USING MASSIVELY PARALLEL EVOLUTIONARY COMPUTATION ON GPUS FOR BIOLOGICAL CIRCUIT RECONSTRUCTION

by

CHULWOO LIM

(Under the Direction of Khaled Rasheed)

ABSTRACT

A fundamental and ubiquitous difficulty of systems biology is identifying relevant model parameters. A genetic network model of the biological clock of Neurospora crassa that is quantitatively consistent with the available RNA and protein profiling data was proposed. However, the oscillating nature of biological models poses more challenge for identifying model parameters due to the high dimensional complex search space and computational cost of numerically solving ODEs. In this work, an Evolutionary Algorithm leveraging the GPU architecture is proposed. Our implementation identified promising model parameters with a speedup of two orders of magnitude compared to the CPU implementation.

INDEX WORDS: Evolutionary Computing, Biological networks, Graphical processing unit, Runge-Kutta method

USING MASSIVELY PARALLEL EVOLUTIONARY COMPUTATION ON GPUS FOR BIOLOGICAL CIRCUIT RECONSTRUCTION

by

CHULWOO LIM

B.S., Korea University, South Korea, 2008

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2013

USING MASSIVELY PARALLEL EVOLUTIONARY COMPUTATION ON GPUS FOR BIOLOGICAL CIRCUIT RECONSTRUCTION

by

CHULWOO LIM

Major Professor: Khaled Rasheed

Committee: Thiab Taha
Tianming Liu

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
December 2013

ACKNOWLEDGEMENTS

I would like to thank Dr. Rasheed for his guidance and patience in helping me complete this thesis. I would like to thank Dr. Bernd Schüttler for providing the biological data and guiding me through understanding and solving the problem. I would also like to thank Dr. Taha and the NVIDIA CUDA teaching center at UGA for the support on CUDA programming and providing the device for running the experiments. I would like to thank William Dale Richardson for his help with proofreading and correcting the English for this thesis.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Page

CHAPTER 1

INTRODUCTION

1.1 The biological circuit reconstruction problem

A living system can be viewed as a chemical reaction network described as a "biological circuit," in which genes and their products are represented by nodes in the circuit. Constructing an operating circuit model requires the computation of effective rules to simplify complex circuits. A genetic network is introduced as a hypothesis to explain how genes and their products control the biological system to define a complex trait [1]. A complex organization of gene regulatory networks controls the overall behavior of an organism, and allows it to adapt to the surrounding environment and stimuli. A gene regulatory network allows the individual organism to regulate gene expression starting from the interacting DNA segments (genes) in a cell and controlling expression levels of mRNA and proteins (signaling pathway). A single, simple network controlling a model cannot explain or predict the entire complex nature of a gene regulatory network. Mathematical models of gene regulatory networks should function as physical models of natural gene networks. In some cases of modeling it was proven to give a reliable prediction which can be tested experimentally. One of the dependable approaches to modeling is using Ordinary Differential Equations (ODEs) [2]. Several other modeling methods involve Boolean Networks [3], Petri Nets [4], Bayesian Networks [5], graphical Gaussian models [6], Stochastic [7], and Process Calculi. Figure 1 presents a genetic network model of the biological clock of Neurospora crassa in ODEs.

This model proposed in Yu, Y. et al. [9], is quantitatively consistent with the available RNA and protein profiling. In the figure, boxes represent molecular species, arrows entering circles identify reactants, arrows leaving circles identify products, and bidirectional arrows identify catalysts. The relationships represented by arrows are defined by the rate equations in Figure 1. Identification of the genetic network is important in molecular level quantitative genetics because it enables one to integrate diverse experimental information and predict the behavior of complex traits, such as the biological clock in this model. The circadian clock has a characteristic of interconnected network portraying system approach. A systems approach can be used to identify oscillator circuits. Various kinetic models were applied to explain the periodic nature of the biological clock.

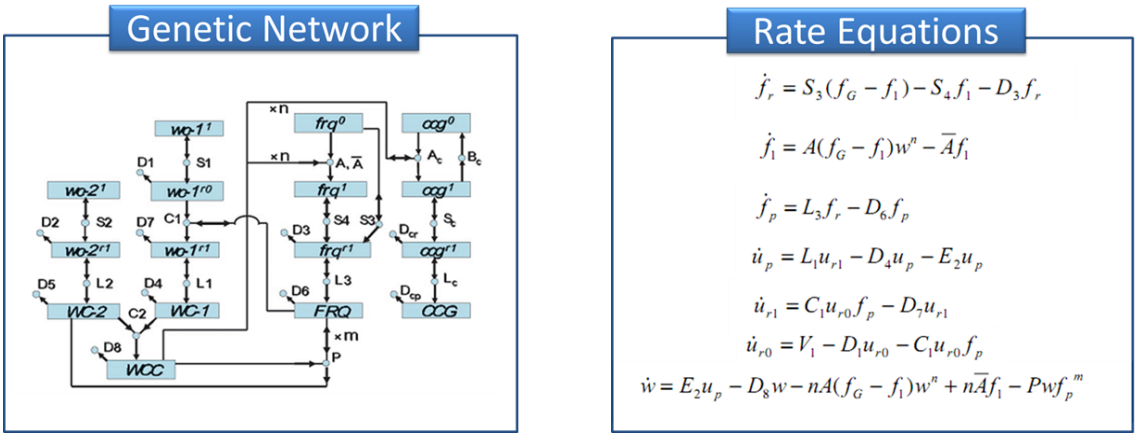

Figure 1: Genetic Network Model and Rage Equations(ODEs) defining the system

Yu, Y. et al. [9] have applied a human analytical assistant approach to guide the search for model parameters that present the oscillating behavior of the network model. The mathematical characteristic of the network model producing oscillation was enforced to the parameters selected. The goal of our method is to perform the search with purely

data-driven guidance (i.e. without any human enforcement) to find more accurate solutions more quickly. Neurospora crassa is a type of red bread mold that is often used in biological experiments because it is easy to grow and has a haploid life cycle, which makes genetic analysis simple [10]. A biological clock is an internal mechanism in organisms that controls the periodicity of various functions or activities, such as metabolic changes, sleep cycles, etc [11]. This periodicity appears with 24 hour frequency of oscillation as shown in Figure 2. The circadian clock triggers rhythms of biological activities including cellular functions, development and growth in all aspects of a living system. The assembly of a consistent, functional nature of the biological clock warrants the extensive measurement of various parameters of the clock function. In addition, oscillations should be mathematically identified by amplitude and phase, and it is absolutely necessary to maintain consistent periodicity in the circadian oscillation. Although much experimental data on the biological clock have been compiled at great effort and expense, mathematical modeling may not predict the deeper nature of the biological clock. Recently devised kinetic linear mathematical models of the clock network lack a rationally-definable overall integration of the models. The massive divergent and disparate signaling to and from the clock should be appropriately interconnected to the clock system. The circadian network model should resolve modeling and experimental efforts.

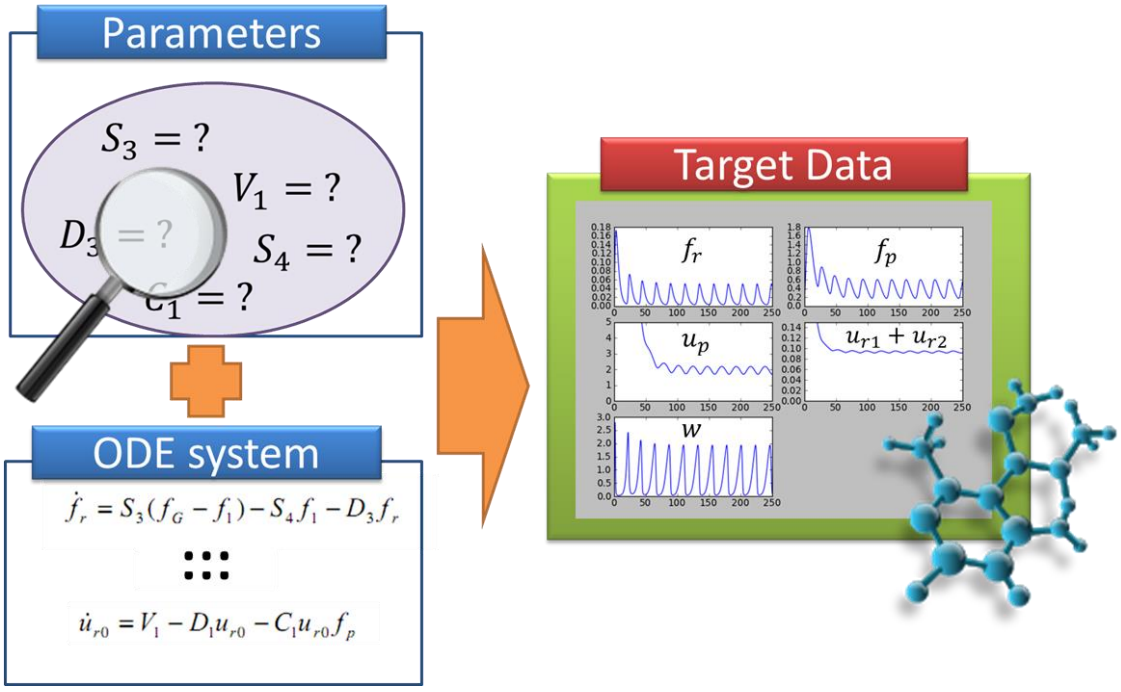

Figure 2: Finding Parameters for Genetic Network Model

1.2 Searching model parameters and evolution strategy

A fundamental and ubiquitous difficulty of system biology is that the relevant model parameters are not completely known while the available and meaningful experimental data may be sparse and noisy. This makes it difficult to unravel the basic nature of the biological activities such as the circadian clock network. There have been numerous attempts to connect the oscillator circuit to the kinetic modeling, but they appear to be fragmentary to explain circuit mechanism. An integrative approach to build up the circuit model through the suitable algorithms is necessary. The search space of model parameters is extremely complex due to the oscillating nature of biological clock models. The relationships between variables are non-linear and highly complex, and defined by an ODE system. The biological clock model targeted in this thesis has 20

coefficients and 7 initial conditions. This high dimensionality of the search space makes exhaustive search impractical. The only known way of evaluating the genetic network is to plug the coefficients and initial conditions into an ODE solver which attempts to numerically compute the output of the defined system to compare it with the profiling data. The ODE system is very sensitive to changes in the coefficients. Small changes in parameters may yield drastic changes in model behavior, making the system numerically unstable. This makes our search space very discontinuous with regions of un-evaluable points. Population based evolutionary algorithms are suitable for problems with such complexity, high dimensionality and discontinuity in the search space. In particular, the algorithms appear to exhibit reliable performance in such discontinuous spaces where analytical search would fail. Karr, C. L. et al. [12] demonstrated the potential for searching for parameters for curve fitting with evolutionary algorithms. Simonsen, Martin, et al.[13] efficiently used the GPU computation power in Differential Evolution. Their work demonstrates the potential of GPU acceleration for real world application of Evolutionary computing. Many Evolutionary algorithms were implemented on GPUs. Longo, G., & Ventre, G. [14] implemented a general framework for genetic algorithms on a GPU architecture which was intensively tested and validated on massive Astrophysical data classification problems. Cano et al. [15] implemented three different approaches to run Genetic Programming on GPUs. Cárdenas-Montes, Miguel, et al. [18] implemented a Particle swarm algorithm on GPUs. Franco et al. [16] parallelized the fitness evaluation on a GPU demonstrating the potential for speedup of evolutionary learning systems. Jaros,J. et al. [17] demonstrated an effective use of Multiple GPUs with Island-based genetic algorithms. Ramirez-Chavez et al. [19] implemented GPU-Based

Deferential Evolution for solving the gene regulatory network model inference problem achieving significant speedups by utilizing the GPU. Their work is on a similar problem with GPU and evolutionary computation as ours, but the network model we are dealing with is much more complex, and their approach is therefore not suitable. We have selected an Evolution Strategy to utilize the GPU on fitness evaluation of the circuit reconstruction with an Island model approach to utilize both a multi-core CPU and GPU. We have selected the Evolution Strategy to maximally utilize the GPU architecture, as explained in the next section.

1.3 Evolution strategy and GPU architecture

Computer algorithms can be applied to exploit the biological circuit starting from the initial conditions and all the way to the output of the system under diverse emergent behaviors. It can highlight the relevance of setting a variable parameterization of the model. The exploration of dimensional parameter spaces would permit us to explore the system functioning throughout the broad spectrum of conditions and to produce statistically meaningful results. The methodologies include parameter involving analysis, sensitivity analysis, and parameter estimation. Most of these methodologies involve the repetitive application of simulation. General purpose computing on graphics processing units (GPUs) is an emerging technology that reduces computation time for many applications. GPUs are designed specifically for processing graphics which requires the parallel processing of thousands of triangles and rasterizing them to millions of pixels in a fraction of a second. A stream is a set of records that require similar computation. The GPU is designed to utilize this data parallelism in the streams of computation required for

rendering thousands of triangles. Data parallelism has focuses on distributing the data and is different from Task parallelism. Task parallelism is a common form of parallelism when we have for example multiple cores or servers running in parallel to achieve fast computation. However, in data parallelism each thread will focus on distributing large amounts of data and applying similar tasks to the data. To deal with the demands for processing thousands of triangle meshes to render 60 or more frames per seconds, GPUs have evolved to become massive parallel streaming processors which can handle GFLOPS of stream operations with high bandwidth as shown in Figure 3. To achieve this special purpose, the GPU architecture became quite different in architecture from the traditional CPU as shown in Figure 4. The GPU devotes transistors to data processing more than data caching and flow control that CPUs do because it is specialized for computation-intensive, highly parallel computation – exactly what graphics rendering is about. The GPU is especially well-suited for problems that fall into the category of data-parallel computation with high arithmetic intensity [20]. A GPU is built around an array of Streaming Multiprocessors (SMs) and each SM runs the program in units of blocks as shown in Figure 5. The streams of work get grouped into multiple blocks and get dispatched to SMs by the GPU scheduler. This enables the GPU architecture to be scalable to different hardware settings ranging from supercomputers to mobile devices.

Another reason for adopting GPUs in computations is their low energy consumption. The Top500 supercomputers list shows that GPU powered systems are growing. China's Tianhe-1A (one of the supercomputers in the top of the list), which uses more than 7,000 GPUs, consumes about half as much power as the CPU powered Jaguar does. The GPU based Tsubame has 92% fewer servers and consumes only 1/7th the

power that the CPU based Jaguar consumes and still shows competitive performance compared to the Jaguar. Industries outside the supercomputing community are also adapting GPUs to yield benefits. HESS (a major oil and gas firm in the US) replaced a 2,000 CPU cluster with 32 Tesla S1070 (GPU) servers and reduced power consumption from 1340 kwatts to 47 kwatts thus reducing an operational cost of 2.3 million dollars to only 82 thousand dollars. Today, more than 20 energy firms are in the process of migrating to GPU-based processing, including Chevron, Schlumberger and BR Petrobras [21].

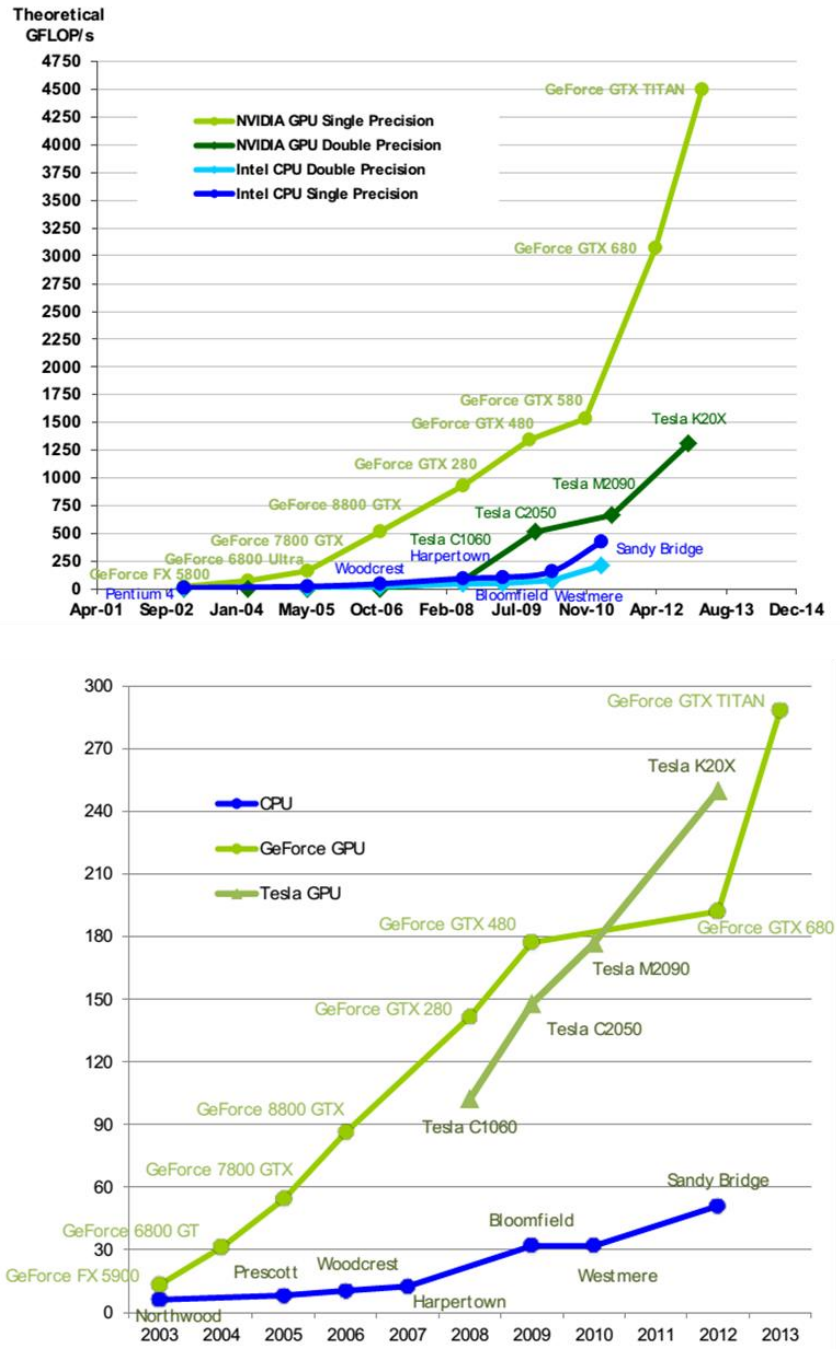

Figure 3: FLOPS and Memory Bandwidth for CPU and GPU (Adapted version source: NVIDIA CUDA Programming Guide Version 5.5 [20] available at http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf. Last accessed November 2013)

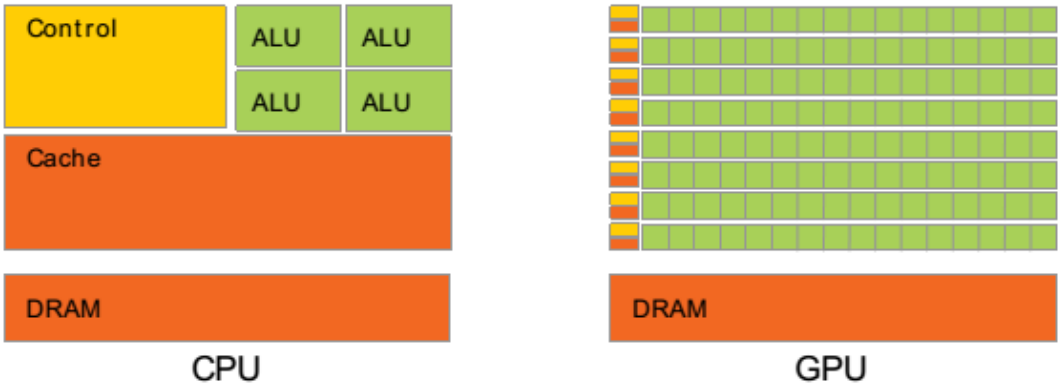

Figure 4: CPU and GPU architecure (Adapted version source: NVIDIA CUDA Programming Guide Version 5.5)

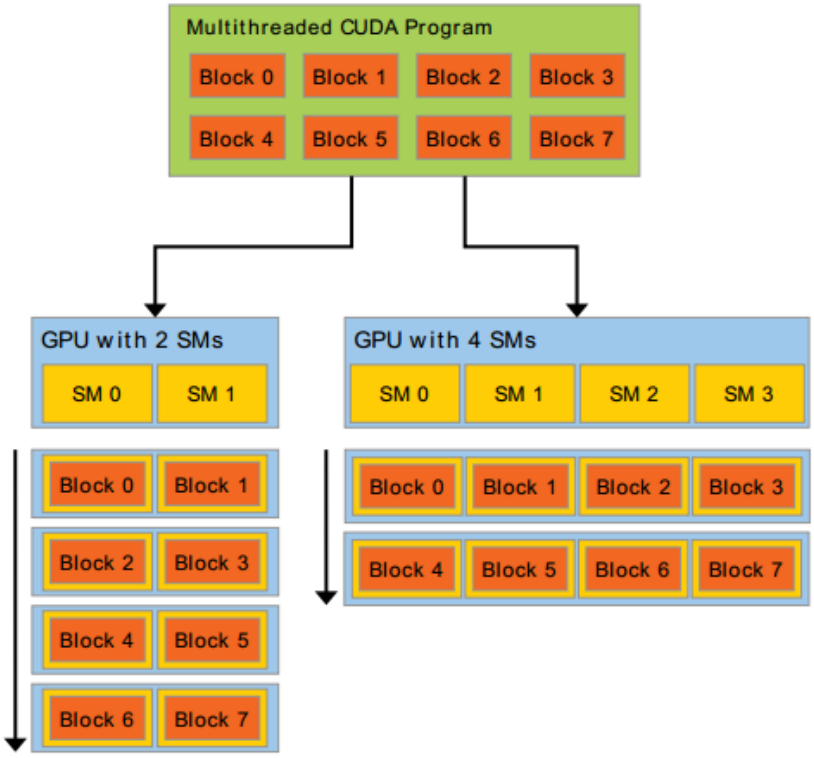

Figure 5: Automatic scalability with Streaming Multiprocessors on different GPUs (Adapted version source: NVIDIA CUDA Programming Guide Version 5.5 [20])

GPUs have multi-level parallelism in the form of threads, blocks and grids as shown in Figure 6. A thread is the smallest unit that can execute a sequence of

instructions. A block is a group of threads which is assigned to a SM (Streaming Multiprocessor). Multiple blocks are executed in parallel by multiple SMs, while threads run in parallel within the block. However, threads within the block cooperate closely by having a synchronization barrier point and shared memory space which enables fast communication between them. Our fitness calculation for the Evolution Strategy especially fits well to massive streaming GPU architecture because an Evolution Strategy requires fitness evaluation of all children only at one point every generation. This is important because we need to stream enough parallel work to the GPU to saturate the computing power of the device. Otherwise we have to frequently launch the GPU calls from the CPU, and the GPU computation-launching overhead will be more than the performance benefit we gain from running the task on the GPU. In our implementation, we can stream a massive population to the GPU and execute massively parallel fitness evaluations. This will return fitness values to the CPU for the entire child population at one point in time in each generation. In this manner, the parallel tasks streamed to the GPU are sufficient to saturate the full GPU computing power. In our method, we assign 16 individuals to each block with each individual utilizing 8 threads (8 threads communicating through shared memory within the block). To get the fitness of a total of 3584 children (the parent population contains 512 individuals and the child to parent ratio is set to 7 as explained in the next chapter) we create 224 blocks (16 x 224 =3584) and stream them to the GPU to run the computation at each fitness evaluation step. The fitness of 3584 children consumes 8 threads for each which results in a total of 28672 threads launched on the GPU. When there is a sufficient number of thread blocks assigned to the GPU device, the GPU scheduler assigns multiple blocks to a single SM.

The benefit of having multiple blocks on a SM is that the SM can juggle the blocks hiding the latency of any operation by utilizing the hardware. For example, if block A is fetching some data from the global memory (which takes hundreds of clock cycles), the SM will set aside block A for a moment and run block B's arithmetic operations and possibly block C next, etc. Multiple blocks will be assigned to one SM to always keep the SM busy. This way, even though some memory operations are very slow, one can hide that latency by executing other available instructions while waiting for the data to arrive.

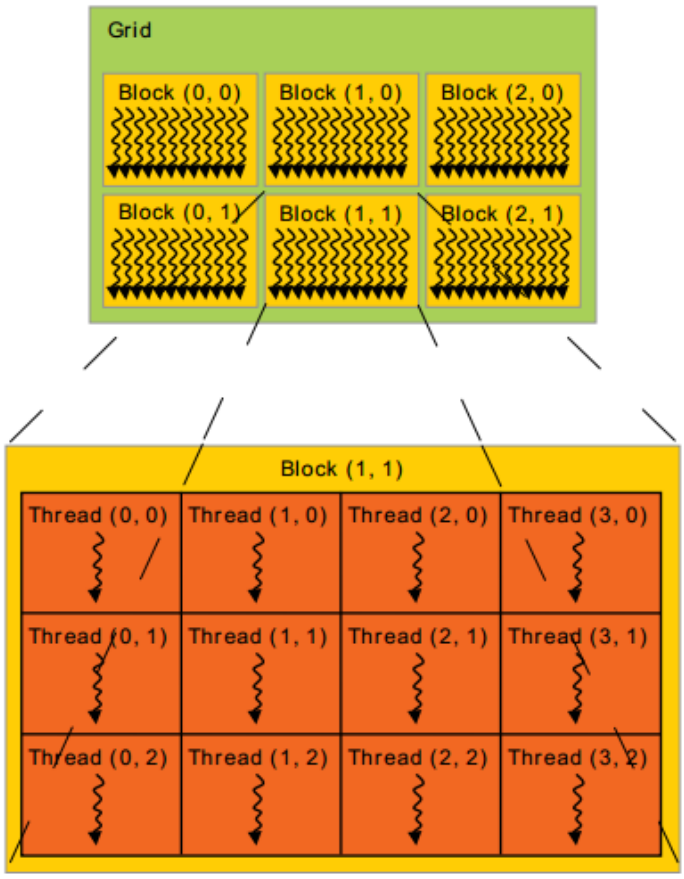

Figure 6: Grid of thread blocks

(Adapted version source: NVIDIA CUDA Programming Guide Version 5.5 [20])

The rest of the thesis is organized as follows: Chapter 2 begins with an introduction to the Evolution Strategy. Then the implementation details and our strategy are described. Chapter 3 continues with utilization of the massive parallel GPU architecture in our work. Chapter 4 presents the experimental results. Chapter 5 contains the conclusion and suggested future work.

CHAPTER 2

EVOLUTION STRATEGY

2.1 Overview

The general algorithm of the Evolution Strategy that we have adopted for biological circuit reconstruction is depicted in Figure 7. An initial population of randomly generated candidate solutions forms the first seeded generation. Then, it goes through the evolutionary process of parent selection, recombination, mutation, fitness evaluation and survivor selection to produce the next generation [22]. Multiple iterations of this evolutionary process lead the population search through the solution space towards an optimal solution.

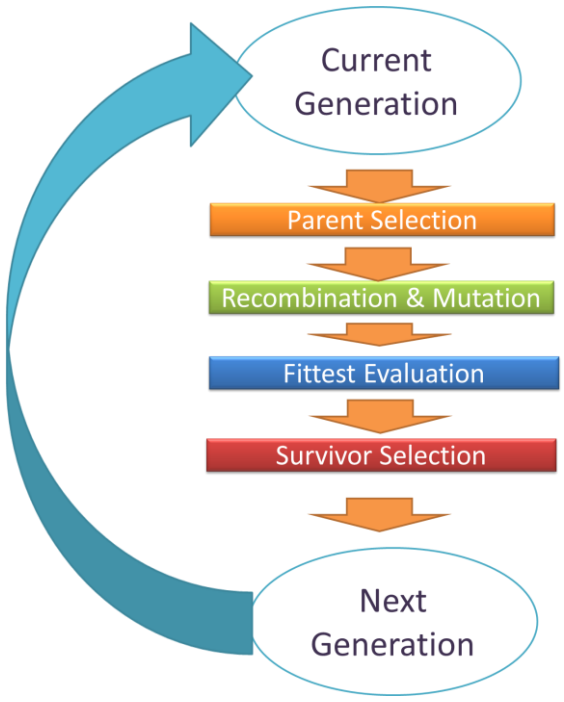

Figure 7: Overview of Evolution Stretegy

A technical overview of the Evolution Strategy used for our circuit reconstruction is given in Table 1. By leveraging the massive parallel computation power of the GPU, we were able to handle a large population size (512 on four islands) efficiently.

Table 1: Technical Overview

Representation	Continuous real value for rate coefficient and initial condition
Recombination	Local discrete on object variables & Global intermediary on strategy variables
Mutation	Gaussian perturbation
Parent selection	Uniform random
Survivor selection	(μ,λ)
Population size	Four islands with 512 on each
Niching	Fitness Sharing

The circuit reconstruction process with the Evolution Strategy is depicted in Figure 8. Each individual of the population encodes the realized unknown parameters of the ODE system. The genes define the numeric ODE system which can be solved by numerical methods. We assign a fitness value to each individual by actually solving the ODE system numerically and comparing the result with the target data. Having a fitness for each individual, we can go through the evolutionary process to evolve the population towards the matching solution for the targeted system.

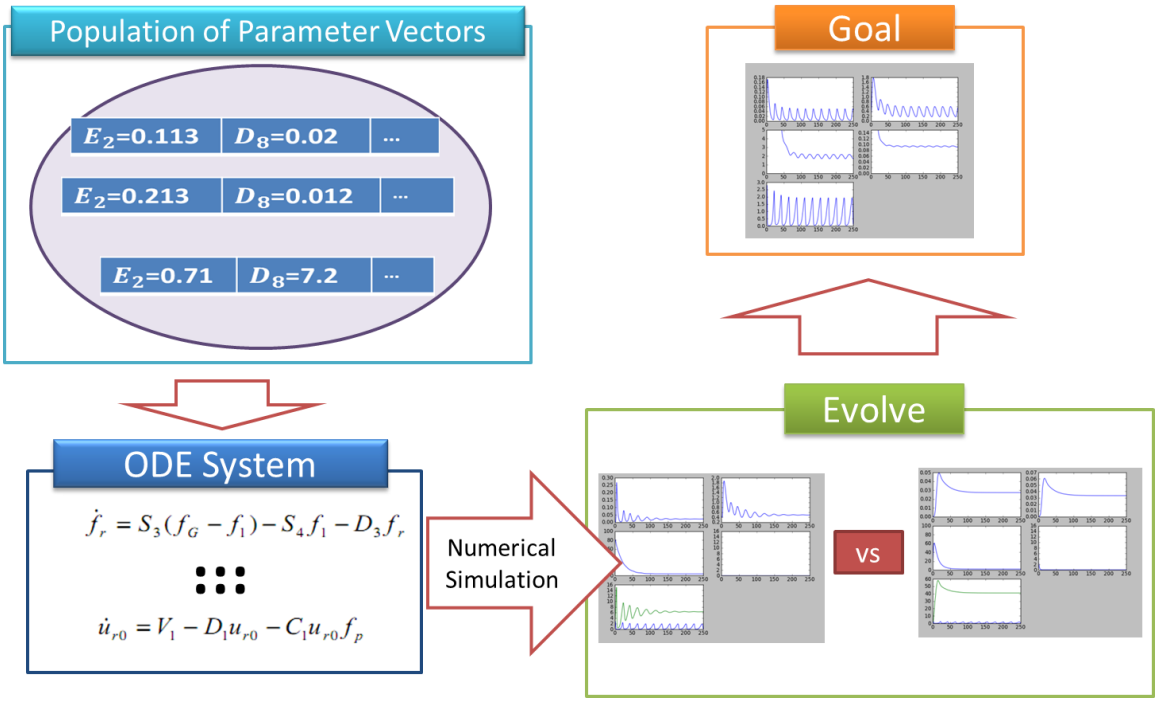

Figure 8: Overall Circuit reconstruction process

2.2 Representation

Each individual consists of real-valued genes representing rate coefficients and initial values of the ODE system as shown in figure 9. The values have bounds of 0.0 and 100.0, but in the implementation we used a very small floating point value ϵ instead of 0.0 to avoid numerical instability. Each individual also has a set of strategy variables representing the standard deviations for mutating the object variables which represent the ODE parameters.

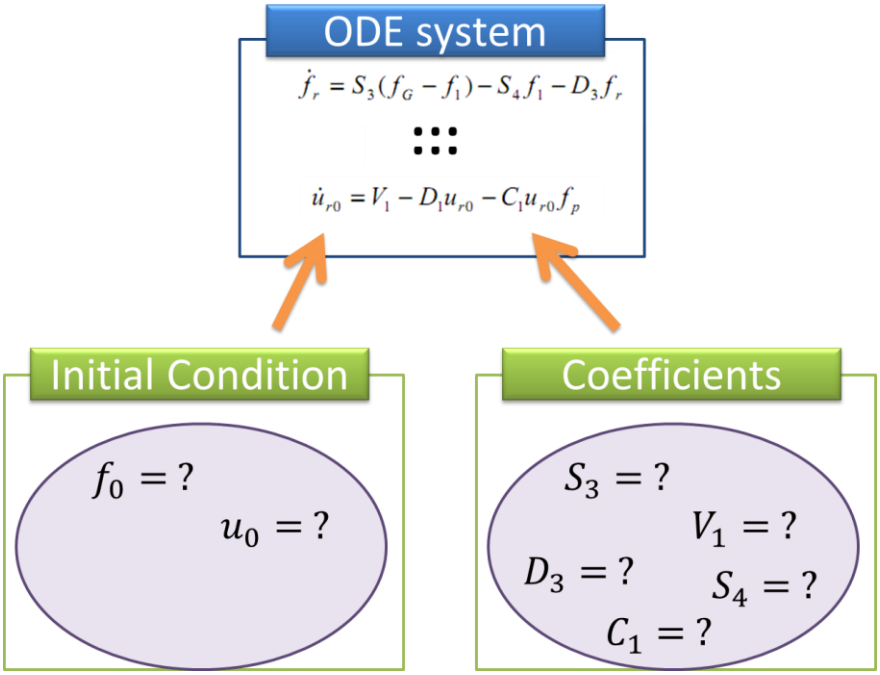

Figure 9: Initial Condition and Coefficients to define a Genetic Network Model

With the given genotype we can solve the ODE system numerically to produce the phenotype as shown in Figure 10. The genotype in evolutionary terminology is an organism's full hereditary information which is encoded by genes. The phenotype is an organism's actual observed properties. There is not a one-to-one mapping between genotypes and phenotypes because it is possible for different parameters to produce the same series as output. However, our target is to find one gene encoding for a given protein profiling data series. The existence of at least one genotype for a targeted phenotype is sufficient for our goal. Initially, candidate solutions will form a population to start the evolutionary process (Figure 11).

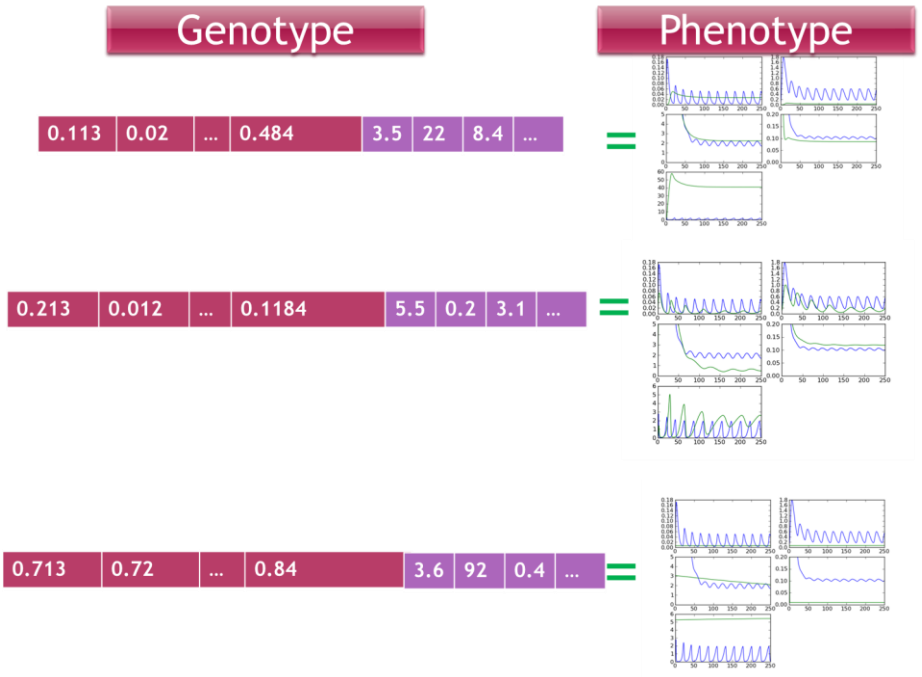

Figure 10: Genotype and Phenotype of representation

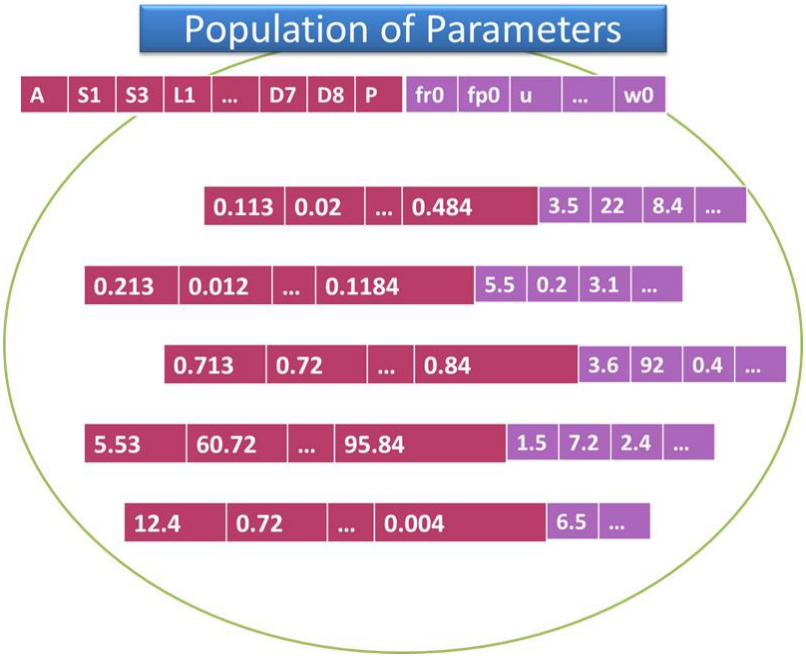

Figure 11: Population of candidate solutions for circuit reconstruction

2.3 Operations

2.3.1 Recombination

In our Evolution Strategy, the object variables are the rate coefficients and initial condition of each variable. Rate coefficients are constants which define the reaction rate between chemicals on genetic network model. Rate coefficients have very close relationship to each other in a system, and they fall into certain scaling range for each system. For this reason, we perform local recombination by selecting genes from only two parents for object variables. Also, the rate coefficient search space is not smooth and continuous, meaning that from range A to B there could be critical points which break the whole system even though most of the values between A and B may give stable solutions to the system. Consequently, we use discrete recombination to randomly choose one gene from either parent which allows us to avoid invalid ones in between. The experiments below confirm that this local discrete recombination strategy works fairly well for our problem domain.

In contrast, we use global intermediary recombination for strategy variables. Global intermediary recombination generates the values from any pair of parents for each gene and selects the intermediary values of the selected genes from the parents. This is much more aggressive than local discrete recombination. Aggressive evolution of strategy variables results in a more diverse perturbation space for mutation, resulting in a better chance of leaving local optima. The recombination probability is set differently on each island with a strategy that is discussed in section 2.5.

2.3.2 Mutation and parent selection

We use uncorrelated mutations with *n* strategy variables followed by the corresponding object variables which gives our chromosome a $\langle x_1,\ldots,x_n, \sigma_1,\ldots, \sigma_n \rangle$ format. There is an overall learning rate τ' and a coordinate-wise learning rate τ. The learning rate controls the range over which the variable is portable, indirectly controlling the mutation range. The formula used for mutation is described in Figure 12, (N(0,1) is a sample drawn from a Normal distribution with mean 0 and standard deviation 1). Our search space has significant slope differences by gene and by region of the search space. It is therefore appropriate to have a strategy variable for each object variable. As in a typical Evolution Strategy, we select parents from the population with uniform random probability. Each selected candidate will go through mutation to produce a child and would go through recombination according to the recombination probability we have setup for each island.

$$\sigma'_i = \sigma_i \times e^{\tau' N(0,1) + \tau N_i(0,1)}$$

$$x'_i = x_i \times \sigma'_i \times N(0,1)$$

$$\tau' \propto \frac{1}{\sqrt{2n}}, \tau \propto \frac{1}{\sqrt{2\sqrt{n}}}$$

Figure 12: Mutation formula

2.3.3 Survivor selection

A typical Evolution Strategy is designed to create λ children from μ parents with the children being more than the parents. We adopted the recommended default setting of $\lambda \approx 7 \bullet \mu$, having 7:1 ratio. The (μ,λ) selection strategy, selects survivors (the μ parents of the next generation) from among the λ children, excluding all parents from the

competition. Another common selection strategy is $(\mu+\lambda)$, in which parents are included in the competition. As will be shown below, (μ,λ) gives much better performance in our experiments because (μ,λ) selection gives a better chance to leave the local optima.

2.4 Fitness evaluation

Measuring the similarity of output signals for different ODE systems is a challenging task. We have to carefully measure multiple sets of time series data to rate the similarity of the ODE systems' outputs. The naïve idea of measuring the accumulated sum of differences of each point between the two series is very deceptive and likely to lead us to local optima. In this section, we describe a fitness measurement formula specially designed for a class of ODE systems which generates oscillating behavior in the output signal.

2.4.1 Shape matching - Correlation and Detrending

Shape and pattern are both important features of any signal. In a biological clock system, the oscillating behavior and its frequency are key features of the system. Our focus is to quantitatively measure the similarity of the shape and pattern from the output signal of our candidate solutions. As in previous works [9, 19], this shape measuring was often ignored. Thus, we have adopted the Pearson Correlation measurement in our fitness function aiming for this feature. The Pearson Correlation coefficient is a well-designed statistical tool to measure linear correlation between two variables. It gives a quantitative measurement in the range from 1.0 to -1.0 with 1.0 being most positive and -1.0 being most negative correlation. This measurement lets us screen the oscillation feature of each output series.

Figure 13 shows curves having high correlation in different frequency ranges. To measure the oscillating feature, we have to target the most relevant frequency range. The Pearson Correlation is a good measurement for shape, but it often gets dominated by a low frequency factor in the shape. For example, if there is a global increasing trend like the green curve shown in Figure 13(a), the green curve and blue curve have a high correlation even though they are completely off on higher frequency factors. To target the higher frequency, we need to remove the low frequency factor (the global trend) from the signal and measure correlation. Figure 13(b) shows two signals with low frequency factor removed and now they will have high correlation only when they match in high frequency factors as shown in the figure. We have used b-spline curve fitting to detect the global trend and removed it from the signal when measuring the shape with the Pearson Correlation function.

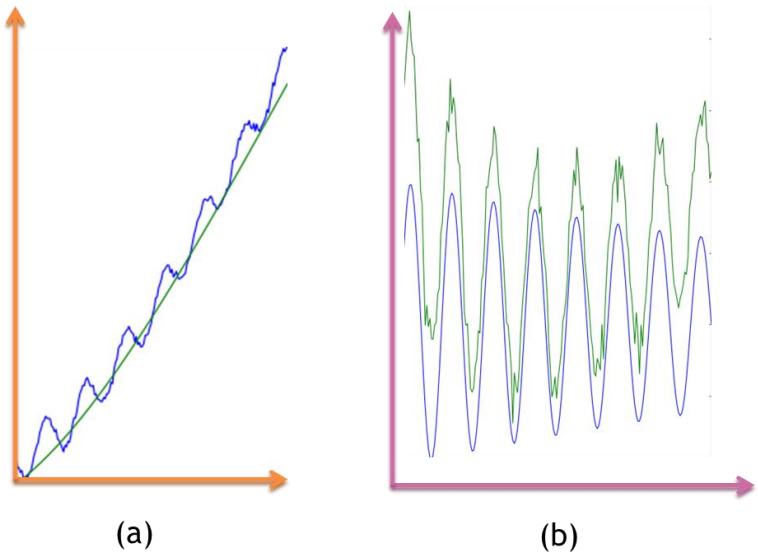

Figure 13: Global trend and higher frequency component of the signal

(a) low frequency global trend line (green) on the curve (blue)

(b) high frequency sine curve(blue) fitting the noisy target curve(green)

2.4.2 Absolute difference – Chi-square

With Pearson correlation measuring the shape of the output signal from the ODE system, we still cannot ignore the magnitude of the signal. Figure 14 demonstrates the effect of the χ^2 monument on our fitness measure. The blue curve is the target fitting curve and the green curve is the solution found from an evolutionary process guided by the Pearson Correlation error measure. The signal matches perfectly in shape and frequency but the magnitude gets really off track. However, without the correlation measurement, the Evolution Strategy can be easily deceived by local optima as shown in Figure 15. The left-hand side of Figure 15 demonstrates local optima where the search can fall when the fitness lacks the correlation measure. On the other hand, the right-hand side of Figure 15 shows a point which is much closer to the global optimum than the left-hand side. Even though this candidate solution has a much lower score on the χ^2, the actual parameters are very similar to the global optimum.

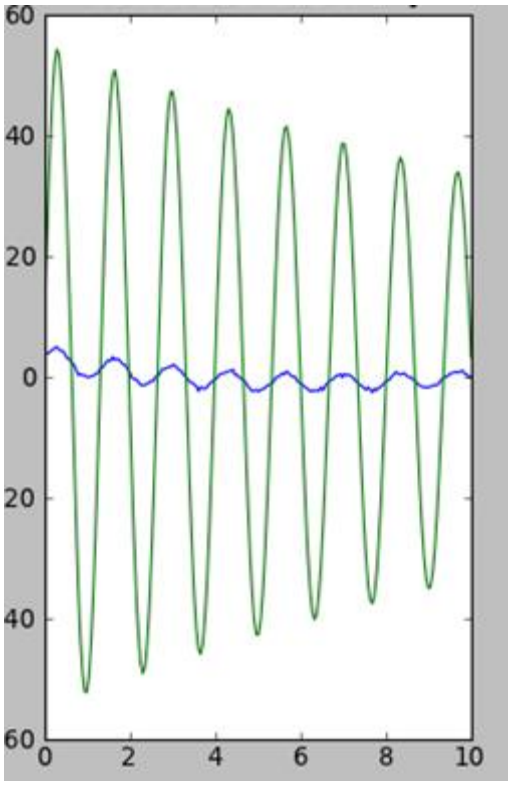

Figure 14: Measuring without χ^2

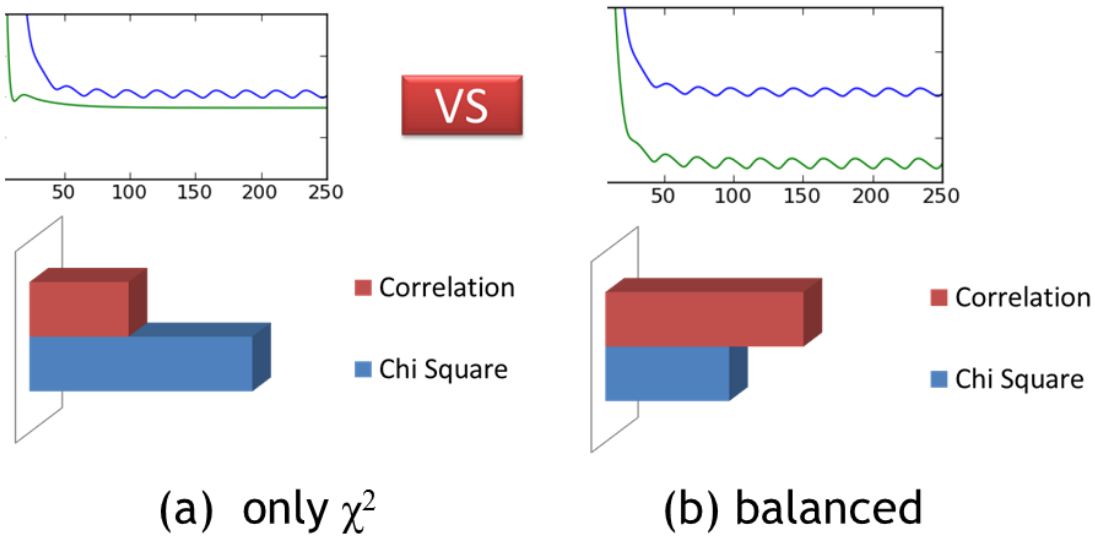

Figure 15: Strong χ^2 score candidate and balanced score candidate

2.4.3 Fitness function for circuit evaluation

In our biological clock ODE system model, there are a total of seven variables, each of which represents a biological species. Among those seven, only four species can be quantitatively measured from protein profiling data. For this reason, only four observable variables can be used to guide the circuit reconstruction. By combining the correlation factor and χ^2 factor of four variables, we define the fitness function f(θ) shown in Figure 16.

$$f(\theta) = (3.0 - 2.0 \cdot C_0)^4 (3.0 - 2.0 \cdot C_2)^4 (3.0 - 2.0 \cdot C_4)^4 (3.0 - 2.0 \cdot C_5)^4 \times (K_0 dy_0 + K_2 dy_2 + K_4 dy_4 + K_5 dy_5 + KA)/KA$$

$$where \quad dy_i = \sum_{t=1}^{n} \left| \frac{(F_t^i(\theta) - y_t^i)^2}{y_t^i} \right|$$

$$and \ C_i = \frac{\sum_{t=1}^{n} (F_t^i(\theta) - \mu_{F^i(\theta)})(y_t^i - \mu_{y^i})}{\sqrt{\sum_{t=1}^{n} (F_t^i(\theta) - \mu_{F^i(\theta)})^2} \sqrt{\sum_{t=1}^{n} (y_t^i(\theta) - \mu_{y^i(\theta)})^2}}$$

Figure 16: Fitness function for circuit reconstruction

2.5 Island model and reseeding

The Island model is a popular approach for evolutionary computation in parallel computing environments. The Island model helps maintain diversity by imposing separation between populations while they evolve. Each island evolves for a number of generations and then undergoes a periodic migration of individuals to stimulate evolution.

We have characterized islands as progressive and conservative islands. Progressive islands keep a high recombination rate to aggressively explore the search space and retain the diversity of the population. Conservative islands focus on searching intensively near promising candidates by having low recombination rate. The migration is performed in a ring structure as shown in Figure 17. The conservative solutions (green

arrow) will migrate to stimulate progressive islands and progressive solutions (red arrow) will migrate to stimulate conservative islands on each epoch. We achieved the best results when our epoch was 150 generations.

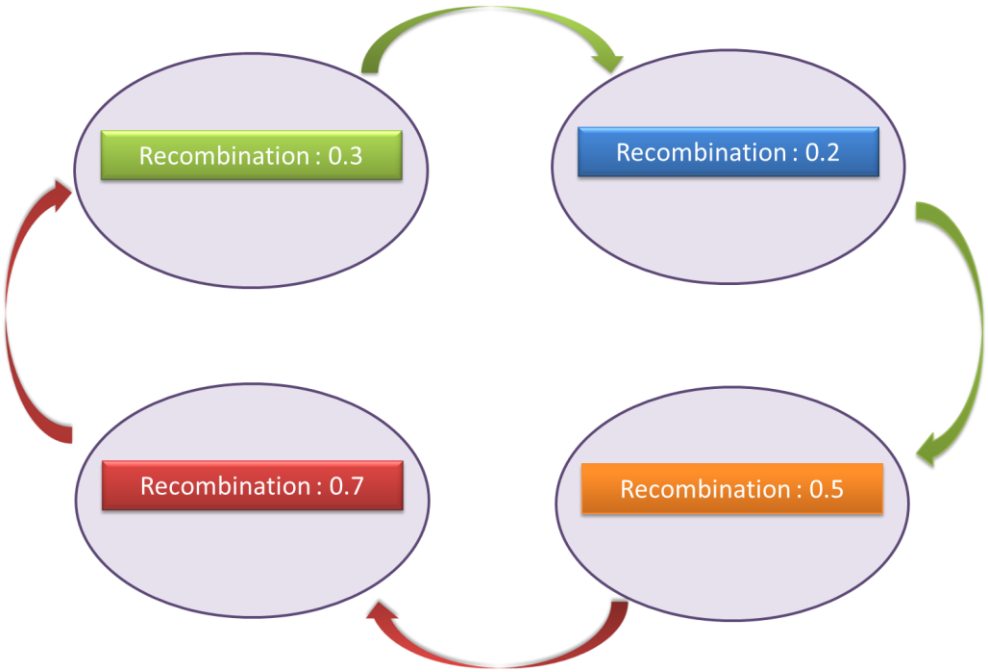

Figure 17: Progressive and Conservative Islands stimulating each other

.

Reseeding the population is an effective way of escaping local optima. When there is no improvement on the evolution process for many generations, it is helpful to start the evolution again with a new random population. Our islands fully restart when there is no significant improvement for over 30% of the evolutionary process. This mechanism enables our island model to be fault-tolerant to poor initial seeding that may lead the population to local optima. Without the acceleration by the GPU architecture, reseeding would be a challenging task because of the computational overhead, but GPU acceleration enables us to reseed the population many times within a couple of hours.

CHAPTER 3

ACCELERATING EC ON GPU ARCHITECTURE

Solving an ODE system with the Runge-Kutta Numerical method is inherently a sequential process. In our ODE system, computing the first order derivatives of seven equations could be parallelized, but this is not sufficient for the massive parallelism of a typical GPU architecture. On the other hand, solving a population of ODEs with different parameters (rate coefficients and initial conditions) can effectively leverage the massive parallelism of the GPU architecture.

3.1 Exploiting the massive parallelism

The GPU architecture is designed to operate on SIMD instructions (SIMD is an acronym for Single Instruction, Multiple Data) which exploit data level parallelism. The graphics card we are using (the GTX 480) follows the Cuda 2.0 compute capability standard which groups every 32 threads as one unit called “Warp”. This means that it is most efficient when 32 threads run the same instruction on multiple data fed to each thread. When a conditional branch inside this unit (one warp) diverts threads into two separate instruction paths, a set of threads which fulfill the condition will take path A and the others will take path B. As shown in Figure 18, threads go to sleep when they are not executing instructions according to a branch condition. If we assign seven equations from our ODE into one warp to compute, this will result in running seven times slower.

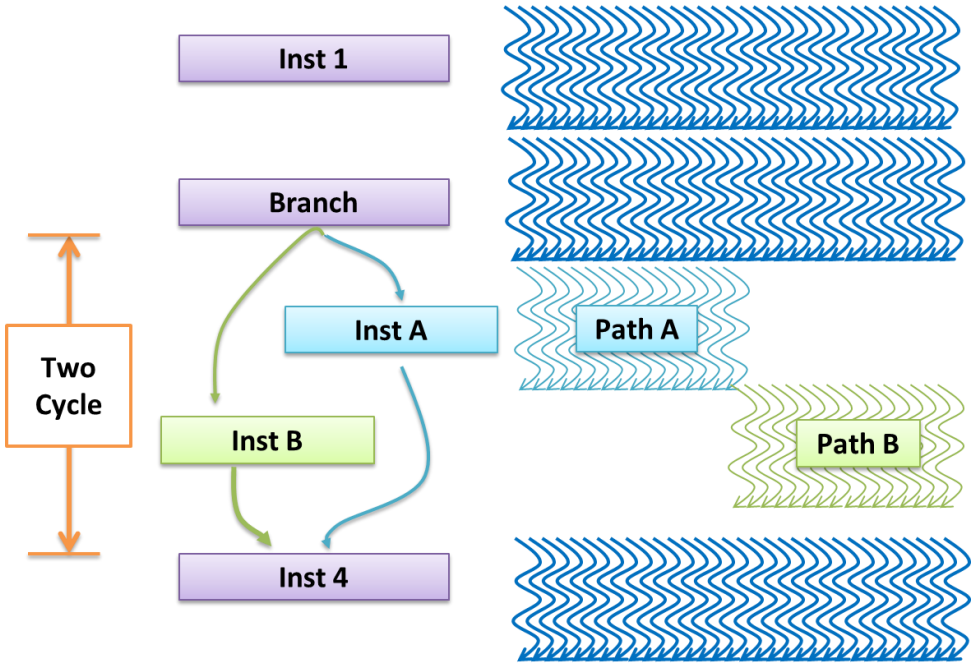

Figure 18: GPU branch divergence

The major computational cost of solving ODE systems often resides in computing the Y Prime portion. In our biological clock model, Y Prime functions are defined by the rate equations. Our biological clock genetic network model also has a high computational load for computing this Y prime portion. The major challenge of branch divergence comes in when we try to distribute the seven equations to parallel threads. Simply assigning each equation to a thread will cause seven way branch divergences which may take seven times more time and have only 1/7 of each thread being activated at a time. To avoid branch divergence and fully exploit the massive parallelism, we have grouped 16 different ODE systems having 16 sets of different parameters (each $|\theta|=24$) to assign for each thread block and run on SIMD fashion as a group. The 16 individuals (which are ODE systems having each different coefficients and initial conditions) are assigned to one block in GPU. Then each ODE system occupies 8 threads for computing the seven

equations which makes the thread block size 8×16. It is possible to assign 32 sets to a block, with no branch divergence on each warp, but assigning 16 sets of individuals per block was found to be more efficient. The details are given in section 3.2. Overall, the computation of the Runge-Kutta order 4 algorithm simultaneously solving 16 different systems on each streaming multiprocessor of GPU is depicted in Figure 19. Figure 19(a) shows the pseudo code of the Runge-Kutta order 4 algorithm which the 16 different systems go through. Figure 19(b) depicts the Runge-Kutta execution of 8×16 threads in a single GPU block having 8 threads assigned for each ODE system. A total of 8×16 threads proceed through one time step at a time to compute the value of each variable at a given time point. First, each group of 16 threads is split into running Y $Prime_1$ for seven equations. The 16 threads will run in SIMD fashion, running the same instructions used to compute equation *x* in Y $Prime_i$ (i is 1 to 4 as described in Figure 19) but having different data for the coefficients and variables according to which individual the thread belongs to. Then, each thread performs a common Runge-Kutta operation like multiplying k_i by h or dividing in half and storing the computed k_i value in the GPU block-wise shared memory so that other threads can access it. The next Y $Prime_{i+1}$ function splits again but all the values of k_i are computed from Y $Prime_i$ which were stored in shared memory on the previous step. One important thing to recall is to synchronize all the threads after each Runge Kutta operation by placing a block-wide synchronize barrier. Otherwise, the value fetched from shared memory will fall into race condition, and the correct value will not be guaranteed.

(a)

```
while t < tEnd:
    k1=yPrimeFunc(t, Yi, p)*h;
    k2=yPrimeFunc(t+h/2,Yi+k1/2,p)*h;
    k3=yPrimeFunc(t+h/2,Yi+k2/2,p)*h;
    k4=yPrimeFunc(t+h,Yi+k3,p)*h;
    Yi=Yi+(k1+2*k2+2*k3+k4)/6;
    t+=h;
```

(b)

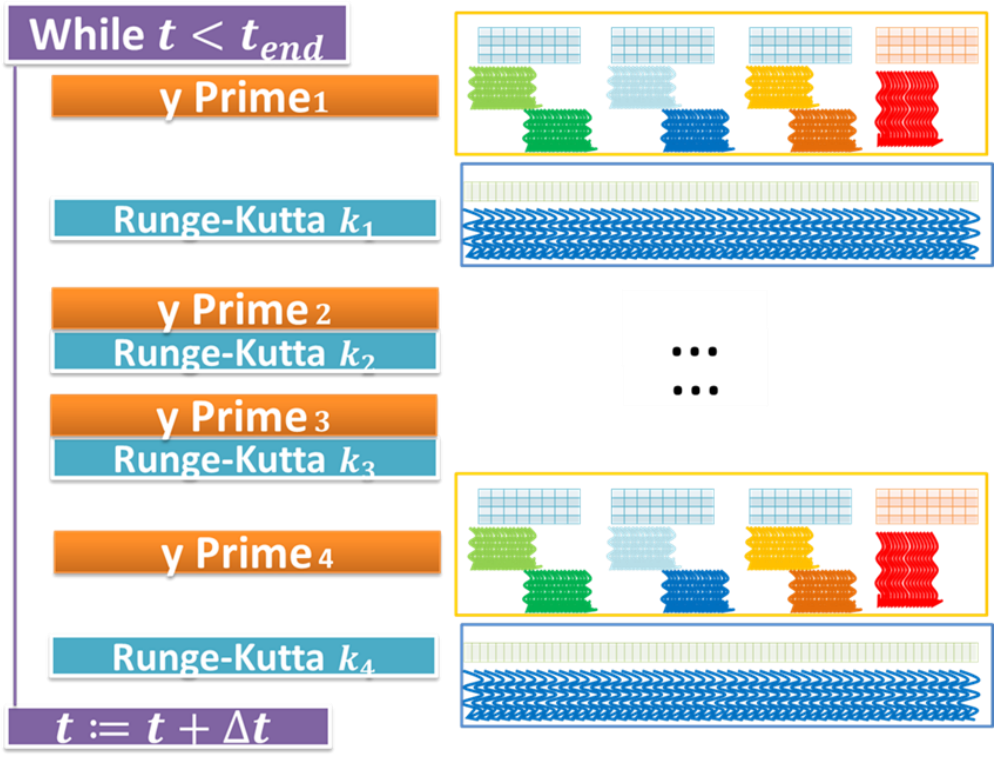

Figure 19: Overall ODE solving algorithm on GPU

3.2 Load balancing the computation

In each block, we have to compute 16 instances of rate equation 1 for each individual. The parallel computation of equations 1 through 6 is shown in Figure 20. However, Equation 7 has twice as much computational load. If we assign the same resources to equation 7, the other threads must wait for equation 7 to be computed. So our strategy is to balance the workload by allowing time sharing (one branch divergence) on equations 1 through 6 and giving full compute time to equation 7 as shown in Figure 20.

One warp will compute equations 5 and 6 while another warp computes equation 7. This is the reason that we did not assign 32 individual ODE systems to a block, which would not allow any branch divergence. We gained almost 1.5 times speedup by assigning 16 individuals to each block.

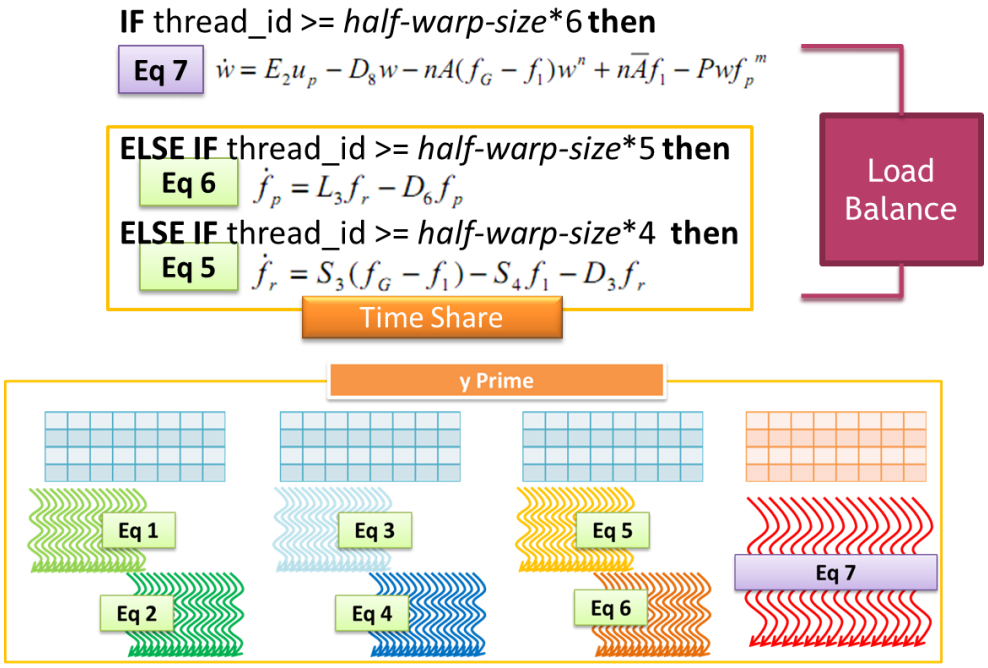

Figure 20: Load balancing the computational load on rate equations

3.3 Efficient shared memory communication

On Y Prime computation, many equations require the values of other variables from previous compute steps. Since the result of each computation is saved to a register which is private to each thread, we need to efficiently communicate those results between threads. The GPU memory access hierarchy is shown in Figure 21. We can store the results of each calculation in the global memory of the GPU so that every other thread can gain access to them, but that would be very inefficient. The GPU has shared memory inside each streaming multiprocessor for this special purpose which is much more

efficient. This shared memory's access scope is limited to the GPU block, which works in our case. The result of the computation is copied into shared memory after each Y Prime calculation. Then we put a block-wise synchronization barrier to make sure that the data is multi-thread safe. Each Y Prime computation will now read other variables' up-to-date values by accessing the shared memory.

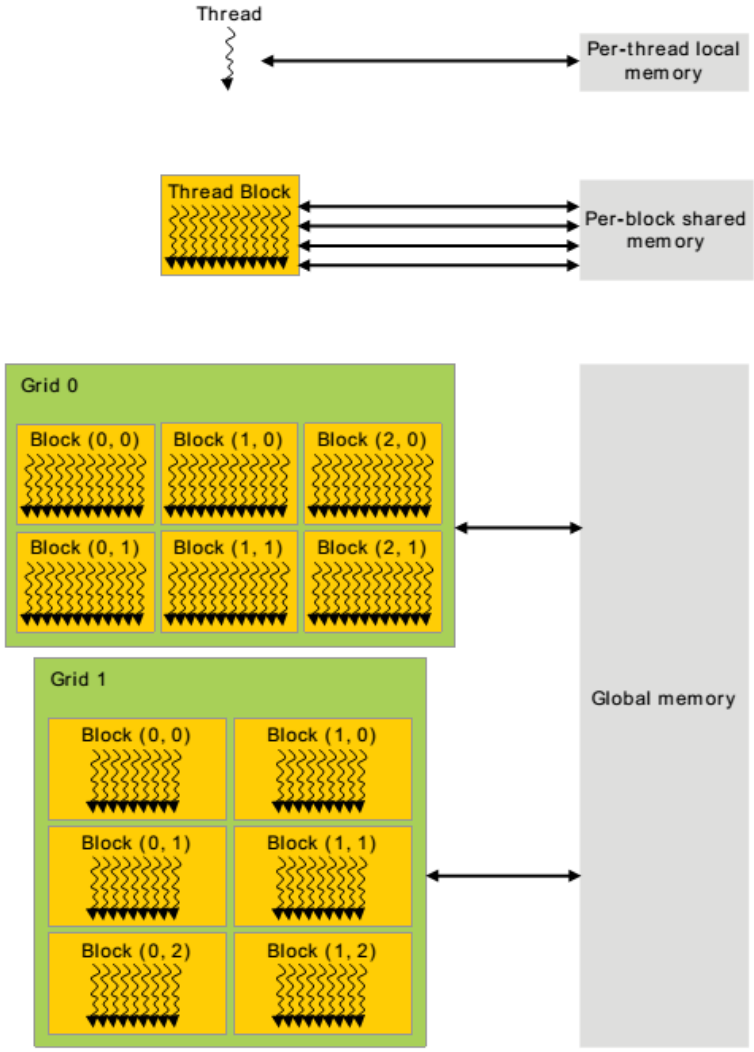

Figure 21: GPU memory access hierachy (Adapted version source: NVIDIA CUDA Programming Guide Version 5.5 [20])

3.4 Hierarchical caching

In massive parallel computation, memory bandwidth is often a bottleneck. An efficient strategy to resolve this issue on GPU architectures is hierarchical memory caching. The GPU architecture provides a hierarchical memory structure as shown in Figure 22. With this architecture, we transfer the massive data in chunks from global memory to shared memory and cache most frequently used data into registers. Since accessing global memory takes hundreds of clock cycles, it is very important to avoid frequent data fetching from global memory. It is better to fetch large chucks of useful data at each access. In our implementation, the parameter values stored in global memory are transferred to shared memory as a large, aligned chunk at the initial phase to fully utilize the global memory bandwidth.

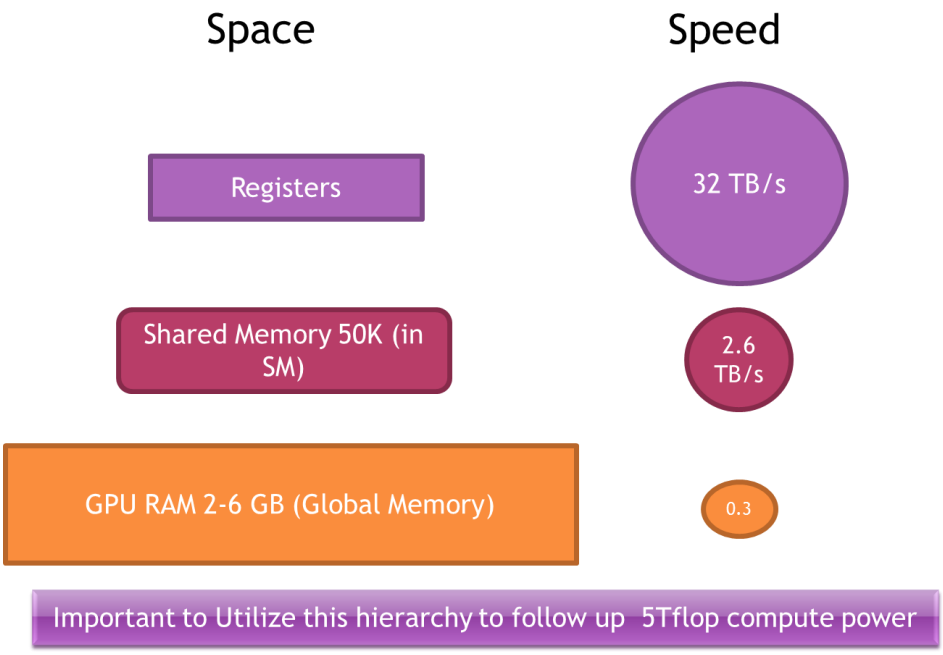

Figure 22: GPU Memory size and bandwidth hierarchy

We have applied a special strategy for register caching. In our genetic network model, each individual has 18 parameter values which are used in Y Prime computation throughout the loop from the start time point to the end time point. To reach a high degree of precision, we have set the time step as 0.005 time unit. Thus it takes 50,000 loops to compute the whole time span of 250 units. Each GPU thread has a very small number of registers available which makes it impossible to save all 18 parameter values of the ODE system to registers. However, for each rate equation, only a few coefficients are required for computation. We utilize only four registers per rate equation and save only up to four parameters that are essential for each rate equation as shown in Figure 23.

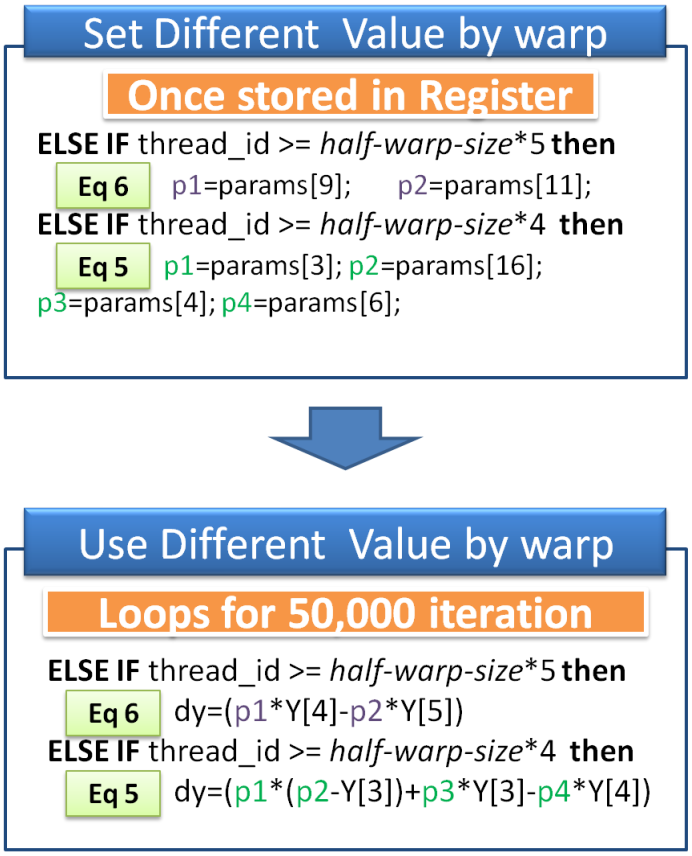

Figure 23: Caching different parameter variables for each thread

3.5 Distributing computational load on GPU and CPU with Island model

Using the massively parallel strategy described above, we have greatly accelerated the massive Runge-Kutta algorithm which is the core of the fitness evaluation function. Since most of the computational load is in fitness evaluation, this already gives us a significant speedup. However, with the island model, we can achieve further gain by executing CPU and GPU computation in parallel. As shown in the two-island model in Figure 24, CPU Core 1 executes all the genetic operations for Island 1, excluding fitness evaluations, and then it streams the data to the GPU. The GPU then runs all fitness evaluations of the submitted individuals from CPU core 1. While the GPU is computing fitness, CPU Core 2 runs genetic operations for island 2. Again, when CPU Core 2 streams the individuals to the GPU, CPU Core 1 gets the fitness values streamed out from the GPU. In this manner Islands 1 and 2 run genetic operations in parallel with GPU fitness computations. We therefore gain the extra benefit of earning more CPU time to spend on genetic operations. With more CPU time being available, we were able to use a high level language (Python) for genetic operations without sacrificing performance. Using Python gives us considerable flexibility in implementation and a fast prototype development cycle. Python lets us pass function arguments as map data structures, giving us the freedom to pass in any configuration for any step as a single set instance. Even though Python has poor performance as a disadvantage, our island model alternating GPU and CPU hides that disadvantage and lets us enjoy the full flexibility of the high level language. In our final implementation, we fully utilized the four CPU cores and the two GPUs in our machine to run four islands of Evolution Strategy as in the four-island model in Figure 24.

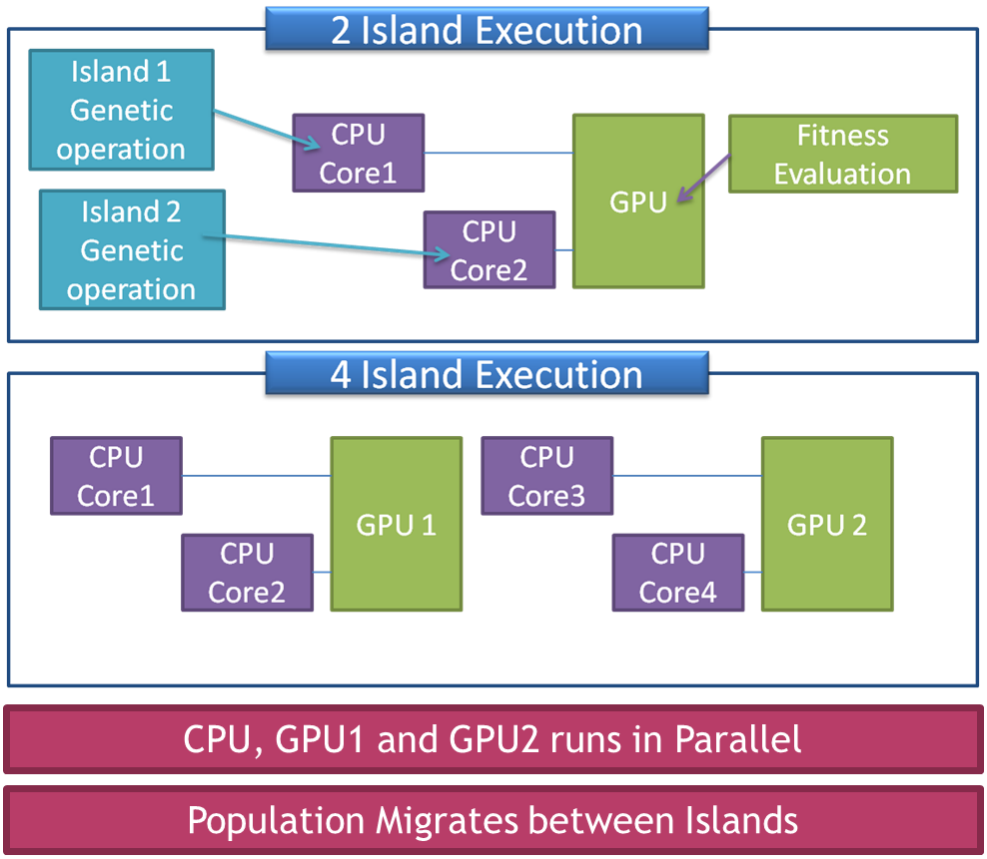

Figure 24: Utilizing both CPU and GPU computing power with Island model

CHAPTER 4

EXPERIMENTS AND RESULTS

4.1 Circuit reconstruction for biological clock system

We have generated synthetic data as our target data series from one of the parameter sets in an ODE system in [9] to evaluate the circuit reconstruction ability of our method. The ODE system parameters in [9] were computed by a Monte Carlo algorithm with strong constraints on the search space with analytical background knowledge of the ODE system itself. The Monte Carlo method used was not truly data driven because of the influence of a human expert on the search process. In contrast, our method is purely data driven having no analytical constraints at all. Our only constraint on the search space was setting the boundaries of the range to 0.0 and 100.0. This range was chosen because a rate coefficient is positive by definition and 100.0 is a reasonable limit that cannot be exceeded in a realistic genetic network. Our method is able to consistently find the solution shown in Figure 25 within 2 hours on multiple trials whereas the method in [9] was never able to find that solution without enforced analytical constraints to assist in searching the space. The green curves are the outputs from our identified solution and the blue curves are the target data series given to our method. In our search process we do not include the variable w in our fitness function because this is a species we cannot directly measure. However, it is an important modulation variable which controls the oscillation behavior, and as Figure 25 shows, the frequency perfectly matches the target data series.

The experiments we use to evaluate our method are described in the following sections. GPUs played a vital role in this work, because the GPU parallel implementation cut down the run time of the algorithm from 310 hours to 2 hours, which enabled us to run multiple runs with various experimental settings.

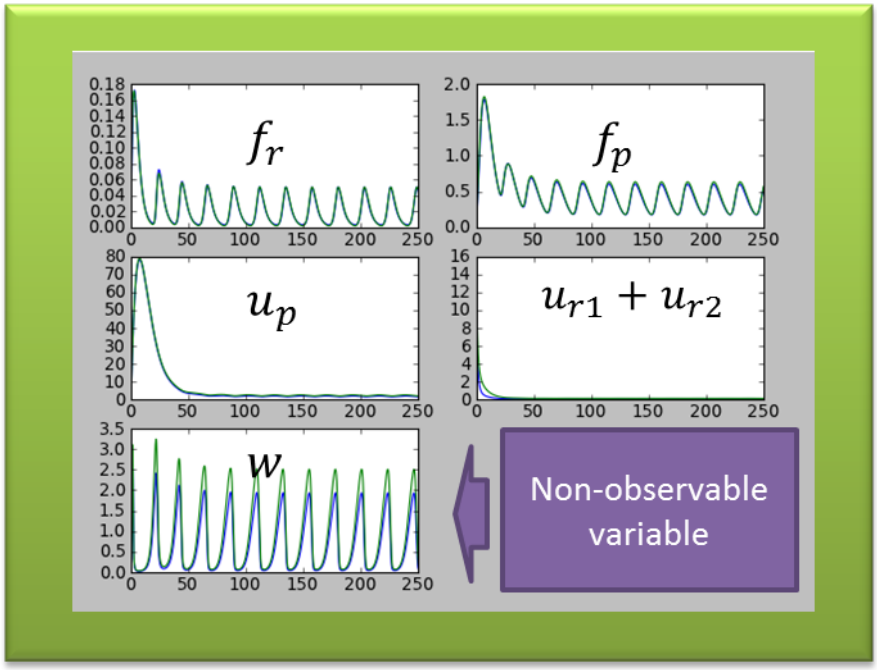

Figure 25: Identified solution by our method

4.2 Comparison with Particle Swarm Optimization

To examine the performance of our Evolution Strategy, we compared it with the Particle Swarm Optimization [23] approach. The same population of 512 particles was generated and a ring topology was used to select the neighborhoods of the particles. We experimented with neighborhood sizes of 5, 10, 15 and 20 respectively. Each particle maintains its own velocity, though it also gets influenced by local and global best velocities and locations. Particles will move around the search space with their velocity to reach the optimal solution. The best solutions found with Particle Swarm Optimization

and Evolution Strategy in multiple runs are shown in Figure 26. Particle Swarm Optimization did reach promising solutions for two runs, but on the rest of the runs it was apparently trapped in local optima. It seems that the PSO evolutionary process finds relatively low error solutions in the early stages with no sustained improvement over time.

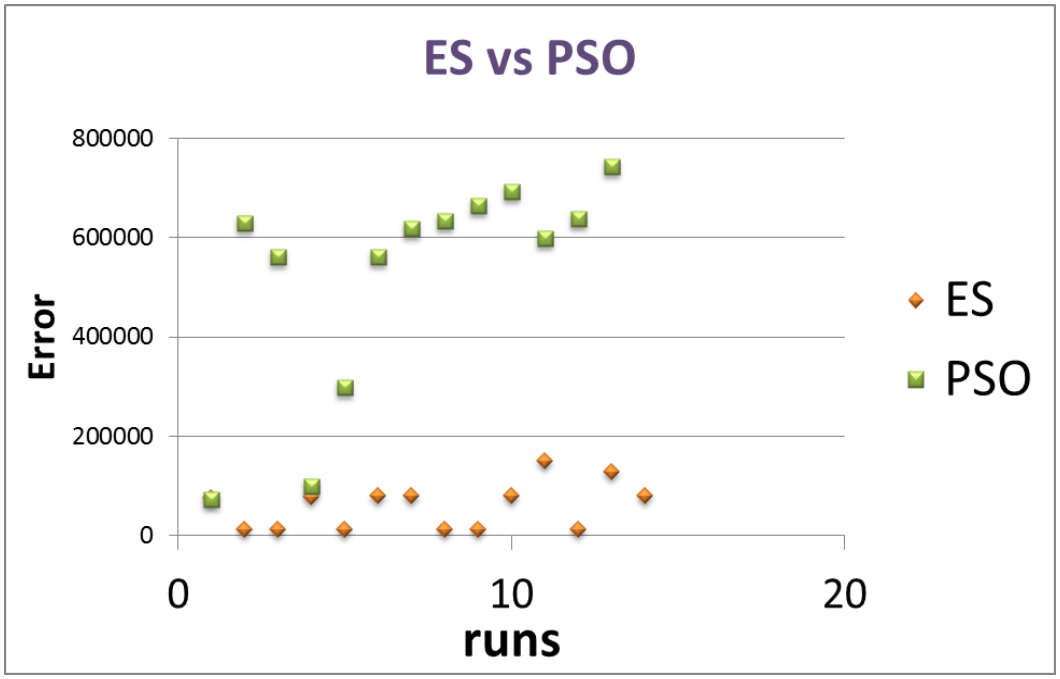

Figure 26: Fitting Errors of Evolution Strategy and Particle Swarm Optimazation

4.3 Island model and single population comparison

Of the 24 runs of ES with a single population performed, only 29% are successful in finding compatible solutions. Assuming this 29% to be the general success rate, running 4 independent islands would have only 74% probability of success, which is inferior to running 4 islands having 100% success rate with the reseeding approach. We have migrated 5 individuals every 150 generations. The migrations require the process-to-process communication because each island runs on a different CPU core. This is a

huge bottle neck in computation speed when migration happens frequently. Limiting migration to once every 150 generations helped reduce the overall migration cost. Once every 150 generations was frequent enough to stimulate the evolutionary process.

4.4 Experiments on Evolution strategy

As described in section 2.3.3, we have experimented with the (μ,λ) and (μ+λ) survivor selection strategies. The results are shown in Figure 27. Intuitively, the (μ,λ) strategy has a better chance to forget and leave the local optima because it does not include the parents in the survivor competition. In Figure 27 there are 3 runs where (μ+λ) errors are above 1,000,000. In those 3 runs, it seems that the (μ+λ) approach had the entire population fall into local optima and lost all diversity. Therefore, our choice of the (μ,λ) strategy is justified.

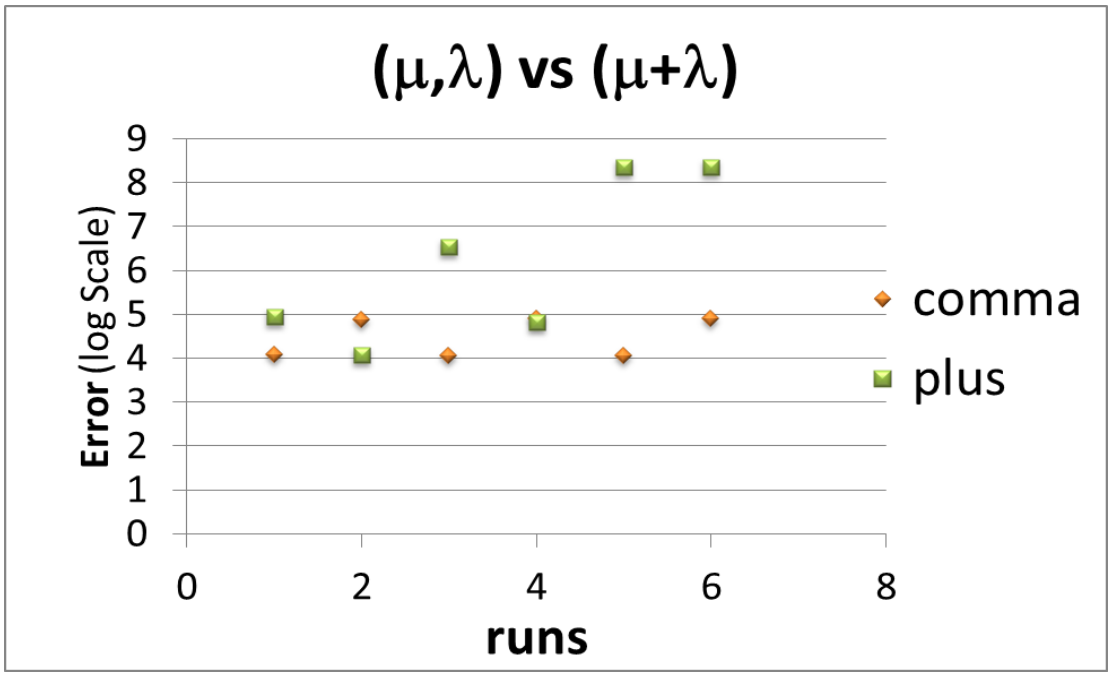

Figure 27: Performance of (μ,λ) and (μ+λ)

We described the recombination choices on section 2.3.1. Each object variable can chose either two parents (Local) to select the value for all gene positions or select any random parents (Global) for each and every gene position to inherit from them. As shown in Figure 28, Global recombination resulted in very poor performance. In 8 out of 12 runs it did not even converge to a fair solution. Therefore, our choice of local recombination is justified.

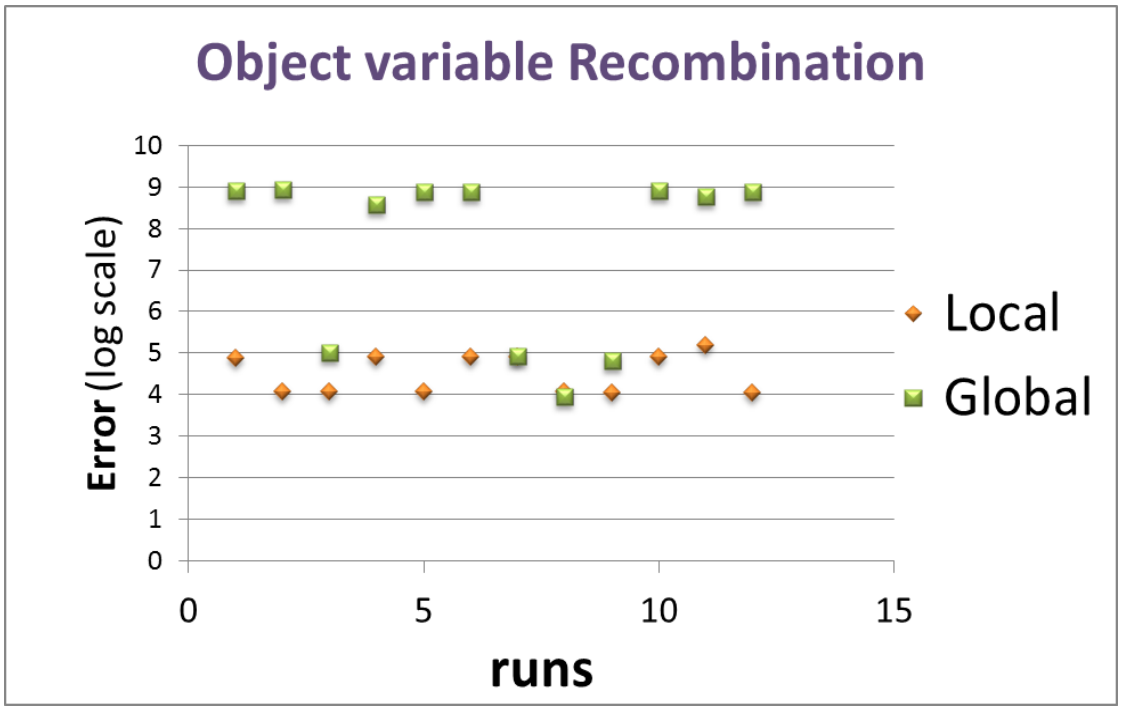

Figure 28:Object variable recombination Strategy

With four islands, we can either set all islands to uniform parameters or set them to diverse parameters. One diverse approach setting the role of Progressive and Conservative islands with different recombination rates is described in section 2.5. In this experiment we compare the uniform island model and the diverse island (Progressive and

Conservative island) model. As shown in Figure 29, both approaches had similar performance with just one exceptional fail on the 5th run for the uniform model. However, the evolutionary process and the diversity state of the islands on each model were quite different.

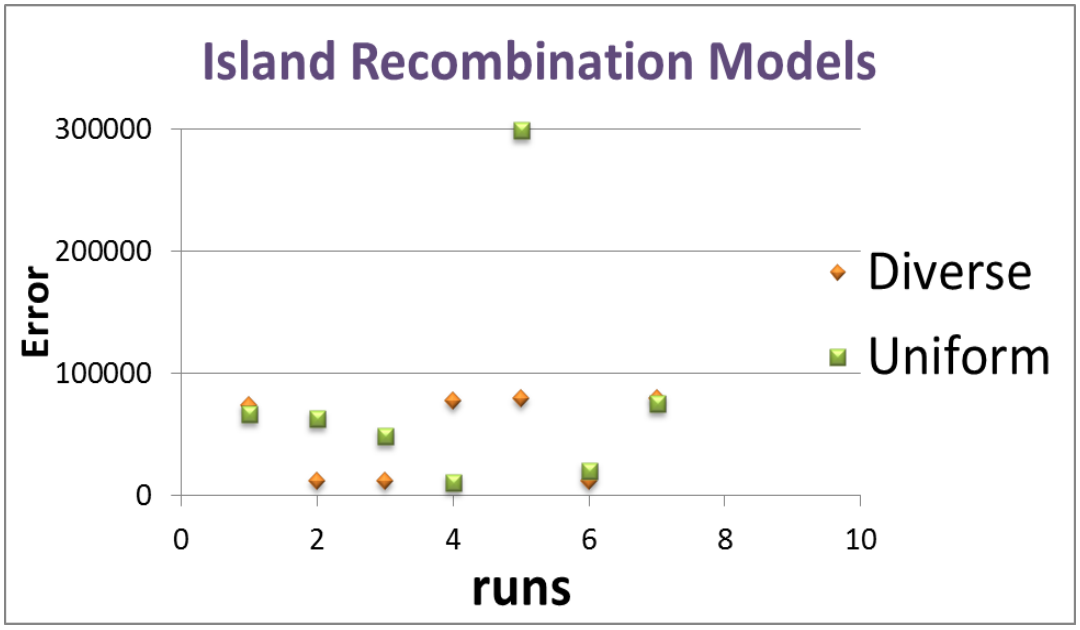

Figure 29: Comparesion of uniform and diverse island models

Figure 30 shows the evolutionary process of a set of four islands from one experimental run with uniform settings. Although there are some varying delays on convergence, the islands generally follow similar water fall curves down to the convergence point. On the other hand, Figure 31 shows the evolutionary process of a set of progressive and conservative islands in an experimental run. In contrast to the uniform model, we can observe that the progressive island experiences a much slower fall, maintaining a more diverse population on the island until it converges to the best solution.

There is also a noticeable bouncing pattern on the curve climbing uphill trying to escape from a local optimum and maintain diversity.

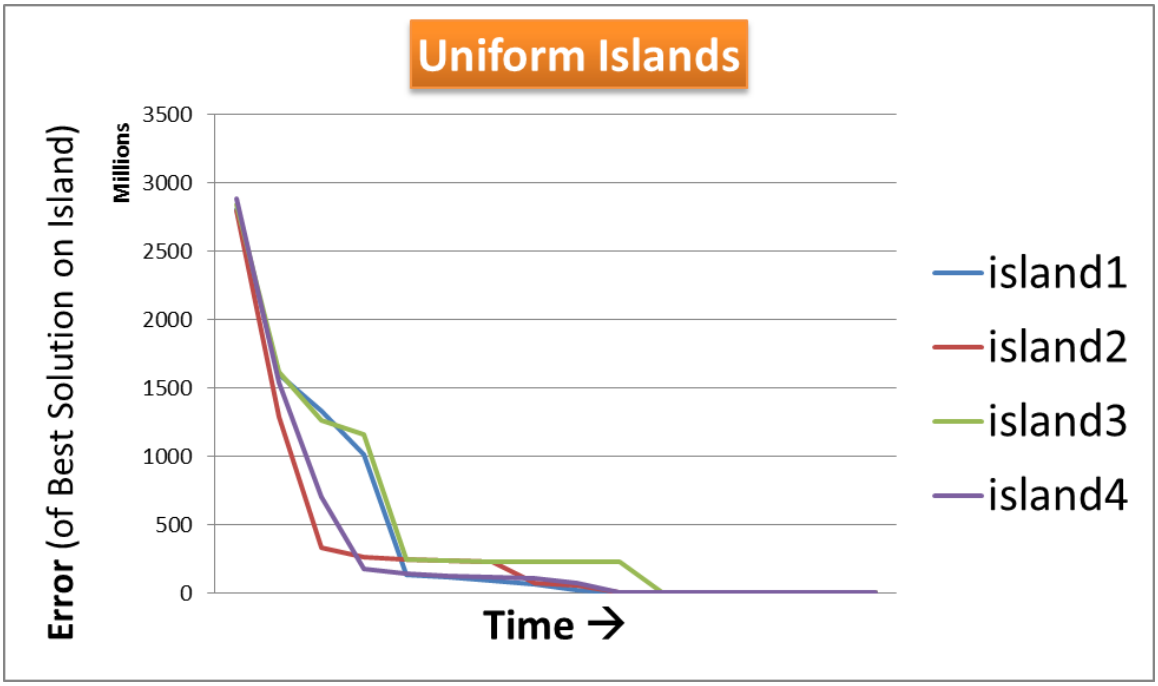

Figure 30: Evolution process of uniform islands

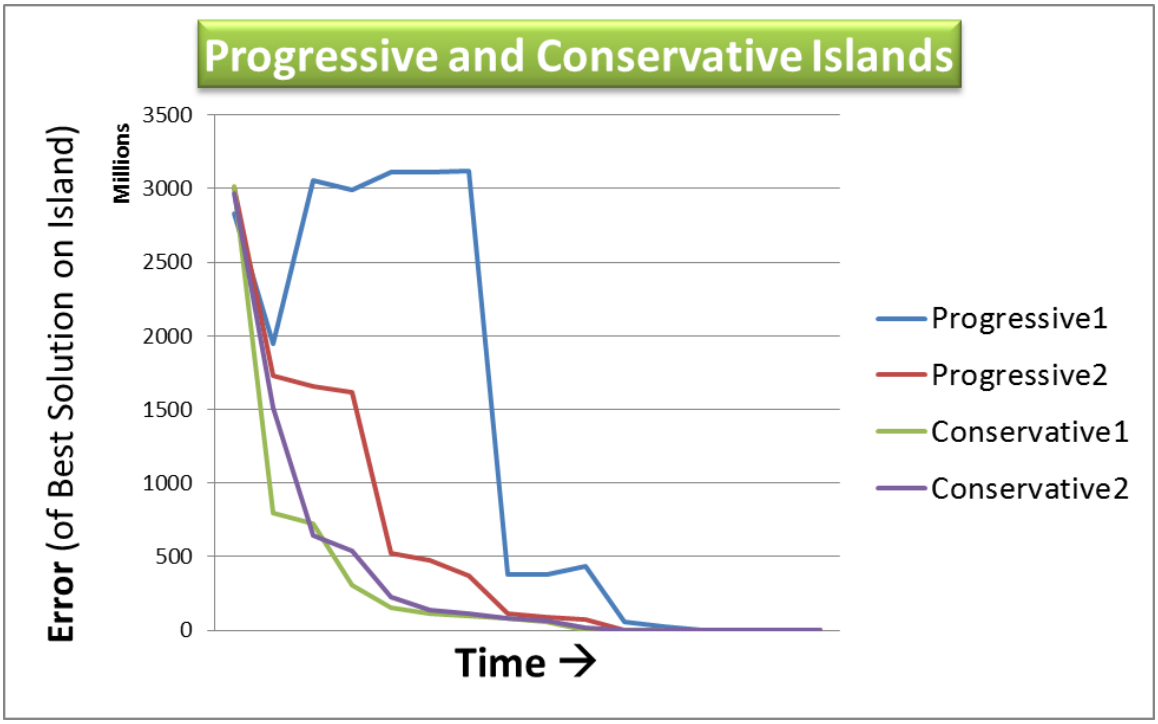

Figure 31: Evolution process of progressive and conservative islands

We measure the diversity of the population quantitatively by calculating the standard deviation of the error scores of the population. As shown in Figure 32, islands 1 and 3 maintain diversity for a longer period than islands 2 and 4. However, we can observe a longer-lasting and greater diversity maintained on progressive islands 1 and 2 in Figure 33. With 70% probability of recombination on the progressive island 1, it does not quickly converge to current optimal solutions from other islands. Instead, it maintains the diversity of the population while it explores uncovered search areas with the help of migrated promising solutions from other islands. Although both island models showed similar performance, having the Progressive Island provides more diversity in the population which is often preferred in evolutionary computation.

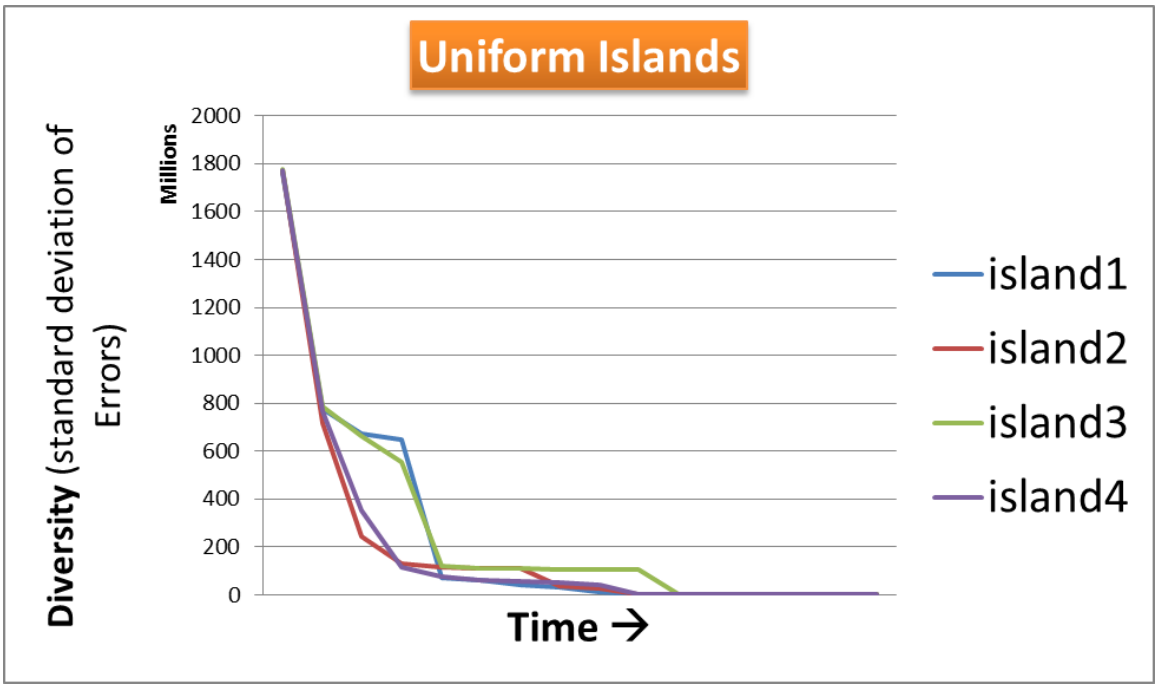

Figure 32: Diversity of population on each islands through evolutinary process

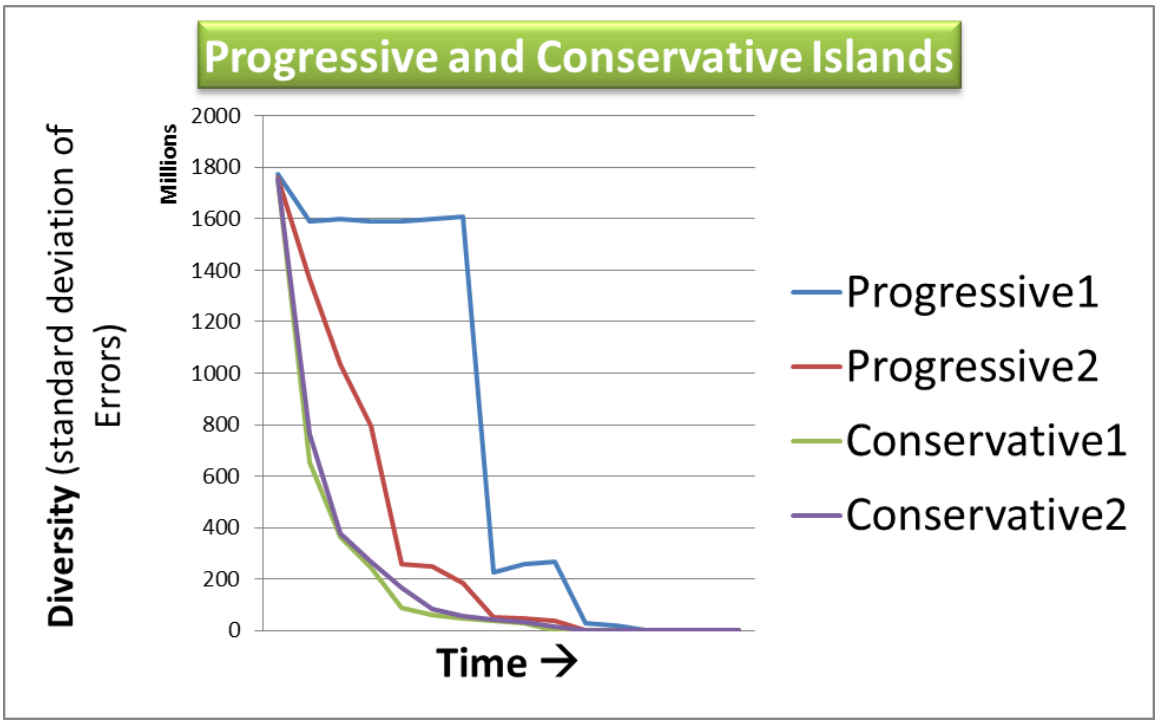

Figure 33: Diversity of population on progressive and conservative islands through evolutinary process

4.5 Speedup from utilizing the GPU architecture

Solving a massive population of ODE systems with the Runge-Kutta method is the core computational portion of our evolutionary algorithm. This fitness evaluation step has to run on each generation for thousands of generations to reach convergence. The speedup from using our GPU implementation compared to an optimized single threaded C code implementation (compiled with gcc having –O2 option) of the Runge-Kutta method is shown in Figure 34. We gained a speedup up to 81 times on evaluating 7,680 individuals as shown in the figure.

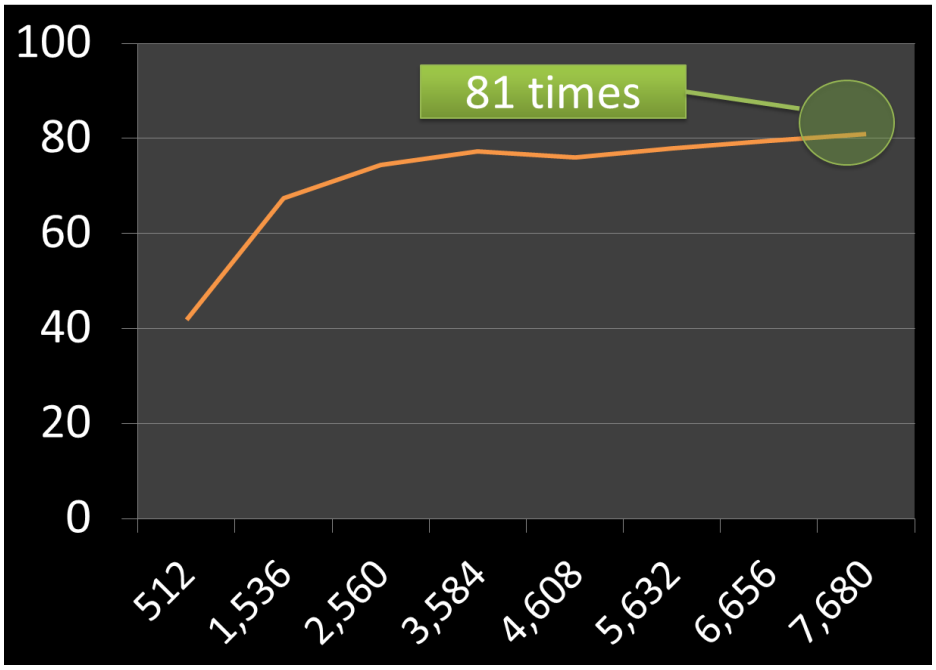

Figure 34: x81 Speed up evaluating 7,680 ODE systems

The island model utilizing 2 GPUs + 4 CPU cores gained x155 speedup compared to using a single threaded CPU implementation. The entire work is implemented on a PC workstation with a GTX 480 GPU and an Intel(R) Core(TM) i5-2400@3.6 GHz CPU.

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

We have introduced a new fitness function which captures the shapes and magnitudes of the signals generated from a biological clock system. We have shown that a correlation measure with de-trending is effective for capturing the quantitative similarity of high level frequency features of the series. We have also demonstrated that the conventional approach based on chi square error [9, 19] is not sufficient to accurately measure the full similarity of the outputs of the ODE system especially for biological clock model. A massive population-based searching Evolution Strategy was applied to the circuit reconstruction problem, and the proposed fitness function shows a consistent ability to find very good solutions. We have utilized a GPU implementation of massively parallel evolutionary algorithms for the biological circuit reconstruction. Our tests show that the GPU implementation is efficient and suitable for application in the investigation of the biological clock circuit system. The GPU serves a purpose, through the mathematical formalization of complex biological networks, of the understanding of the emergent and dynamic control of the biological system. The GPU based parallel implementation of the Evolution Strategy resulted in up to a 155 times speedup in our experiments. The speedup gave us the opportunity to run multiple experiments with different settings, helping us to gain insight into the problem and the selected algorithm.

The GPU-powered Evolution Strategy clearly demonstrated that it is a powerful tool in building a genetic network model of the biological clock of Neurospora crassa.

5.2 Future work

Working with real protein profiling data, there is another unknown variable we need to fit called the Scaling Factor. Often the data is generated from multiple different environments. The Scaling Factor is a scalar value used to modulate the different magnitude scale of the experimental data which varies by each environment. Our next step would be encoding this scaling factor into an additional gene so that we can run our circuit reconstruction targeting protein profiling data.

Furthermore, the typical protein profiling data measure is far sparser than the sampling rate we used on this work. Some robust interpolation method or down sampling should be implemented to fill the gaps introduced by this factor.

REFERENCES

[1] D'haeseleer, P., Liang, S., & Somogyi, R. (2000). Genetic network inference: from co-expression clustering to reverse engineering. Bioinformatics, 16(8), 707-726.

[2] Chen, T., He, H. L., & Church, G. M. (1999, January). Modeling gene expression with differential equations. In Pacific symposium on biocomputing (Vol. 4, No. 29, p. 4).

[3] Liang, S., Fuhrman, S., & Somogyi, R. (1998, January). REVEAL, a general reverse engineering algorithm for inference of genetic network architectures. In Pacific symposium on biocomputing (Vol. 3, No. 18-29, p. 2).

[4] Goss, P. J., & Peccoud, J. (1998). Quantitative modeling of stochastic systems in molecular biology by using stochastic Petri nets. Proceedings of the National Academy of Sciences, 95(12), 6750-6755.

[5] Friedman, N., Linial, M., Nachman, I., & Pe'er, D. (2000). Using Bayesian networks to analyze expression data. Journal of computational biology, 7(3-4), 601-620.

[6] Toh, H., & Horimoto, K. (2002). Inference of a genetic network by a combined approach of cluster analysis and graphical Gaussian modeling. Bioinformatics, 18(2), 287-297.

[7] Elowitz, M. B., Levine, A. J., Siggia, E. D., & Swain, P. S. (2002). Stochastic gene expression in a single cell. Science, 297(5584), 1183-1186.

[8] Blossey, R., Cardelli, L., & Phillips, A. (2006). A compositional approach to the stochastic dynamics of gene networks. In Transactions on Computational Systems Biology IV (pp. 99-122). Springer Berlin Heidelberg.

[9] Yu, Y., Dong, W., Altimus, C., Tang, X., Griffith, J., Morello, M., ... & Schüttler, H. B. (2007). A genetic network for the clock of Neurospora crassa. Proceedings of the National Academy of Sciences, 104(8), 2809-2814.

[10] Vollmer, S. J., & Yanofsky, C. (1986). Efficient cloning of genes of Neurospora crassa. Proceedings of the National Academy of Sciences, 83(13), 4869-4873.

[11] Johnson, C. H., & Hastings, J. W. (1986). The Elusive Mechanism of the Circadian Clock: The quest for the chemical basis of the biological clock is beginning to yield tantalizing clues. American Scientist, 74(1), 29-37.

[12] Karr, C. L., Weck, B., Massart, D. L., & Vankeerberghen, P. (1995). Least median squares curve fitting using a genetic algorithm. Engineering Applications of Artificial Intelligence, 8(2), 177-189.

[13] Simonsen, M., Pedersen, C. N., Christensen, M. H., & Thomsen, R. (2011, July). GPU-accelerated high-accuracy molecular docking using guided differential evolution: real world applications. In Proceedings of the 13th annual conference on Genetic and evolutionary computation (pp. 1803-1810). ACM.

[14] Longo, G., & Ventre, G. Genetic Algorithm Modeling with GPU Parallel Computing Technology.

[15] Cano, Alberto, Amelia Zafra, and Sebastián Ventura. "Speeding up the evaluation phase of GP classification algorithms on GPUs." Soft Computing 16.2 (2012): 187-202.

[16] Franco, María A., Natalio Krasnogor, and Jaume Bacardit. "Speeding up the evaluation of evolutionary learning systems using GPGPUs." Proceedings of the 12th annual conference on Genetic and evolutionary computation. ACM, 2010.

[17] Jaros, J. (2012, June). Multi-GPU island-based genetic algorithm for solving the knapsack problem. In Evolutionary Computation (CEC), 2012 IEEE Congress on (pp. 1-8). IEEE.

[18] Cárdenas-Montes, M., Vega-Rodríguez, M. A., Rodríguez-Vázquez, J. J., & Gómez-Iglesias, A. (2012). GPU-Based evaluation to accelerate particle swarm algorithm. In Computer Aided Systems Theory–EUROCAST 2011 (pp. 272-279). Springer Berlin Heidelberg.

[19] Ramírez-Chavez, L. E., Coello Coello, C. A., & Rodríguez-Tello, E. (2011, October). A GPU-based implementation of differential evolution for solving the gene regulatory network model inference problem. In Proc. of the 4th

International Workshop on Parallel Architectures and Bioinspired Algorithms (WPABA'2011) (pp. 10-14).

[20] NVIDIA Inc. (2013) NVIDIA CUDA Programming Guide v5.5. Resource document. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf. Accessed November, 2013.

[21] NVIDIA Inc. (2013) GPU energy efficiency. Resource document. http://www.nvidia.com/object/gcr-energy-efficiency.html. Accessed November, 2013.

[22] Eiben, A. E., & Smith, J. E. (2003). Introduction to Evolutionary Computing. Springer Berlin Heidelberg.

[23] Kennedy, J. (2010). Particle swarm optimization. In Encyclopedia of Machine Learning (pp. 760-766). Springer US.