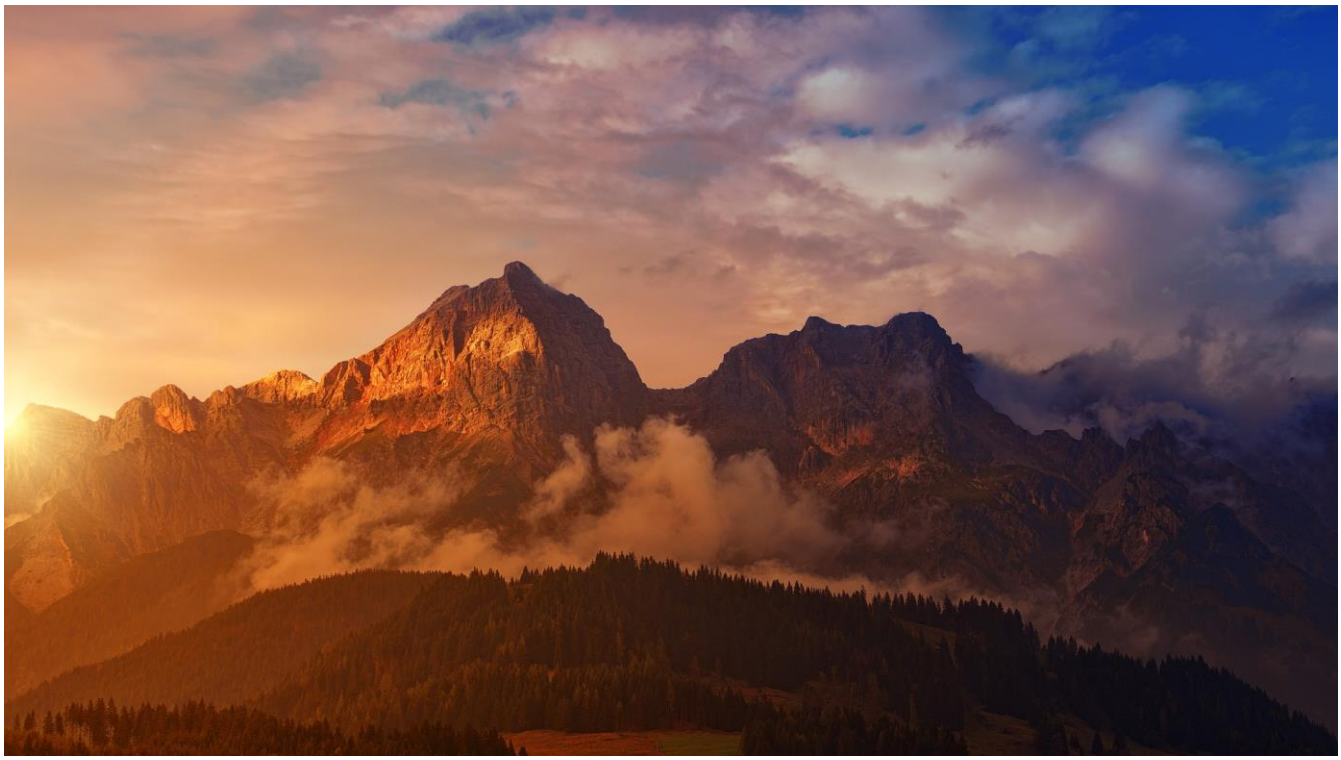
CHALMERS
UNIVERSITY OF TECHNOLOGY



Fast GPU simulations of the FRAP experiment

Master's thesis in Engineering Mathematics and Computational Science

LEANDER LACROIX

Department of Mathematics
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2018

Abstract

In this thesis we study several models of the Fluorescence Recovery After Photobleaching (FRAP) experiment. FRAP is a technique used to estimate the diffusion coefficient of fluids based on Confocal Laser Scanning Microscopy (CLSM). A fluorescent sample is first photobleached on a user defined region. Then, by studying the recovery of fluorescence in the bleaching region, one can retrieve important parameters of the fluid such as the diffusion coefficient and binding constants by fitting a model to the data. We implemented and compared three models of the FRAP experiment. The first model assumes bleaching and image acquisition is an instantaneous process. The second model, based on the first one, introduces multiple bleach frames. The final model takes into account the scanning movement of the CLSM and is computationally much more complex. For the instantaneous models, two schemes are introduced and compared against each other to ensure correct implementation of the algorithms. The first scheme uses the spectral method to solve the diffusion-reaction equations and the second uses a stochastic formulation of the problem. The last model, due to its complexity, has only been implemented stochasticaly. All three models have been implemented on Graphical Processing Units (GPUs) using the OpenCL API in C++. The GPU has a massively parallel architecture that can be exploited for scientific computing. These schemes are "embarrassingly parallel" and thus suitable for a GPU implementation.

By comparing the different models, we see that a good compromise between precision and computing resource is given by the instantaneous bleaching with multiple bleach frames model. Because of the scanning nature of the CLSM, we would expect the last model to reveal some asymmetry in the results. These were only found for extreme and unrealistic parameters and it is thus not necessary to simulate the FRAP experiment with such complexity.

Index terms— FRAP, Spectral method, GPGPU, OpenCL

Contents

Introduction

Fluorescence Recovery After Photobleaching (FRAP) is a method used in order to experimentally estimate the kinetics of the diffusion coefficient of fluorescent particles in liquid. In a confocal laser scanning microscope, we first bleach in a well defined circular or rectangular spot, using a laser, the sample and then save snapshots of it at regular intervals. Two classes of methods have been developed in order to compute diffusion parameters from data. The first simplest one is to fit a model on the recovery curve, which is the evolution of the mean concentration of the sample in the bleaching area. This class of method can be enough for complex fluids (such as in cells) where the signal over noise ratio is poor and is also the standard method used in different scientific fields. A higher accuracy can be achieved by using pixel based methods. Instead of fitting a model on a reduced data set (the recovery curve), we fit a more accurate model over the full image data. Pixel-based methods are more computationally demanding and as such fitting over a large data set can take some time. The currently implemented pixel based model makes a lot of simplifying assumptions that can potentially be detrimental for the accuracy of the fitting.

The evolution of graphic cards designed for the gaming market boomed the past decade, with more computing power for smaller cost at each product iteration. Compared to Central Processing Unit (CPU) which are general purpose and the heart of every computers, the architecture of Graphics Processing Unit (GPU), optimised for computer graphics, is highly parallel and very efficient power consumption wise. Their potential for general purpose and scientific computations became apparent during the mid-2000s. In 2007, Nvidia introduced CUDA, the first framework designed for the GPGPU (General Purpose GPU) computing model, only working on their graphic cards. In 2008, the Khronos group introduced OpenCL, an open standard for heterogeneous computing, which can run on GPUs, manycore CPUs and even whole supercomputers.

The goal of this master thesis is to develop fast parallel FRAP simulations on GPUs. Evolution of the concentration of the fluorescent fluid can be described by a system of PDE and, in order to validate the method, we compare the results with the solution given by a stochastic simulation also implemented on GPU. We then delve into more complex models of the FRAP experiment by taking into account more physical constraints of the laser of the microscope, such as the non-instantaneous nature of the bleaching process or the scanning movement of bleaching and reading which can skew the results. These models, computationally much more intensive, can potentially only be comfortably explored using GPUs. The aim of implementing and exploring these different models is to investigate which features of the experience are important to model.

Chapter 1

CLSM and the FRAP experiment

1.1 Fluorescence microscopy and Confocal Laser Scanning Microscopy

Fluorescence microscopy is an optical microscopy technique that uses fluorescence and phosphorescence of the sample as light source for the study of organic or inorganic substances [9]. The sample is excited at a specific wavelength which is absorbed by the fluorophores, bound to the sample. In order for the fluorophores to return to its ground state, it has to emit light corresponding to the energy lost, which is done spontaneously at a known rate. This emission spectrum is then captured by a sensor (such as a CCD camera) forming the final image. One can also use autofluorescence if the studied sample has this property, meaning that fluorophores are no more needed.

The very high selectivity of fluorescence microscopy makes it a standard tool in the biology field. Indeed, non-fluorescent objects are ignored during the measurement making the signal/noise ratio advantageous. Unfortunately, this technique is not without drawbacks. Sample preparation is more complicated when the addition of fluorophores is needed. Also, its large unfocused background can be problematic when doing measures on large 3D objects, as large unfocused area of the sample can appear.

Confocal Laser Scanning Microscopy is built on top of the idea of fluorescence microscopy [13]. The basic idea is to add a pinhole to the light path in order to reject the unfocused images. A much better focus plan selectivity means that it also become possible to sample at different depth. Using a pinhole to filter out unfocused part of the image also means that only a small part of the sample is measured at a time and therefore it is no longer possible to use a camera as a detector. Instead, a photomultiplier tube (PMT) is usually used. The resulting image is discretized into pixels and the laser sweeps the field of view thereby forming an image. The sweeping is usually performed by slightly moving or rotating one or several mirrors and lenses in the light path. For each pixel, the PMT measures the light and is saved on the final render target. This sweeping can have an impact on the reading, as pixels are measured at different time. Objects with an important dynamics at this time-scale can thus appear distorted.

1.2 Fluorescence After Photobleaching (FRAP)

One way to experimentally measure the diffusion coefficient of a sample is to use the Fluorescence after Photobleaching (FRAP) method ([10] for an up to date review, the papers [1], [8] pioneered the technique). Using a confocal microscope, we first photobleach a spot of the sample, reducing locally the concentration of bounded particles. The non bleached area, having a higher concentration of the sample, will then diffuse back into the previously bleached area, in order to find an equilibrium. This diffusion process is recorded in real time, and the set of images taken is then fitted to a model in order to estimate the diffusion coefficient.

There exist different kinds of models and fitting techniques. The simplest method fits a model of the recovery curve, which is the evolution of the mean concentration in the bleached region in time, with the experimental recovery curve. The recovery curve can be modelled by a function (usualy based on exponential or bessel functions) or by computing it from a full simulation of the experiment. This class of method can be enough for complex fluids e.g. in cells, where the signal over noise ratio is poor. This is the standard method used in different scientific fields such as cell biology or food science which deals with complex colloids.

More recently, attempts are made to fit whole simulations to the data. As more information is available, a higher accuracy can be achieved by these so called pixel based methods [5], [6]. Instead of fitting a model on a reduced data set (the recovery curve), we fit a more accurate model over the full image data. It is still unclear of how much more information is used for fitting or of the gained accuracy. Pixel based methods can be more computationally demanding and as such fitting over a large data set can take some time. Indeed, we now need to generate a lot of models, which are full simulations of the whole experiment, in order to estimate residuals and minimise them. However, for complex models of the recovery curve, such as the one based on Bessel functions, computation time can also be substantial.

These methods also permit to quantitatively estimate the binding and unbinding rate constants characterising the interaction between the sample and the fluorophores [7], [15].

1.2.1 Experimental data

In order to match the experimental data (figure 1.1), the simulation developed throughout the work should output the concentration $c(x, y, t)$ of the studied sample in n snapshots on a $N \times N$ discrete grid. Simulation occurs in two phases: first photobleaching and then free diffusion. Snapshots are taken after every timestep Δt during diffusion, each of them representing a frame given by the CLSM running the FRAP experiment.

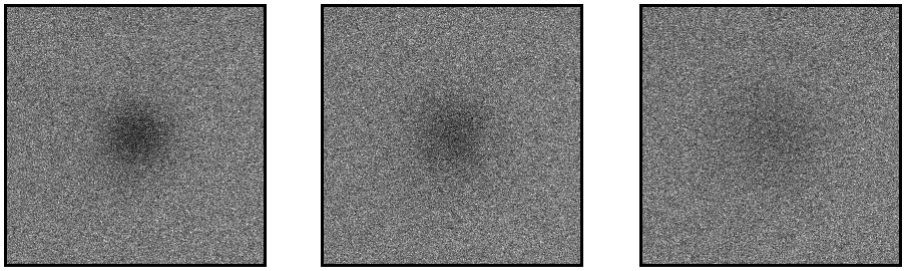


Figure 1.1: Experimental FRAP data, with $\Delta t = 0.5\,\mathrm{s}$ (from [14]). Each pixel is $0.732\,\mu\mathrm{m}$ with a disk shaped bleaching region of radius $15\,\mu\mathrm{m}$.

Chapter 2

The OpenCL framwork

The evolution of graphic card designed for the gaming market boomed the past decade, with more computing power for smaller cost at each product iteration. The architecture of Graphics Processing Unit (GPU), optimised for computer graphics, is highly parallel and very efficient power consumption wise. Their potential for general purpose and scientific computations became apparent during the mid-2000s. In 2007, Nvidia introduced CUDA, the first framework designed for the GPGPU (Global Purpose GPU) computing model, only working on their graphic cards. In 2008, the Khronos group introduced OpenCL, an open standard for heterogeneous computing, which can run on GPUs, manycore CPUs and even whole supercomputers.

2.1 The heterogeneous programming model

OpenCL is a standard framework designed for general purpose and scientific computing on a variety of devices [3]. These devices include CPU, GPU, FPGA and more generally accelerator cards. It is possible to use specific functions of a given device by the use of extensions, e.g. a specific video encoder hard wired circuit on an accelerator device. This library is designed to profit from the highly parallel architecture of these devices by the use of a specific memory and computing architecture we will describe below. Kernels, which are the specific functions executed by the devices, are written in a superset of C99, adapted for computation usages. In the current version of OpenCL (2.2), it is possible to write kernels in a superset of C++ 14. However, this version of the standard is not well supported at this time. This work we will use OpenCL 1.2 which is the most supported version of OpenCL, were kernels are written in a superset of C99, adapted for computation and parallel computations.

The OpenCL standard is maintained by the Khronos group, a consortium of company and academic members founded to promote open standards. Since OpenCL is a standard, this means that the implementation of the framework is vendor dependent. Being an open and general purpose standard, it is possible to run the same code for different devices for different vendors and different operating systems. In our work, we will write code for two NVidia Titan Xp graphic cards and two Intel Xeon CPUs on a computer running Linux Ubuntu 16.04.

Given the heterogeneous nature of OpenCL, a fairly high level memory and computing architecture has to be introduced [16]. The workload consist of functions called *kernels* that are executed either on a line, a grid or a cube, depending on the way we want to

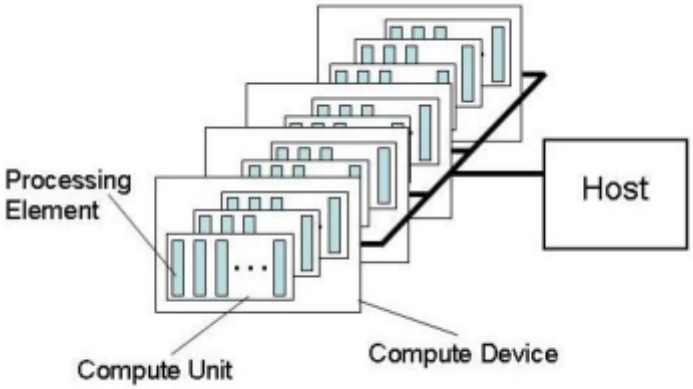


Figure 2.1: Host and devices in OpenCL (from [3])

solve the problem.

OpenCL makes the distinction between the *host* and the *device*. The host is the entity that will send commands to the device which represent the accelerator card that will execute the commands. A device is divided into *Compute Units* which are independent processing components that will execute the mapped kernel (figure 2.1). What a Compute Unit exactly is depends on the targeted device and will be described in more detail in the section below, but can be seen as something that execute kernels. A Compute Unit can usually process the workload further in parallel in Processing Element lanes. One instance of a kernel execution is called a *workitem*. These are executed in parallel in groups called *workgroup* which can have shared memory and communication abilities (figure 2.2). A workgroup is executed by one or several Compute Unit. Synchronisation between workitems can only take place inside the same workgroup, i.e. we cannot synchronise two different workgroups together as there is no guarantee that workgroups will be executed at the same time.

2.2 Memory organisation in the OpenCL programming model

In the OpenCL model, device memory is divided into 5 types with different usages and physical localisation (figure 2.3). Exact behaviour and localisation depends on the device and on its OpenCL implementation.

- Global memory - High latency, large capacity, memory used to communicate data between the host and the device. For CPUs this corresponds to the RAM, and for GPUs to the VRAM.
- Local memory - Memory shared between workers into the same work group, usually implemented into the cache of the Compute Unit. Very fast but of low capacity.
- Private memory - Memory only accessible to a worker (like local variables or arrays on the stack), usually mapped on the registers of the thread (but may be mapped on global memory if the compiler run out of addressable registers).
- Constant memory - Constant memory accessible by each workitem, usually mapped on the global memory but with caching, enabling fast reading.

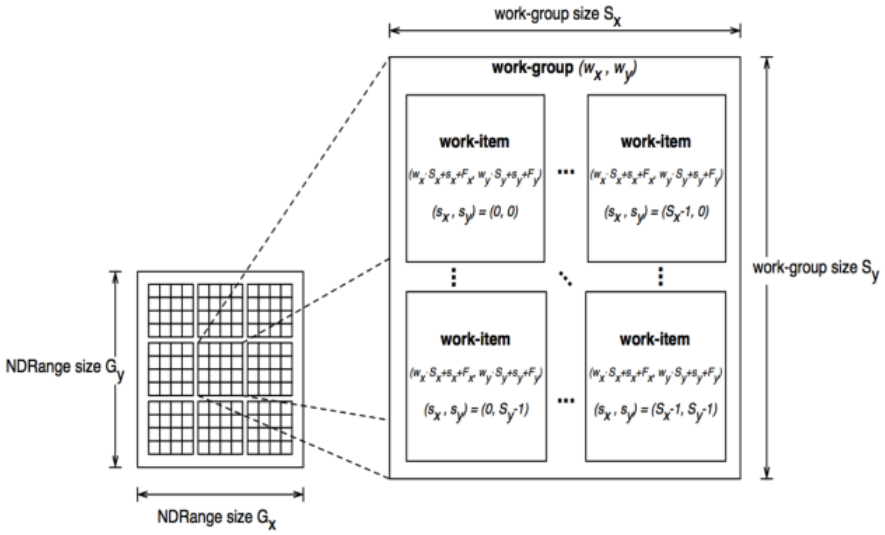


Figure 2.2: Distribution of the workload in the OpenCL execution model (from [3])

- Texture memory - 2D memory usually used to represent images (an heritage from GPUs who are specialised into computer graphics), with sampling capability and caching.

We see that OpenCL offers a great deal of memory spaces and thus many device-dependent optimisation opportunities.

2.3 Architecture of the targeted devices

Simulation developed throughout the work will run on the CPU and on the GPU. The devices differ greatly in architecture, the CPU being the general purpose processor of a computer and the GPU a processor specialised into 3D graphic display and is highly parallel. As the theoretical computing speed of one Titan Xp GPU (12 TFlops) is much greater than for the CPUs (around 800 GFlops for both units), we focus our optimisation effort for the GPU. First naive implementations of the algorithms would not give a clear advantage to the GPU, as they relied a lot on global memory reading (fast and cached on CPU, slow and uncached on GPU) and branching (for which CPUs excel).

We are using three different implementations of OpenCL. The first two implementations target the CPU. We first use the Intel implementation of OpenCL [17], which exploit special features of the Intel CPUs, and then POCL (Portable OpenCL) which tries to be as portable as possible between all supported devices, including performance portability [4]. For the GPU, we use the NVidia implementation that come with the graphics driver [12], [11]. Since compilation is managed by the implementation, compilation error catching and messages are different and often complementary.

Nvidia GPUs are SIMT - Single Instruction Multiple Thread. Each Compute Unit (*warps* in Nvidia lingo) consists of 32 scalar threads running in parallel. The Nvidia Titan Xp has 30 Compute Unit giving a total of 960 threads. Every instruction is executed in parallel, meaning that if a branching occur in one thread, then the other threads in the

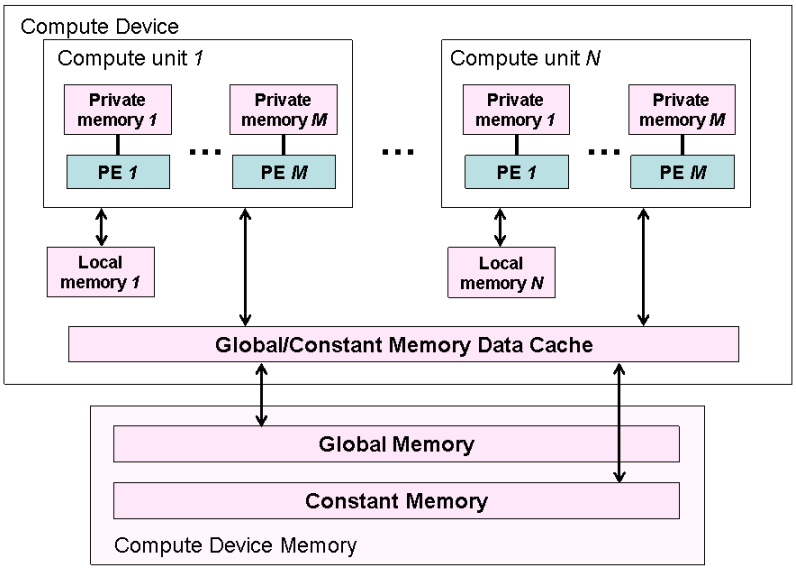


Figure 2.3: The OpenCL memory model (from [3])

same *warp* are halted and only resumed once the branch converge back to the original instruction flow. This behaviour results in bad performance for algorithms that depends a lot on branching, such as rejection sampling. The compiler, depending on the implementation, can optimise these branches or remove them if they are simple enough. Still, it is best to manually optimise simple branches using ternary conditions (which do not result in branching) or `select` instructions which are a vectorised equivalent of ternary conditions. If some global data needs to be written or read a lot, it may be best to first cache the corresponding memory region into local memory. Threads are specialised in computation and have efficient hardwired mathematical functions. Launching and scheduling workitems on the GPU is very fast, meaning that it is usually best to divide the problem into much more workitems than threads e.g. in this work, we launch around 1 millions workitems for one of the simulations. It is also important to divide workgroup with a multiple of 32 workitems so we use the full computation capabilities of every Compute Unit. Letting the Nvidia implementation automatically choose these parameters usually gives good results.

The Intel CPUs (and modern CPUs in general) have a low count but very fast cores, where each core form a Compute Unit. These cores excel in branch predictions and memory reading with automatic and efficient memory caching. Scheduling workitems is usually slower than on GPUs as threads and context switching are managed by the OS. Being general purpose, they are not as specialised in mathematical computations as for GPUs. Modern CPUs have SIMD instructions (Single Instruction Multiple Data) which can boost floating point computations by doing them in parallel on one core. Depending on the implementation of OpenCL, SIMD capabilities can be automatically exploited without prior vectorisation of the code. This is true for the Intel OpenCL implementation but not with the current POCL implementation. Intel CPUs such as the one used for this project have the Hyper Threading technology enabled, doubling the number of threads a core can manage in parallel. Since there is 2 Xeon E5-2699 V4 each having 22 cores, we have total of 88 threads. Hyper Threading is advantageous in situations with mixed kind of

instructions or asynchronous operations, such as in databases or for virtualisation, but is inefficient in a heavy computation context such as in this project. From a test made on our code, enabling Hyper Threading only increase the speed by 5-10% when theoriticaly we would naively expect a 100% speed increase. The implementations on CPUs are much less forgiving when writing on unallocated memory area than on GPUs. On a CPU, memory is allocated in pages of 4 KB or 2 MB on RAM [1] and is virtualised so these blocks seem contiguous from the application's point of view. Writing outside these allocated blocks will result in a segmentation fault. On GPU, as memory is not virtualised and not managed by the host OS, writing to unallocated memory is permitted as long as it correspond to a a physical address. This rises unpredictable behaviour and generally breaks the algorithm, usually by giving erroneous results or a segmentation fault. This important distinction has been exploited for debugging purpose.

[1] This depend on the CPU architecture. Values given above are for X86-64 architectures such as the Xeon used for this work.

Chapter 3

Model of the FRAP experiment

In a FRAP experiment, a fluorescent species is irreversibly photobleached in the bleaching region. Unbleached particles outside of the bleaching region will then move towards the bleached area and the particles from the bleached area will move towards the unbleached area. This convergence to an equilibrium leads to a recovery of fluorescence with time evolution, although, technically, the total amount of fluorescence in the whole system is reduced and as such full recovery is not possible (but this is negligible for the studied systems as they are large enough). Throughout the work, we assume that the bleaching region is sufficiently extended in the axial dimension and thus the diffusion in the z direction can be neglected.

3.1 Diffusion

In order to validate the simulations, we develop and solve, for each model, two formulation of the problem. The first formulation, deterministic, solves a PDE system. The second formulation solves the problem by a stochastic simulation.

3.1.1 System of PDEs

We will consider two cases of equal interest. We either suppose the fluorescent species follows a pure diffusion process or that it follows a reaction-diffusion process.

For pure diffusion with diffusion coefficient D, the evolution of the concentration of the fluorescent species is given by the diffusion PDE without source:

$$\frac{\partial c}{\partial t} = D\nabla^2 c. \tag{3.1}$$

In order to formulate the reaction-diffusion process into a PDE system, we complement the diffusion with an interaction between unbound particles and binding sites forming together a bound complex, characterised by the following chemical reaction equation:

$$U + S \underset{k_{\text{off}}}{\overset{k_{\text{on}}}{\rightleftharpoons}} B,$$

where k_{off} is the off-binding rate and k_{on} is the on-binding rate. From this reaction, we

get a system of coupled reaction-diffusion PDE, where $u = [\mathrm{U}]$, $s = [\mathrm{S}]$ and $b = [B]$:

$$\begin{cases} \frac{\partial u}{\partial t} = D_u \nabla^2 u - k_{\mathrm{on}} us + k_{\mathrm{off}} b \\ \frac{\partial s}{\partial t} = D_s \nabla^2 u - k_{\mathrm{on}} us + k_{\mathrm{off}} b \\ \frac{\partial b}{\partial t} = D_b \nabla^2 u + k_{\mathrm{on}} us - k_{\mathrm{off}} b \end{cases}$$

With ∇^2 the Laplacian operator and D the diffusion coefficient of each species.

We can simplify the system by using two assumption that are reasonable in the context used. We first suppose that before bleaching the system is at equilibrium, that is $u = u_{\mathrm{eq}}$, $s = s_{\mathrm{eq}}$ and $b = b_{\mathrm{eq}}$. Bleaching changes the local concentration of bound and unbound particles but does not change the concentration of free binding sites. Therefore, during the recovery, $s = s_{\mathrm{eq}}$ and thus $\partial_t s = 0$. We also set $k^*_{\mathrm{on}} = s_{\mathrm{eq}} k_{\mathrm{on}}$ which we call the pseudo-first-order rate constant. As a second simplification, we suppose that the diffusion of the bound complex particles is negligible, giving us $D_b = 0$. We then have the following system:

$$\begin{cases} \frac{\partial u}{\partial t} = D_u \nabla^2 u - k^*_{\mathrm{on}} u + k_{\mathrm{off}} b \\ \frac{\partial b}{\partial t} = k^*_{\mathrm{on}} u - k_{\mathrm{off}} b \end{cases}$$

We also set $c = u + b$ the total concentration of particles. We simplify the notation by setting $D_u = D$ and $k^*_{\mathrm{on}} = k_{\mathrm{on}}$, yielding:

$$\begin{cases} \frac{\partial u}{\partial t} = D \nabla^2 u - k_{\mathrm{on}} u + k_{\mathrm{off}} b \\ \frac{\partial b}{\partial t} = k_{\mathrm{on}} u - k_{\mathrm{off}} b \end{cases} \tag{3.2}$$

This is a linear parabolic system of PDEs which can be solved numerically. In this work we use the spectral method.

3.1.2 Stochastic diffusion and dynamics

In parallel with the numerical PDE solving, we simulate the models using a stochastic approach which is used to validate the deterministic implementations of the simulations.

In the case of pure diffusion, a particle is only defined by its position. At each timestep Δt, the particle is displaced in each direction by a normal distribution with variance $2D\Delta t$.

We simulate the diffusion-reaction process using a two state hidden Markov model. Each particle now has a state, either bound or unbound. Bound particles are freezed in their position and unbound ones diffuse as in pure diffusion. The transition probabilities between the states are given by:

$$p_{\mathrm{u}\to\mathrm{b}} = \Delta t / \mu_{\mathrm{u}}$$

$$p_{\mathrm{b}\to\mathrm{u}} = \Delta t / \mu_{\mathrm{b}}$$

The average time of a fluorophore being unbound and bound are given by:

$$\mu_u = 1/k_{\mathrm{on}}$$

$$\mu_b = 1/k_{\mathrm{off}}.$$

It follows that the residence time distributions are geometric with mean μ_u and μ_b. In this work, all simulations are done with $k_{\mathrm{on}} = k_{\mathrm{off}} = 1$ 1/s.

The equilibrium concentrations of unbound and bound species are given by $u = \pi_u c$ and $b = \pi_b c$, where π_u and π_b are respectively the unbound and bound proportion of species:

$$\pi_u = \frac{k_{\text{off}}}{k_{\text{on}} + k_{\text{off}}}$$

$$\pi_b = \frac{k_{\text{on}}}{k_{\text{on}} + k_{\text{off}}}$$

3.2 Bleaching

Throughout this work we explore several models for the bleaching. We refer to instantaneous bleaching when the sample is bleached at once in time. A first simple bleaching model is to set, with α the bleaching coefficient:

$$c(x, y, 0) = \begin{cases} \alpha c_0, & (x, y) \in \Omega \\ c_0, & (x, y) \notin \Omega \end{cases} \tag{3.3}$$

With Ω a binary mask representing the bleaching region. In order to reduce the sharpness of the edges, and thus reduce numerical difficulties, we first generate an upsampled version of the initial conditions by some factor (3 for samples generated in this work) and then downsample it back to the simulation size by using bilinear downsampling.

In this work, we will explore models implementing bleaching such as in equation 3.3 then with multiple instantaneous bleach frames and finally with bleaching occurring in time by sweeping through the sample as a way to simulate the laser scanning motion of the CLSM.

Chapter 4

Single instantaneous bleach frames

We first assume that bleaching occur in one frame and that the experiment only start once bleaching is done.

4.1 Implementation of the deterministic model

This simulation numerically solves the pure diffusion or reaction-diffusion equations using the spectral decomposition method just after bleaching.

Simulation is performed on a $[0, N+2M] \times [0, N+2M]$ periodic, discrete domain, with N the final image size and M the padding in order to minimise border effects. A frame is computed every Δt. The concentration c of the fluorescent specie is thus computed for every pixel and every time step of the domain and given by $c(x, y, t)$.

4.1.1 Generation of the initial conditions

Initial conditions $c(x, y, 0)$ given by equation 3.3 are generated in two steps. We first generate an upsampled version of the initial conditions and then downsample it in order to reduce aliasing effects for non rectangular, sharp initial conditions such as disks.

We first implemented initial conditions generation on CPU but quickly found out that it was a major bottleneck in the simulation, especially the downsampling phase of the algorithm. We thus decided to move the code to the GPU. We first call a kernel to generate the initial upsampled version of the initial conditions. The algorithm is implemented naively: the work is distributed on a 2D grid where each worker represents a pixel and test it against the geometry of choice. We then call a kernel mapped on a 2D grid with size of the final image dimension and perform block downsampling, i.e. the downsampled pixel value is equal to the mean values of the pixel of a square block of size equal to the zoom factor. The generated initial conditions can also work as a mask and will be used as such for subsequent implementations of the FRAP experiment model. We implemented masks for disk, rectangle and Gaussian distribution.

4.1.2 Solving the pure diffusion equation using the spectral method

Our goal is to solve equation (3.1) using the spectral method. We first transform the equation into the spectral domain:

$$\frac{\partial \hat{c}}{\partial t} = -(\xi^2 + \eta^2) D \hat{c}(t)$$

The numerical transformation from pixel space (x, y) to frequency space (ξ, η) is done using the discrete Fourier transform on a discrete grid of same size.

This gives $(N + 2M)^2$ simple independent ODEs that can be solved analytically. We get:

$$\hat{c}(\xi, \eta, t) = \hat{c}(0)e^{-(\xi^2+\eta^2)Dt}$$

The concentration $\hat{c}$ is then computed at every pixel value in frequency space (corresponding to values of ξ and η) for every timestep. This problem is "embarrassingly parallel" and suitable for an OpenCL implementation. We then transform the result back into the spatial domain. For the forward and backward numerical Fourier transform, we use the clFFT library [2] which implement the Fast Fourier Transform (FFT) algorithm using OpenCL. FFT computation is done in-place, directly into the device memory, without the need to allocate a new memory block. We also found out during development that the clFFT library is not working as it should with the Intel OpenCL implementation and we could not find a way to solve it (some warnings are spewed during compilation and results are wrong).

4.1.3 Solving the reaction-diffusion equations using the spectral method

Our goal is now to solve equation (3.2) using the spectral method. We first transform the equation into the spectral domain:

$$\begin{cases} \frac{\partial \hat{u}}{\partial t} = -[(\xi^2 + \eta^2)D - k_{\text{on}}]\hat{u} + k_{\text{off}}\hat{b} \\ \frac{\partial \hat{b}}{\partial t} = k_{\text{on}}\hat{u} - k_{\text{off}}\hat{b} \end{cases}$$

Again, we get $(N + 2M)^2$ ODEs, one for each (ξ, η) in frequency space. Which can be rewritten into matrix form as:

$$\frac{\partial}{\partial t}\begin{pmatrix} \hat{u} \\ \hat{b} \end{pmatrix} = A\begin{pmatrix} \hat{u} \\ \hat{b} \end{pmatrix}$$

With,

$$A = \begin{pmatrix} -(\xi^2 + \eta^2)D - k_{\text{on}} & k_{\text{off}} \\ k_{\text{on}} & -k_{\text{off}} \end{pmatrix}$$

This is an ODE with solution:

$$\begin{pmatrix} \hat{u} \\ \hat{b} \end{pmatrix} = e^{At}\begin{pmatrix} \hat{u}_0 \\ \hat{b}_0 \end{pmatrix}$$

The matrix exponential can be computed analytically using the the eigendecomposition $A = Q\Lambda Q^{-1}$, yielding:

$$e^{At} = Q\begin{pmatrix} e^{\Lambda(1,1)t} & 0 \\ 0 & e^{\Lambda(2,2)t} \end{pmatrix}Q^{-1}$$

This means that to compute the solution, we first need to compute the eigenvalues, eigenvectors (without a need to normalise them) and compute a matrix inverse. Then, for every timestep and every pixel in frequency space, we compute the diagonal matrix and two matrix multiplications. These operations can easily be implemented on the GPU and, as each computation does not depend on results from other computations, the resolution of the system of ODE is trivially parallel as for the pure diffusion case. Once timestepping in the spectral domain is done, we compute the inverse Fourier transform to get the results back into the spatial domain.

Linear algebra computations are done using a small linear algebra library developed beforehand.

4.1.4 Discontinuity of the initial conditions

How to cope with discontinuities in numerical solutions of PDEs is still an open problem and an important area of research. Because the initial conditions are discontinuous, we expected some problems with the spectral method. Indeed, Gibbs phenomena, which takes the form of rings/waves around discontinuities, should manifest itself because of the finite aspect of the discrete Fourier transform. In actual results, using real life parameters, no such observation can be made. Finally, oscillation could be found only with extreme parameters i.e. very low resolution and diffusion coefficient. This behaviour can be explained by the diffusive nature of the system: oscillation is corrected by diffusion, which can be seen as a low pass filter removing the high frequency features of the signal.

Before realising Gibbs phenomena would not be problematic, we experimented with other numerical schemes in order to solve the system. We explored finite difference methods and implemented a first order in time, second order in space, explicit solver (Euler's method). The main argument for this choice is that this method avoid transforming back and forth to the spectral domain, avoiding costly FFT operations when simulating the bleaching. We found out that this method introduced non negligible numerical diffusion and performed even worse with discontinuities. A solution would be to increase either globally or locally the grid resolution, which we want to avoid at all cost as it would increase computational cost. Before experimenting with implicit methods (Crank-Nicolson scheme), or higher order in space methods (which would also introduce oscillations), we finally decided to stick with the spectral method which offer excellent global error properties when compared to finite differences, even in the presence of discontinuities.

4.2 Implementation of the stochastic model

We also implement a stochastic solution to the model. Inherently slower, it's only purpose is to validate the solution generated by the spectral method. We first need to generate a large amount of particles that we then simulate. Here, the problem is 1D as particles are stored into an array. Each worker manages a fixed amount of particles. For 1 billion particles, around 9 GB of device memory is needed: 16384 workers are launched where each one of them manages 65536 particles.

4.2.1 Initial conditions and rejection sampling

We first need to generate initial conditions for the simulation, i.e. the initial position and state of the particles. For this, we implement rejection sampling (algorithm 1). We first draw a random position over the frame from a uniform distribution and then compare an uniform distribution draw with the value given by the geometry mask at the same position. If the position is not accepted i.e. the random value is higher than the mask value, we repeat the whole operation. For reaction-diffusion process simulation, state of the particle is determined from the equilibrium concentration of unbound and bound species. Because of the stochastic nature of the loop, rejection sampling is not very efficient on GPU architecture as lots of path divergences can occur i.e. when we reject a particle and need to sample a new one. However, this is compensated by the raw speed of massively parallel devices. Speedup using OpenCL is marginal compared to a classic CPU approach, but it still proves faster than doing memory transfer between the host and the device. The

sampling phase takes around 1 second (equally for CPU and GPU) for 1 billion particles so not much optimisation is necessary as it is already fast enough.

input : bleach mask
output: initialised particles

for $i \leftarrow 0$ **to** particlePerWorker **do**
  acceptedSample $\leftarrow$ **false**
  while not acceptedSample **do**
    $(x, y) \leftarrow$ N `UniformSample(2)`
    maskValue $\leftarrow$ mask [`Truncate(`x`)`, `Truncate(`y`)`]
    if `UniformSample(1)` $<=$ maskValue **then**
      acceptedSample $\leftarrow$ **true**
      particlePos $[i] \leftarrow (x, y)$
      if `UniformSample(1)` $<= \pi_{off}$ **then**
        particleState $[i] \leftarrow$ StateBounded
      end
      else
        particleState $[i] \leftarrow$ StateUnbounded
      end
    end
  end
end

Algorithm 1: Rejection sampling based algorithm to initialise the particles used in the stochastic simulation

Position are stored into 32 bit 2D float vectors and the binding state into a 8 bit integer, which means that it can store 8 different on-off states. This will be used later on once we start bleaching particles in the simulation and need to tag bleached particles. This means that in order to simulate the targeted 1 billion particles, the device needs at least 9 GB of memory.

4.2.2 Simulation of the particles

Simulation are performed on a $[0, N+2M] \times [0, N+2M]$ periodic simulation domain, with N the final image size and M the padding in order to minimise border effects, as for the spectral method.

Since the average time of a particle being unbound or bound can be smaller than the timesteps between two frames, we need to subdivide the time between frames. In most of our work, we use $\Delta t_{\text{sim}} = \Delta t/32$.

At each time step subdivisions, we first update the state of each particles and then displace the unbound ones in each direction by a normal distribution with variance $2D\Delta t_{\text{sim}}$ (algorithm 2).

4.2.3 Binning and generation of the frames

We want to generate images out of the particles. For this, we count the numbers of particles that lie on each pixels at each frame. This is called binning and is analogous to construction a histogram. We could not find a suitable library that implements a 2D binning algorithm

```
for i ← 0 to particlePerWorker do
    if particleState [i] = StateBounded then
        if UniformSample(1) ≤ π_off then
            particleState [i] ← StateUnbounded
        end
    end
    else
        if UniformSample(1) ≤ π_on then
            particleState [i] ← StateBounded
        end
    end
    if particleState [i] = StateUnbounded then
        particlePos [i] ← particlePos [i] + NormalSample(2, 0, 2DΔt_sim)
    end
end
```

Algorithm 2: Stochastic simulation of the reaction-diffusion process. Actual implementation of the algorithm also ensures that particles stay inbound by a series of conditional instructions.

so we decided to implement our own. We first implemented a naive algorithm mapped on a 2D grid corresponding to the output frame, were each worker loops on every particles and count the number of particles that lie on the corresponding pixel. Being $O(n^3)$, this brute force algorithm is very inefficient, as every worker has to loop on every particles, and so another method was quickly developed. On the second version, the problem in mapped on a 1D array in the same way as before: each worker manages a fixed amount of particles. Here, each worker loops over its particles and computes by truncation the position in pixel space. We then increment the corresponding pixel on the output frame, effectively giving a $O(n)$ algorithm (algorithm 3). The incrementations occur concurrently, atomic operations are thus needed to make sure each memory location is written correctly e.g. if two threads increment at the same time without using atomics, one of them would be ignored. The second algorithm has a speedup of 3 to 4 order of magnitudes and thus, binning which now takes approximately 1 second for 1 billion particles, is no longer a bottleneck.

```
for i ← 0 to N do
    (x, y) ← Truncate(particlePos [i])
    AtomicIncrement(binOut, x, y)
end
```

Algorithm 3: Efficient 2D binning algorithm

4.3 Results and validation

By comparing both simulations with the same parameters (figure 4.1, 4.2) we directly see, ignoring the noise of the stochastic simulation, strong similarities.

A more rigorous comparison would be to analyse the residuals between both simulations. We see that there is no structure in the residuals and that it follows a normal

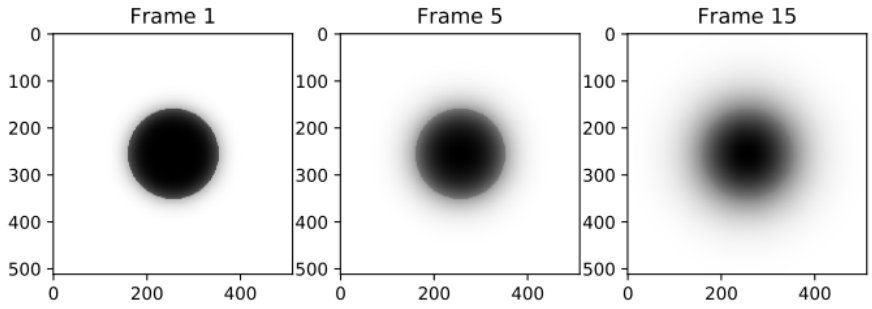


Figure 4.1: Frames for the deterministic single bleach frame simulation, with $D = 800$ pixel/s^2, $k_{\text{on}} = k_{\text{off}} = 1$ and $\Delta t = 0.265$ s.

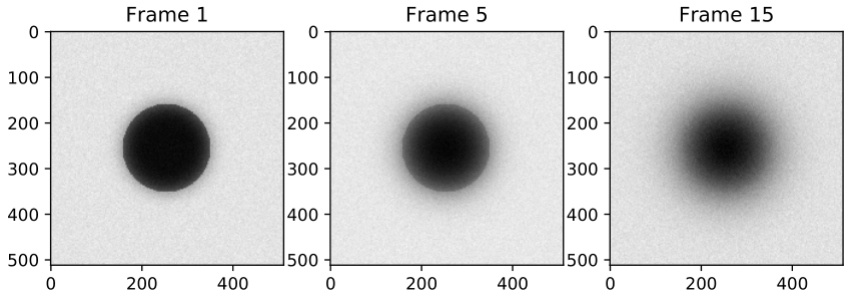


Figure 4.2: Frames for the stochastic single bleach frame simulation, with $D = 800$ pixel/s^2, $k_{\text{on}} = k_{\text{off}} = 1$, $\Delta t = 0.265$ s and 1 billion particles.

distribution, indicating a good fit between both simulations, thus validating both implementations.

Chapter 5

Multiple instantaneous bleach frames

The model developed in the last section describes reality acceptably and works well for the fitting of experimental data. However, the developed framework permit us to better model the FRAP experiment. In reality, multiple bleach frames may be necessary to obtain a satisfactory amount of bleaching e.g. for real experiment, four bleach frames are usually taken. The new presented model will take into account this fact. Bleaching is still assumed to be instantaneous but particles diffuse between bleach frames. This has an important impact on the state of the simulation just after the bleaching as diffusion already started.

5.1 Implementation of the deterministic model

We store the state of the simulation into two field representing the concentration of bounded and unbounded particles, $u(x, y, t)$ and $b(x, y, t)$. We start into space domain and homogeneously fill both fields with the value given by π_u and π_b. Bleaching is done in space domain and is the simple term multiplication between the fields and a mask representing the bleaching profile used for the simulation, with α the bleach multiplier. Once bleached, we then perform diffusion using the spectral method in the same way as for the single bleach frame model.

The bleaching process takes a significantly longer time compared to the simpler model implemented before, where only one mask generation was needed. Here, for each bleach frames, we perform two forward and backward FFT and one term multiplication.

5.1.1 Importance of separating the bounded and unbounded concentration

The first implementation only stored the field of total concentration $c(x, y, t)$ instead of separating them into $u(x, y, t)$ and $b(x, y, t)$ and simulation was done by assuming π_u and π_b is constant throughout time and space. This is a fundamental flaw and break the simulation.

5.2 Implementation of the stochastic model

We can't use rejection sampling for the initial conditions as bleaching is a process taking place in time. Particles get a new state: activated or not. Bleaching a particle then means that we deactivate it. Deactivated particles are ignored during the binning phase but are still simulated: the SIMT nature of GPUs means that using a condition instruction to distinguish between activated and deactivated particles during simulation will slightly negatively impact the execution speed, as workers in the same workgroup are synchronised.

5.2.1 Bleaching phase

We first start by generating particles homogeneously on the simulation domain, with the proportion of bounded and unbounded particles following π_b and π_u. We also generate a mask corresponding to the bleaching profile we choose for the simulation. For each worker, we compute by truncation the particles positions in pixel space. We then, for each particle, draw from from an uniform distribution and compare its result to the one given by the mask at the same position. If the drawn number is higher than the mask value, we then deactivate the particle, thus effectively bleaching it (algorithm 4). This technique has the disadvantage of removing particles from the simulation but this can be compensated by adding more initial particles. Between bleach frames, we let the particles diffuse for a time Δt.

At the end of each bleach frame we can optionally add a consolidation pass which consist of removing deactivated particles. Consolidation occurs on the host side. We first load the position and state of each particle from device memory, loop over them and accumulate the still active particles on a new buffer. We then write back on device memory the new buffer and change the number of particles each worker manages accordingly. For a large number of particles, this operation can be slower than the bleaching and simulation in itself as memory transfer is very slow and the looping is single threaded. Still, as it has been observed that simulation times behaves linearly with the number of particles (as expected), consolidation can be useful if done at the end of the bleaching phase, especially if bleaching was aggressive or for a large number of bleach frames thus giving an important part of deactivated particles.

```
for i ← 0 to particlePerWorker do
    (x, y) ← particlePos [i]
    maskValue ← mask [Truncate(x), Truncate(y)]
    if UniformSample(1) ≤ maskValue then
        particleState [i] ← StateDeactivated
    end
end
```

Algorithm 4: Stochastic bleaching

5.2.2 Simulation phase

The simulation code for the diffusion after bleaching is exactly the same as the one used for the single bleach frame stochastic model.

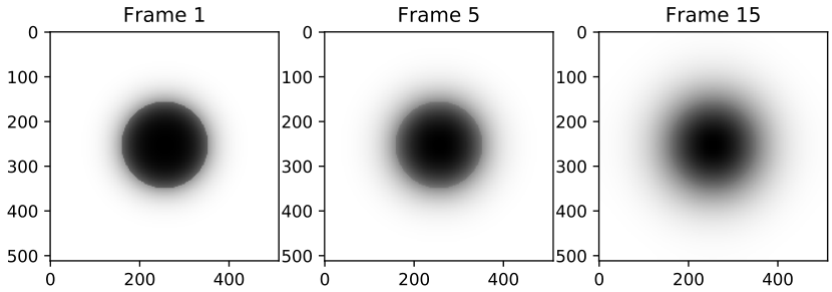


Figure 5.1: Frames for the deterministic multiple bleach frames simulation, with $D = 800$ pixel/s^2, $k_{\text{on}} = k_{\text{off}} = 1$, $\Delta t = 0.265$ s, 4 bleach frames and $\alpha = 0.7$.

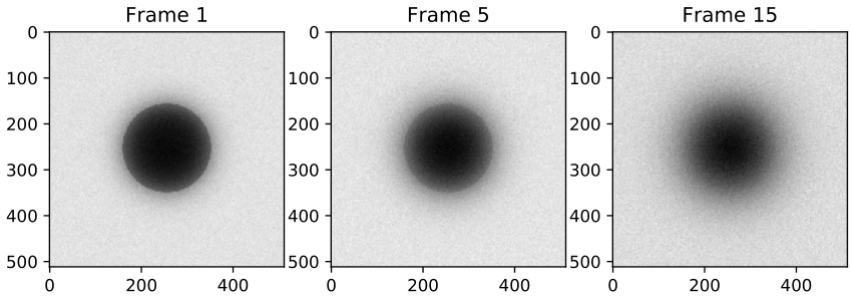


Figure 5.2: Frames for the stochastic multiple bleach frames simulation, with $D = 800$ pixel/s^2, $k_{\text{on}} = k_{\text{off}} = 1$, $\Delta t = 0.265$ s, 4 bleach frames, $\alpha = 0.7$ and 1 billion particles.

5.3 Results, validation and comparison

For both versions of the algorithm, the residuals between the spectral method (figure 4.1, 5.1) and stochastic implementation (figure 4.2, 5.2) are pure Gaussian noise without structure, validating the implementations.

We can then compare the multiple bleach frames simulation with the single bleach frame simulation by computing the relative change considering the instantaneous bleaching model as the reference (figure 5.3):

$$\Delta = \frac{c_{\text{multiple}} - c_{\text{single}}}{c_{\text{single}}}$$

For this comparison, we choose, with n then number of bleach frames:

$$\alpha_{\text{single}} = \sqrt[n]{\alpha_{\text{multiple}}}. \tag{5.1}$$

Without surprise, most of the difference between both simulations is in the vicinity of the discontinuity in the bleaching spot. Positive relative change is mainly found near the inner border of the disk. We see that the relative change is mainly positive, meaning less concentration is lost for the multiple bleach frames algorithm.

More generally, because of the arbitrariness of equation 5.1, it is difficult to compare the first frames of the comparison. However, it seems clear that the choice of the algorithm is less and less important as t grows.

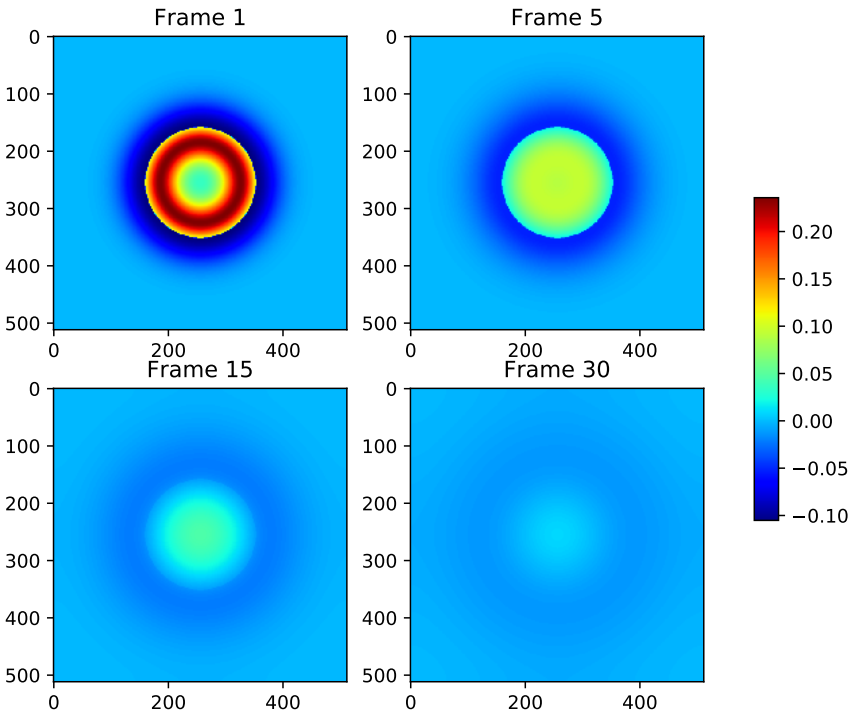


Figure 5.3: Comparison between instantaneous and multiple bleach frames, with with $D = 800$ pixel/s^2, $k_{\text{on}} = k_{\text{off}} = 1$, $\Delta t = 0.265$. For instantaneous bleaching $\alpha = 0.24$ and for multiple bleach frames (4), $\alpha = 0.7$.

Chapter 6

Modelling the scanning motion of the CLSM

As described when presenting the Confocal Laser Scanning Microscope, bleaching and acquisition are done by a laser scanning the sample. This scanning is not instantaneous, which results in a time lag between each pixel reading or bleaching.

6.1 Laser position

The laser follows a path over the image that can be approximated by a sinusoidal function with frequency ω. Experiments are usually done with $\omega = 1000$ Hz. The laser stays over a pixel for a defined period of time (so called *pixel dwell time* - PDT), dependent on the position. Although not completly accurate, it can be assumed that the pixel dwell time is constant and be approximated using the following formula (in seconds):

$$\mathrm{PDT} = \frac{0.295}{1.2\,\omega\,N}. \tag{6.1}$$

As $0.295/1.2 < 0.5$, and hence that the sum of the PDT of each pixels in one line is less than $2/\omega$, we see that only a part of the sinusoid is used. Indeed, in order to homogenise pixel dwell time across the line, only one of the two approximately linear part of the sinusoid is used (the other part being used to move the laser to the next line at pixel 0). The time moving around (*new line time* - NLT) can then be computed using the following formula (in seconds):

$$\mathrm{NLT} = \frac{1}{\omega} - N \cdot \mathrm{PDT} = \frac{0.7542}{\omega} \tag{6.2}$$

Finally, once every line have been bleached or read, we need to return to the origin (*back to origin time* - BTOT). This time is constant and given by the microscope manufacturer. For a 256×256 image, the laser takes 10 ms.

For a given N and ω, we can compute the timing of each laser states. Total frame time is then given by:

$$\Delta t = N\omega + \mathrm{BTOT}$$

For $\omega = 1000$ Hz and $N = 256$, which are the settings used by the real experiments, we can precompute all the timing constants (table 6.1).

	PDT	NLT	BTOT	Total
Timing (s)	$9.6\ 10^{-7}$	$7.5424\ 10^{-4}$	$1\ 10^{-2}$	0.266

Table 6.1: Timings for $\omega = 1000$ Hz and $N = 256$.

6.1.1 Bleaching and zooming

When bleaching, the CLSM will zoom on the bleaching region defined by the bleach radius. This is done in order for the bleaching to be more accurate and limit asymmetry from unwanted diffusion during the scanning. Also, since the bleach region is small compared to the unbleached one, zooming will permit the laser to be more effectively used. Finally, zooming ensures the lines the laser will follow are close enough and thus will not miss any spots which might materialise as unbleached lines.

In the simulation, particles are first generated into zoomed space, i.e. on a $Z(N + 2M) \times Z(N + 2M)$ continuous space with Z the zoom factor. After bleaching, an affine transformation is applied on the particle positions in order to transform the positions from zoomed space into unzoomed space.

6.2 Stochastic simulation

Because the exact or numerical solution to the equations 3.1 or 3.2 with the scanning movement proved to be very difficult, we decided to only implement a stochastic simulation of the new model. As for the previous models, we separate the simulation into two phases, first bleaching and then free diffusion. Bleaching is done by taking several bleach frames. Because of the scanning nature of the CLSM, both bleaching (algorithm 5) and free diffusion (algorithm 6) schemes are very similar.

6.2.1 Scanning movement

One way to simulate the scanning movement for each frame is to sequentially, using two loops, visit each pixel and do work corresponding to the current simulation phase, and then diffuse particles for a time equal to the pixel dwell time. Once at the end of the horizontal line, we do a diffusion equal to the new line time and at the end of the frame, a diffusion for a time equal to back to origin time.

6.3 Optimisation of the algorithms

The first implementation of the algorithm gave bad performance e.g. 12 hours were needed to simulate 200 M particles using both GPUs and CPUs. Depending on parameters such as worker/workgroup distribution, CPU timings could be better than GPU ones by a factor of 3. Thus, substantial effort has been put into optimisation of the algorithms.

We first noticed that a very large number of read and write operations, particularly costly on GPUs, are done. This was fixed by first caching particle positions and states into local memory, work on it, and then loading it back into global memory. Because of the small size of the local memory, workers can now only handle a very small amount of particles. This is compensated by the very large numbers of workers GPUs can handle.

In our work, for 2^{25} particles we found out having 2^{20} workers each handling 32 particles gave the best results.

The most costly operations are diffusion, particularly the drawing from a normal distribution. In the original algorithm, all particles were simulated for each pixels. For 256×256 images, this makes 2^{16} steps (to be compared with the 32 steps of previous models for each frames). This can be mitigated by simulating pixels in stripes i.e. blocks of horizontal pixels. Care should then be taken when choosing the stripe size as it may blur the suspected asymmetric nature of the output frame.

The OpenCL mathematical functions do not necessarily map to the hardwired equivalent implemented on the Compute Units and this behaviour depends on the OpenCL implementation. Indeed, in order for an implementation to be compliant to the standard, floating operations and functions need to follow a set of specified precision and the IEEE 754 standard for floating numbers, which is not necessarily the case of hardwired mathematical functions. For GPUs, originally designed for the gaming industry, these functions do not need as much precision for 3D graphics but need to be fast. Software implementation of these functions can take up to 1000 cycles, and hardwired ones, called *native* in OpenCL jargon, usually takes 1 or 2 cycles. Native mathematical functions are precise enough for our work (the normal sampling algorithm and stochastic simulation seems to be robust enough), and are thus used in place of compliant functions. For an unknown reason, speedup is not spectacular but still enough (around 5 %) for it to be kept.

For floating operations, lots of instructions are executed in order for them to be IEEE 754 compliant. OpenCL gives the developer the choice, at the compilation stage, to relieve some of these checks in order to make computations faster. Setting the `-cl-fast-relaxed-math` compilation flag will, in particular, assume all floating-point arithmetic arguments and results are finite, ignore the sign of 0 in operations [1] and automatically enable mad operations [2]. This has the effect of significantly speedup the computations.

Given the high computation cost given by a relatively small number of particles, it is interesting to remove the bleached particles. We thus introduce a *consolidation* pass at the end of each bleach frames which consist in the removing of particles tagged as non active i.e. bleached. For 2^{25} particles, consolidation pass takes less than 1 second, and gives a 5-10% speedup for each bleach frame, depending on the bleaching coefficient and bleaching region surface. Also, since we are guaranteed a particle is active during the diffusion pass, there is no need to check particle state thus saving us from a costly comparison instruction.

Finally, all branching were taken care of by using vectorised ternary operations.

All these optimisations lead to great improvements in timings (table 6.2). We see that we are close to a one order of magnitude speedup between the first and final implementation of the algorithms. The optimisations done had for effect to change the CPU/GPU ratio giving a better advantage to the GPU, as they could better profit from the manual caching than CPU, which do not need it. Results are highly dependent on the way the workers and workgroups are distributed and scheduled. Before optimisations and for some ratio of workers/workgroup, one could get a CPU/GPU ratio of 0.3.

[1] In the IEEE 754 standard, 0 has two possible representations, -0 and +0 with specific behaviour for each of them for some simplifications.

[2] Hardwired multiply-add operation, usually computed with a reduced accuracy.

	GPU	CPU	CPU/GPU
Original	17420	30352	1.74
Optimised	2000	3840	1.92
Speedup	8.7	7.9	

Table 6.2: Approximate timings (in s) for original implementations of the simulation and after the optimisation described above. GPU timing is for one NVidia Titan Xp, CPU timing is for both Xeon processors totalling 88 cores. The simulation consisted of 4 bleach frames with 25% of particles lost and 10 diffusion frames for 2^{25} particles.

```
input : pos and state of n particles
for i ← 0 to N do
    for j ← 0 to N with j += stride do
        SimulateAllParticles(stride × PDT)
        for k ← 0 to stride do
            for l ← 0 to particlesPerWorker do
                pixelPos ← Truncate(particlePos [l])
                if pixelPos = (j, i) then
                    if UniformSample() > bleachMask [j, i] then
                        state[l] = bleached
                    end
                end
            end
        end
    end
    SimulateAllParticles(NLT)
end
SimulateAllParticles(BTOT)
```

Algorithm 5: Bleaching algorithm modelling the laser scanning of the CLSM.

```
input  : pos and state of n particles
output: frame
for i ← 0 to N do
    for j ← 0 to N with j += stride do
        SimulateAllParticles(stride × PDT)
        for k ← 0 to stride do
            particlesInPixel ← 0
            for l ← 0 to particlesPerWorker do
                pixelPos ← Truncate(particlePos [l])
                if pixelPos = (j, i) then
                    particlesInPixel ++
                end
            end
            frame [j, i] += particlesInPixel
        end
    end
    SimulateAllParticles(NLT)
end
SimulateAllParticles(BTOT)
```

Algorithm 6: Diffusion algorithm modelling the laser scanning of the CLSM.

6.4 Results

Simulating for realistic parameters, we find no asymmetric effects as expected (figure 6.1).

By playing with extreme parameters, it was possible to get asymmetry. Those where found by setting a low scanning frequency and low frame resolution. We compared the state of the simulation after one and four bleach frames, adapting the bleach coefficient using equation 5.1. It was found that adding more bleach frames does not mitigate the asymmetry (figure 6.2). Asymmetry appears more clearly when taking the instantaneous state of the system after photo bleaching (figure 6.3). The asymmetric effect seems to be mitigated by the scanning movement during reading. This would mean that it does not seem necessary, for a set of realistic parameters, to simulate the scanning movement in order to capture all the details of the FRAP experiment.

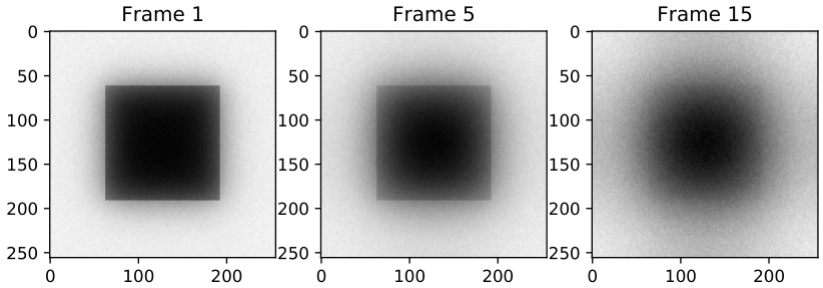


Figure 6.1: Several frames of the scanning simulation with timing parameters from table 6.1), $D = 800$ Hz, 4 bleach frames and a square bleaching region which should be optimal to discern asymmetry.

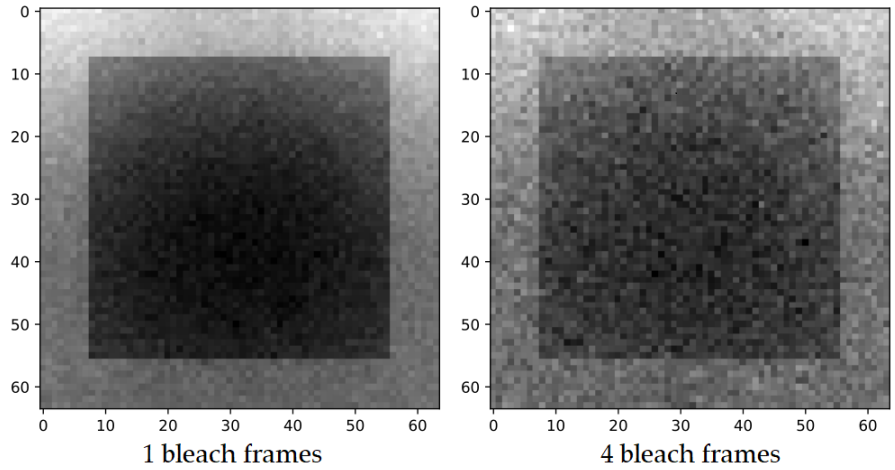


Figure 6.2: First frame after photo bleaching, with $\omega = 32$ Hz, $N = 64$. Some asymmetry is visible.

We see that the bleaching is more aggressive when taking four bleach frames compared to one, even when adapting the bleaching coefficient. This leads to less particles (80M compared to 50M) and thus noisier simulated images.

6.5 Comparison with previous models

We now compare results given by the scanning model with the multiple bleach frames model in the same way than between the single and multiple bleach frames comparison, by computing the relative change between both model (figure 6.4), where parameters are similar and with the same number of particles. In this case, we will suppose the scanning model is the reference, as it was designed to be the closest to the reality:

$$\Delta = \frac{c_{\text{multiple}} - c_{\text{scanning}}}{c_{\text{scanning}}}$$

As between the single and multiple bleach frames models comparison, it is clear that the choice of the model does not matter too much in the long term. We see that in the bleaching region, the relative change is more or less homogeneous (especially visible in the frame 5 and 15) and positive, meaning the scanning model do bleach more than its competing model. This effect is also visible on frame 30 where the mean noise level is around 0.058.

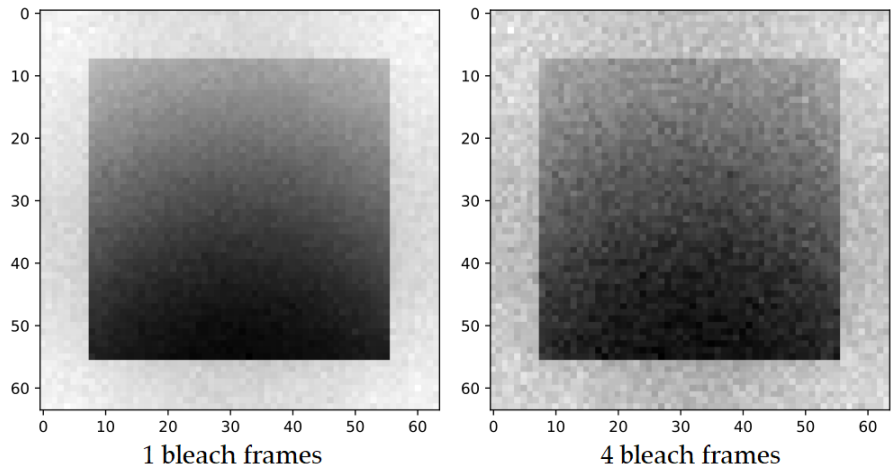


Figure 6.3: Instantaneous state of the system after photobleaching, with $\omega = 32$ Hz, $N = 64$. We clearly see asymmetry due to the scanning movement.

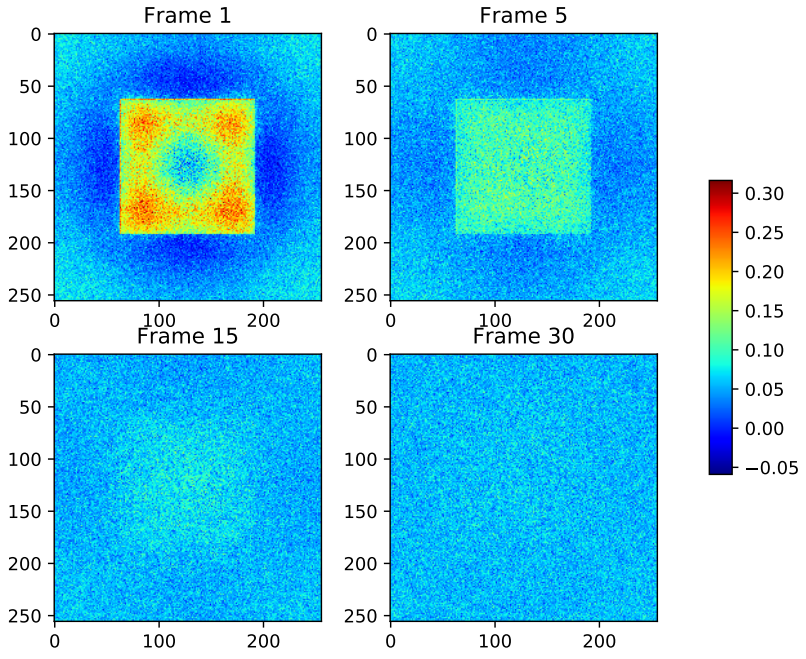


Figure 6.4: Comparison between the scanning and the multiple bleach frames model.

Conclusion

In this work, we first implemented a simple model of the FRAP experiment both by a deterministic approach using a spectral method and by a stochastic approach. This simple model summarised the whole bleaching process into one bleach frame. We then extended the model to take into account multiple bleach frames. We finally simulated the scanning movement of the CLSM.

By comparing both instantaneous bleaching models, substantial differences arise, especially in the first frames, hence it seems clear that it is necessary to simulate all bleach frames and not just one. Further we implemented the scanning movement of the laser of the CLSM into the model, and concluded incorporating this feature did not give a substantial impact on the results. While the scanning movement during bleaching yields an asymmetry in the simulated images (which cannot be captured by the instantaneous model previously implemented), this effect seems to be mitigated by the same scanning motion during the acquisition phase. In fact, asymmetry caused by the non-instantaneous scanning movement was only found with extreme and non realistic parameters. Hence it appears it is not crucial to implement this feature in the model.

Given that simulating the laser movement is computationally heavy, and with the previous results in mind, it appears that the best trade-off between accuracy and computational cost is given by the multiple instantaneous bleach frames model.

Both the deterministic and stochastic versions of the models benefited substantially from the massively parallel architecture of the GPUs, in particular for the last model which would have not been practically feasible to explore on CPU. Further taking the cost efficiency into consideration, we could get a 2x speedup by using a $ 2000 gaming graphic card compared to $ 10000 worth of CPUs.

The work can be extended and improved in many ways. First in order for the CLSM to acquire an image it has to emit a faint laser pulse for each pixel, thus slightly bleaching the sample. Simulating this behaviour would make the scanning model a bit more realistic. Also, in the model implementing the scanning movement, padding was not implemented meaning border effects could perturb the results. It could also be interesting to develop various mathematical tools for a more rigorous analysis of a possible asymmetry in the results and model comparisons.

Bibliography

[1] Daniel Axelrod et al. "Mobility measurement by analysis of fluorescence photobleaching recovery kinetics". In: *Biophysical journal* 16.9 (1976), pp. 1055–1069.

[2] *clMathLibraries/clFFT: a software library containing FFT functions written in OpenCL.* URL: `https://github.com/clMathLibraries/clFFT` (visited on 04/18/2018).

[3] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.2.* Khronos. Nov. 2012.

[4] Pekka Jääskeläinen et al. "pocl: A performance-portable OpenCL implementation". In: *International Journal of Parallel Programming* 43.5 (2015), pp. 752–785.

[5] Jenny K Jonasson et al. "A pixel-based likelihood framework for analysis of fluorescence recovery after photobleaching data". In: *Journal of microscopy* 232.2 (2008), pp. 260–269.

[6] Jenny K Jonasson et al. "Pixel-based analysis of FRAP data with a general initial bleaching profile". In: *Journal of microscopy* 239.2 (2010), pp. 142–153.

[7] Minchul Kang et al. "A quantitative approach to analyze binding diffusion kinetics by confocal FRAP". In: *Biophysical journal* 99.9 (2010), pp. 2737–2747.

[8] DE Koppel et al. "Dynamics of fluorescence marker concentration as a probe of mobility". In: *Biophysical Journal* 16.11 (1976), pp. 1315–1329.

[9] Jeff W Lichtman and José-Angel Conchello. "Fluorescence microscopy". In: *Nature methods* 2.12 (2005), p. 910.

[10] Niklas Lorén et al. "Fluorescence recovery after photobleaching in material and life sciences: putting theory into practice". In: *Quarterly reviews of biophysics* 48.3 (2015), pp. 323–387.

[11] *OpenCL Best Practice Guide.* NVidia. May 2010.

[12] *OpenCL Programming Guide for the CUDA Architecture.* 2.3. NVidia. Aug. 2009.

[13] Stephen W Paddock et al. "Confocal laser scanning microscopy". In: *Biotechniques* 27 (1999), pp. 992–1007.

[14] Erich Schuster et al. "Interactions and diffusion in fine-stranded β-lactoglobulin gels determined via FRAP and binding". In: *Biophysical journal* 106.1 (2014), pp. 253–262.

[15] Brian L Sprague et al. "Analysis of binding reactions by fluorescence recovery after photobleaching". In: *Biophysical journal* 86.6 (2004), pp. 3473–3495.

[16] J. Tompson and K. Schlachter. "An Introduction to the OpenCL Programming Model". In: (2012).

[17] *Writing Optimal OpenCL Code with Intel OpenCL SDK*. 1.3. Intel. 2011.