Bachelor Thesis

GPU Acceleration in Large Computational Models

Yecheng Yang

Date: May 12, 2017

Advisor:
Ahmet Ay, John Stratton, Vijay Ramachandran

Department of Computer Science
Colgate University
Hamilton, New York

Abstract

Complex biochemical systems such as gene regulation networks are often represented as delay differential equations (DDEs). However, unlike ordinary differential equations (ODEs), there are few large-scale systems available for simulating DDEs, and fewer still for high-performance simulations. To meet the growing need for simulating tissue-level systems of intercellular signaling gene regulation networks, I present a high-performance DDE simulation framework. By using compile-time specialization and optimization, my framework creates highly compact and efficient data structures with a small amount of configuration for the model being simulated. It also supports GPU acceleration of highly parallel models, which can be used to increase the simulation speed of large tissues, or increase the search speed among independent simulations in a reaction parameter search. I demonstrate congruence of my simulation results with prior special-purpose simulations, and the superior CPU and GPU performance of my framework over those same frameworks, while also supporting greater flexibility and configurability of the systems being modeled.

Aknowledgments

I would like to thank my research supervisors, Professor Ahmet Ay, Professor John Stratton and Professor Vijay Ramachandran. Without their assistance and dedicated involvement in every step throughout the process, this paper would have never been accomplished. I would like to thank you very much for your support and understanding over these past years.

Contents

1 Introduction

This honor thesis is motivated by my previous research on building a computer system for simulating and analyzing a regulatory biological network called *zebrafish segmentation clock network*. In particular, the computer system builds on top of a list of species (such as mRNAs and proteins of *her1, her7, her13* genes) and a set of differential equations describing reactions (such as protein synthesis, protein degradation and mRNA synthesis) in this regulatory network. The system workflow consists of several phases: parameter estimation, simulation, feature extraction, and testing. Parameter estimation is used to generate random parameter values within a certain range through a genetic algorithm in hopes of finding a “desirable” parameter set whose simulation result “fits” the biological model. Simulation, based on a particular parameter set, happens at tissue level (multiple cells) and is responsible for replicating the entire biological process for a certain period of time. This simulation is also tasked with recording concentration levels of each species during this process. Feature extraction retrieves concentration levels recorded in the previous step and turns those values into properties such as period, amplitude, and synchronization score. At last, during testing, the properties produced in the previous step are compared to standards created based on experimental data; this comparison determines if the results are “desirable” and “fit” the model.

1.1 Motivation

Various systems and packages have been created to meet the need of research in regulatory biological networks. Some of the created packages provide a user-friendly interface, yet they either conduct simulations on a small-scale basis, or they focus solely on ordinary differential equations, and thus ignoring delay differential equations. Both of the practices effectively prevent researchers from building accurate and comprehensive models. Multiple cell simulation is important in analyzing the effects of intercellular communication on various species in the system, and due to the time delay in certain biological processes, delay differential equations are sometimes crucial in analyzing such network structures. For example, gene expression in zebrafish segmentation clock system consists of two main segments, gene transcription and mRNA translation. Because of the biologically complex nature of gene transcription and mRNA translation, such reactions typically take five to ten minutes to finish in the cell. Thus, the corresponding computer simulation relies on DDEs to mimic the time delay introduced by such processes in a biological system. Other packages, though never previously applied in this particular field, can offer potentially desirable runtime for large-scale simulations through GPU acceleration. However, since those packages are often presented as libraries, they usually require a decent understand of GPU programming for researchers to enjoy such benefits. Under such circumstances, many researchers are forced to spend prolonged periods of time creating systems to simulate specific models. Such systems generally lack flexibility or configurability, and without GPU acceleration, tissue level simulations running on CPU are far less efficient and extremely time-intensive, even on computer cluster systems. Here, I built a system that is compatible with DDEs, flexible, easily configurable, and accessible to researchers without substantial programming skills.

1.2 Benefits

This new system will provide researchers with considerable advantages in three key areas. First, the cost of creating a running system for a comprehensive large-scale model will be significantly lower, and when researchers are able to configure and simulate their own model, more time can now be devoted to alternative research tasks. Second, runtime of the new system shows significant improvement; it is five to ten times faster than CPU only systems. This is important to researchers since parameter estimation usually takes days to complete on computer clusters, and much longer on usual workstations. Third, requirements on hardware for efficient usage of the systems will be greatly reduced. While previous systems usually require clusters for parameter estimations, the new developed system will be compatible with most workstations that contain a powerful GPU card. Compared to a cluster of powerful cores that typically cost 100,000+ USD, a GPU card is economically efficient (typically < 5,000 USD), requires less maintenance, and is widely accessible to researchers.

2 Related Work

Developers have created various packages to provide simulation of regulatory systems to aid researchers. The current packages, however, all have certain limitations and fail to fully satisfy the demands of researchers. In this section, I will review the advantages and disadvantages of existing programs and packages.

2.1 COPASI

COPASI, for example, is one of the most widely used CPU based software for analyzing biological regulatory networks [?]. COPASI includes user-friendly interfaces for model input to cater to all researchers with varied programming ability. Simulation and analysis sections of COPASI perform efficiently on small-scale models, namely, one-cell models. However, the features included in the software strictly limit a pre-compiled program like COPASI and thus it is impossible for researchers to extend its function to multi-cell simulation of biological networks. For the same reason, researchers are not able to analyze complex system involving DDEs because COPASI only supports simulations with ordinary differential equations.

2.2 GPU Accelerated Packages

On the other hand, simulation software packages that utilize GPU acceleration have shown considerable speedups. For example, cupSODA achieved“a 86x speedup on GPUs with respect to equivalent executions of LSODA on the CPU” and Murray’s package achieved “speedups of up to 115-fold over comparable serial CPU implementations, and 15-fold over multithreaded CPU code” [?, ?, ?]. However, very few of the existing packages support partial differential equations and none of them support DDE. Lack of functionality in these systems also exclude a large portion of researchers from using this category of packages. In addition, most of the existing packages are presented as libraries and are not nearly as user-friendly as COPASI; often written in languages like C/C++, those packages require even higher levels of programming skill, which is not common among researchers in this field. Until now, there is virtually no usage of such packages in this field.

2.3 Other Software

Because of the aforementioned limitations of the packages described above, researchers interested in creating large and complex models are forced to create unique simulations. Professor Ay and I were tasked with building a specialized system. The system was constructed gradually over the past four years. One of the major data structure in the system is a large three-dimensional array named *baby_cl*, which holds concentration levels over several delayed time steps for all species and all cells. It supports DDE and multi-cell simulations, particularly for the segmentation clock project [**?**, **?**]. There are a total of six mutants for each parameter set. The system uses thirteen differential equations to represent rate change at any time step. Historical data, stored in concentration levels, and differential equations together predict the concentration level of next time step. ODEs in the system only rely on the data in the last time step whereas DDEs in the system may request concentration levels from over one thousand time steps ago. There are some limitations to this system, however. Written for a CPU only environment, the system runs slowly and parameter estimation takes days to execute on Colgate's computer cluster. Furthermore, because all data structures and functions were hard coded for a particular regulatory network, updating the system corresponding to a model update was extremely inconvenient and highly time-consuming.

3 Contribution

Given the limitations of the currently available alternatives discussed earlier, I decided that the new program should address all of them. In particular, this new program will:

(i) Accept delay differential equation models as inputs,
(ii) Includes user-friendly interface for entering model information,
(iii) Automatically generate necessary data structures and corresponding functions,
(iv) Simulate using desired numerical solvers, including both deterministic and stochastic methods,
(v) Be highly modularized and can be extended or updated easily in the future,
(vi) Support GPU for better performance.

The entire software will be developed over the next few years, and for the purpose of this thesis, I will mainly focus on the design and development elements of the simulation function within the new system. Simulation is the most complicated and time-consuming section of the entire system because it mimics the entire biological process for every species, cell, and time step. In addition, it provides a foundation for feature extraction and testing, as well as connection between parameter estimation and the rest of the system. Therefore, the simulation section is essential in the construction of the entire software and its completion can provide insights for building other parts of the system moving forward.

The whole project starts with preliminary conversion of the original system into a GPU accelerated system and I expect to discover flaws of the original system as well as exploring potential solutions to fix them. The second half of the project will include construction of a whole new system that will address the flaws identified during preliminary work as well as difficulty in model switch and system update.

4 Preliminaries

As mentioned above, the existing system simulates and analyzes a tissue-level model of segmentation clock network. In this stage, I will focus on improving runtime of the existing CPU only system by introducing GPU acceleration to the system.

4.1 Methodology:

4.1.1 CUDA Platform

In this study, I chose CUDA by NVIDIA as the platform for GPU computing. As a minimal extension of C and C++, CUDA works well with the existing system and it has been applied to create scalable parallel programs in similar disciplines such as computational chemistry [?]. There are three key abstractions in the CUDA platform: a hierarchy of thread groups, shared memories, and barrier synchronization. Together, those three abstractions create a parallel structure to C/C++ code. This structure allows coarse-grained parallelism on the high level and more fine-grained parallelism on lower levels. It is a perfect candidate for executing simulations in a biological regulatory network since the structure within the CUDA platform corresponds well to a large computational model. In particular, CUDA platform is organized as grids of blocks where each block contains a set of parallel threads, the lowest level of parallelism in the system. The set of threads in a block cooperates well with barrier synchronization and shared access to a memory space private to the block. Each thread is analogous to a cell inside one single simulation since cells are simulated in parallel to each other and the set of all cells in a simulation are connected and need to share common memory. On a higher level, parameter estimation contains a set of simulations that are not related to each other and can execute in parallel, exactly corresponding to a gird of independent block that can be executed independently. Overall, CUDA platform and its design correspond with a biological system from low level to high level simulations.

4.1.2 Initial Attempt

Conversion of the original system to GPU started with simulation of all six mutants (of a parameter set) in parallel for one parameter set. Under this scheme, the system is broken into three major parts: All inputting parameters, creating new data structures, and copying original code to new copies are executed in the original order for each of the mutants. A *for loop* is added to iterate and setup all six mutants before the system moves to the next phase. Next, the system transfers all the data necessary for simulation (particularly data needed by protein synthesis, dimer synthesis and mRNA synthesis) to GPU and the system starts one block on GPU for simulating each of the six mutants. During simulation of a single time step, the system transfers relatively small portions of data between CPU and GPU for the purposes of updating rates and storing intermediate concentration levels. This part of the program is executed in parallel on GPU. After all simulations of this time step have completed on GPU, the system then copy all data back to CPU. The same process repeats for all time steps, and once simulation of all time steps is complete, the system starts to perform feature extraction and testing for all mutants. Similar to the first phase, all processes in the last phase are executed in the same order as they were originally, except there is a new *for loop* over six mutants.

The major difficulty in this part of the project is minimizing data transfer between CPU and GPU. Since rates of reaction, concentration level of mRNAs and proteins all need to be updated and stored at each occurrence. Large amounts of data transfer may likely be inefficient and dismiss the benefit of incorporating GPU computing into the system. This is because GPU has to constantly wait until all data transfer is completed and thus its computation power is far from being fully utilized. Therefore, I applied several methods to prevent data transfer to be the bottleneck of the heterogeneous computing system.

4.1.3 Improved Version

To achieve an improvement of data transfer, I first integrated memory transfer of all six mutants together because transfer bundling is faster than transferring the same amount of data over separate occurrences. Next, I moved data transfer outside of the *for loop* over all time steps. Now, the system transfers all data onto GPU in the beginning of the large *for loop* over all time steps. Now data transfer of the entire data set only occurs twice for all time steps, instead of twice for each time step. In addition, I divided the transfer functions into two separate functions. One function will allocate a CPU array on the GPU and the other function will transfer the data and swap pointers to the array on GPU and CPU. I then added another function that will only swap pointers to the array. Through those separated functions, the system only needs to allocate the array once and can access the array repeatedly later using the pointer and swapping functions. The allocation process is now omitted during each data transfer, and the system only needs to overwrite the data at the given location; thus, each data transfer can be accelerated.

Less communication between GPU and CPU will obviously reduce unnecessary overheads and improve runtime of each individual simulation. However, less data transfer between CPU and GPU will require more data to stay on GPU throughout each simulation and will directly affect memory requirement of each process. When I was trying to simulate multiple parameter sets at the same time, simulation of multiple parameter sets lead to another challenge – reducing the memory requirement of the simulation on GPU. Limited by the structure of the current system, the system keeps a copy on CPU for every copy of data structure existing on GPU. With large requirement on memory for each simulation, only a small number of simulations can fit on the GPU card simultaneously, and thus, utilization of GPU card does not reach its full capacity.

There are two viable ways to reduce the memory requirement of the system on GPU. The first way to reduce memory requirement is to increment time step and thus indirectly decrease the size of *baby_cl*. If the time step is increased to 0.02, then size of *baby_cl* can reduce by half. Similarly, if time step if four time larger, then size of *baby_cl* can be reduced to a quarter of the original size. This change can be easily achieved; however, time step granularity (size of a time step) is directly determined by the accuracy requirements of the system. In other words, increasing step granularity may lead to less accurate simulation results. The second solution involves directly reducing the size of main data structure that will be copied to GPU. Currently, the number of time steps that every *baby_cl* keeps is the same and is equal to the maximum delay. However, most reactions require data from a small number of past time steps. If *baby_cl* for each of the reactions can be tailored according to the delay size of that reaction, memory requirement on GPU can reduced by a factor of six or ten in the worst and best case scenarios, respectively. Redesigning the data structure and changing time step sizes for differential equations require various changes over the entire program, and thus, they were not implemented in this preliminary stage. Instead, they will be addressed and incorporated during the later stages of the project.

4.2 Results

At the end of the preliminary stage, I obtained the following results: runtime of the system simulating one parameter set sequentially and in parallel are 6:15 and 4:18, respectively. Runtime for simulating two parameter sets in parallel is 7:26, which is roughly 3:43 per parameter set (Figure 4.1). Although the simulation runtime for one parameter set increases on GPU, improvements can be seen when multiple parameter sets are simulated. Instead of a

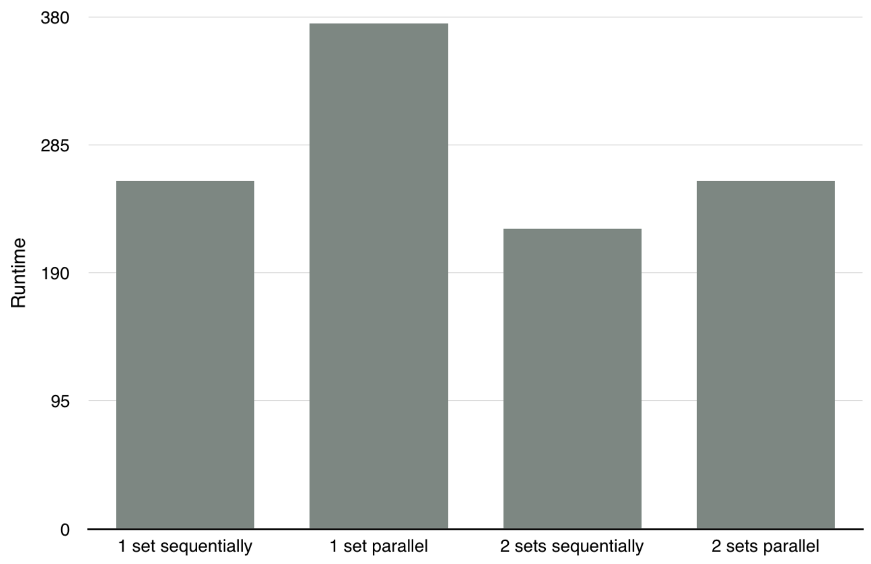


Figure 4.1: Runtime Per Parameter Set for Sequential and Parallel Simulations

linear increase in total runtime, GPU accelerated system has a much smaller increase since simulation, the most time-consuming part, is now executed in parallel with other sets and thus, does not incur additional runtime increase. Only feature extraction and analysis result in a light increase in average runtime. This pattern runtime increase continues to hold true with larger number of parameter sets, but before significant increase, the memory on GPU exhausts due to the reasons mentioned above.

Overall, this is a successful experiment and reveals several factors in the current system that may affect the runtime. Data structures are larger than they need to be and thus inefficient. Due to this reason, number of parameter sets to be simulated on GPU at the same time is strictly limited. Furthermore, there are two copies of some data structures in the system, which increases memory requirement. Problems identified in this stage provide insights about limiting factors of current system and will be addressed during development of the new system.

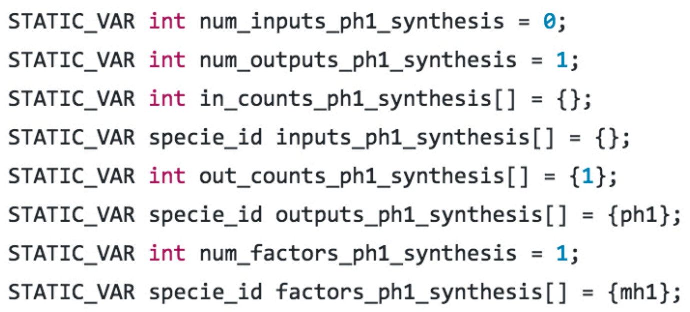
5 Model Separation

In the original system, model information and simulation implementations deeply integrated, and model updates require changes to be made throughout the entire system. This prevents the system from having better flexibility. The goal of the new system is to accept mathematical models as input; based on the model input, the system should be able to dynamically adapt itself by creating corresponding data structures and function calls simulation. In this section, I aim to separate all model information from simulation methods and data structures. Through model separation, the future system users will be able to easily update or switch biological models while keeping the rest of the system untouched.

5.1 Methodology

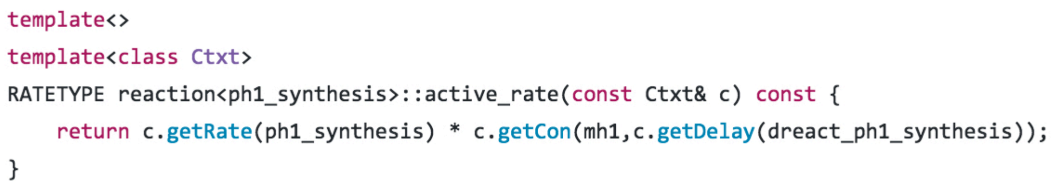
A new form of model representation is designed for model information in the new system. In this representation, the basis of the model consists of a list of species and list of reactions related to those species (*species_list.hpp* and *reaction_list.hpp*, respectively). Each of the reaction has its specific *reaction_id*. On top of those two files, there are two additional structures. The first file, *reaction.cpp*, (Figure 5.1) contains data structure describing the relations between species and reactions, which involve input, output and determining species of each reaction and their effects. The second file, *model_impl.hpp*, (Figure 5.2) describes mechanisms of each reaction, which are the related active rates and concentration levels used in this reaction. Notice here that each function in this file is templated on two parameters and one of them is *reaction_id*, and it will be useful later.

Since the system is trying to dynamically create data structures and functions though this representation of model information, some level of polymorphism is required to accomplish this task. A common way to construct this class is to use virtual functions. Then, different kinds of reactions will specify different functions in order to calculate reaction rate. This is undesirable in our system, however, because all virtual functions provide only run-time polymorphism. Since virtual functions are small, the overhead of looking up corresponding member function at runtime is high relative to the amount of work the functions do. In other words, the system will be much less efficient since those functions cannot be optimized



```
STATIC_VAR int num_inputs_ph1_synthesis = 0;
STATIC_VAR int num_outputs_ph1_synthesis = 1;
STATIC_VAR int in_counts_ph1_synthesis[] = {};
STATIC_VAR specie_id inputs_ph1_synthesis[] = {};
STATIC_VAR int out_counts_ph1_synthesis[] = {1};
STATIC_VAR specie_id outputs_ph1_synthesis[] = {ph1};
STATIC_VAR int num_factors_ph1_synthesis = 1;
STATIC_VAR specie_id factors_ph1_synthesis[] = {mh1};
```

Figure 5.1: Snippet of Code from *reaction.cpp*



```
template<>
template<class Ctxt>
RATETYPE reaction<ph1_synthesis>::active_rate(const Ctxt& c) const {
    return c.getRate(ph1_synthesis) * c.getCon(mh1,c.getDelay(dreact_ph1_synthesis));
}
```

Figure 5.2: Snippet of Code from *model_impl.hpp*

during compile time. To overcome the loss of complier optimization, I introduced *x-macros* into the system and define a reaction and species list as an enumerated list. Through *x-macros*, resulting data structures and function calls can be generated during pre-processing, thus granting polymorphism while enjoying compiler optimization. Details of *x-macros* usage will be clarified in the next section due to its close interaction with simulation.

5.2 Results

In this section, we choose a much higher level of generalization of model information at the cost of not continuing to use design and manual optimization of the previous system. Usage of *x-macros* provides enough compensation to the system to make generalization viable, though. It is important to note here that manual optimization of this particular section of the system is impractical. Manual optimization requires all data structures and functions to be static and written in the system before compilation. Yet the ultimate goal of the system is to keep simulation section independent of model input, thus allowing convenient model switch. Manual optimization is contradictory to the purpose of the study and the model information can only be restricted to a certain degree to include as many biological models as possible. As for the restriction imposed on model information currently, manual optimization in simulation is already completed and will be explained in detail in the next section.

6 Simulation Structure Change

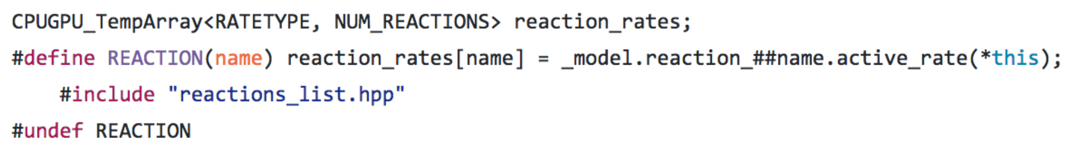
The work in this section involves both re-writing existing functions for more general usage and redesigning the structure of the system to further support improvements made on the model side. As illustrated earlier, model part of the system is now declared through x-macros and thus, it was necessary to redesign simulation sections to connect to and optimize changes made on means of model representation. In addition, to support a special biological mode, the old simulation section as well as numerical methods have minimal configurability and reusability. To address this problem, I intend to design new classes and functions to reimplement the *Euler's Method* integration, and added modularity so that the new system is no longer model specific, but instead can be re-applied to or easily updated based on other biological models.

6.1 Methodology

For further isolation of reactions and other simulation process, a new *context* class is created. In a biological system, it provides a scope for various simulations, and in *segmentation clock network*, each instance of *context* class is equivalent to a cell. There is a unique identifier inside *context* class and it is used as the second template parameter for functions in *model_impl.hpp* shown earlier.

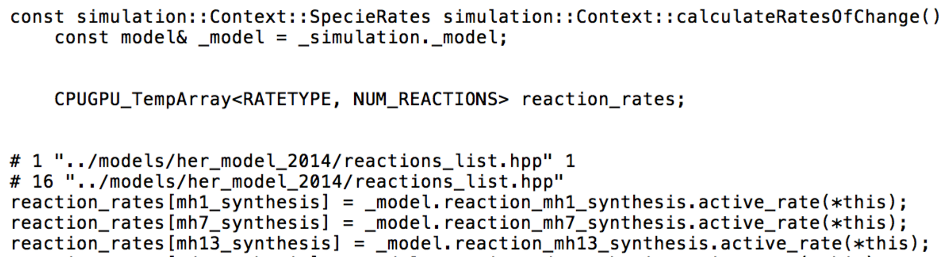
Placed inside simulation, the *context* is essentially a wrapper class inside simulation that interacts with reactions. It encapsulates function implementation and is responsible for calculating rate change and updating concentration levels. It also provides necessary information for calculation and updating concentration levels, including step size and time step. With such functions and a unique identifier, the system can easily iterate through all *context* instances and perform designated tasks with enough information from both simulation and model.

Updating active rates of each reaction is the first step of simulation process. Active rate describes the rates of change at one particular time step for all reactions and is equivalent to the notion of derivative in a differential equation. To update the active rates, functions are designed to cooperate well with *x-macros* used during model declaration; in particular, function is placed inside a for loop over all reactions and is able to automatically generate lines of code to invoke functions for calculating rates for each of the reactions. An example of how *x-macros* works in the system in is shown in *context* implementation file (Figure 6.1). The code is written according to *x-macros*. Earlier the system iterates through each *context* instance to invoke member functions and now the system iterates through all *reaction_ids* and invoker each member function for that reaction. Thus two template parameters together provide easy access to all combinations of member functions and reactions. As shown in Figure 6.2, the preprocessor is able to automatically generate specific code based on the *context* and the *reaction_id*. Along with model separation, new simulation mechanism allows high level of flexibility.



```
CPUGPU_TempArray<RATETYPE, NUM_REACTIONS> reaction_rates;
#define REACTION(name) reaction_rates[name] = _model.reaction_##name.active_rate(*this);
    #include "reactions_list.hpp"
#undef REACTION
```

Figure 6.1: Function for Updating Reaction Rates



```
const simulation::Context::SpecieRates simulation::Context::calculateRatesOfChange(){
    const model& _model = _simulation._model;

    CPUGPU_TempArray<RATETYPE, NUM_REACTIONS> reaction_rates;

# 1 "../models/her_model_2014/reactions_list.hpp" 1
# 16 "../models/her_model_2014/reactions_list.hpp"
reaction_rates[mh1_synthesis] = _model.reaction_mh1_synthesis.active_rate(*this);
reaction_rates[mh7_synthesis] = _model.reaction_mh7_synthesis.active_rate(*this);
reaction_rates[mh13_synthesis] = _model.reaction_mh13_synthesis.active_rate(*this);
```

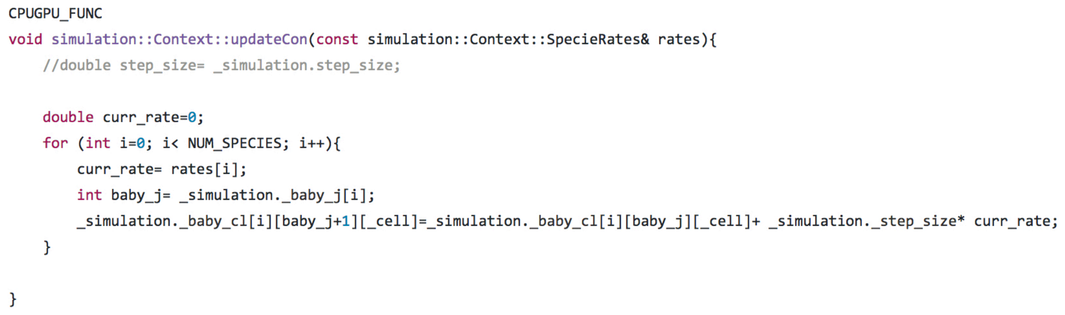
Figure 6.2: Function for Updating Reaction Rates After Preprocessing

X-macros provide a foundation for the reusability of the simulation section as it ensures that the template function is configured to run with any biological models as long as model information is input under specified restrictions.

The second step of simulation is to apply numerical methods to predict concentration levels of all species at the next time step and this is where active rates at current time step is necessary along with concentration level of each species. The new system also implements a simple *Euler's Method* as its numerical solver. In the new system, I first aggregate all reactions as well as their influences on each of the species through *x-macros* (Figure 6.1, 6.3) and then apply them collectively along with concentration level of current step to estimate the concentration level of next time step (Figure 6.4). Through those methods applied, two subsections of updating concentration level is entirely separated from each other, which will then render much high level of freedom in either part.

```
#define REACTION(name) \
const reaction<name>& r##name = _model.reaction_##name; \
for (int j = 0; j < r##name.getNumInputs(); j++) { \
    specie_deltas[inputs_##name[j]] -= reaction_rates[name]*in_counts_##name[j]; \
} \
for (int j = 0; j < _model.reaction_##name.getNumOutputs(); j++) { \
    specie_deltas[outputs_##name[j]] += reaction_rates[name]*out_counts_##name[j]; \
}
#include "reactions_list.hpp"
#undef REACTION
```

Figure 6.3: Function for Collecting Derivatives at a Time Step



```
CPUGPU_FUNC
void simulation::Context::updateCon(const simulation::Context::SpecieRates& rates){
    //double step_size= _simulation.step_size;

    double curr_rate=0;
    for (int i=0; i< NUM_SPECIES; i++){
        curr_rate= rates[i];
        int baby_j= _simulation._baby_j[i];
        _simulation._baby_cl[i][baby_j+1][_cell]=_simulation._baby_cl[i][baby_j][_cell]+ _simulation._step_size* curr_rate;
    }

}
```

Figure 6.4: Implementation of *Euler's Method*

6.2 Results

Results for usage of *x-macros* is clearly illustrated above. This is extraordinarily helpful because iteration through species or reactions takes place across the entire system and now the system is capable of generating code based on the model input.

Updating active rates and concentration level were integrated in form of differential equations in the original system and each reaction may be calculated multiple times since it may contribute to concentration level changes of various species. Below is one of the thirteen differential equations in the original system (Figure 6.5) along with its corresponding implementation of this equation (Figure 6.6). All model information is hard coded into the implementation of *Euler's Method* and almost all differential equations are implemented separately in the original system. Thus, to change the numerical solver, the methodology applied to update concentration levels, of the original system was extremely complicated and time comsuimg.

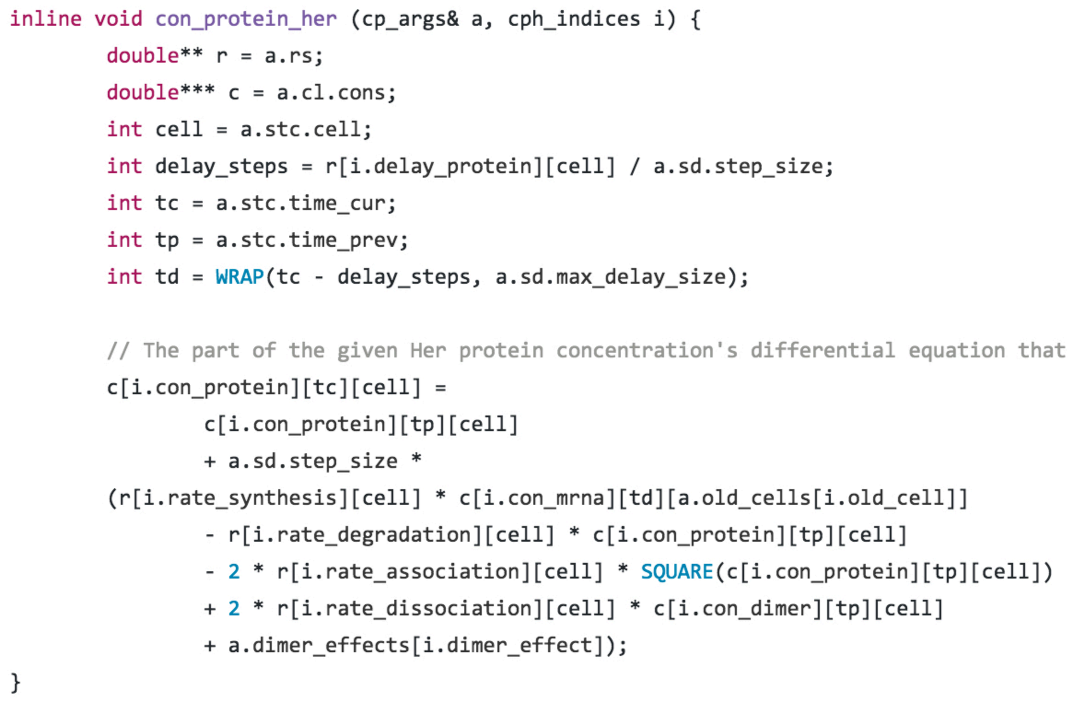
With the changes made in the new system, *Euler's Method* is implemented in one place and could be applied for all reactions. Therefore, if a new numerical solver, such as *Runge-Kutta Method*, is used, implementation of this solver will take place inside just one function and can be applied to all reactions with no further adjustments except storing more past concentration levels and active rates (for example *Runge-Kutta Method* requires concentration levels from multiple time steps).

Dimer Protein Levels

$$\frac{\partial ph_{i,j}(c_k,t)}{\partial t} = dah_ih_j \cdot ph_i(c_k,t) \cdot ph_j(c_k,t) - ddih_ih_j \cdot ph_{i,j}(c_k,t) - ddgh_ih_j \cdot ph_{i,j}(c_k,t)$$

$$\text{where } i \le j \text{ and } i, j \in \{1, 7, 6\}$$

Figure 6.5: Differential Equation for One Reaction



```
inline void con_protein_her (cp_args& a, cph_indices i) {
        double** r = a.rs;
        double*** c = a.cl.cons;
        int cell = a.stc.cell;
        int delay_steps = r[i.delay_protein][cell] / a.sd.step_size;
        int tc = a.stc.time_cur;
        int tp = a.stc.time_prev;
        int td = WRAP(tc - delay_steps, a.sd.max_delay_size);

        // The part of the given Her protein concentration's differential equation that
        c[i.con_protein][tc][cell] =
                c[i.con_protein][tp][cell]
                + a.sd.step_size *
        (r[i.rate_synthesis][cell] * c[i.con_mrna][td][a.old_cells[i.old_cell]]
                - r[i.rate_degradation][cell] * c[i.con_protein][tp][cell]
                - 2 * r[i.rate_association][cell] * SQUARE(c[i.con_protein][tp][cell])
                + 2 * r[i.rate_dissociation][cell] * c[i.con_dimer][tp][cell]
                + a.dimer_effects[i.dimer_effect]);
}
```

Figure 6.6: Implementation of *Euler's Method* for Rection

7 Memory usage

In this section, I will discuss the design pattern used in the new system to cover the flaws in memory usage in the original model. The new design aims to further reduce the unnecessary memory requirement for better space efficiency. While space efficiency is an important factor in system design in general, it is particularly important in our system because less memory requirement for each individual simulation will allow us to run a larger number of simulations in parallel(or only CPU). Smaller memory requirement for each simulation also means less data transfer between GPU and CPU, which is another bottleneck identified during the preliminary stage of the project. Given those two factors, it is easy to see that space efficiency is important to performance of the new system.

7.1 Methodology

One of the main data structure in the system is a large three-dimensional array named *baby_cl*, which holds concentration levels over several delayed time steps for all species and all cells. In parameter input, the length of delay is directly associated with each individual reaction. Since concentration level in *baby_cl* is stored for each species, however, there is no direct attribute in model information that will determine the delay size in *baby_cl*. To handle this problem, original systems decides the number of time steps to keep based on the maximum delay size across all reactions to ensure that each species is kept in the system long enough for possible reactions. However, not all species are related in a reaction with the maximum delay. The maximum delay can be as long as $1,200$ time steps for some reactions such as gene transcription and much shorter for reactions such as mRNA translation; some reactions in the system may not even have a delay. The original system does not fully utilize memory allocated not all species need to be stored for maximum delay to provide history data for future simulations.

To decrease the size of allocated memory for *baby_cl*, I redesigned data structures so that the system will now only allocate necessary space to hold enough history data for future usage. I first added a related species section in the data structure (*reaction.cpp*) describing relations between species and reactions. Next, the system iterates through all reactions to find the species according to reaction.cpp and add delay size of the reaction to a set specific to each species. After all iterations, the maximum delay size needed for a species is the maximum delay within the subset. Based on this information, the system will then create a large one-dimensional array to hold all concentration levels and place different wrappers according to its maximum delay size for each species. For the system to access concentration levels of each species, I created another much shorter one-dimensional array to hold the pointers to the beginning point of each species in the larger array. Each concentration level access starts by locating subsection of *baby_cl* through species id and completes through cell number and time step.

7.2 Results

This design will reduce size of *baby_cl* on running environment greatly and thus allow simulation of more parameter sets at the same time. Using the segmentation clock project for example, the new *baby_cl* data structure uses 70% less memory than the original one. Through this improvement, the number of simulations that can stay on the GPU at the same time increases more than two folds and consequentially, the average runtime for each of the parameter sets will be reduced to one third of its original runtime.

8 Time efficiency

During the preliminary phase of GPU acceleration, several factors prevented us from an efficient GPU implementation. Thus, in the new system, a great amount of effort was spent to address those problems, and now with simulation improvements and space efficiency, GPU acceleration is again implemented to the system, free of previous limitations.

8.1 Methodology

Methodology in converting CPU code into CPU-GPU code follows a similar pattern as demonstrated in the preliminary phase and thus will not be re-introduced here.

On top of the changes mentioned earlier, another attempt to minimize data transfer between CPU and GPU is made through initiating some data structures on GPU to reduce both memory requirement on CPU and data transfer required during simulation. For example, copies of *baby_cl* or *active_rates* may be created on GPU and no initial transfer of those structures from CPU to GPU will be needed during simulation process. Other necessary data transfer during simulation is now performed by CUDA managed memory. It allows data structure to be accessed on CPU when required.

8.2 Results

Simulation of CPU only code runs 1.6 to 1.9 times longer than the original system. Simulation of CPU-GPU code runs roughly 2 times longer than the original code for simulation of a parameter set, which does not fill all the parallel computational units on the GPU. However, parallel simulation of multiple parameter sets on GPU environment shows significant improvement in the new system. As a rough test case, it shows that when simulating twenty parameter sets simultaneously, the original code takes twenty times longer than single simulation, while doing so in the new system takes roughly double the time as running one simulation. Overall, simulation runtime of twenty parameter sets is five times faster than the same simulation through the original system, and with multiple GPU cards connected to the same work station, the new code can run efficiently on the station instead of relying on cluster.

The slight increase in runtime for CPU only code of the new system was a big concern for us and after profiling I found that the extra layer of indirection added to save memory requirement was a major source of time spent. Note that concentration levels are constantly accessed and updated in each time step for all species and all cells and thus number of memory access is very large. This is a price to pay for much smaller memory requirement on CPU or GPU and as discussed above, it is will be compensated through larger numbers of simulations running in parallel on the GPU.

9 Compilation Challenges

In order to maximize the potential reach of the system, I also included a CPU only version of the system. This imposes another low-level challenge to the system, which is to compile the same functions in a different way with different compilers (*g++* for CPU and *nvcc* for GPU).

There are two main parts of my solution. On the simulation side, for functions to be run on both CPU and GPU, I defined a conditional macro called *CPUGPU_FUNC* for function declaration. If the system will run on GPU, it will be compiled with nvcc and when this is the case, I define *CPUGPU_FUNC* as *__host__ __device__*, which is the function declaration necessary for a function to run both the host (CPU) and the device (GPU). When the user wishes to compile the code only on CPU, *CPUGPU_FUNC* will be defined as blank, not changing anything to function declaration. This conditional macro is important since keyword *__host__ or __device__* would make C++ code invalid if included, but are necessary if using GPU acceleration. Another change was in regards to usage of CUDA managed memory, which is used in the new system to transfer data between CPU and GPU. Following a similar design patter, I added another macro called *STATIC_VAR* and declare it as *__managed__* on GPU and nothing for CPU only simulations. Through these two strategies, functions are declared based on the platform on which the simulation will run.

Turning to the model files, I am forced to create different files for different simulation environments since when a static variable is passed to the compiler, *g++* will make assumptions such as the location of the variable, such assumptions are not valid when *nvcc* is used and thus files have to created in versions, normal implementation files (*.cpp*) for *g++* and CUDA implementation files (*.cu*) for *nvcc*. To work with two versions of the implementation files, I added macro declarations inside the header file so when the header file is linked to the implementation file, it will be compatible to either of them. For a similar reason, two versions of the test files are created for CPU only and CPU-GPU simulations.

Files with the two kinds of macros mentioned above can only be included once among all the object files since multiple inclusions will result in multiple declarations of the macros. The location of inclusion is thus an important choice. For the CPU-GPU simulation, I could not place the macros it in the normal simulation implementation file (*simulation.cpp*) since it also serves as the super class for CUDA implementation file (*simulation_cuda.cu*). Thus included in GPU accelerations as well. Therefore, the only place that will determine if a simulation is CPU only or CPU-GPU simulation is the actual test file. One of the header files also includes implementation details and for that file in particular, it was placed inside simulation_cuda implementation file since that will never be included in a CPU only simulation.

10 Validation

One aspect of the new system that is important to biological users is the validity of the system. Is the new system conducting all parts of simulation correctly? One way to test this is to see if the new system can replicate the simulation results of the original system. Controlled tests are conducted to check the accuracy of the simulation in this system. I passed the same set of simulation results into both the original system and the new system for simulation. After the concentration level of each species and each cell are recorded along the entire simulation, I compare the concentration levels from two systems to see if they are identical to each other. These controlled tests demonstrate a 0.3% discrepancy in final simulation results after $60,000$ time steps (equivalent to 600 minutes, a common length for the *segmentation clock* project).

This slight discrepancy in simulation results may stem from the differences in implementation of the system. As described in the simulation section, the original system updates concentration level of a species based on the corresponding differential equation, which only counts the influences of a reaction on that particular species at a time. This process will be repeated for each of the species and if a reaction is related to multiple species, it will be included in multiple differential equations. On the other hand, the new system gathers all active rate changes that one reaction may impose on various species and then the system updates the concentration level for each of the species collectively. Potential rounding off error may arise since the order of two sub processes is switched. But overall, this is a relatively small error especially after $60,000$ time steps and will not affect any system level characteristics of the biological network.

11 Conclusion

11.1 Contributions

I have constructed a new system that allows for easier model switching and independent simulation updates. Additionally, this system reduces the memory requirement on CPU and GPU computing environments, and improves the overall runtime in simulating multiple parameter sets.

A copy of the system is available at:
https://github.com/CarltonYang/DelayDifferentialEqnSimulator-1
Note to compile the system with GPU acceleration, *nvcc* version 7.0 or above is needed. In addition, a NVIDIA GPU card is need to run the test files with GPU acceleration. A make file is provided with the system.

At this stage of the project, much of the simulation process is complete and desirable results have been attained. There are potential improvements, however, that can be implemented to the system in the future.

11.2 Future Work

One potential improvement to the system is to add a more user-friendly interface for model input. Compared to the original system, the current model information is represented in a systematic and well-organized manner which is already intuitive. Yet there is potential for more user-friendly interface. By implementing such an interface, the software can maximize research potential by reducing the time expended for tedious manual input. Also, the software reduces the possibility of programming error in entering model information.

An alternative improvement would involve the addition of a new simulation mechanism in the system. Currently, a deterministic simulation, solved by Euler?s Method, is employed to mimic the regulatory network of the zebrafish segmentation clock. A more realistic and comprehensive biological model can be constructed by applying probabilistic simulation methods. Under such a simulation scheme, probabilistically determined propensities and reaction times are used to decide which reactions fire at each iteration. Reactions with higher propensities are more likely to fire. Since stochastic simulation typically requires more resources than deterministic simulations, such as memory and computation power, stochastic simulation methods were omitted in the original system. With major improvements in space and time efficiency, stochastic simulations, such as the next reaction method, may now be feasible in the new system.

A further improvement to the system could involve developments to the feature extraction section. Profiling of the original model demonstrated that feature extraction occupying nearly thirty percent of the entire simulation runtime. Under the original system structure, all concentration levels were saved during simulation and then later utilized for feature calculations. Conversion of feature extraction onto GPU might further improve system runtime. This is because feature extraction is necessary for all parameter sets and mutants. In addition, every determined block on GPU will have relatively the same amount of work to complete and thus most of the blocks can run in parallel and resource waste will be kept as minimum.

Bibliography

[1] T. Akiba and Y. Iwata. Branch-and-reduce exponential/FPT algorithms in practice: A case study of vertex cover. *Theoretical Computer Science*, 609, Part 1:211–225, 2016.