Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Parallelized Stochastic Reaction-Diffusion Simulations on GPGPU and multi-core CPU

Bachelor's Thesis

Fabian Güra
Munich School of Engineering
Technische Universität München

somewhen, 2014

Advisors ETH: Prof. Dr. Petros Koumoutsakos
Jana Lipková, M.Sc.
Advisor TUM: Prof. Dr. Hans-Joachim Bungartz

Abstract

In this thesis the deterministic as well as the stochastic approach towards simulating chemical reaction-diffusion systems are presented. Apart from the underlying formalistic concepts two different implementations of the tau-leaping stochastic simulation algorithm are discussed. In addition to theoretical deliberation and accuracy validation, the performance of the algorithm in two parallel computing environments, namely Nvidia CUDA GPGPUs and OpenMP-enabled multicore CPUs, is assessed. Throughout the complete thesis the Gray-Scott model is used as an example to illustrate various results. This reaction-diffusion system is not only subject to pattern-formation, but it is also of great interest due to is computational complexity and practical relevance.

Contents

Chapter 1

Introduction

"I had no idea this was going to be an accurate prediction, but amazingly enough instead of 10 [years] doubling, we got nine over the 10 years, but still followed pretty well along the curve."

– Gordon E. Moore[1]

In 1965 Godon E. Moore, one of the founders of Intel, published his observation that the number of transistors on a chip of same size doubles approximately every 24 months and predicted similar exponential growth for the future [Moo65]. The so-called "Moore's Law" is illustrated in figure 1.2. A higher level of integration usually goes hand in hand with increasing performance. The availability of greater computational capabilities does not only accelerate existing solutions, but may enable applications from a variety of research fields that were previously impossible to realize or had been completely out of scope a few years ago.

One of these fields is biochemistry. Over the years, computational methods have become a valuable and integrative tool that is by no means able to completely replace traditional laboratory experiments, but can help to gain deeper insight into complex reaction mechanisms that can hardly be understood from experiments alone, but dominate the comportment of the system of interest. Nowadays, apart from reproduction and model validation, it is possible to derive predictions for the behaviour of a system subject to parameters that have not yet been studied in experiments. The approach does not only help to cut costs, but can greatly increase flexibility and pace of research in this field.

In the emerging field of computational biochemistry two fundamentally different approaches are used to derive the dynamic evolution of a system over time from a mathematical model: deterministic and stochastic simulation. The former, often considered the 'classical' way, is based on a macroscopic

[1]Computer History Museum Presents The 40th Anniversary of Moore's Law with Gordon Moore and Carver Mead (http://goo.gl/4WT2ux)

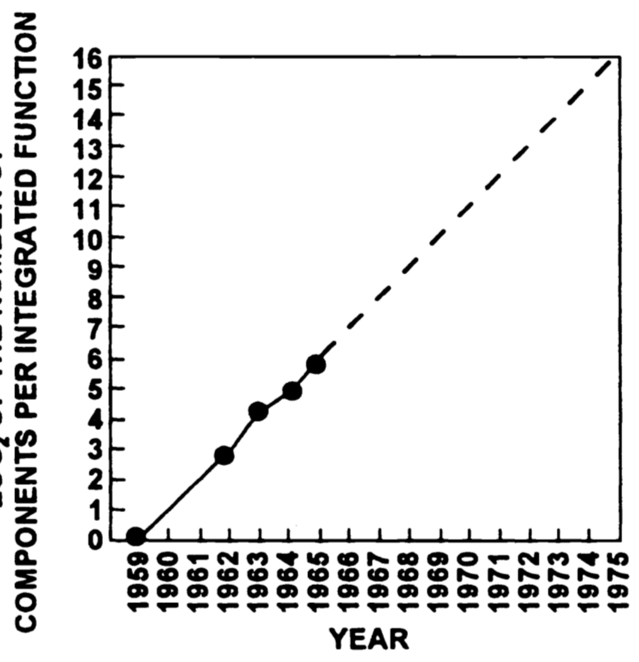

Figure 1.1: Original illustration of "Moor's Law". Figure taken from [Moo65].

description of the reactive species. The system is represented by a set of differential equations which can be solved numerically to gain insight into its dynamic behaviour. The latter, a more recently proposed and popularized approach, is based on the intrinsic randomness chemical species are subject to on a microscopic level. In the context of probability theory and statistics, the next reaction to fire as well as the according point in time can be modeled by means of correctly-distributed random variables.

Over the course of centuries numerous fundamental theoretical results and efficient numerical methods for solving differential equations have been presented. Today, in consequence, systems of reasonable size can be solved accurately on affordable computing hardware in a relatively short amount of time. On the other hand, deterministic methods are considerably more expensive from a computational point of view. Furthermore, for system configurations where macroscopic concepts such as concentration are a valid approximation to the microscopic system dynamics, the averaged results of stochastic simulations often converge to the deterministic solution. One may therefore be tempted to question the relevance of the stochastic approach in practice. However, in cases where system dynamics is dominated by the particles of one or more species available only in small quantities, deterministic models cannot be used. In fact stochastic algorithms are needed to correctly simulate the temporal evolution of most biological systems. Stochastic effects arising from small molecular concentrations can only be considered

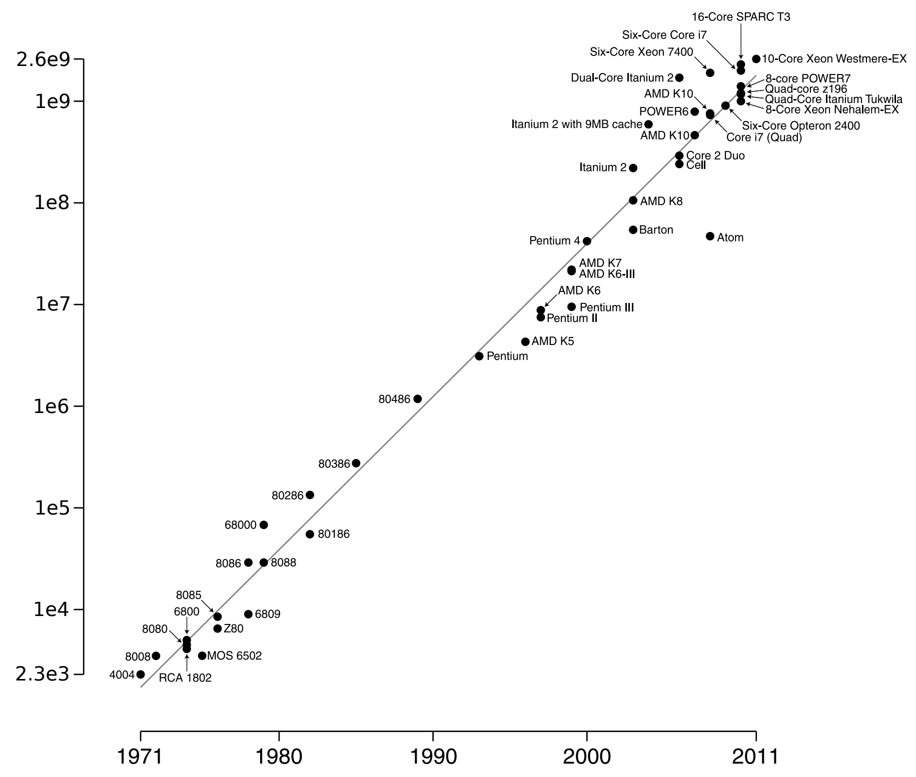

Figure 1.2: Transistor counts of selected microprocessors introduced in the years 1971-2011. The diagonal line represents Moore's law. Figure taken from [Wgs11] (modified).

by a non-deterministic approach. Examples of successful applications include genetics [MA97], cellular signal transduction [EO05] and drug delivery [FPB11].

In practice a great variety of these processes can be modeled in terms of reaction and diffusion. Particles close to one another can collide and react according to prescribed reaction channels. Diffusion, on the other hand, macroscopically describes the spatial movement of molecules in the direction of the negative concentration gradient due to microscopic effects such as Brownian motion. The methods presented in the framework of this thesis are applicable for generic reaction-diffusion systems. To maintain a reasonable scope, however, a special focus is placed on the Gray-Scott model [GS84, Pea93]. This primary example is of great interest due to its ability to exhibit pattern formation caused by an embedded activator-inhibitor process. Pattern formation as a form of self-organization "can be considered the complement of the second law of thermodynamics[2] explaining the emergence of order from disorder instead of disorder from order" [MC13]. In practice this mechanism was observed in follicle spacing on mice skin [SRTS06]. It can be used to describe pigmentation pattering in mammals [Mur02]. Reaction-diffusion models have recently been used to simulate signalling pathways in eukaryotic cells [TTNW10].

For his original prediction Moore considered a time span of ten years into the future (see figure 1.1). As of today (July 2014), Moor's Law is still considered to be valid [Tec14]. However, it is a common misunderstanding that computing power of microprocessors doubles at the same rate. Not the level of integration is the main driver of performance, but the processor's clock speed is. The latter has an enormous influence on the system's energy efficiency and since it is bound by the amount of heat a CPU cooler can dissipate, (starting in around 2005/06) a novel approach was introduced to the mainstream market: multi-core CPUs. This paradigm shift requires developers to explicitly write code that can make use of multiple parallel execution units. Another more recent trend in scientific computation is General-purpose Computing on Graphics Processing Units (GPGPU). Considering raw computational power (i.e. the number of floating point operations per second, FLOP/s) GPUs easily outperform CPUs.

Due to the great computational complexity of stochastic simulation algorithms parallel platforms must be considered in order to simulate systems whose enormous complexity was previously considered prohibitive. In this thesis a simulator is presented that can make use of both, multi-core CPU and GPU, to accelerate stochastic reaction-diffusion simulations. The implementation depends on the OpenMP standard and Nvidia CUDA, respectively. Even though optimization techniques were used and the results of performance analysis is given, this work will not answer the question of which platform is suited better for stochastic simulations. The presented application is far from being perfect on both platforms. The goal of the project is merely to give an overview about available techniques and demonstrate results that can be achieved in a time-limited academic project.

The structure of this thesis is as follows: In chapter 1, a compact overview about the topics and the scope of this thesis was given. In chapter 2, theoretical aspects of both approaches are derived and important results are cited. It is therefore the foundation for the two following chapters: In chapter 3, the implementation details and ideas incorporated into the developed programmes as well as the platforms they are targeted at are presented. In the subsequent chapter the tools are validated by comparison to analytic solutions of model examples. In addition, the more complex Gray-Scott model is simulated to assess performance on the different platforms and

[2]"The entropy of a closed system never decreases"

to demonstrate the pattern formation mechanism. The final chapter briefly summarizes the results of the thesis and gives an outlook on possibilities, trends and ideas to further improve the presented methods.

Chapter 2

Approaches to Simulating Chemical Reaction-Duffusion Systems

"Theory without practice cannont survive and dies as quickly as it lives. He who loves practice without theory is like the sailor who boards ship without a rudder and compass and never knows where he may cast."

– Leonardo da Vinci [BL06]

In this chapter, the two fundamental approaches towards modeling chemical reaction-diffusion systems are presented. On the one hand, deterministic models based on rate laws and differential equations are the traditional way of describing the dynamic evolution of a system over time. Stochastic models, on the other hand, are based on the intrinsically random behavior of such systems. Unlike in the deterministic case, running a stochastic simulation algorithm twice gives two possible, but usually completely different solutions, each representing one possible scenario of how the system may develop over time in reality. Often averaging the states obtained by a great number of stochastic simulations yields the solution of the deterministic simulation, however, this is not always the case. For some models, especially those representing biochemical systems, stochastic simulations can give more insight. Whenever the presence of one or more species can heavily affect the dynamic evolution of the overall system (i.e. by enabling or disabling a specific reaction channel), but the absolute number of molecules is low, the intrinsic randomness in particle movement has to be considered. Stochastic effects can only be modeled by a stochastic approach, deterministic algorithms are not applicable in this case.

2.1 Illustrative Model: The Gray-Scott System

Throughout this thesis, the Gray-Scott reaction system described in [Pea93] will be used as a primary example to illustrate the presented techniques, algorithms and results.

The chemical reaction system consists of two species U and V subject to the reactions as described in the following:

$$\emptyset \xrightarrow{F} \mathrm{U} \tag{2.1}$$

$$\mathrm{U} \xrightarrow{F} \emptyset \tag{2.2}$$

$$\mathrm{V} \xrightarrow{F+\kappa} \emptyset \tag{2.3}$$

$$\mathrm{U} + 2\,\mathrm{V} \xrightarrow{\rho} 3\,\mathrm{V} \tag{2.4}$$

with positive reaction rate constants F, κ and ρ.

The two species U and V are also subject to diffusion. Approaches towards modelling this spatial dependency of the system will be discussed in chapters (ref) and (ref) for the deterministic and the stochastic case.

The reactions given above can be interpreted as follows: Reactions (2.1) and (2.2) describe degradation and production of U particles, respectively. Since they are the opposite to each other, the process of production and degradation of U is reversible. Equation (2.3) describes degradation of V particles. Note that there is no direct production of V in the system. Finally, reaction (2.4) describes the autocatalytic[1] conversion of U particles to V particles. The symbol $\emptyset$ represents species that are available in excess (i.e. not rate-limiting), but are of no further interest for the modeled process.

The Gray-Scott model is of great interest since it

- can be used to describe real-world phenomena.
- is subject to pattern formation (for certain parametrizations).
- is computationally expensive due to spatial dependencies.

All these properties will be examined more closely throughout the thesis.

2.2 Deterministic Simulation

The traditional way of simulating chemical models is to solve a system of differential equations. The solution obtained is deterministic in the sense that it is, for given parameters, unique and accurately reproducible (up to possible numerical errors).

[1]A reaction is said to be autocatalytic if it is catalyzed by its products [AP09, pg. 907]. In this example, V particles can only be generated if there are already some available in the system to catalyze the reaction.

2.2.1 Reaction Rate Equations

Let there be a system with species X_i, $i = 1, \dots, N$ and reactions R_j, $j = 1, \dots, M$. The state of the system $\vec{x}(t)$ at time t is determined by the concentrations[2] of its species X_i. Let $x_i = [X_i]$ denote the concentration of species X_i. For the moment, the system is considered to be well-stirred, i.e. the system is spatially homogeneous with respect to the concentrations of all species. A spatial effect in inhomogeneous systems, namely diffusion, will be considered in chapter 2.2.3.

By applying the *Law of mass action*, one can find N differential equations that describe the rate of change of concentration over time for every species X_i:

$$\frac{dx_i}{dt} = \sum_{j=1}^{M} v_{ij} f_j(\vec{x}) \tag{2.5}$$

v_{ij} denotes the number of particles of species X_i that are created ($v_{ij} > 0$) or consumed ($v_{ij} < 0$) in reaction R_j. f_j is the reaction rate of R_j which in general depends on the current state of the system.

For simple ("elementary") reactions, the reaction rate is proportional to the product of the concentrations of the reactant species raised to the power of the respective stoichiometric coefficients. Considering for example the reaction $2\,\mathrm{A} \longrightarrow \mathrm{B}$, one has $\frac{da}{dt} = -2ka^2$ and $\frac{db}{dt} = ka^2$.

Applying this formalism to the reactions (2.1) - (2.4) from the Gray-Scott model, the system can be represented by the following differential equations:

$$\frac{du}{dt} = -\rho u v^2 + F(1-u) \tag{2.6}$$

$$\frac{dv}{dt} = \rho u v^2 - (F+\kappa)v \tag{2.7}$$

In order to uniquely determine the solution of the system, the mathematical model is completed by introducing initial conditions:

$$\vec{x}(t=0) = \vec{x}_0 = \begin{bmatrix} u_0 & v_0 \end{bmatrix}^T \tag{2.8}$$

2.2.2 Steady State Analysis

Steady state analysis is a practical way to gain an initial insight into how a chemical system may develop in time. In order to better understand the primary example of this thesis, the Gray-Scott model shall now be analyzed with this technique.

[2]Unlike in chemical literature where concentration is usually defined as "number of moles/volume", in this thesis a more general concept of concentration scaling is introduced (see chapter 2.3.5).

A steady state $\vec{x}_s$ of a system of ordinary differential equations (ODEs) of the form $\frac{d\vec{x}}{dt} = f(\vec{x}, t)$ has to satisfy $f(\vec{x}_s, t) = \vec{0}\ \forall t \geq 0$. Considering the equations (2.6) and (2.7), it is obvious that $\vec{x}_{s1} = \begin{bmatrix} 1.0 & 0.0 \end{bmatrix}^T$ is a trivial steady state. When all molecules of species V are gone (or there have never been any), the autocatalyzed reaction (2.4) cannot fire any more and the concentration v will remain at zero. The system is therefore reduced to the single ODE $\frac{du}{dt} = F(1-u)$, where the right hand side is 0 iff[3] $u = 1$.
Throughout the simulations which are carried out in chapter 4, the parameter values proposed in [RBK08] ($F = 0.04$, $\kappa = 0.06$ and $\rho = 1.0$) will be used. For this setup, another steady state is located at $\vec{x}_{s2} = \begin{bmatrix} 0.5 & 0.2 \end{bmatrix}^T$. Detailed calculations can be found in the appendix (A).

To asses the stability of the steady states, the eigenvalues of the Jacobian evaluated at these points in phase space are needed. In the trivial case on obtains $\lambda_{11} = -0.1$ and $\lambda_{12} = -0.04$. Since the real part of both eigenvalues is less than zero (i.e. $\Re(\lambda_{1i}) < 0, i = 1, 2$), the steady state is stable. The eigenvalues of the non-trivial pole are $\lambda_{21} = 0$ and $\lambda_{22} = 0.02$, in consequence the steady state is unstable.

Considering initial conditions $\vec{x}_0 = \begin{bmatrix} 0.5 & 0.25 \end{bmatrix}^T$ and the results of the calculations above one can predict that the system will move towards the steady state $\vec{x}_{s1}$ over time. This behaviour can be observed in figure 2.4 after the transient response has gone away. Figure 2.1 illustrates the corresponding trajectory in phase space. It can be concluded that diffusion is necessary for complex pattern formation in the Gray-Scott model. An advanced analysis of the involved mechanism (i.e. Turing bifurcation) can be found in [VPHS88].

2.2.3 Diffusion and Fick's Law

The term diffusion describes the phenomenon that matter migrates in the direction of the negative concentration gradient. In consequence, inhomogeneities in the system tend to spread over time and gradients disappear (cite Atkins, pg. 770). Diffusion is an integral part of the Gray-Scott model since it is essential for the pattern formation mechanism. In the following, a deterministic description for diffusion (Fick's law) will be derived to incorporate the phenomenon into the reaction rate equations (2.6) and (2.7) given above.

Let there be a cubic tube filled with a solution of some species A. The tube stretches along the x-Axis. The extent in the two other directions is negligibly small and so are possible concentration inhomogeneities in this plane. On the left end, the concentration $a = [A]$ is high, on the right end it is low.

[3]"if and only if"

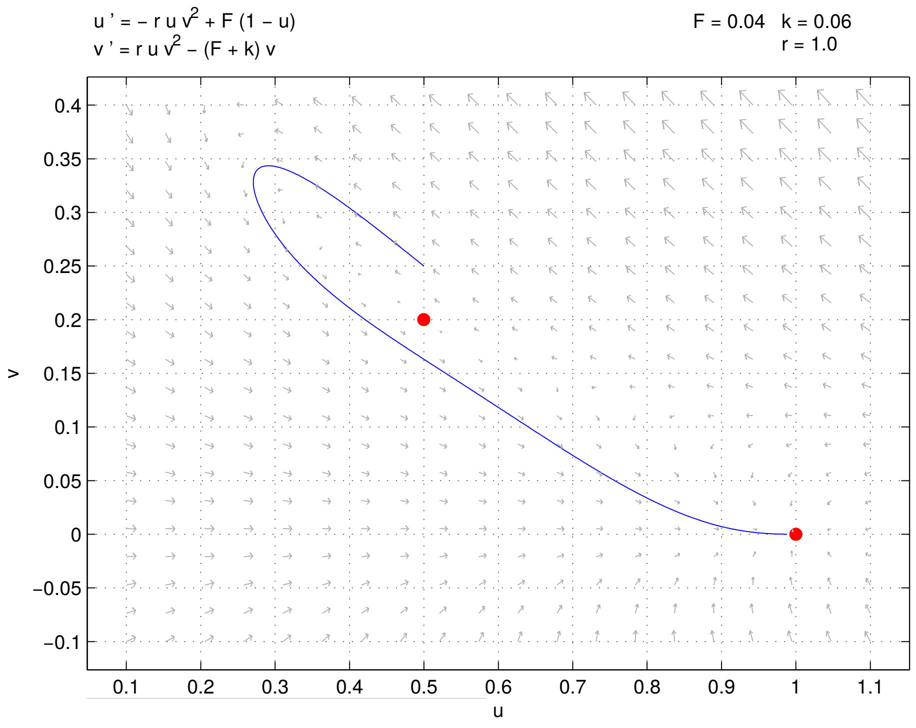

Figure 2.1: Phase space plot of the Gray-Scott local reaction system. Trajectory starts at $\vec{x}_0$. Red dots mark steady states.

Figure (2.2) illustrates the setup. Considering a slab of length l ranging from x_s to $x_s + l$ perpendicular to the x-Axis, the change of concentration inside the slab in some infinitesimal time interval is equal to the influx through the cross-sectional area A at $x = x_s$ minus the flux out of the slab at $x = x_s + l$:

$$\frac{\partial a}{\partial t} = \frac{J(x_s)Adt}{Aldt} - \frac{J(x_s + l)Adt}{Aldt} = \frac{J - J'}{l} \tag{2.9}$$

The flux $J(x)$ at position x is proportional to the negative concentration gradient $\frac{\partial a}{\partial x}$ at x. Using this relation gives

$$\begin{aligned} J - J' &= -D\frac{\partial a}{\partial x} + D\frac{\partial a'}{\partial x} \\ &= -D\frac{\partial a}{\partial x} + D\frac{\partial}{\partial x}\left\{a + \frac{\partial a}{\partial x}l\right\} = Dl\frac{\partial^2 a}{\partial x^2} \end{aligned} \tag{2.10}$$

with positive diffusion constant D.

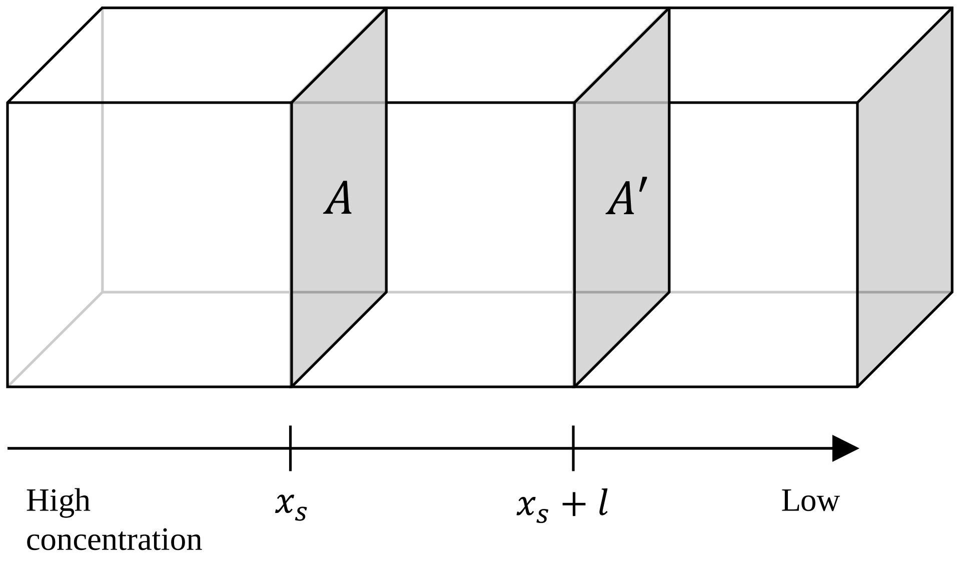

Figure 2.2: Illustration of the setup used to derive Fick's law of diffusion

By substituting this relation back into (2.9) and by passing the limit $l \to 0$, one obtains the one-dimensional version of Fick's law of diffusion:

$$\frac{\partial a}{\partial x} = D\frac{\partial^2 a}{\partial x^2} \tag{2.11}$$

In the generalized 3D case, the spatial derivative term turns into the Laplace operator $\Delta u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}$:

$$\frac{\partial a}{\partial t} = D\Delta u \tag{2.12}$$

The complete Gray-Scott model is then discribed by the following partial differential equations:

$$\frac{\partial u}{\partial t} = D_u \Delta u - \rho u v^2 + F(1-u) \tag{2.13}$$

$$\frac{\partial v}{\partial t} = D_v \Delta v + \rho u v^2 - (F+\kappa)v \tag{2.14}$$

where D_u and D_v are positive diffusion constants of the species U and V, respectively.

2.3 Stochastic Simulation

Another, more recently proposed way of simulating chemical systems are stochastic simulation algorithms (SSA) [Gil76]. Unlike in the deterministic case, the solution obtained is not unique, but it only represents one possible way the system can develop in time. In theory, i.e. on a computer with infinite precision equipped with a perfect source of random numbers, the probability that a SSA will give exactly the same result in two or more independent runs is zero. In practice, however, a computer is a highly deterministic machine with finite precision. Therefore, the number of possible solutions of a SSA is finite. Since most computers don't have true random generators built in, pseudo-random number generators (PRNG) have to be used. PRNGs can create a sequence of numbers Z_i that is random in the sense that it is compliant with a given probability distribution (usually $Z \sim \mathcal{U}(0,1)$) according to a variety of statistical tests (i.e. the TestU01 Big Crush testing battery(cite!)). However, a PRNG is completely defined by its state structure so that for a given state, the sequence of numbers generated thereafter is deterministic. The initial state is usually derived from a seed value. If a SSA is run with a fixed seed value, its output will always be the same. From now on, for the sake of simplicity, the terms "random number generator" and "random number" will be used as synonyms for "pseudo-random number generator" and "pseudo-random number", respectively.

Compared to the deterministic approach towards simulating chemical reaction systems, the stochastic approach is computationally more demanding. The main reason for this is the high expense at which random numbers are generated on a computer. Despite this disadvantage, the demand for good SSAs in research and industry has been on the rise over the past decade. For a large number of particles of every species in the system, it can be shown that the two approaches are identical, for low particle counts, however, this is not the case. Especially in biochemistry and genetics, some systems are described insufficiently by traditional deterministic models. Often particles which dominate the dynamic behavior of the system are only present in small numbers. In such situations, SSAs are an important tool to enable a more detailed understanding of the system. Noise introduced by the randomness of microscopic processes such as Brownian motion cannot be neglected but is an integral factor that determines system behaviour in practice.

2.3.1 Definitions

Let there be a chemical reaction system with species X_i, $i = 1, \ldots, N$ and reactions R_j, $j = 1, \ldots, M$. Now, the state of the system $\vec{x}(t)$ at time t is determined by the number of particles x_i of the species X_i, i.e. $\vec{x}(t) = \begin{bmatrix} x_1(t) & x_2(t) & \ldots & x_N(t) \end{bmatrix}^T$. The initial state of the system at time t_0 is $\vec{x}_0 = \vec{x}(t_0)$. The state change vector $\vec{v}_j$ of reaction R_j is defined such that when the reaction takes place, the state of the system changes from $\vec{x}$ to $\vec{x} + \vec{v}_j$. The i-th component of the state change vector gives the number of particles that are consumed ($\nu_{ij} < 0$) or created ($\nu_{ij} > 0$) when reaction R_j occurs. Again, we will only consider spatially homogeneous (i.e. well-stirred) systems in the beginning.

The fundamental assumption of the deterministic description is that the probability of reaction R_j taking place during the time interval $[t, t + dt)$ is (cite!)

$$\alpha_j(\vec{x}(t))\, dt + \mathcal{O}(dt^2) \tag{2.15}$$

The propensity function α_j of reaction R_j is dependent on the state of the system $\vec{x}(t)$. In its most general form, it is given as

$$\alpha_j(\vec{x}(t)) = \Omega k_j \prod_{i=1}^{N} \binom{x_i}{r_{ij}} \Omega^{-r_{ij}} \tag{2.16}$$

where k_j is the reaction rate of reaction R_j, x_i the number of molecules of species X_i at time t. Reaction R_j consumes r_{ij} molecules of species X_i. The binomial coefficient $\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$ gives the number of distinct subsets of size k from a set of size n. In the context of chemical reactions this is the number of different scenarios that can occur when a reaction requires the collision of k out of n available particles. Ω is the number of molecules per volume unit associated to unit concentration in the deterministic model. Chapter 2.3.5 gives a detailed explanation of why this scaling factor is needed.

Considering the local reactions from the Gray-Scott model, one gets the following propensity functions:

Generation of U (2.1):

$$\alpha_1 = F\Omega \tag{2.17}$$

Degradation of U (2.2)

$$\alpha_2 = Fu \tag{2.18}$$

Degradation of V (2.3)

$$\alpha_3 = (F + \kappa)v \tag{2.19}$$

Autocatalytic Conversion (2.4)

$$\alpha_4 = \rho \frac{uv(v-1)}{2\Omega^2} \tag{2.20}$$

Remark: (Work in progress!) Be careful about the 2! Limit of propensity formula, Ω, equivalence of both models. In order to simulate the same system, the parameter $\rho = 1.0$ in the deterministic model corresponds to $\rho = 2.0$ in the stochastic case.

2.3.2 Chemical Master Equation

The following derivations are based on ideas presented in [Lip11]. Reconsidering equation (2.15), the probability that a reaction R_j with propensity function α_j takes place in a time interval $[t, t+dt)$ is approximately $\alpha_j dt$. Let the time step dt be short enough such that the probability of R_j occurring more than once during the interval is negligible. Then the conditional probability of the system being in state $\vec{x}$ at time $t + dt$ given that it was in state $\vec{x_0}$ at time t_0 is

$$\begin{aligned}\mathcal{P}(\vec{x}, t+dt|\vec{x}_0, t_0) = \mathcal{P}(\vec{x}, t|\vec{x}_0, t_0)[1 - \sum_{j=1}^{M} \alpha_j(\vec{x})dt] \\ + \sum_{j=1}^{M} \mathcal{P}(\vec{x} - \vec{v}_j, t|\vec{x}_0, t_0)\, \alpha_j(\vec{x} - \vec{v}_j)dt\end{aligned} \tag{2.21}$$

The first term of the equation represents the case where the system is in state $\vec{x}$ at time t and no reactions take place in the interval $[t, t+dt)$. The second term takes account for the cases where exactly one of the M reactions occurs. The system must have been in the state $\vec{x} - \vec{v}_j$ at time t to reach the state $\vec{x}$ after the firing of R_j.

By rearranging equation (2.21) a difference quotient can be obtained. Passing the limit $dt \to 0$ yields the Chemical Master Equation (CME):

$$\begin{aligned}\frac{\partial}{\partial t}\mathcal{P}(\vec{x}, t|\vec{x}_0, t_0) = \sum_{j=1}^{M} \mathcal{P}(\vec{x} - \vec{v}_j, t|\vec{x}_0, t_0)\, \alpha_j(\vec{x} - \vec{v}_j) \\ - \sum_{j=1}^{M} \mathcal{P}(\vec{x}, t|\vec{x}_0, t_0)\, \alpha_j(\vec{x})\end{aligned} \tag{2.22}$$

The CME of the homogeneous Gray-Scott model in the state $\vec{x} = \begin{bmatrix} U & V \end{bmatrix}^T$ is

$$\begin{aligned}\frac{\partial \mathcal{P}(U,V)}{\partial t} &= F\Omega \cdot \mathcal{P}(U-1, V) \\ &+ (U+1)F \cdot \mathcal{P}(U+1, V) \\ &+ (V+1)(F+\kappa) \cdot \mathcal{P}(U, V+1) \\ &+ \rho\frac{(U+1)(V-1)(V-2)}{2\Omega^2} \cdot \mathcal{P}(U+1, V-1) \\ &- \left(F\Omega + UF + V(F+\kappa) + \rho\frac{UV(V-1)}{2\Omega^2}\right) \cdot \mathcal{P}(U,V)\end{aligned} \tag{2.23}$$

$\mathcal{P}(U,V)$ is used as an abbreviation for $\mathcal{P}(U, V, t|U_0, V_0, t_0)$.

The solution of the CME of a system with initial conditions $\vec{x}_0 = \vec{x}(t_0)$ gives the conditional probability $\mathcal{P}(\vec{x}, t|\vec{x}_0, t_0)$ that the state of the system is $\vec{x}$ at time t. Since the number of molecules in the system is in general not limited (i.e. in the Gray-Scott model nothing stops reaction (2.1) from creating U particles), the dimension of the system of ODEs is infinite. In practice, an exact solution can only be obtained for some simple systems. Therefore, in the next chapter a computational algorithm of great practical importance will be presented.

2.3.3 Gillespie Algorithm

The Chemical Master Equation presented in the previous chapter gives an exact description of the time evolution of a chemical reaction system. In practice, however, it is impractical (or even impossible) to obtain the solve explicitly. Since it is based on the same basic assumptions, the Gillespie Algorithm is a simulation method that is equivalent to the CME. However, the approach does not try to solve the equation explicitly, but simulates the underlying Markov process the CME describes analytically. The main idea of the algorithm can be summarized by two questions that have to be answered in every simulation step: "What is the next reaction that takes fires?" (i.e. what is the next state of the system) and "When will this happen?" (i.e. at what time will the state transition take place). In the following the Gillespie SSA will be derived based on basic probability theory concepts.

Let there be a chemical reaction system with species X_i, $i = 1, \dots, N$ and reactions R_j, $j = 1, \dots, M$. The state of the system $\vec{x}(t)$ at time t is determined by the number of particles x_i of the species X_i. The initial state of the system at time t_0 is $\vec{x}_0 = \vec{x}(t_0)$. The state change vector $\vec{\nu}_j$ of reaction R_j is defined such that when the reaction takes place, the state of the system changes from $\vec{x}$ to $\vec{x} + \vec{\nu}_j$. The i-th component of the state change vector gives the number of particles that are consumed ($\nu_{ij} < 0$) or created ($\nu_{ij} > 0$) when reaction R_j occurs. In the following derivation, for the sake of readability, the dependency of the propensity functions α_j on the state of the system $\vec{x}(t)$ will be omitted.

For the moment, just a single reaction R_j of the system equipped with the propensity function α_j is considered. Let $f_{0,j}(\tau + d\tau)$ be the probability that the reaction does not fire in the time interval $[t, t + \tau + d\tau)$, where $d\tau$ is an infinitesimal time step and $\tau > 0$. This is equivalent to the formulation that R_j does not occur in the interval $[t, t + \tau)$ and in the subsequent instant $(t + \tau + d\tau)$:

$$f_{0,j}(\tau + d\tau) = f_{0,j}(\tau) \cdot (1 - \alpha_j d\tau). \tag{2.24}$$

Rearranging and passing the limit of $d\tau \to 0$ leads to

$$\lim_{d\tau \to 0} \frac{f_{0,j}(\tau + d\tau) - f_{0,j}(\tau)}{d\tau} = \frac{df_{0,j}(\tau)}{d\tau} = -\alpha_j f_{0,j}(\tau) \tag{2.25}$$

Solving the differential equation with the initial condition $f_{0,j}(0) = 1$ yields

$$f_{0,j}(\tau) = \exp(-\alpha_j \tau) \tag{2.26}$$

Considering the complete system, in the case that none of the M reactions of the system fires in the interval $[t, t+\tau)$, one has

$$\begin{aligned} f_0(\tau) &= \mathcal{P}(\tau_1 > \tau \wedge \tau_2 > \tau \wedge \ldots \wedge \tau_M > \tau) \\ &= \mathcal{P}(\min(\tau_j) > \tau) = \prod_{k=1}^{M} \alpha_k \exp(-\alpha_k \tau) \\ &= \exp(-\alpha_0 \tau) \end{aligned} \tag{2.27}$$

where $\alpha_0 = \sum_{k=1}^{M} \alpha_k$.

The probability density $f_1(\tau)$ that any reaction fires at time $t+\tau$ is given by the probability that none fired in the interval $[t, t+\tau)$ and the probability that exactly one fires in the instant $(t+\tau)$:

$$f_1(\tau) = \exp(-\alpha_0 \tau) \cdot \alpha_0 \tag{2.28}$$

The waiting time τ until any reaction R_j, $j = 1, \ldots, M$ fires is therefore exponentially distributed, i.e. $\tau \sim \text{Exp}(\alpha_0)$. The corresponding cumulative distribution function (CDF) is given by

$$F(\tau) = \int_0^{\tau} f_1(t)dt = 1 - \exp(-\alpha_0 \tau) \tag{2.29}$$

Since most random number generators can only generate random numbers $r_1 \sim \text{Unif}(0, 1)$, inverse probability integral transform (cite the paper!!!) can be used to generate $\tau \sim \text{Exp}(\alpha_0)$:

$$\tau = F^{-1}(r_1) = \frac{1}{\alpha_0} \ln\left(\frac{1}{1 - r_1}\right) \tag{2.30}$$

In this case, the (excluded) values $r_1 = 0$ and $r_1 = 1$ are mapped to $\tau = 0$ and $\tau = \infty$, respectively. A computationally favorable but otherwise equivalent mapping is

$$\tau = \frac{1}{\alpha_0} \ln\left(\frac{1}{r_1}\right) \tag{2.31}$$

where $r_1 = 0$ and $r_1 = 1$ are mapped to $\tau = \infty$ and $\tau = 0$. Considering only the real part of the complex logarithm, it can be obtained by moving the original function to the left by 1 (i.e. $r' = r + 1$) and then mirroring it on the ordinate axis. Figure 2.3 illustrates this process.

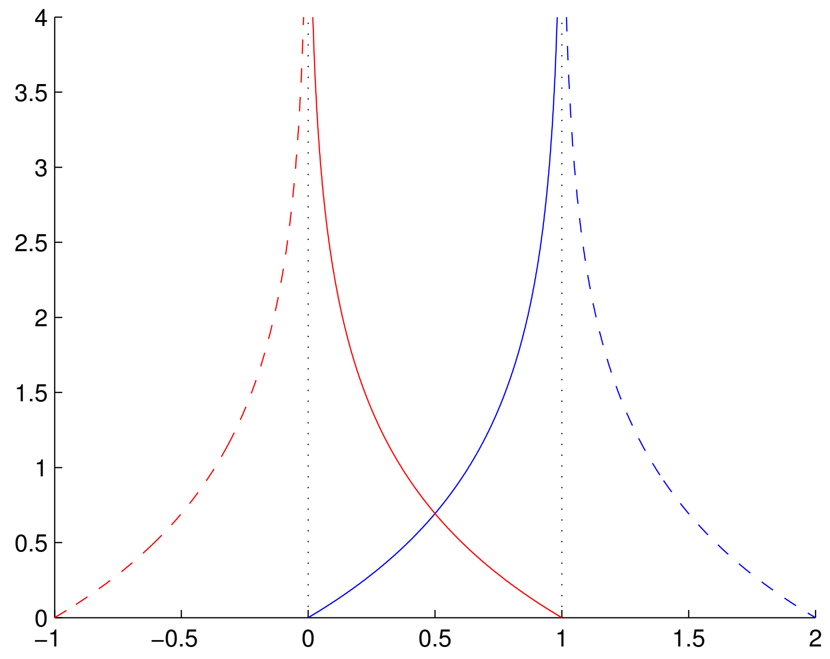

Figure 2.3: Inverse probability integral transform: original mapping drawn in blue, modified function in red. Solid lines are used in the domain of interest, dashed ones illustrate the symmetry of the function (imaginary parts omited). Asymptotes are indicated by dots.

Now that a way to find the time τ when the next reaction takes place has been derived, the next step is to determine which reaction actually fired. This, however, is straightforward:

Let r_2 be a random number uniformly distributed in the interval $(0, 1)$, i.e. $r_2 \sim \text{Unif}(0, 1)$. Then R_j is the reaction which fired in the interval $[t, t + \tau)$ if the index j fulfills

$$\frac{1}{\alpha_0} \sum_{k=1}^{j-1} \alpha_k \leq r_2 < \frac{1}{\alpha_0} \sum_{k=1}^{j} \alpha_k \tag{2.32}$$

Combining the results from above, the Gillespie Algorithm can be outlined as follows:

```
**Input:** Initial state $\vec{x}_0$, propensity functions $\alpha_j$, time $t_{end}$
**Output:** State $\vec{x}(t)$ for $t \in [t, t_{end}]$
**Initialization:** Set time $t = 0$ and state $\vec{x} = \vec{x}_0$.
while $t < t_{end}$ do
    Generate random numbers $r_1, r_2 \sim \mathcal{U}(0,1)$.
    for $j = 1$ to $M$ do
        Compute the propensity function $\alpha_j(\vec{x})$.
    end
    Compute the sum of all propensities $\alpha_0 = \sum_{j=1}^{M} \alpha_j$.
    Compute the time step $\tau = \frac{1}{\alpha_0} \ln\left(\frac{1}{r_1}\right)$.
    Find $j \in [1, M]$ such that $\sum_{k=1}^{j-1} \alpha_k \leq r_2\alpha_0 < \sum_{k=1}^{j} \alpha_k$ holds.
    Set $\vec{x} = \vec{x} + \vec{v}_j$.
    Set $t = t + \tau$.
end
```

Algorithm 1: Gillespie Algorithm

Figure 2.4 shows three independent Gillespie SSA samplings of the temporal evolution of the well-stirred Gray-Scott model. The number of molecules is rescaled to obtain concentrations, the factor $\Omega = 1000$ was used (see chapter 2.3.5). Reaction constants are $F = 0.04$, $\kappa = 0.06$ and $\rho_s = 2.0$. The initial condition is given as $u_0 = 500$ and $v_0 = 250$ which is equivalent to initial concentrations of 0.5 and 0.25, respectively. The dashed line indicates the deterministic solution obtained numerically using a fifth-order Runge-Kutta scheme. In Figure 2.5 a histogram of the number of U molecules at time $t = 10$ is shown. The results from 100000 samples are partitioned in 30 bins.

2.3.4 Tau-leaping Algorithm

The Gillespie Algorithm presented in the previous chapter enables exact numerical simulations of well-stirred chemical reaction systems. It takes account for the inherent randomness in such systems and obeys the same microphysical principles that underlay the Chemical Master Equation. In addition, the algorithm is relatively easy to implement. In practice, however, it turns that even for moderately-sized systems the computational costs of applying the Gillespie algorithm are prohibitive. This is especially true for

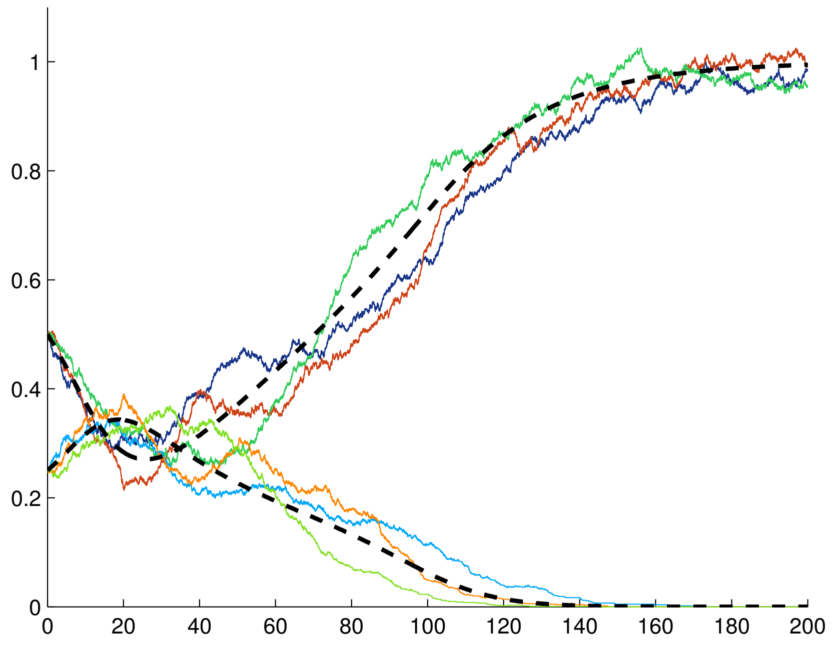

Figure 2.4: Stochastic simulation of the well-stirred Gray-Scott model with $F = 0.04$, $\kappa = 0.06$, $\rho_s = 2.0$, $u_0 = 500$ and $v_0 = 250$. The concentrations ($\Omega = 1000$) of both species over times ($t \in [0, 200]$)is plotted. Color codeing: dark colors represent species U, bright colors represent species V.

real-world applications where not just one single trajectory is needed, but a great number of samples must be obtained to estimate probability densities. For every single reaction the system needs to be updated and (some) propensity values have to be recalculated. Consequently, the temporal evolution is obtained up to a level of detail that is neither useful nor necessary for typical applications. It is obvious that there is a need for accelerated (parallelizable) stochastic simulation algorithms.

In the following, the tau-leaping algorithm originally proposed in [Gil01] is presented. "By making a minor sacrifices in simulation accuracy, major gains in performance can bet obtained." Considering the overall principle of the algorithm, a clear similarity to the explicit Euler method applied to deterministic models can be observed.

Equation (2.26) in the previous chapter gives the probability that a reaction R_j does not fire in an interval $[t, t + \tau)$. The probability density that it fires in the instant $t + \tau$ is

$$f_{1,j}(\tau) = f_{0,j}(\tau) \cdot \alpha_j = \alpha_j \exp(-\alpha_j \tau) \tag{2.33}$$

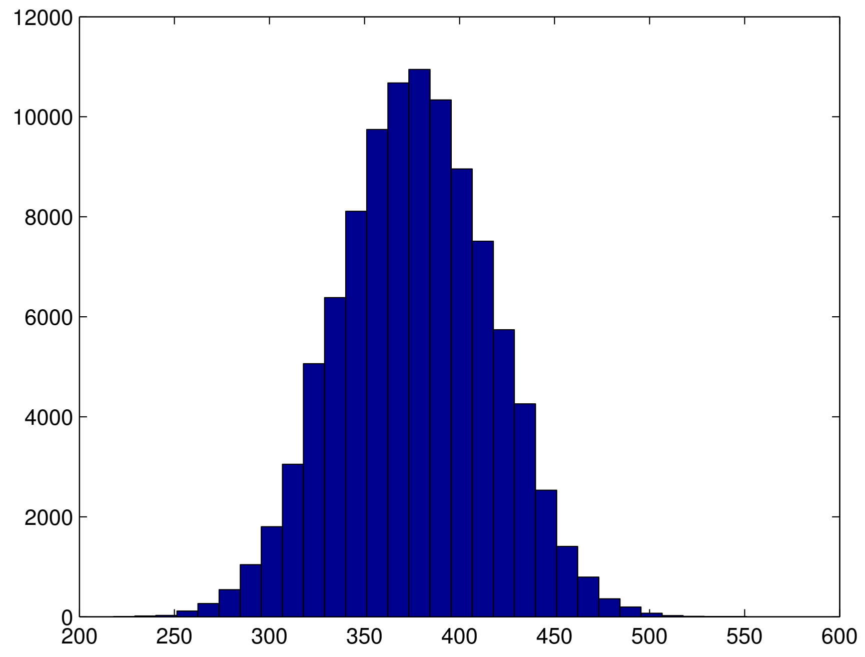

Figure 2.5: Histogram obtained by 100000 Gillespie SSA samples evaluated at $t = 10$. Parameters and initial conditions similar to figure 2.4

The time one has to wait until reaction R_j fires is therefore exponentially distributed, i.e. $\tau \sim \text{Exp}(\alpha_j)$. Without loss of generality, due to the memorylessness of the exponential distribution, i.e. $\mathcal{P}(\tau > t + s | \tau > t) = \mathcal{P}(\tau > s)$ the interval $[0, \tau)$ can be considered. Let k be the number of times the reaction takes place in this interval. Then the probability that R_j fires exactly zeros times is

$$\mathcal{P}(k = 0) = \int_\tau^\infty f_{1,j}(t)dt = \int_\tau^\infty \alpha_j \exp(-\alpha_j t)dt = \exp(-\alpha_j \tau) \tag{2.34}$$

The probability that R_j fires exactly once in the interval is

$$\begin{aligned}\mathcal{P}(k = 1) &= \int_0^\tau f_{1,j}(x) \left(\int_{\tau - x}^\infty f_{1,j}(t)dt \right) dx \\ &= \int_0^\tau \alpha_j \exp(-\alpha_j x) \left(\int_{\tau - x}^\infty \alpha_j \exp(-\alpha_j t)dt \right) dx \\ &= \alpha_j \tau \exp(-\alpha_j \tau)\end{aligned} \tag{2.35}$$

where $x \in [0, \tau)$ is the time when R_j actually fires. The inner integral gives the probability that the reaction does not fire thereafter.

The probability that R_j fires exactly twice in the interval is

$$\begin{aligned}\mathcal{P}(k=2) &= \int_0^\tau f_{1,j}(x)\left(\int_x^\tau f_{j,1}(y)\left(\int_{\tau-x-y}^\infty f_{1,j}(t)dt\right)dy\right)dx \\ &= \frac{(\alpha_j\tau)^2}{2}\exp(-\alpha_j\tau)\end{aligned} \tag{2.36}$$

It can be shown that for arbitrary $l \in \mathbb{N}_0$ the following formula for the probability of counting exactly l firings in the interval is (cite Grimmett, pg. 247, 248)

$$\mathcal{P}(k=l) = \frac{(\alpha_j\tau)^l}{l!}\exp(-\alpha_j\tau) \tag{2.37}$$

The number of times reaction R_j takes place in the interval $[0,\tau)$ is therefore Poisson distributed, i.e. $l \sim \text{Pois}(\alpha_j\tau)$. This result is the main idea used in the tau-leaping algorithm.

Up to now, the implicit assumption that $\alpha \neq \alpha(\vec{x}(t))$ was made. Reconsidering equation (2.16), this is not the case. After each firing of a reaction the state changes and so do (in general) the propensities. For the tau-leaping algorithm, however, the assumption is made that the propensities remain approximately constant during a time interval of length τ. Mathematically this can be formulated as follows:

$$|\alpha_j(\vec{x}(t+\tau)) - \alpha_j(\vec{x}(t))| \leq \varepsilon\alpha_0(\vec{x}(t)) \tag{2.38}$$

where $0 < \varepsilon \ll 1$ is a parameter defined by the user to control model accuracy.

If this so-called Leap Condition (cite Gillespie) is satisfied, the number of firings for each of the M reactions can be approximated by a Poisson random variable with mean $\alpha_j\tau$. The state of the system at time $t+\tau$ is

$$\vec{x}(t+\tau) = \vec{x}(t) + \sum_{j=1}^M k_j\vec{v}_j \tag{2.39}$$

where $k_j \sim \text{Pois}(\alpha_j\tau)$.

The task of finding a suitable τ, however, is nontrivial. A time step that is too long makes the simulation inaccurate, one that is too short increases computing time. Over the years a variety of different procedures have been proposed (cite some). One of the most sophisticated τ-selection formulas is described in (Cao et al. from R-leap). It is given as follows:

$$\tau = \min\left\{\frac{\max\left\{\frac{\varepsilon x_i(t)}{g_i},1\right\}}{|\mu_i(\vec{x}|}, \frac{\max\left\{\frac{\varepsilon x_i(t)}{g_i},1\right\}^2}{|\sigma_i^2(\vec{x}|}\right\} \tag{2.40}$$

where I_{rs} is the set of all reactant species in the system. For these species the parameter g_i is defined as follows:

$$g_i = h_i + \frac{h_i}{n_i} \sum_{j=1}^{n_i - 1} \frac{j}{x_i(t) - j} \tag{2.41}$$

h_i denotes the highest order of reaction in which species X_i appears as a reactant, n_i is the number of X_i molecules that are consumed in any of the highest order reactions (Check with 19 of R-leaping paper). The terms μ_i and σ_i^2 are given by

$$\mu_i(\vec{x}) = \sum_{j=1}^{M} \nu_{ij} \alpha_j(\vec{x}) \tag{2.42}$$

$$\sigma_i^2(\vec{x}) = \sum_{j=1}^{M} \nu_{ij}^2 \alpha_j(\vec{x}) \tag{2.43}$$

Remark: Negative populations Due to the unboundedness of the Poisson distribution, it can happen that in a leaping step more molecules are consumed than there are available in the system. The resulting negative population of the species is unphysical since the system cannot be in such a state in reality. A lot of different measures to avoid this problem have been proposed (e.g. [CGP05, And08]). For this thesis, however, the simplest resolution strategy is chosen: A step which creates negative populations is rejected, the proposed tau is reduced, e.g. $\tau_{new} = \tau_{old}/2$.

The tau-leaping algorithm can be outlined as follows:

```
**Input:** Initial state $\vec{x}_0$, propensity functions $\alpha_j$, time $t_{end}$
**Output:** State $\vec{x}(t)$ for $t \in [t, t_{end}]$
**Initialization:** Set time $t = 0$ and state $\vec{x} = \vec{x}_0$.
while $t < t_{end}$ do
    for $j = 1$ to $M$ do
        Compute the propensity function $\alpha_j(\vec{x})$.
    end
    Compute the time step $\tau$ according to (2.40).
    for $j = 1$ to $M$ do
        Generate a random number $k_j \sim \text{Pois}(\alpha_j \tau)$, i.e. the number of
          time reaction $R_j$ fires.
    end
    if any of the populations would be negative then
        Set $\tau = \tau/2$
        Continue at the beginning of the loop
    end
    Set $\vec{x} = \vec{x} + \sum_{j=1}^{M} k_j \vec{v}_j$.
    Set $t = t + \tau$.
end
```

Algorithm 2: Tau-leaping Algorithm

By leaping over a different time slots instead of considering every individual reaction the computational effort of stochastic simulations can be reduced significantly. The losses in simulation accuracy must be considered, but usually remain within a tolerable range. In the next chapter parallel implementations for multi-core CPU and GPU of the tau-leaping algorithm derived above will be presented.

solve gray-scott and compare to SSA

2.3.5 Parameter Rescaling

The scaling constant Ω introduced in chapter 2.3.1 gives the number of molecules per volume unit associated with unit concentration in the deterministic model. It can also be though of as the fixed subvolume in the complete domain within which the molecules can move. Concentration is then defined as

$$c = \frac{x}{\Omega} \tag{2.44}$$

for x molecules of species X. In chemistry, the product of Avogadro's constant $N_A = 6.022 \cdot 10^{23}\,\mathrm{mol}^{-1}$ and a fixed volume of unit size (e.g. 1l) is often used as a value for Ω. This, however, is mainly a historic convention and in general, Ω can be any arbitrary value $\omega > 0$.

The concept of concentration by definition only makes sense for a macroscopic description of a system. It gives the average number of particles in a fixed control volume. For a system that fulfills the continuum assumption, i.e. one for which a macroscopic description is a valid approximation to the microscopic, particle-based reality, the concentration $c(\vec{x})$ at a point $\vec{x}$ in the system barely depends on the chosen control volume containing $\vec{x}$. For small values of Ω, the continuum assumption is violated. In this case the presented deterministic methods are not applicable. If, on the other hand, the number of molecules in a system is large compared to is volume (i.e. $\Omega =\gtrsim 1000$), it can be shown that the deterministic approach based on concentrations and the stochastic approach considering the interaction of individual particles converge [Gil09]. This result will be used in chapter 4.1 to validate the stochastic simulation tools presented in this thesis by comparing them to the deterministic analytic solutions of several model problems.

In order to ensure that both, the deterministic and the stochastic model describe the same system, the reaction rates have to be rescaled. The autocatalytic conversion reaction (2.4), for example, is represented in the deterministic model by the term $\rho_d uv^2$. In the stochastic case, the propensity function of the reaction is $\alpha_4 = \rho_s uv(v-1)/(2\Omega^2)$. For both description to be equal, the relation $\rho_s = 2\rho_d$ has to hold. For the choice of parameters in this thesis one has $\rho_d = 1.0$ and $\rho_s = 2.0$.

2.3.6 Compartment-based Stochastic Simulation of Diffusion

To simulate spatially inhomogeneous chemical systems, the concept of compartments has to be introduced. A compartment c is a subvolume of the the complete domain of interest V. The set of all compartments C is a partition of the system domain, i.e. $c \cap d = \emptyset\, \forall c, d \in C, c \neq d$ (Compartments don't overlap) and $\bigcup_{c \in C} c = V$ (The complete domain is covered by the compartments). The assumption that only particles close to one another can collide and consequently react is reflected by the principle that only molecules

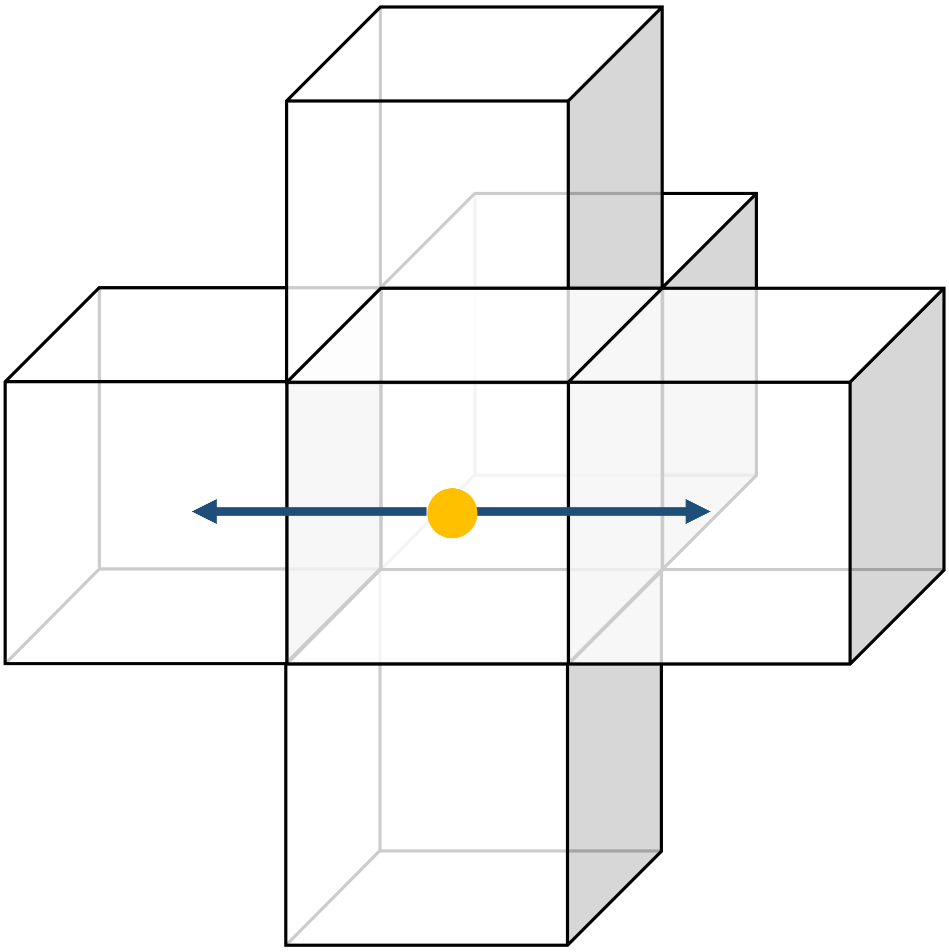

Figure 2.6: Illustration of the jump process used to model diffusion in stochastic simulations. The front compartment is omitted for reasons of visual clarity. The arrows indicate two out of six possible directions in which the particle can move.

within a compartment can interact. Diffusion, on the other hand, is modeled as a jump process of molecules migrating from a compartment to one of its neighbours. Figure 2.6 illustrates the idea.

In general, the shape of the compartments is arbitrary. In this thesis, however, a cubic domain $[0,1]^3$ will be decomposed into L^3 uniform cubes with side length $h = 1/L$. Every compartment is identified by its set of indices $(i, j, k) \in [1, \dots, L]^3$. The volume in space that is covered by the compartment is $[(i-1)h, ih] \times [(j-1)h, jh] \times [(k-1)h, kh]$. By introducing stochastic diffusion constants d_U and d_V a modified description of the Gray-Scott model presented in chapter 2 can be obtained:

Reactions

$$\emptyset \xrightarrow{F} \mathrm{U}_{\mathrm{i,j,k}} \tag{2.45}$$

$$\mathrm{U}_{\mathrm{i,j,k}} \xrightarrow{F} \emptyset \tag{2.46}$$

$$\mathrm{V}_{\mathrm{i,j,k}} \xrightarrow{F+\kappa} \emptyset \tag{2.47}$$

$$\mathrm{U}_{\mathrm{i,j,k}} + 2\,\mathrm{V}_{\mathrm{i,j,k}} \xrightarrow{\rho} 3\,\mathrm{V}_{\mathrm{i,j,k}} \tag{2.48}$$

with positive reaction rate constants F, κ and ρ.

Diffusion

$$\mathrm{S}_{\mathrm{i,j,k}} \xleftrightarrow{d_S} \mathrm{S}_{\mathrm{i\pm1,j,k}} \tag{2.49}$$

$$\mathrm{S}_{\mathrm{i,j,k}} \xleftrightarrow{d_S} \mathrm{S}_{\mathrm{i,j\pm1,k}} \tag{2.50}$$

$$\mathrm{S}_{\mathrm{i,j,k}} \xleftrightarrow{d_S} \mathrm{S}_{\mathrm{i,j,k\pm1}} \tag{2.51}$$

for the two species $\mathrm{S} \in \{\mathrm{U}, \mathrm{V}\}$ with positive diffusion constants d_{U} and d_{V}. The spatial dependency of the system is therefore reflected by the compartment index (i,j,k). It is assumed that only particles within a compartment can react according to equations (2.45) - (2.48). Equations (2.49) - (2.51) describe how particles can migrate from one compartment to another.

It shall be noted that the chosen discretization approach introduces artificial anisotropy to the system in the sense that the number of directions a particle can move in is not infinite, but limited to 6. Consider, for example, a particle that moves along the vector $[h, h, 0]^T$. In reality, the distance the particle travels is $d_r = \sqrt{2}h$, in the discretized model it has to travel $d_d = 2h$. For small h, however, this effect can be neglected. Furthermore, the analysis of the possible consequences is beyond the scope of this thesis.

In order to be able to compare the results of the deterministic and the stochastic approach towards diffusion, it remains to derive a relation between the diffusion parameters in both types of simulation. It is obvious that when the compartment length h is reduced, the diffusion constant has to be increased to keep the "average diffusion velocity" of the particles constant. It has been shown (cite) that deterministic and stochastic simulation are equivalent when d is chosen as

$$d = \frac{D}{h^2} \tag{2.52}$$

An illustrative way to derive this relation is as follows: Considering Fick's law of diffusion $\frac{\partial u}{\partial t} = \Delta u$ and applying a central finite-difference approximation for the Laplace operator leads to:

$$
\begin{aligned}
\Delta u &\approx D\frac{u_{i+1,j,k} + u_{i-1,j,k} + u_{i,j+1,k} + u_{i,j-1,k} + u_{i,j,k+1} + u_{i,j,k-1} - 6u_{i,j,k}}{h^2} \\
&= \frac{D}{h^2}u_{i+1,j,k} + \frac{D}{h^2}u_{i-1,j,k} + \frac{D}{h^2}u_{i,j+1,k} + \frac{D}{h^2}u_{i,j-1,k} + \frac{D}{h^2}u_{i,j,k+1} + \frac{D}{h^2}u_{i,j,k-1} \\
&\quad - 6\frac{D}{h^2}u_{i,j,k}
\end{aligned}
\tag{2.53}
$$

It is obvious that this is equivalent to the stochastic approach if d is chosen as in equation (2.52).

Considering the results stated above, it turns out that spatially inhomogeneous systems can be simulated with both the Gillespie and the tau-leaping algorithm without any changes just by applying the compartment approach. However, the resulting system is by far more complex than its spatially homogeneous equivalent (in both, a mathematical and a computational sense). Considering L^3 cubic compartments in a three-dimensional cubic domain and a reaction system that consists of N local species subject to M reactions, one has NL^3 species and $(M + 3N)L^3$ reactions.

Chapter 3

Stochastic Simulations on Parallel Platforms

"The way the processor industry is going, is to add more and more cores, but nobody knows how to program those things. I mean, two, yeah; four, not really; eight, forget it."

– Steve Jobs [Cla11]

In accordance with Moore's law exponential growth in the level of integration of microprocessors has been observed for more than five decades. As briefly discussed in the introduction to this thesis this does not necessarily imply equal growth rates in processor performance. In order to harness the enormous raw computing power of modern architectures, it is necessary to develop parallel applications.

In this chapter two different parallel platforms are presented that are used to enable accelerated implementations of the tau-leaping algorithm. At first a general overview about features, differences and similarities of both platforms is given. Subsequently a more detailed account on ideas, considerations and techniques incorporated into the final implementation follows.

A lot of research about stochastic simulation algorithms on various parallel platforms has been done in the recent years. Selected examples include [TB05] (OpenMP), [KHM12], [BBHT05] (distributed memory cluster), [Mac08] (FPGA) and [NCB+14, LP10, KEGM11, VLM11] (GPGPU). Depending on application, platform and algorithm significant speedups could be achieved. A novelty of the simulator presented in this thesis is - to the extent of the author's knowledge - the exclusively stochastic handling of diffusion in combination with the tau-leaping algorithm in an approach based on data parallelism.

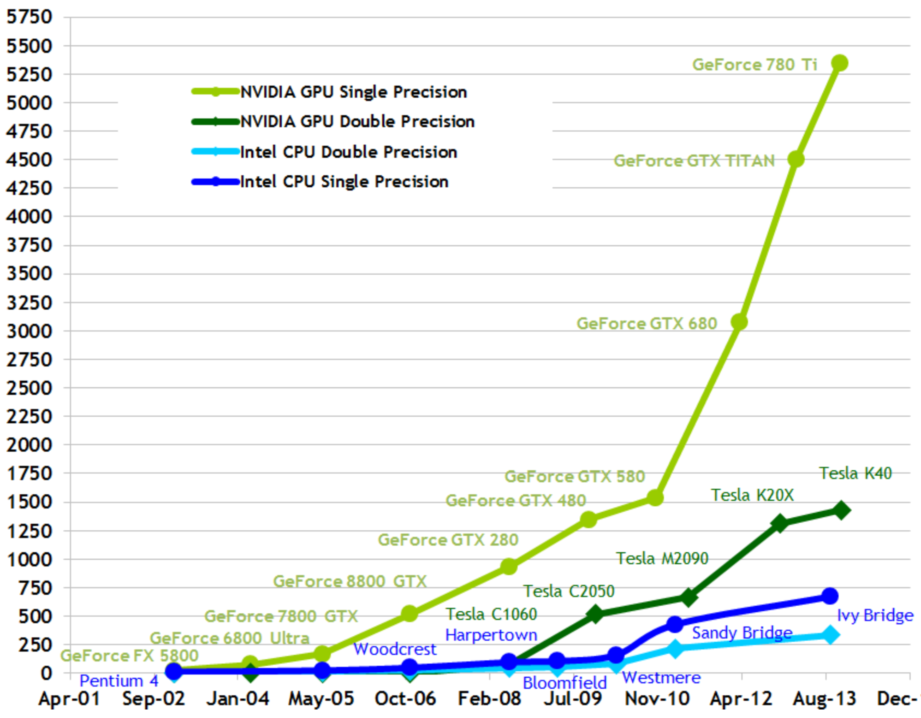

Figure 3.1: Theoretical peek performance of selected Intel CPU and Nvidia GPU models measured in Giga-Floating-point Operations Per Second (GFLOP/s). Image by Nvidia (cite!)

3.1 GPGPU

A Graphics Processing Unit (GPU) is traditionally considered to be an integral part of a workstation computer mainly because it enables accelerated or even real-time rendering of three-dimensional computer graphics. As such, the device relies on a highly specialized and massively parallel architecture. Within a single second hundreds of millions of geometric operations (i.e. translation, rotation, scaling) and rendering procedures (i.e. interpolation, shading) are applied independently to the polygon primitives of a model by the numerous streaming microprocessors the GPU consists of. Due to the application specific design and since almost all parts of the rendering pipeline are hardware-accelerated, i.e. functionality is directly implemented in the physical circuit layout, throughput and computing performance of GPUs are orders of magnitude larger than what a common Central Processing Unit (CPU) can achieve for the same computation. Figures 3.1 and 3.2 show the theoretical peak performance and the memory-bandwidth of selected CPU and GPU models over time.

In order to benefit from this enormous computational power, research has been conducted to investigate if GPUs can be used to accelerate computations beyond the intended scope [HCSL02, Owe04]. Since the application

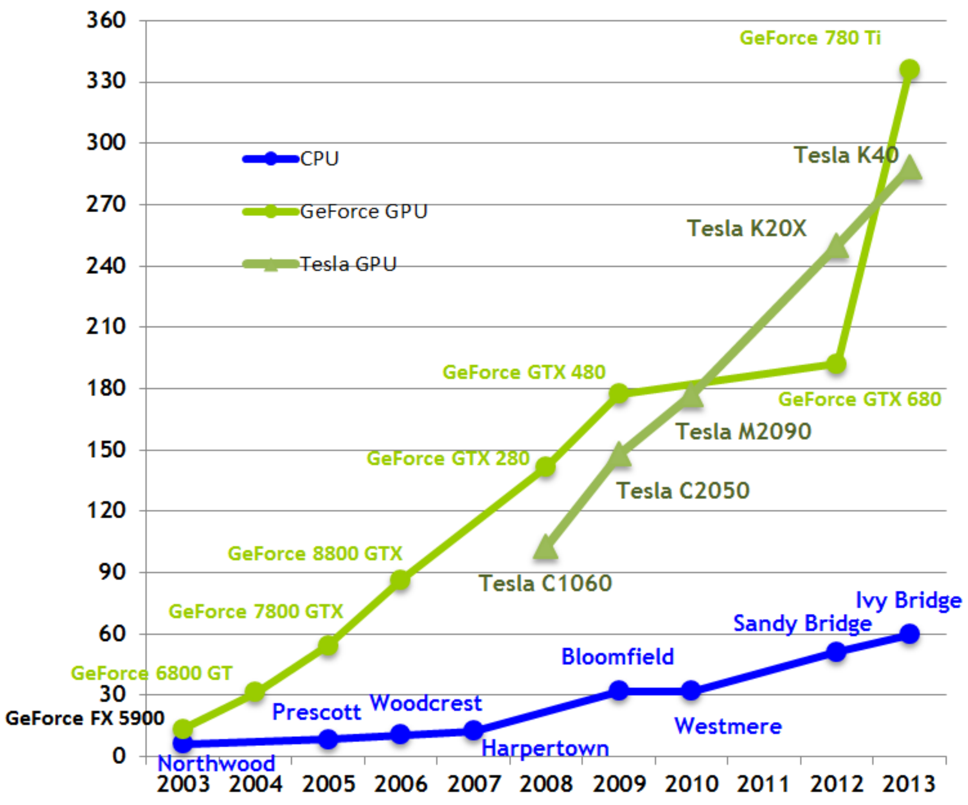

Figure 3.2: Memory bandwidth of selected Intel CPU and Nvidia GPU architectures measured in Gigabytes Per Second (GB/s). Image by Nvidia (cite!)

programming interface (API) of the devices only supported computer graphics development in these days and due to the fact that a lot of the functionality on the device is hardwired, a general problem has to expressed in terms of pixels, textures and shaders. This process requires a lot of experience, detailed knowledge about the hardware and computer graphics programming as well as trial and error optimization. Apart from these practical hurdles, it turned out that a lot of parallel algorithms, i.e. algorithms that consist of independent subproblems that are processed in parallel, can outperform an equivalent CPU implementation by orders of magnitude.

The term General Purpose Computation on Graphics Processing Unit (GPGPU) was coined by Mark Harris in 2006 [LHG$^+$06]. At around the same time, GPU manufacturers realized that scientific computing might be a new market for their products. However, in order to enable the average researcher to harness the computing horsepower of the GPU, improvements in both hardware and software had to be made:

Hardware Over the course of subsequent GPU generations, the flexibility as well as the programmability of the streaming multiprocessors (also called shading units or shader) has increased. Furthermore, essential primitives for concurrent programming such as synchronization and atomics were introduced and optimized. In addition, standard CPU features that were formally not available on GPUs like an automatically managed cache or double-precision floating-point arithmetic were added. Both major vendors of GPUs, AMD and Nvidia, have introduced special versions of their devices for professional high performance computing (HPC) applications. Those products, branded AMD FireStream and Nvidia Tesla, respectively, are optimized for use in multi-GPU cluster located in computing centers.

Software At an even faster rate than hardware improvements, software innovations were presented. At first, native development in widely-known high-level languages such as C or C++ was enabled[1], later bindings for Python, Java, MATLAB and others were provided. In a next step, highly optimized compilers, debuggers, performance analysis tools and complete integrated development environments (IDE) were shipped to further improve development efficiency and product performance. Furthermore, libraries for applications such as linear algebra (BLAS, SPBLAS), signal processing (FFT) and random number generation were introduced. Now, an existing application can be accelerated just by replacing the library used, without writing a single line of GPU code.

Over the years, several programming frameworks and standards for GPGPU were introduced. In the following a short overview is given.

OpenCL The Open Computing Language is an open standard for computation on heterogeneous platforms. As such, it can not only be used for GPU programming, but also to create applications that make use of systems that consist of numerous CPUs and so-called accelerators[2]. It is based on the C/C++ programming language. The standard has been adapted by over 30 companies, including AMD, Intel and Nvidia. OpenCL 1.0 was introduced in December 2008, the latest version OpenCL 2.0 published in November 2013 includes novel execution models and introduces features of C++11. (cite)

OpenACC The Open Accelerators standard is developed by several companies including Cray, Nvidia and PGI. OpenACC is conceptually similar to the widely-used Open Multi-Processing (OpenMP) standard for parallel programming in shared memory systems (see chapter ??). The

[1]The language dialect used for GPU development is usually not absolutely compliant with the original standard language. On the one hand some minor language features are sometimes ommited (i.e. CUDA-C does not support function pointers), on the other hand small additions are added to account for GPU-specific features and properties.

[2]Field Programmable Gate Array (FPGA), Digital signal processor (DSP), GPU

framework primarily consists of a set of compiler directives that can be used to mark regions that are to be run on an accelerator device. The main advantage of this approach is that the developer does not need to know the details of a potentially complicated framework, but can still significantly improve the performance of existing code just by adding compiler directives. Version 2.0a of the standard was ratified in August 2013.

CUDA The Compute Unified Device Architecture is a proprietary framework developed by Nvidia. An initial beta version was published in February 2007. Due to its widespread and early distribution on the market and the large number of libraries, tools and resources available, CUDA is considered to be the marking leader for GPU computation. The GPU implementation of the tau-leaping algorithm presented in this thesis is based on CUDA. The most recent version, CUDA 6.0, was published in April 2014. It provides a unified view on CPU and GPU memory and improved scaling for multi-GPU systems. In the following section more detailed information on the paradigms and features of CUDA will be given.

3.1.1 Nvidia CUDA

The Nvidia CUDA computing platform combines two major paradigms of modern data processing devices: *Single Instruction, Multiple Data* (SIMD) and *multi-threading*. Following the data parallelism approach, the former principle describes the ability of a processor to apply the same operation to different pieces of data at once. The later, on the other hand, distributes independent instruction streams (*threads*) to the available processors.

The abstract principles mentioned above can be derived directly from the actual hardware architecture of the device: A Nvidia GPU consists of an array of *Streaming Multiprocessors* (SM) which in turn are formed by a number of so-called *CUDA-cores*. Following the SIMD principle, a SM can only execute one instruction at a time, i.e. all cores have to apply the same operation to their current data point. If there is code branching within the instructions issued to one SM, individual cores may skip the operations of a non-applicable branch. This leads to serialization of the execution so that in general branching should be avoided whenever possible.

Within the instruction stream running on a SM data parallelism is expressed using the SIMD principle. On a larger scale, considering only the multiprocessor units and their in general different instruction streams, task parallelism can be observed. Analogous to the cores within a conventional CPU, all SMs execute at the same time and so do the threads running on them. On Nvidia GPUs, however, communication and synchronization between threads is limited.

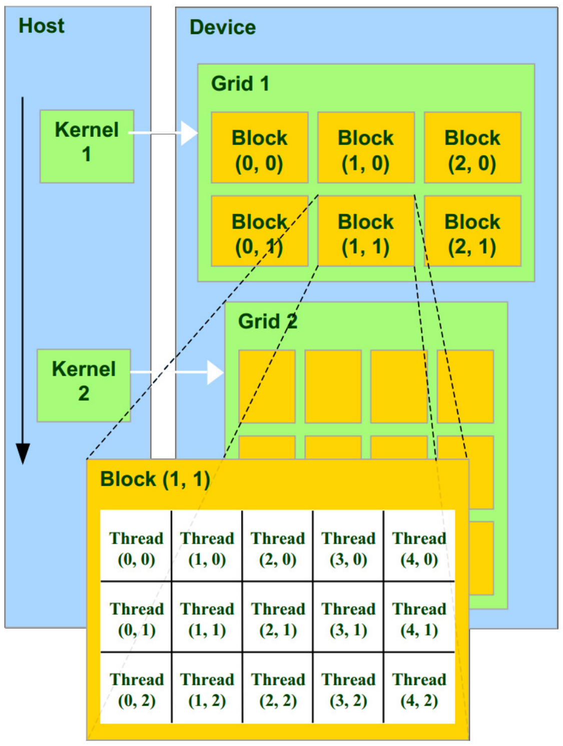

Figure 3.3: Illustration of the CUDA programming model. Figure taken from [Nvi12].

From the point of view of a software developer, a CUDA program consists of code for the host (i.e. the CPU) and the device (i.e. the GPU). Device procedures, so-called *kernels*, must be launched in the host section of the application. The kernel is then automatically replicated many times to create individual threads. In CUDA, threads are organized in one-, two- or three-dimensional structures called *blocks*, which in turn are part of a *grid*. The size of a block is limited by the hardware and so is the size of blocks and grids. Threads within a block can be synchronized during execution of the kernel, on a global level, however, the only way of synchronizing all threads is the invocation of two destinct kernels. The device assigns blocks to SMs and further splits them to form batches of 32 threads named *warp*. Since the number of threads per warp is equal to the number of cores per SM, a warp can be executed directly. Each thread can determine its position in grid, block and warp at runtime. The complex dispatch and scheduling procedure allows CUDA programs to achieve good performance on different GPUs independent of the properties of the specific model (e.g. number of SMs). Figure 3.3 visualizes the concepts introduced above.

One of the most important aspects to consider when developing GPU applications is the memory hierarchy of the device. Figure 3.4 gives an overview over that different kinds of memory on the device, table 3.1 lists bandwidth and latency.

Global memory Global memory can be considered to be the GPU equivalent of the CPU main memory. Its capacity usually exceeds several gigabytes, the bandwidth is in the order of 100 GB/s (see figure 3.2). It is the only kind of memory that can be accessed directly from the host CPU via the PCI-Express port (~6 GB/s). (some additional sentences)

Shared memory Data in shared memory is private per thread-block, i.e. it can only be accessed by the threads in a block for data exchange. After the complete block has finished execution, its data in shared memory cannot be accessed any more. Physically shared memory is located in banks close to the individual SMs. The bandwidth is usually around 7-10x faster than global memory bandwidth. Since accessing the latter often limits the overall performance of a CUDA application, shared memory can help to reduce the number of expensive global memory transactions by keeping data that is used more than once close to the SM (spatial locality). It is often considered to be a programmer-managed L1 cache. In fact, modern GPUs have an automatically managed L1 cache which physically shares its banks with the latter. (cite cuda handbook, kaufmann)

Registers One of the main difference between GPUs and CPUs is the number of registers available. Registers are small but extremely fast memory units that usually only store one single value. They are located very close to the processor such that its content can be fetched almost instantly (i.e. within a single clock cycle). A standard multi-core CPU performs best if it can execute a small number of threads in parallel. If the number of threads occupying the processor grows, the overall performance decreases since frequent context switches are needed to ensure that every thread gets a fair share of the computing power. In this process it is necessary to restore the state of a suspended thread before it can execute again, i.e. the register values of the previously active thread have to be stored and those of the next thread have to be restored. On a CPU, this is a length process, but on a GPU ever thread has its own set of registers so that context switching does not involve refilling registers with values stored in slower memory regions. Fast context switching can therefore be used to hide memory latencies: Whenever a warp stalls because it has to wait for a memory transaction to finish, another warp can be executed. The delay of this switch is negligible.

For the sake of completeness it has to be mentioned that there are other kinds of memory (e.g. constant and texture memory) that can be accessed in CUDA. They can be useful for some applications, especially for image

Storage Type	Registers	Shared Memory L1 Cache	Global Memory
Bandwidth	~8 TB/s	~11.5 TB/s	~200 GB/s
Latency	1 cycle	1-32 cycles	400-600 cycles

Table 3.1: Bandwidth and access time by memory time (Adapted from [Coo12], fixed)

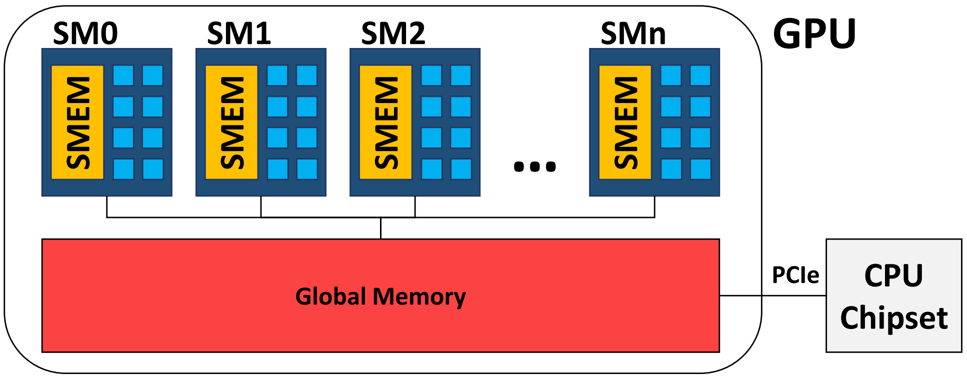

Figure 3.4: The memory hierarchy of Nvidia GPUs (simplified view, adapted from [Nvi12]

processing tasks, but are of no great use for the application presented in chapter 3.1.2.

Furthermore, it is important to note that registers and shared memory have to be considered precious resources in CUDA development. Writing, for example, a kernel that uses to many registers limits the number of blocks that can reside simultaneously on a SM. This, in turn, reduces the number of warps that are available for execution. If there are no alternatives for the processor available to choose from when one warp stalls, the latency of the memory access cannot be hidden.

3.1.2 Implementation

In this chapter the tau-leaping simulator developed in the context of the thesis will be presented. At first, general execution steps of the program and implementation details are discussed. Afterwards, some of the main ideas that were incorporated into the application are explained in more detail. This chapter is solely explanatory in character, the results of numerical validation and performance evaluation experiments will be presented in chapter 4.

Algorithm 3 gives a pseudocode description of the GPU implementation of the tau-leaping algorithm developed for this thesis. Considering the Gray-Scott model as the primary example of this work, the description below represents the special case for $N = 2$ species. It can, however, easily be generalized. In the following the central parts of the implementation are presented in more detail.

1. **Initialization:** During the initialization stage of the application memory for the simulation state is reserved. On the device, two arrays of datatype Integer (int) are reserved to store the number of particles of each species. In the concrete example the arrays are named `d_X1a`, `d_X1b`, `d_X2a` and `d_X2b` where `X1` and `X2` represent species U and V, respectively. The size of the arrays depends on the number of compartments the system consists of. For example, if the cubic domain of a system is divided into 128x128x128 subvolumes (and assuming `sizeof(int) = 4`), one array occupies 8 MB of global memory space. To allow parallel execution of the algorithm, two fields for every species are needed to store the previous and the modified state of the system. Furthermore, if negative populations are observed after a leaping step, the old state can easily be reverted.

   Furthermore, two additional arrays have to be allocated on the GPU: `d_states` and `d_tau_per_cell`. The former is used to store the state structures of the pseudo random generators used in the simulation. For the choice presented in the next section, the maximum size is 48 Byte times the number of compartments. The latter is an integral part of the procedure to calculate the leap-length τ according to equation (2.40). Its size depends on the number of CUDA-blocks spawned on the device during the simulation, it contains floating point numbers, i.e. `float` in the given example. The array contains the minimum value derived from the state of the individual compartments of a block. To determine the value of τ that is used for the next simulation step, the minimum of the values in the array has to be found. A more detailed explanation of the procedure is given in part 5 of this enumeration.

   After being allocated, the values of the device arrays are set by invocating an initialization kernel. It creates and correctly seeds the PRNG states. In addition, the prescribed initial conditions are evaluated and stored in the species state arrays.

   In order to write the state of the simulation to a file, it is necessary to copy the data from the GPU's global memory to the host main memory. This is the reason why on the host an array of the same size has to be allocated for each species of interest. The prefixes '`d_`' and '`h_`' are used by convention to distinguish between host and device arrays. To achieve best data transfer performance, it is important to use so-called pinned memory, i.e. host-side memory that cannot be swapped out to disk by the operating system so that it is instantly available for Direct Memory Access (DMA) copying. Pinned memory can be allocated by library functions that are part of the CUDA framework.

2. **CPU-GPU-interaction:** One of the traditional shortcomings of CUDA and GPU computing in general is the fact that complex control flows cannot be handled on the device. Furthermore, kernels can only be launched by the host, not from within GPU code. For the presented tau-leaping implementation this means that the main simulation loop (`while t < t`$_{\texttt{end}}$) is run on the CPU. Even though all important computations are done in GPU memory, some values such as the proposed tau value or the information if negative populations have been detected must be transferred to the host. If the whole simulation procedure was be manged by the device itself, a lot of time-consuming communication could be avoided. In practice however, implementing the whole algorithm in one complex kernel would neither help achieve high-performance (due to unavoidable branching) nor would it be possible at all (lack of global synchronization).

   On modern Nvidia devices (compute capability $\geq$ 3.0) a feature called Dynamic Parallelism enable kernel invocation within GPU code. Since the tau-leaping simulator presented in this thesis was developed for previous generation hardware (i.e. compute capability 2.0), the feature could not be exploited. However, it seems to be promising and should be considered for future work (see 5).

3. **Random number generation:** To imitate the stochastic behaviour of the simulated system, any SSA is highly dependent on availability, quality and performance of random number generators. In practice random number generation is usually the computationally most demanding part of the algorithm. The cuRAND library by Nvidia offers several different pseudo-random number generator implementations that support various statistical distributions. The manufacturer claims that the library can outperform CPU-based solutions on current-generation hardware by a factor of almost two orders of magnitude. In the tau-leaping simulation a great number of Poisson-distributed random numbers with varying mean are needed, a functionality that cuRAND offers since version 5.0. XORWOW [SM10] is used as the underlying PRNG since it is a good compromise between performance (i.e. gigasamples/second), state size and statistical correctness.

4. **Thrust library:** In CUDA, "common and important data parallel primitives [are] easy to implement [, but it is] harder to get it right" (Harris, 2007). To increase developer efficiency as well as application performance, several CUDA libraries have been introduced to provide easy access to highly-optimized algorithms and subroutines. The Thrust library of parallel algorithms [HB10], for example, is shipped directly with the CUDA toolkit. It closely resembles the C++ Standard Template Library (STL) known to many developers. Its performance in

benchmarks as well as in real-world application is considered to be out of reach for the average researcher who uses CUDA for simulations. In the tau-leaping implementation presented in this thesis two operations provided by thrust are of great importance: parallel minimum reduction and predicate-based logical reduction. The former is used to find the minimum value of τ proposed by the compartments stored in the `d_tau_per_cell` array (see 1). The latter, more precisely the function `thrust::any_of` together with the unary predicate `is_negative` is applied to the modified state arrays of both species X1 and X2. It is used to determine if the proposed step has produced negative molecule populations and, if that is the case, it has to be rejected.

5. **Tau Selection Procedure:** A good example of how library functions and custom CUDA-kernels by the developer can go hand in hand is the implementation of the tau-selection procedure according to equation 2.40. In a first step, the values for g_i and x_i can be determined by the individual thread. Due to the spatial dependency in the simulated system introduced by diffusion, to calculate the mean μ_i and the variance σ_i^2 it is necessary that the molecule population counts of the neighbouring compartments are known. To reduce the number of expensive accesses to global memory, shared memory is used to exploit the principle of data locality within the three-dimensional blocks. Further on, after every thread has calculated its individual tau value, the block-wise minimum of the proposed values is found via reduction in shared memory. Once the kernel has finished execution, the final value for τ is determined by the high-performance reduction routines provided by Thrust.

6. **Reaction kernel:** Implementing the in-compartment reactions (2.1) - (2.4) is straightforward. In the common framework to classify parallel functions `kernel_reactions` implements a map operation. In the concrete example this means that every thread reads two values from distinct positions in two arrays (i.e. the molecular population counts of its compartment) and writes two values to distinct positions in two other arrays (i.e. the molecular counts after the reactions have fired). The operation does not require any synchronization, nor can GPU resources apart from global memory be used to speed up the computation.

7. **Diffusion kernel:** Implementing the diffusion kernel and achieving good performance, on the other hand, is much more difficult. Several different approaches can be taken and features of the underlying hardware can be used to accelerate the computation. In the following the main ideas of three different diffusion kernels will be presented. The results of performance analysis will be presented together with other optimization techniques in chapter 4.3.

a) **diffusion_naive:** This first approach does not make use of shared memory and relies heavily on atomic operations (also called atomics). At first, every thread loads the population counts of the own and neighbouring compartments from global memory. These accesses are not only redundant (every value is needed by 7 threads in 3D), but also strided due to the layout of the compartments in memory (x is the fastest direction, z the slowest). If a thread with three-dimensional index (i, j, k) needs to access a value at $(i + 1, j, k)$ (i.e. the neighboring compartment in positive x-direction), it can be found right next to its own value. If, however, $(i, j, k + 1)$ is accessed, L^2 values are in between. Since memory is always accessed in lines, this will definitely affect performance. Considering, for example, Double Data Rate (DDR) memory and a 384 bit wide memory bus, with every memory transaction 24 integer values can be accessed. If just one value is actually needed, only a small fraction of the theoretical peak bandwidth is used. When a thread has fetched all the values it needs, it uses Poisson-distributed random variables to determine how many molecules leave the compartment through the six (2D: 4) faces. These numbers then need to be added to or subtracted from the respective values in the state arrays. These accesses are again strided and atomic operations need to be used to avoid data races. Every position in the array is written to seven times and if two or more threads try to modify the value at the same time, incorrect result may be obtained.

   Even though no performance data will be given in this chapter, it is obvious that this first naive approach leaves a lot of space for improvements. Due to the simplicity of the actual code it is suitable to show correctnes of the algorithm and it can act as a reference for more sophisticated approaches.

b) **diffusion_shared** One obvious way to improve the naive approach is to make use of shared memory. If all population counts belonging to threads in a three-dimensional cubic block are stored locally at the SM, almost 6 out of 7 global memory transactions can be saved. Furthermore, the relative performance of strided access patterns and atomic operations in shared memory is overproportionally good compared to global memory. To ensure that in a first step all necessary values have been loaded, all threads in a block have to be synchronized by a barrier. Afterwards, diffusion can be simulated the same way it was described above, this time in fast shared memory instead of slow global memory. When all those operations have finished, the resulting delta-values (i.e. the difference between old and new state) are added to the global memory arrays. Compartments at the outer shell of one block are accessed by threads from at least one neighbour (up to three at the corners) so that atomics are needed to ensure correct results.

An additional challenge that arises from this approach is question of how to handle accesses to neighbouring compartments that are not part of the same block. The simplest idea to handle 'out-of-block reads' is to just fall back to global memory. Since these values are only needed by one thread, there is no use in storing them in shared memory. For 'out-of-block writes' so-called overflow buffers can be introduced. Whenever a delta-value is supposed to be added to an 'out-of-block' compartment, it is stored in this temporary field. After all threads of the block have finished their computation of diffusion, the overflow buffers are added to global memory together with the regular delta values.

c) **diffusion_four** The main idea of the third approach presented is to avoid atomic operations in global memory altogether. If the pattern of active blocks resembles that of a three-dimensional chessboard, the overflow buffers are written to positions in the state arrays that are not owned by an active thread. It is termed `diffusion_four` due to the four distinct kernel launches that are needed for the approach. The pattern of active blocks is illustrated in figure 3.5. Apart from a more complicated index calculation section in the beginning, the kernels work exactly the same like the previously presented approach. Avoiding atomic operations comes at the coast of the overhead introduced by three additional kernel launches.

It shall be noted that in practice, since all the data that is needed for `kernel_reactions` is available within the diffusion kernel, the two are combined to reduce the number of kernel launches and global memory transactions.

8. **Output file:** In order to visualize the temporal evolution of the system over time, it is necessary to save the state of the system to the hard drive disk (HDD) in user-defined intervals. For best performance the arrays are written to files in binary, i.e. the sequence of bits in the files and in memory is equal. Whenever the control loop on the host enters the output section, data has to be copied from GPU to CPU memory. Since kernel launches in CUDA are asynchronous and input/output operations are managed by the operating system in the background, the application can continue regular execution even before the file has been stored on the comparably slow HDD. Apart from performance considerations one of the main advantages of the binary file format

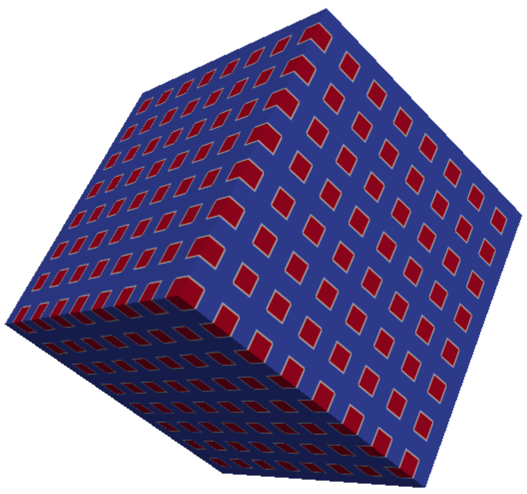

Figure 3.5: Illustration of the access pattern of `kernel_four`. Red blocks are active.

(.bin, .raw) is that it can be read by almost all programs one may use for post-processing. If that is not the case, data can easily be converted with tools such as MATLAB. On the other hand, it shall be noted that binary files are not necessarily interoperable (e.g. depending on system architecture, operating system and compiler different result may be produced), they are not readable for humans and a lot of implicit information such as data type, endianness, system dimension and coordinate axis ordering has to be provided to be able to work with the file.

```
Initialization:
Allocate state arrays d_X1a, d_X1b, d_X2a and d_X2b on the device.
Allocate random generator state array d_states on the device.
Allocate local tau array d_tau_per_cell on the device.
Allocate state arrays d_X1 and d_X2 on the host.
Execute kernel_init_device(d_X1a,...,d_X2b,d_states).
Write initial state to file.
Simulation loop:
Set pointer d_X1_old $\to$ d_X1a.
Set pointer d_X1_new $\to$ d_X1b.
...[a]
while t $<$ $t_{end}$ do
    Execute kernel_tau_local(d_X1_old,d_X2_old,d_tau_per_cell).
    Set tau = thrust :: reduce(d_tau_per_cell,thrust :: minimum).
    while true do
        Execute kernel_reactions(d_X1_prev,...,tau,d_states).
        Execute kernel_diffusion(d_X1_prev,...,tau,d_states).
        if thrust :: any_of(d_Xi_new,is_negative())[b] then
            tau = tau/2
        else
            break
        end
    end
    if t $\geq$ write_counter $*$ write_every then
        Write current state to file.
    end
    Swap d_X1_old and d_X1_old.
    Swap d_X2_old and d_X2_old.
end
Finalization:
Write final state to file.
Deallocate arrays and reset device.
```

Algorithm 3: Overview for GPU implementation !!!

[a]Analogous for d_X2_old and d_X2_new.

[b]For any of the two arrays d_X1_new and d_X2_new, i.e. $i = 1, 2$.

3.2 Multi-core CPU

For a long time the main driver of CPU performance growth has been the clock rate. Within the first years of the new millennium, however, it became obvious that this principle could not be exploited for any further progress. For CPU frequencies beyond 3-4 GHz, it is practically impossible to dissipate the heat released in the chip with common CPU coolers, the so-called *power wall* was reached [Kur01]. In order to enable further performance growth, multi-core architectures were introduced. Instead of trying to improve a single core at enormous expenses, several potentially smaller cores are placed on a single chip. The resulting CPU can then execute several threads truely in parallel, i.e. not only just by assigning timeslots to individual threads. The novel approach turned out to be very successful so that the great majority of processors sold nowadays implement a multi-core architecture.

The downside of this paradigm change from the point of view of a developer is that programs do not automatically execute faster when a new generation of CPUs is introduced. The individual core does not necessarily get any faster and since they are executed one step at a time, sequential programs cannot make use of the increasing number of concurrent computation units in modern chips.

To allow an application to benefit from the parallel architecture, the developer has to explicitly declare sections that can be run concurrently. In the majority of today's most popular programming languages little to no means of expressing parallelism are provided. Even though this is about to change (C++11 has introduce threads, Java's concurrency package is planned to be given a major overhaul), still a lot of work has to be done in this field.

In general, there are two different approaches towards enabling concurrency in an application: Using explicit threading subroutines provided by the operating system (OS) or relying on semi-automatic compiler functionality. The former means that the developer has to explicitly create threads, implement scheduling routines and manage synchronization. On Microsoft Windows systems threading is provided as part of the Base Services included in the Windows API, in a Unix-like environment POSIX Threads (Pthreads) can be used. The latter approach is considered to be significantly faster to apply and easier to learn. It is especially useful when already existing legacy code should be parallelized since the structure of the program does not need to be changed significantly. Two of the most commonly used frameworks are Intel Threading Building Blocks (TBB) and the Open Multi-Processing standard (OpenMP). In the back end semi-automatic solutions depend on the native threading routines available on the targeted operating system. Therefore in practice the performance of both approaches is comparable. The additional level of control obtained by explicitly using OS routines comes at the cost of greater development effort.

3.2.1 OpenMP

"OpenMP is a specification for a set of compiler directives, library routines, and environment variables that can be used to specify high-level parallelism in Fortran and C/C++ programs. [Boa14]" A first version for the Fortran programming language was published in October 1997, support for C/C++ was added in October 1998. The latest version, OpenMP 4.0, was released in July 2013. OpenMP (OMP) implements the fork-join multithreading model, i.e. there is one master thread which can be forked to create several other threads. A region of parallel execution is ended by joining all threads with the master. To distribute the work available in a parallel region, OpenMP comes with a variety of scheduling strategies such as static (i.e. compiletime) and dynamic (i.e. runtime) scheduling. Outside of this region the program is executed sequentially.

Creating a parallel region in OpenMP is straighforward: In C/C++ the directive `#pragma omp parallel` orders the compiler to create several threads by forking the master thread. The number of threads can be prescribed by the developer, set at runtime or determined automatically by the system taking into account the number of physical execution units. If a compiler does not know the standard (or the necessary flags to enable it are not set), the directives will be ignored and in most situations a working serial version of the application can be created.

By default all the code enclosed by the the braces ({. . . }) specifying beginning and end of the parallel region is executed by ever thread. To tell the compiler that a specific subsection should only be executed once, directives such as `#pragma omp single` or `#pragma omp master` can be used. The work within a for-loop is shared by adding `#pragma omp for` in front of it. If the parallel region consist just of a single loop, this can be shortened to `#pragma omp parallel for`. Apart from simple loop parallelization OpenMP offers more advanced features such as reduction, atomic operations, explicit thread synchronization and (debuting in version 3.0) tasks. For an in-depth introduction to all the features of OpenMP the reader shall be referred to an online course by Tim Mattson from Intel[3].

3.2.2 Implementation

In the following the implementation details of the multi-core CPU version of the developed tau-leaping simulator will be presented. The OpenMP standard is used for parallelization. In the pseudocode description given in algorithm 4 a lot of conceptual similarities to the GPU implementation (see

[3]Tim Mattson is one of the original developers of OpenMP. His video lectures can be found online at http://goo.gl/VObNd1, the complementary slide at http://goo.gl/bDWka6.

algorithm 3) can be observed. There are, however, certain differences that directly reflect the differences of the underlying hardware.

1. **Initialization:** Just like in the GPU version, two arrays of type `int` are used in the algorithm to represent the number of molecules of one species in the individual compartments. For the two species in the Gray-Scott example, these fields are called `h_X1a`, `h_X1b`, `h_X2a` and `h_X2b`. Even though all simulation data resides in CPU main memory, the `h_` prefix is used to underline the similar structure of both implementations.

   The required reduction operation to find the minimum proposed tau value is a built-in feature of OpenMP (`#pragma omp for reduction (min:tau)`) so that there is no need for an additional array to explicitly store candidate values. Furthermore, unlike in the GPU case, no temporary storage for IO operations is needed since data from main memory can be written directly to the HDD. The PRNG states are encapsulated in a helper object of type `paraRNG`, a class that was developed to enable thread-safe access to a variety of RNG implementations (see section 3).

2. **Parallel region:** An important characteristic of the presented algorithm is the fact that there is solely one large parallel region. It may seem counter-intuitive to make all threads deal with the work introduced by evaluating loop control conditions. However, this additional effort is negligible compared to the overhead that would be introduced if threads were continuously created and destroyed. Sections that shall be executed exactly one are guarded by the directive `#pragma omp single`. Unlike in the CUDA implementation where separate kernel invocations are needed, to achieve global synchronization in OpenMP it is not necessary to leave the parallel region. In fact, thread barriers are introduced automatically at reasonable places in the code (e.g. after directives such as `parallel for` or `single` and at the end of a parallel region). Implicit synchronization can be disabled in some cases by adding the keyword `nowait`.

3. **Random number generation:** Within the CUDA framework, due to its good documentation, large number of features and outstanding performance, the cuRAND library is a reasonable choice when random numbers are needed in an application. Considering a generic C++ environment, the situation is not so clear since a great variety of libraries is available. The performance of any tau-leaping implementation relies heavily on the underlying PRNG's ability to provide Poisson-distributed random numbers with varying mean. Benchmarking was used to find a suitable solution for the CPU version of the presented simulator. Candidate libraries were chosen according to

Library	Generator	Uniform	Poisson
C++11	mt19937	5.07	3.06e-2
GSL	mt19937	5.56	3.73e-2
TRNG	mt19937	4.15	12.6
TRNG	lagfib2xor_19937_ull	2.67	11.0
TRNG	lagfib4plus_19937_ull	3.31	12.0

Table 3.2: Performance benchmark of selected random number engines. The test program was compiled and executed on an Intel Core i5-2430M @ 2.4 GHz running Ubuntu 14.04 using the GNU C++ compiler in version 4.8.2 with parameter -O3. Execution times measured for four threads with OMP host functions (values rounded, unit: seconds). Testcase 'uniform' takes 10^9 samples, 'poisson' requests 10^5 values from Poisson distributions with mean linearly spaced in [4,1200].

a) availability of the required features, b) stochastic correctness (i.e. performance in empirical testing toolkits), c) performance in available benchmarks and d) ease of incorporation into existing code. Three libraries were considered for the benchmark: Tina's Random Number Generator Library (TRNG) [Bau], the GNU Scientific Library (GSL) [Gou09] and the pseudo-random number generation library that come with C++11. The results are presented in table 3.2.

It is interesting to see how the TRNG library outperforms the two alternatives when a random variable $X \sim \text{Unif}(0, 1)$ is simulated, but is far behind when Poisson values with varying mean are needed. The reason for this behavior is the enormous expense at which the distribution objects are created that are used to generate $Y \sim \text{Pois}(\lambda)$ from $X \sim \text{Unif}(0, 1)$. Since the mean λ (i.e. the propensity values in the simulation) changes constantly, this cost is never amortized. The C++11 library also needs to create a new (relatively lightweight) distribution object at every access, GSL provides a method that does not explicitly have this requirement. Nevertheless, the built-in library turns out to perform best in a testcase that closely resembles the demand of the real application and was therefore chosen for use in the simulator. Unlike for the GPU implementation, the size of a PRNG state is not of great importance. Due to the small number of threads (orders of magnitued smaller than in GPU applications), the space occupied is negligible.

To enable safe parallel access to the random number generator, the wrapper class `paraRNG` was introduced. It initializes a number of PRNGs depending on how many OMP threads are to be created. A unique seed value is derived from the thread IDs. Later, each thread can access the correct state structure by obtaining its ID by calling `omp_get_thread_num()`.

The idea that all threads use the same PRNG but different initial states is not the only way to generate random number in a thread-safe way. In fact, sometimes this approach is considered to be flawed. "The hope is that [generators seeded in this way] will generate non-overlapping and uncorrelated subsequences of the original PRNG. This hope, however, has no theoretical foundation." [Bau] For methods such as *Leapfrogging* or *Block splitting* it can be shown that uncorrelated streams of random numbers are created. The clear disadvantage of these approaches is the greater computational complexity of random number generation. Considering the cuRAND library (see [Nvi14]), the seed used for the individual generators is by design dependent on the thread index. To maintain comparability between GPU and CPU implementation and since statistical flaws in the presented simulations could be observed by comparison with results from literature, it is justified to use the parallelization approach proposed above.

4. **Reaction and diffusion simulation:** Similar to the GPU implementation reaction and diffusion can be combined to be simulated simultaneously avoiding unnecessary overhead. Again, finding a good way to handle the complex spatial dependencies that need to be resolved when simulating diffusion is key to good performance. In the following two approaches are presented, data and considerations on performance are give in chapter 4.3.

   a) **Atomics** Similar to the naive kernel presented in the GPU context (see chapter 3.1.1), this approach makes use of atomic operations to solve situations when two threads want to change the state of a compartment at the same time. The task of simulating reactions and diffusion for all compartments prescribed by a for loop is shared among all OMP threads. Again every thread needs to access and modify the values of all neighbouring compartments. The advanced caching functionalities available on modern CPUs help to lessen the negative effects of the strided access pattern. In OpenMP operations such as incrementation can be made atomic by the directive `#pragma atomic`. On most platforms hardware support is available to accelerate atomic operations.

   b) **Sixtyfour** Even if atomics are hardware-accelerated, such operations take significantly longer than regular memory transactions (factors of 2-25 are usually observed in practice). Complementary to the approach termed `kernel_four` on the GPU atomic operations can be avoided altogether at the cost of a more complex access pattern. If one considers the fact that a thread which simulates a specific compartment needs to update the values of all the six neighbours, it can be concluded that when there are two inactive compartments between two active ones in all three coordinate directions, no conflicting memory operations (i.e. race conditions) can occur. Some of the presented approaches for the GPU introduce constraints on the allowed number of compartments (e.g. divisibility by 8 of the extent in any direction). In order to maintain a high degree of comparability between CPU and GPU version of the application, it should be avoided to further reduce the number of possible domain configurations that can be handled by both simulators. Therefore, a spacing of three inactive compartments is chosen, i.e. only every fourth compartment in any direction is processed by threads in parallel. The activation pattern is visualized in figure 3.5. the procedure results in 64 separate stages, in between a synchronization barrier is needed to ensure consecutive execution.

5. **Differences between OpenMP and CUDA:** OpenMP is an easy to learn high-level approach towards thread-based parallelism while CUDA is data centered, more complex and offers the developer fine-grained control. In consequence, a parallel application can be developed rather quickly with OpenMP. At some point, however, it becomes very complicated to implement complex ideas that would require a greater degree of control over behavior and collaboration of the individual threads. For the concrete example of the thesis this means that concepts such as tiling and vectorization that come naturally in CUDA code can hardly be exploited in the OpenMP version to increase performance.

The application was programmed in C++ and built using Nvidia CUDA compiler 6.0.1 and GNU C++ compiler 4.8.3. The code together with build scripts is provided in the supplementary material of this thesis.

```
Initialization:
Allocate state arrays h_X1, h_X1b, h_X2a and h_X2b.
Initialize random generator, apply initial conditions and write state to file.
Simulation loop:
Set pointers h_X1_old, h_X1_new, h_X2_old and h_X2_new (Analogous to alg. 3).
#pragma omp parallel
while $t < t_{end}$ do
    #pragma omp for reduction(min:tau)
    for every compartment do
        Compute local proposal for tau.
    end
    while true do
        #pragma omp for
        for every compartment do
            Simulate reactions and diffusion.
        end
        #pragma omp for reduction(|:any_is_negative)
        for every compartment do
            Check if any of the populations is negative.
        end
        #pragma omp single
        if any_is_negative then
            $tau = tau/2$
        else
            break
        end
    end
    #pragma omp single
    if $t \geq write\_counter * write\_every$ then
        Write current state to file.
    end
    Swap h_X1_old and h_X1_old. Swap h_X2_old and h_X2_old.
end
Finalization: Write final state to file and deallocate arrays.
```

Algorithm 4: Overview for CPU implementation !!!

Chapter 4

Application

"In theory, there is no difference between theory and practice. But, in practice, there is."

– Jan L. A. van de Snepscheut [RS07]

The purpose of this chapter is to apply the developed methods to the Gray-Scott model. At first, systems for which analytic solutions are known are used to validate the individual components of the application. Afterwards simulations of the Gray-Scott model in two and three dimensions are used to illustrate the pattern formation mechanism. Last but by no means least results from performance analysis and optimization techniques applicable for the GPU implementation are discussed.

4.1 Validation

Simple degradation Probably the simplest chemical system one can think of is given as follows:

$$\mathrm{A} \xrightarrow{k} \emptyset \tag{4.1}$$

The system consists solely of one reaction in which molecules of species A are degraded. The parameter k determines the rate at which this process takes place. The corresponding ordinary differential equations is given as

$$\frac{da}{dt} = -ka \tag{4.2}$$

Analogous to [ECM07] initial condition $a(0) = 20$ (number of particles) and parameter $k = 0.1$ are chosen.

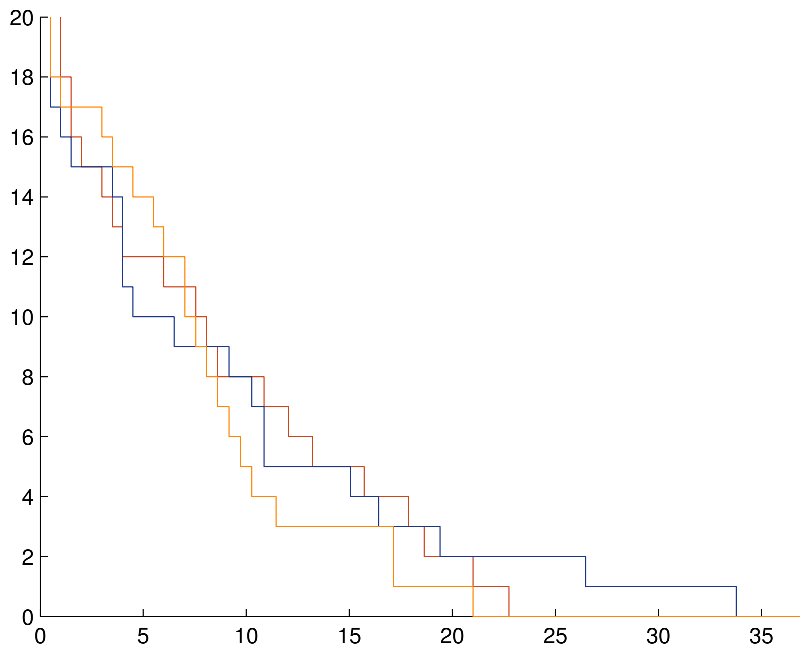

Figure 4.1: Three independent realizations of the simple degradation system (4.1). Parameterization: $k = 0.1$, $a(0) = 20$, $\varepsilon = 0.05$

An analytic solution can be obtained:

$$a(t) = 20 \cdot \exp(-0.1t) \tag{4.3}$$

Figure 4.1 shows three individual trajectories generated by the tau-leaping algorithm with adaptive step size selection ($\varepsilon = 0.05$), in figure 4.2 a plot of both the analytic solution (4.2) and the average of 128^3 stochastic simulation runs is given. The maximum absolute error is 0.1974, the maximum relative error is 0.1138 for $t \in [0, 40]$.

Production and degradation Another system that is commonly used to validate stochastic simulation algorithms is the production-degradation system:

$$\mathrm{A} \xrightarrow{k_1} \emptyset \tag{4.4}$$

$$\emptyset \xrightarrow{k_2} \mathrm{A} \tag{4.5}$$

For the reaction rates k_1 and k_2 values 0.1 and 1.0 are chosen, initial condition is $a(0) = 0$. Considering the two propensity functions $\alpha_1(t) = k_1 a(t)$ and

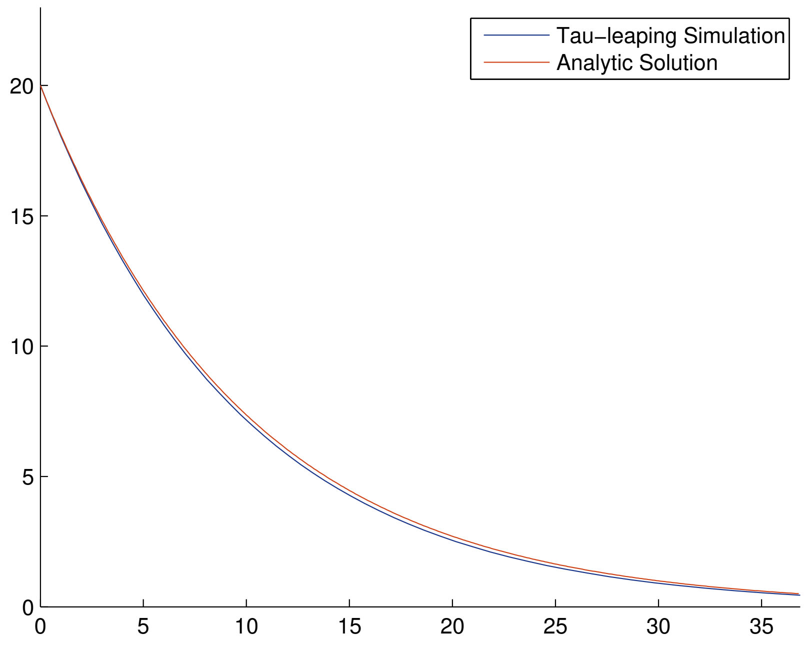

Figure 4.2: Analytic solution of the simple degradation system (4.1) and average of $128^3 = 2097152$ tau-leaping simulations.

$\alpha_2(t) = k_2$ it is trivial to see that the system is in equilibrium for $a = 10$. The same observation can be made from the differential equation:

$$\frac{da}{dt} = -k_1 a + k2 \tag{4.6}$$

The analytic solution is

$$a(t) = -10 \exp(-0.1t) + 10 \tag{4.7}$$

Figure 4.3 again shows a plot of the analytic solution and the average of 128^3 tau-leaping simulations ($\varepsilon = 0.05$). The maximum absolute error for $t \in [0, 100]$ is 0.0525, the maximum relative error is 0.0525. In figure 4.4 a histogram on the number of molecules at $t = 100$ clearly shows the equilibrium for 10 particles.

2D heat equation In order to validate the diffusion mechanism of the stochastic simulators the parabolic partial differential equation know as heat equation (in this context also called diffusion equation) is considered:

$$\frac{\partial\rho(x,y,t)}{\partial t} = D\Delta\rho(x,y,t) = D\left(\frac{\partial^2\rho(x,y,t)}{\partial x^2} + \frac{\partial^2\rho(x,y,t)}{\partial y^2}\right) \tag{4.8}$$

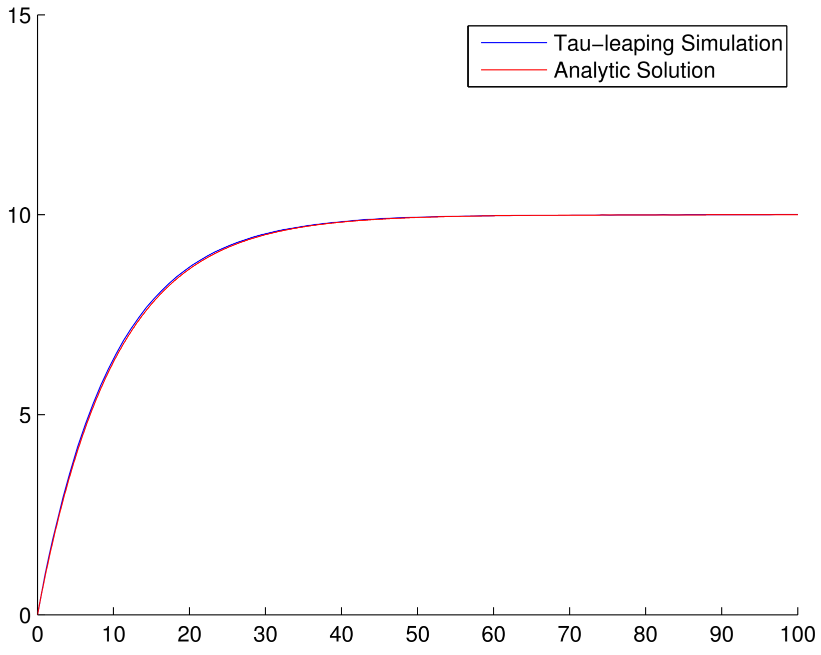

Figure 4.3: Analytic solution of the production-degradation system (4.4)-(4.5) and average of $128^3 = 2097152$ tau-leaping simulations. Parameterization: $k_1 = 0.1$, $k_2 = 1$, $a(0) = 0$, $\varepsilon = 0.05$

$\rho(x, y, t)$ can either be considered the amount of heat or the concentration of some species at time t in a point at (x, y). The diffusion constant D is a positive constant. To facilitate visualization two-dimensional diffusion is considered.

The domain Ω shall be the unit square, i.e. $x, y \in [0, 1]$. The system is endowed with Dirichlet boundary conditions

$$\rho(0, y, t) = \rho(x, 0, t) = \rho(1, y, t) = \rho(x, 1, t) = 0 \; t > 0 \tag{4.9}$$

and an initial distribution

$$\rho(x, y, 0) = \sin(x\pi) \cdot \sin(y\pi) \tag{4.10}$$

Under these conditions an analytic solution can be obtained:

$$\rho(x, y, t) = \sin(x\pi = \cdot \sin(y\pi) \cdot \exp(-2\pi^2 Dt) \tag{4.11}$$

Figures 4.5 and 4.6 show the state of the system ($D = 10^{-3}$) at $t = 60$ obtained by the analytic solution and tau-leaping stochastic simulation, respectively. For the latter 128x128 compartments, the accuracy control parameter $\varepsilon = 0.05$ and the scaling constant $\Omega = 1000$ were used.

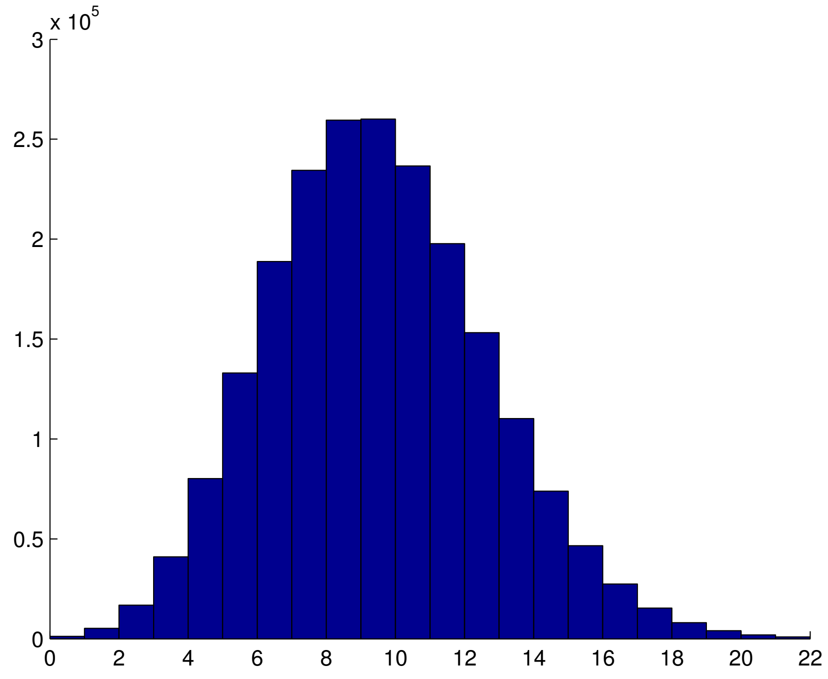

Figure 4.4: Histogram of the production-degradation system (4.4)-(4.5). Parameterization analogous to figure 4.3.

The number of molecules is proportional to the volume integral over concentration. For the 2D case one has the following area integral:

$$C_{analytic} = \int_0^1 \int_0^1 \rho(x, y, 60) = \frac{4}{\pi^2} \cdot \exp(-0.12 \cdot \pi^2) \approx 0.12399 \qquad (4.12)$$

For the democratized stochastic approach it is equivalent to consider the sum over all compartments:

$$C_{discrete} = \cdot \sum_{i,j} v_{i,j} \cdot \frac{x_{i,j}}{\Omega} \qquad (4.13)$$

where v is the volume of the compartment and x the number of molecules in it. Considering the presented two-dimensional example with 128x128 uniform square-shaped compartments one has for the absolute error $\Delta C = |C_{analytical} - C_{discrete}| \approx 1.4232 \cdot 10^{-5}$, for the relative error $\delta C = \Delta C / C_{analytic} \approx 1.1478 \cdot 10^{-4}$.

4.2 Pattern formation in the Gray-Scott System

One of the main reasons for the great interest in the Gray-Scott model is its ability to develop complex spatio-temporal patterns. Pattern formation is a

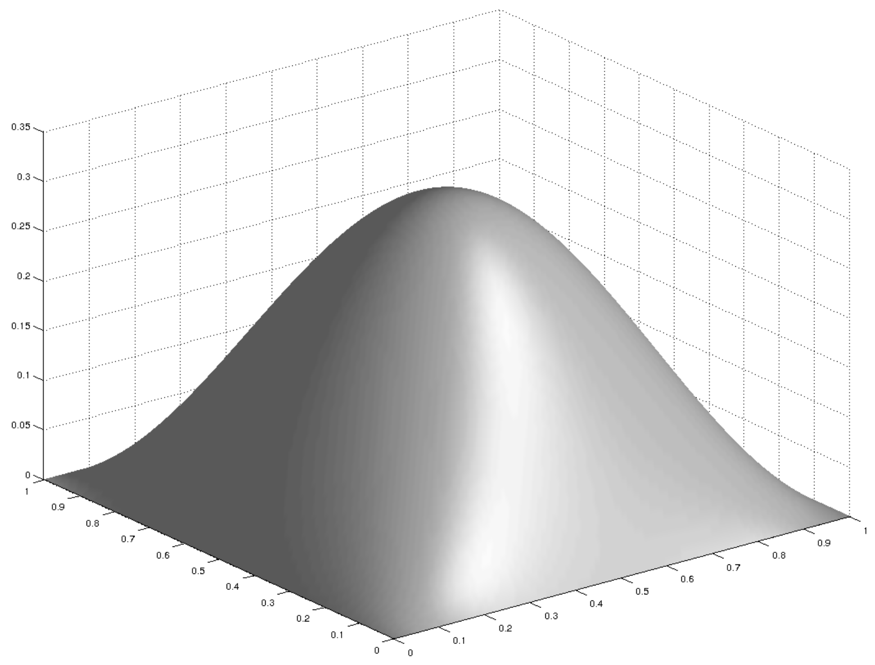

Figure 4.5: Analytic solution of heat equation (4.8) for $D = 10^{-3}$ with boundary condition (4.9) and initial condition (4.10) at $t = 60$.

form of self-organization which plays a central role in a variety of processes observed in nature that are crucial for the existence of live in any form (see chapter 1 for examples). In this section it is shown how the developed stochastic simulator can be used to illustrate the temporal evolution of the system.

The following initial conditions for the number of particles of species U and V are used throughout the whole thesis:

$$u_0 = \begin{cases} \Omega\left(\frac{1}{2} + \xi \cdot 10^{-2}\right) & \text{if } \mathcal{G}_{int} \\ \Omega\left(1 + \xi \cdot 10^{-2}\right) & \text{otherwise} \end{cases} \tag{4.14}$$

and

$$v_0 = \begin{cases} \Omega\left(\frac{1}{4} + \xi \cdot 10^{-2}\right) & \text{if } \mathcal{G}_{int} \\ \Omega\left(\xi \cdot 10^{-2}\right) & \text{otherwise} \end{cases} \tag{4.15}$$

with random variable $\xi \sim \text{Unif}(0,1)$ and inner region $\mathcal{G}_{int} = [1/4, 3/4]^d$ where d is the dimension of the system under consideration. The initial conditions were chosen in dependence on **??**. The parameters used are $F = 0.04$, $\kappa = 0.06$, $\rho_d = 1 - 0$ (i.e. $\rho_s = 2.0$), $D_u = 2 \cdot 10^{-4}$.

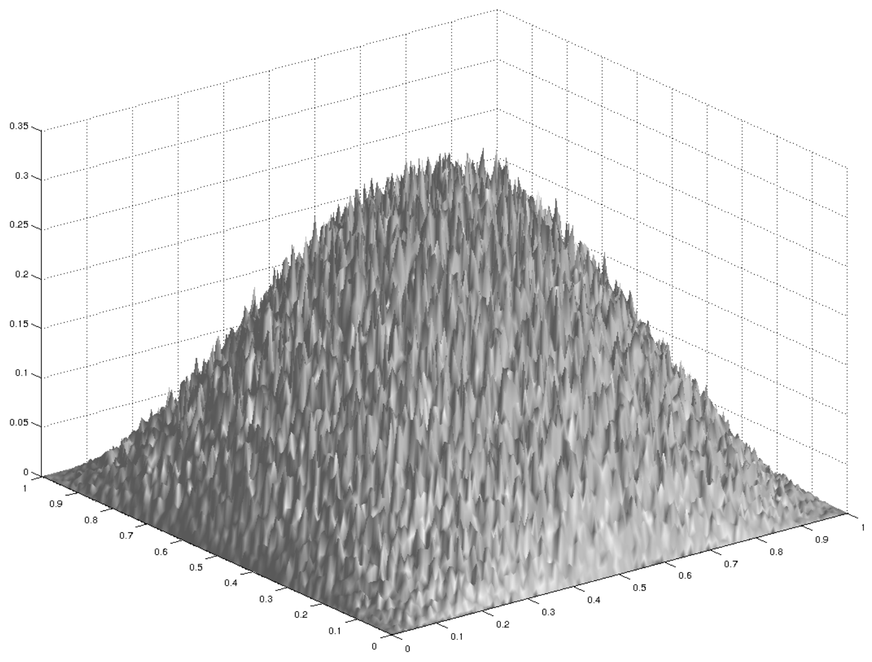

Figure 4.6: Stochastic solution of heat equation (4.8) for $D = 10^{-3}$ with boundary condition (4.9) and initial condition (4.10) at $t = 60$. Parameterization: $\varepsilon = 0.05$, $\Omega = 1000$, spacing h=1/127

Figure 4.8 illustrates the influence of the scaling parameter Ω. For large values (e.g. $\Omega \gtrsim 10000$), the system is close to the thermodynamic limit, i.e. the deterministic approach is a valid approximation to the stochastic microscopic phenomena **??**. Figure **??** gives insight into the temporal evolution of the system, figure 4.7 shows the isosurface for the concentration of species U at time $t = 1000$ with respect to an isovalue of 0.5.

In the literature and on the Internet numerous articles and websites on Gray-Scott pattern formation exist. Two especially illustrative resources on the topic shall be mentioned here: Project *Xmorphia*[1] by Roy Williams at Caltech and the *Amorphous Computing*[2] project by Abelson et al. at MIT CSAIL. Both websites enable interactive exploration of the system behaviour under different conditions. The former resource includes a map of the parameter space which directly links to videos that visualize the temporal evolution of the system for the selected setup. The latter includes interactive simulators and videos that illustrate the great variety of patterns that can be obtained.

[1]Original website unavailable. Extended version by R. Munafo: `http://goo.gl/vFnOU8`
[2]`http://goo.gl/4iZUb1`

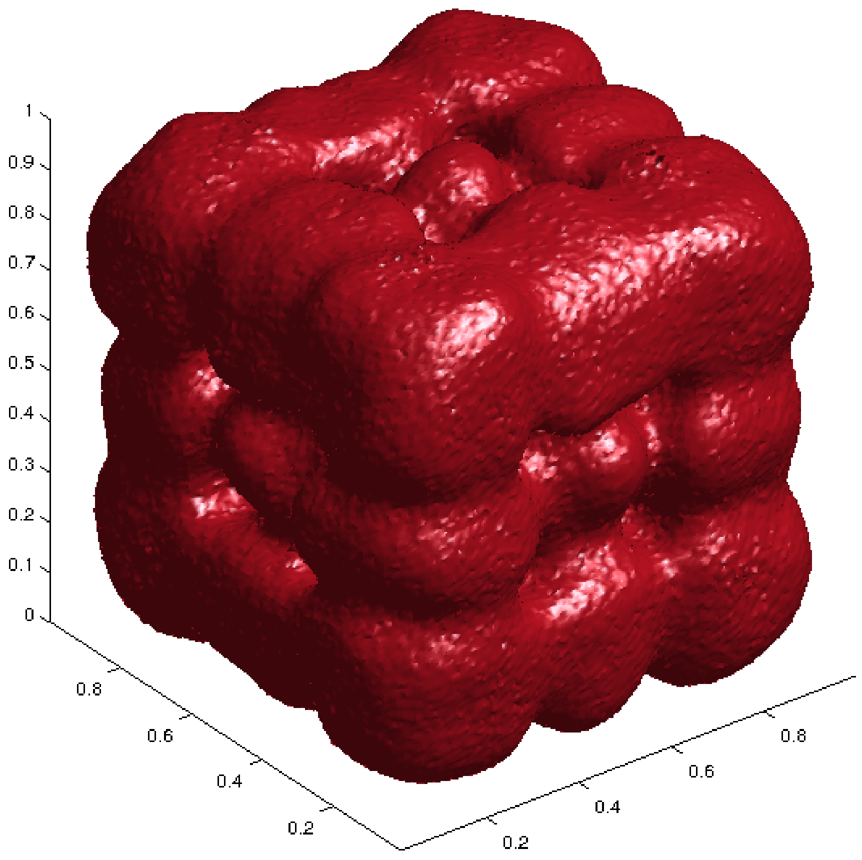

Figure 4.7: Isosurface representation of the concentration of species U at $t = 1000$ for an isovalue of 0.5.

4.3 Performance Analysis

Simulation platforms: GTX 580, ...

Optimization of the GPU version A GPU is a powerful but highly complex device. In order to ensure that an application uses the available resources efficiently performance evaluation and iterative optimization must be an integral part of the development process. The CUDA framework comes with advanced profiling tools (e.g. the Nvidia Visual Profiler) that automatically detect some of the worst pitfalls in GPU programming and can support the developer in trying to avoid them.

In the following some optimization ideas that arouse during the development process of the GPU tau-leaping simulator are discussed.

Initial In chapter 3.1.2 three different approaches towards implementing

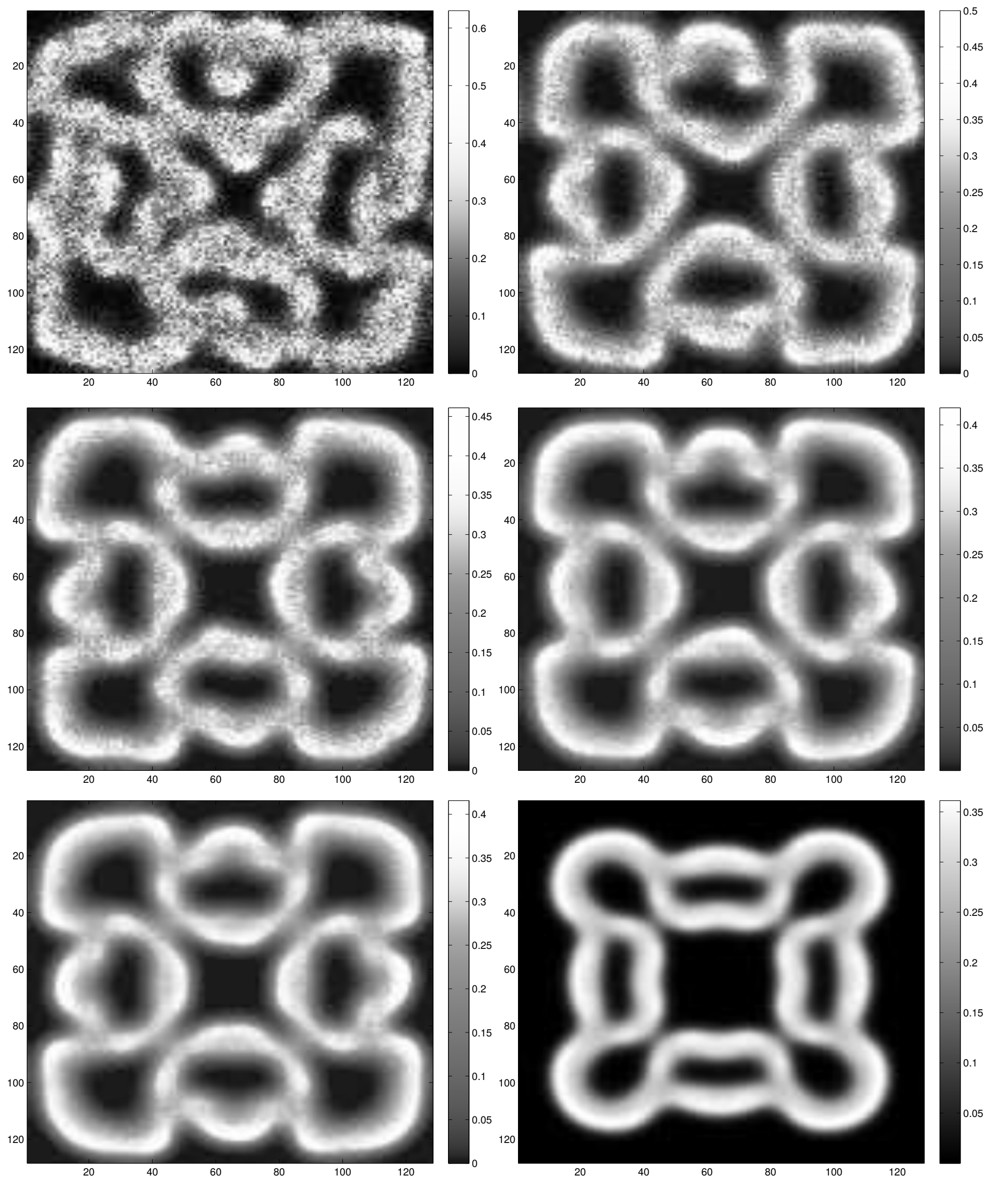

Figure 4.8: Pattern formation under the influence of varying molecule numbers. Figure shows simulation domain $[0,1]^3$ at time $t = 1000$ divided in 128^3 compartments and sliced at $x = 0.6$. Parameters $F = 0.04$, $\kappa = 0.06$, $\rho_d = 1 - 0$ (i.e. $\rho_s = 2.0$), $D_u = 2 \cdot 10^{-4}$ and $D_v = 1 \cdot 10^{-5}$ are used. From left to right and top to bottom the following values for Ω are used: 100, 500, 1000, 5000, 10000. The plot in the bottom right corner shows the deterministic solution (subject to finite-amplitude random perturbation) obtained with a finite difference approximation. Accuracy control parameter: $\varepsilon = 0.05$

diffusion were presented. In table 4.1 execution times of one complete simulation step (i.e. tau calculation, reaction and diffusion) are given for each of the variants. As expected, the naive kernel performs worst since it does not used fast shared memory. The fact that `kernel_shared` outperforms the more sophisticated alternative `kernel_four`, on the other hand, might be surprising. In **??** it has been shown that atomic operations on CUDA-enabled GPUs are significantly slower than regular memory transactions (up to 100x). In reality, however, it seems that the slowdown due to the overhead of three additional kernel launches and a more irregular access pattern overweights the benefits of avoiding atomics in global memory. The main reason for this result is definitely the accelerated hardware support for atomic operations in modern GPUs such as the Nvidia GeForce GTX 580 (in general compute capability $\geq$ 2.0). Another contributing factor is the small degree of the worst case collision, i.e. a maximum of four threads can possibly access a value in global memory at the same time. On GPUs it is often the case that a simple, computationally more expensive algorithm with simple control flow outperforms a complex, more efficient implementation that would be faster on a CPU. In the following only `kernel_shared` is considered since it performed best in all stages of development and in the final simulator.

Combination of reaction and diffusion As mentioned before the reaction and the diffusion operation can be merged. All information that is needed for simulating the local reactions is available within the diffusion kernel. Consequently the overhead of one kernel launch and more importantly several costly operations in global memory can be avoided. The relative speedup of this optimization is $\approx$ 4.3%.

Register count optimization In chapter 3.1.2 the remark has been made that multiprocessor registers have to be considered a valuable resource in GPGPU computing. The reason will become obvious now: The original implementation of the reaction-diffusion kernel occupies 63 registers. Considering a block size of 512 threads (i.e. a 8x8x8 grid is used to reduce the number of global memory atomic operations) and 32768 registers that are available on each SM, only one block can reside on it at a time. This means that the number of threads that are available for execution in case a warp stalls to wait for a memory transaction is limited. As a result of the low *occupancy* memory latency cannot be hidden by fast context switching. By using switches such as `-maxrregcount` the compiler can be forced to reduce the number of registers occupied by the kernel. Values that do not fit into registers any more are spilled to global memory. Trying to hide global memory latency by spilling variables to global memory may sound strange, but since heavily used values are kept in a on-chip L1 cache the approach is promising. By

Optimization	Time / simulation step (ms)	Relative speedup
Initial (naive)	21.98	—
Initial (shared)	16.78	—
Initial (four)	18.35	—
Combination R&D	16.06	4.3%
Register count	14.26	11.2%
Work per thread	8.70	39.0%

Table 4.1: Execution time per simulation step and relative speedup of selected optimization steps. Absolute times are negatively affected by the performance measurement itself since additional synchronization is needed. Values obtained during the standard 128x128x128 compartment simulation of the Gray-Scott model.

limiting the number of registers per thread to 31 two blocks can be scheduled per SM at the same time. The relative speedup obtained for the tau-leaping simulator is 11.2%.

It it obvious that the best alternative to forcing the compiler to lower register counts is not using a great number of registers in the first place. In a complex kernel such as the three-dimensional reaction-diffusion example where local as well as global boundary conditions and special cases have to be handled and comparably large PRNG states have to be kept in local memory reducing the number of registers is a nontrivial task. In future work together with improved compilers that apply more advanced optimization techniques based on compile time knowledge and control flow analysis a further increase in performance can be obtained.

Increase work per thread Even in a framework such as CUDA that heavily relies on data parallelism a solution that exposes as much concurrent work as possible might not be the fastest. A good example for this optimization approach can be found in an example by Nvidia's Mark Harris [**?**]. For the reaction-diffusion kernel under consideration it turns out that the average execution time of a simulation steps is minimized if every thread handles 16 compartments. This values has been obtained by heuristic experiments and depends on GPU generation and model. For the concrete example a considerable relative speedup of 39% is achieved.

Optimization of the CPU version Due to the rather coarse grained task parallelism approach of OpenMP optimizing methods for the CPU version are limited. Avoiding the overhead of one fork-join operation per step by introducing one large parallel region reduces the execution time per step by approximately 50% (1.05s vs. 0.54s on an Intel Core i5-2430M with four threads**??**). Further optimization techniques may involve the use of vector instructions (SSE, AVX) or the introduction of cache-optimized access patterns (tiling, space-filling curve). Those techniques are highly complex to implement and far beyond scope. In this thesis the potential of two parallel platforms for accelerating stochastic simulations in time-limited academic projects is assessed. In this framework an optimal solution in terms of efficiency is neither possible to achieve nor a necessity.

GPU strong domainsize Medium sized system: performance leader
by no means perfect: peak throughput: schoenrechnen

CPU scaling number of threads Memory limited and scaling

resume Difference factor. But in reality highly optimized version. CUDA forces developer to vectorize, makes it easy.

Chapter 5

Conclusion and Outlook

"I think and think for months and years. Ninety-nine times, the conclusion is false. The hundredth time I am right."

– Albert Einstein [?]

What has been done.
Both implementations are not perfect. GPUs are reasonable to use. Worth it to learn. Important skill. Learning ot think in parallel is necessary.
Results: GPU beets CPU for medium sized problems. Interesting devices.
Trend: dynamic parallelism
But don't underestimate the potential of CPUS. Future: Xeon Phi
Stochastic algorithms are great, examples from several scales (universe, cell, ...).
Tau-leaping outperforms classic approaches that consider every individual particle. (more comple implementation)
Computational methods are valuable tool, enables understanding dynamic evolution of biological systems, global-level understanding of emergent dynamics...

Chapter 6

Acknowledgment

First of all sincere gratitude goes to my supervisor Jana Lipková. The contribution of her friendly support and helpful suggestions to this thesis cannot be overestimated. Furthermore, I would like to express my very great appreciation to Petros Koumoutsakos for giving me the opportunity to write this Bachelor's thesis in his lab.

I would like to offer my special thanks to my mentors Hans-Joachim Bungartz and Klaus Mainzer at TUM. Their support before and during my semester in Zurich was enormously beneficial. Professor Mainzer's guidance concerning potential topics for this thesis is greatly appreciated. My thanks to Professor Bungartz for being supervisor to this thesis, for his support with administrative matters in Munich and Zurich as well as for his valuable advice on future studies cannot be overemphasized.

Last but by no means least assistance provided by Gerardo Tauriello, Babak Hejazialhosseini and Christiane Pousa Ribeiro is greatly appreciated.

The semester at ETH was supported by the European Union's Erasmus Program and by the Max Weber-Programme of the sate of Bavaria.

This thesis is dedicated to my parents, my sister and my girlfriend.

Appendix A

Steady State Analysis of the Gray-Scott Reaction System

Calculation of the steady states The steady state condition prescribes

$$0 \overset{!}{=} \frac{du}{dt} = -\rho u v^2 + F(1-u) = f_1(u,v) \tag{A.1}$$

$$0 \overset{!}{=} \frac{dv}{dt} = \rho u v^2 - (F+\kappa)v = f_2(u,v) \tag{A.2}$$

Combining (A.1) and (A.2) leads to

$$u = 1 - \frac{F+\kappa}{F} \tag{A.3}$$

Plugging (A.3) into (A.2) gives

$$-\frac{F+\kappa}{F} v(v^2 + \rho v - (F+\kappa)) = 0 \tag{A.4}$$

The trivial steady state is

$$\vec{x}_{s1} = \begin{bmatrix} 1.0 & 0.0 \end{bmatrix}^T \tag{A.5}$$

The two other steady states are

$$\vec{x}_{s2/3} = \begin{bmatrix} \frac{1}{2}(1 \pm \sqrt{1 - 4\frac{(F+\kappa)^2}{F\rho}}) & \frac{F}{2(F+\kappa)}(1 \mp \sqrt{1 - 4\frac{(F+\kappa)^2}{F\rho}}) \end{bmatrix} \tag{A.6}$$

For the given parameters ($F = 0.04$, $\kappa = 0.06$ and $\rho = 1.0$) one has

$$\vec{x}_{s2} = \vec{x}_{s3} = \begin{bmatrix} 0.5 & 0.2 \end{bmatrix}^T \tag{A.7}$$

Classification of Eigenvalues The Jacobian of the system is

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial u} & \frac{\partial f_1}{\partial v} \\ \frac{\partial f_2}{\partial u} & \frac{\partial f_2}{\partial v} \end{bmatrix} = \begin{bmatrix} -\rho v^2 - F & -2\rho u v \\ \rho v^2 & 2\rho u v - (F + \kappa) \end{bmatrix} \tag{A.8}$$

For the given parameters and at the steady state $\vec{x}_{s1}$ this is

$$J_{s1} = \begin{bmatrix} -0.04 & 0 \\ 0 & -0.1 \end{bmatrix} \tag{A.9}$$

The eigenvalues of this matrix are $\lambda_{11} = -0.1$ and $\lambda_{12} = -0.04$. Since $\Re(\lambda_{11}) < 0$ and $\Re(\lambda_{12}) < 0$ this steady state is stable.

For the given parameters and the steady states $\vec{x}_{s2} = \vec{x}_{s3}$ this is

$$J_{s2} = J_{s3} = \begin{bmatrix} -0.08 & -0.2 \\ 0.04 & 0.1 \end{bmatrix} \tag{A.10}$$

The eigenvalues of this matrix are the roots of the characteristic polynomial

$$(-0.08 - \lambda)(0.1 - \lambda) + 0.08 = \lambda(\lambda - 0.02) \tag{A.11}$$

One obtains $\lambda_{21} = \lambda_{31} = 0$ and $\lambda_{22} = \lambda_{32} = 0.02$.

Bibliography

[And08] David F. Anderson. Incorporating postleap checks in tau-leaping. *The Journal of Chemical Physics*, 128(5):054103, February 2008.

[AP09] Peter Atkins and Julio de Paula. *Physical Chemistry, 9th Edition*. W. H. Freeman, New York, 9th edition edition, December 2009.

[Bau] Heiko Bauke. Tina's random number generator library.

[BBHT05] K. Burrage, P. M. Burrage, N. Hamilton, and T. Tian. Compute-intensive simulations for cellular models. In Albert Y. Zomaya, editor, *Parallel Computing for Bioinformatics and Computational Biology*, pages 79–119. John Wiley & Sons, Inc., 2005.

[BL06] Robert Brym and John Lie. *Sociology: Your Compass for a New World*. Cengage Learning, January 2006.

[Boa14] OpenMP Architecture Review Board. Frequently asked questions on OpenMP, 2014.

[CGP05] Yang Cao, Daniel T. Gillespie, and Linda R. Petzold. Avoiding negative populations in explicit poisson tau-leaping. *The Journal of Chemical Physics*, 123(5):054104, August 2005.

[Cla11] Peter Clarke. Ten quotes on parallel programming | EE times, November 2011.

[Coo12] Shane Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Morgan Kaufmann, Amsterdam ; Boston, 1 edition edition, November 2012.

[ECM07] Radek Erban, Jonathan Chapman, and Philip Maini. A practical guide to stochastic simulations of reaction-diffusion processes. *arXiv:0704.1908 [physics, q-bio]*, April 2007. arXiv: 0704.1908.

[EO05] R. Erban and H. Othmer. From signal transduction to spatial pattern formation in e. coli: A paradigm for multiscale modeling in biology. *Multiscale Modeling & Simulation*, 3(2):362–394, January 2005.

[FPB11] Ashlee N Ford, Daniel W Pack, and RD Braatz. Multi-scale modeling of PLGA microparticle drug delivery systems. In *21st European Symposium on Computer Aided Process Engineering: Part B*, page 1475–1479, 2011.

[Gil76] Daniel T Gillespie. A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. *Journal of Computational Physics*, 22(4):403–434, December 1976.

[Gil01] Daniel T. Gillespie. Approximate accelerated stochastic simulation of chemically reacting systems. *The Journal of Chemical Physics*, 115(4):1716–1733, July 2001.

[Gil09] Daniel T. Gillespie. The deterministic limit of stochastic chemical kinetics. *The journal of physical chemistry. B*, 113(6):1640–1644, February 2009.

[Gou09] Brian Gough. *GNU Scientific Library Reference Manual - Third Edition*. Network Theory Ltd., Bristol, 3rd revised edition edition edition, January 2009.

[GS84] P. Gray and S. K. Scott. Autocatalytic reactions in the isothermal, continuous stirred tank reactor. *Chemical Engineering Science*, 39(6):1087–1097, 1984.

[HB10] Jared Hoberock and Nathan Bell. *Thrust: A Parallel Template Library*. 2010. Version 1.7.0.

[HCSL02] Mark J. Harris, Greg Coombe, Thorsten Scheuermann, and Anselmo Lastra. Physically-based visual simulation on graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '02, page 109–118, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[KEGM11] Guido Klingbeil, Radek Erban, Mike Giles, and Philip K. Maini. STOCHSIMGPU: parallel stochastic simulation for the systems biology toolbox 2 for MATLAB. *Bioinformatics*, 27(8):1170–1171, April 2011.

[KHM12] Edward Kent, Stefan Hoops, and Pedro Mendes. Condor-COPASI: high-throughput computing for biochemical networks. *BMC Systems Biology*, 6(1):91, July 2012.

[Kur01] T. Kuroda. CMOS design challenges to power wall. In *Microprocesses and Nanotechnology Conference, 2001 International*, pages 6–7, October 2001.

[LHG$^+$06] David Luebke, Mark Harris, Naga Govindaraju, Aaron Lefohn, Mike Houston, John Owens, Mark Segal, Matthew Papakipos, and Ian Buck. GPGPU: General-purpose computation on graphics hardware. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[Lip11] Jana Lipková. *On Stochastic Modelling of Reaction-Diffusion Processes in Biology*. PhD thesis, 2011.

[LP10] Hong Li and Linda Petzold. Efficient parallelization of the stochastic simulation algorithm for chemically reacting systems on the graphics processing unit. *Int. J. High Perform. Comput. Appl.*, 24(2):107–116, May 2010.

[MA97] Harley H. McAdams and Adam Arkin. Stochastic mechanisms in gene expression. *Proceedings of the National Academy of Sciences*, 94(3):814–819, February 1997.

[Mac08] L. Macchiarulo. A massively parallel implementation of gillespie algorithm on FPGAs. In *30th Annual International Conference of the IEEE Engineering in Medicine and Biology Society, 2008. EMBS 2008*, pages 1343–1346, August 2008.

[MC13] Klaus Mainzer and Leon Chua. *Local Activity Principle: The Cause of Complexity and Symmetry Breaking*. Imperial College Pr, London : Singapore ; Hackensack, New Jersey, auflage: new. edition, 2013.

[Moo65] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[Mur02] James D. Murray. *Mathematical Biology*. 3 edition, 2002.

[NCB$^+$14] Marco S Nobile, Paolo Cazzaniga, Daniela Besozzi, Dario Pescini, and Giancarlo Mauri. cuTauLeaping: A GPU-powered tau-leaping stochastic simulator for massive parallel analyses of biological systems. *PloS one*, 9(3):e91963, 2014.

[Nvi12] Nvidia. CUDA programming model overview, 2012.

[Nvi14] Nvidia. CURAND LIBRARY programming guide. 2014.

[Owe04] John D. Owens. GPUs tapped for general computing. *EE Times*, 2004.

[Pea93] John E. Pearson. Complex patterns in a simple system. *Science*, 261(5118):189–192, July 1993. arXiv: patt-sol/9304003.

[RBK08] Diego Rossinelli, Basil Bayati, and Petros Koumoutsakos. Accelerated stochastic and hybrid methods for spatial simulations of reaction–diffusion systems. *Chemical Physics Letters*, 451(1–3):136–140, January 2008.

[RS07] Doug Rosenberg and Matt Stephens. *Use Case Driven Object Modeling with UMLTheory and Practice*. Apress, Berkeley, CA : New York, January 2007.

[SM10] Mutsuo Saito and Makoto Matsumoto. Variants of mersenne twister suitable for graphic processors. *arXiv:1005.4973 [cs]*, May 2010. arXiv: 1005.4973.

[SRTS06] Stefanie Sick, Stefan Reinker, Jens Timmer, and Thomas Schlake. WNT and DKK determine hair follicle spacing through a reaction-diffusion mechanism. *Science*, 314(5804):1447–1450, December 2006.

[TB05] Tianhai Tian and Kevin Burrage. Parallel implementation of stochastic simulation for large-scale cellular processes. In *Eighth International Conference on High-Performance Computing in Asia-Pacific Region, 2005. Proceedings*, pages 6 pp.–626, July 2005.

[Tec14] TechRadar. Moore's law: how long will it last?, February 2014.

[TTNW10] Koichi Takahashi, Sorin Tănase-Nicola, and Pieter Rein ten Wolde. Spatio-temporal correlations can drastically change the response of a MAPK pathway. *Proceedings of the National Academy of Sciences*, 107(6):2473–2478, February 2010.

[VLM11] Matthias Vigelius, Aidan Lane, and Bernd Meyer. Accelerating reaction–diffusion simulations with general-purpose graphics processing units. *Bioinformatics*, 27(2):288–290, January 2011.

[VPHS88] John A. Vastano, John E. Pearson, W. Horsthemke, and Harry L. Swinney. Turing patterns in an open reactor. *The Journal of Chemical Physics*, 88(10):6175–6181, May 1988.

[Wgs11] Wgsimon. Transistor counts for integrated circuits plotted against their dates of introduction., May 2011.