcpy(output, dev_output, 1000 * 16 * sizeof(float),
cudaMemcpyDeviceToHost);
        if (cudaStatus != cudaSuccess) { fprintf(stderr, "cudaMemcpy failed!"); goto
Error; }
        FILE* results = fopen("results.csv", "w");
        if (results == NULL) {
                printf("Error acesssing results!");
                exit(1);
        }
        clock_t end = clock();
        double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
        fprintf(results, "TOTAL EXECUTION TIME : %lf\n", time_spent);
        fprintf(results, "time");
        fprintf(results, ", Hsp90");
        fprintf(results, ", HCom");
        fprintf(results, ", HSF1");
        fprintf(results, ", MisP");
        fprintf(results, ", MCom");
        fprintf(results, ", TriH");
        fprintf(results, ", DiH");
        fprintf(results, ", NatP");
        fprintf(results, ", AggP");
        fprintf(results, ", HSE");
        fprintf(results, ", HSETriH");
        fprintf(results, ", X");
        fprintf(results, ", ROS");
        fprintf(results, ", ATP");
        fprintf(results, ", ADP");
        fprintf(results, ", source");
        fprintf(results, "\n");
        for (int i = 0; i < 1000; i++) {
                fprintf(results, "%.10lf", 0.1000000000*i);
                for (int j = 0; j < 16; j++) {
                        fprintf(results, ", %.10lf", output[16 * i + j] / 128);
                }
                fprintf(results, "\n");
        }
        fprintf(results, "\n");
Error:
        cudaFree(dev_output);
        cudaFree(species_global);
        return 0;
}
```