Gpu-accelerated simulations of the chemotactic response of amoeba *Dictyostelium discoideum*.

by

David Casas Vidal

Under the supervision of

Dr. Sergio Alonso, Dr. Blas Echebarria

A thesis submitted in partial fulfillment for the degree of Engineering Physics

in the

Barcelona School of Telecommunications Engineering (ETSETB)

Department of Physics

May 2016
A bad necromancer always blames the corpse.

Oglaf
Abstract

Barcelona School of Telecommunications Engineering (ETSETB)
Department of Physics
Engineering Physics

by David Casas Vidal

Having computer simulations of biophysical systems is an important step for checking the validity of our models, improve our understanding of their behaviour and serve as a first testing ground for further applications. The aim of this project was twofold: firstly, the design and implementation of a generic framework that eases the creation of gpu-accelerated physical simulations. Secondly, it tests a model for the chemotactic response and locomotion of the amoeba *Dictyostelium discoideum*, using the aforementioned framework. The framework has proven itself very useful during the testing of the model, although some improvements could be done. The model used in this project yielded satisfactory results, but should be completed in order to provide more interesting insights.
Acknowledgements

First of all, my deepest gratitude to Dr. Sergio Alonso and Dr. Blas Echebarria for the effort they put into dealing with me and their brilliant guidance all along the project. Secondly, I would like to thank the Biological Physics group at University of Postdam and particularly Prof. Carsten Beta for their generous collaboration, by kindly sharing their experimental data with us. My thanks to Dr. Shuji Ishihara from Ishihara Laboratory at Meiji University and Dr. Satoshi Sawai from Sawai Laboratory at University of Tokyo as well, for their swift responses in answering our many doubts.

Lastly, I would like to thank my father for his unwavering support even when I holed up in my room for days, only venturing outside at night in brief trips to the pantry.
Contents

Abstract ii

Acknowledgements iii

List of Figures vi

List of Tables vii

Introduction 1

I The Framework 3

1 Introducing the Framework 4

2 Interface 6

2.1 The Solver . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

2.2 The Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

2.2.1 New Types . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

2.2.2 Interface visible from a Solver . . . . . . . . . . . . . . . . . . . . . 8

2.2.3 Interface to provide as a Model . . . . . . . . . . . . . . . . . . . . . 10

2.2.4 Interface provided to the Models . . . . . . . . . . . . . . . . . . . . 11

3 Implementation 13

3.1 Registrable . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13

3.2 Solver . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

3.3 Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

3.3.1 Device access . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

3.3.2 pState . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19

4 Performance 22

4.1 Speed . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23

4.1.1 By sections . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23

4.1.2 Scaling of total time . . . . . . . . . . . . . . . . . . . . . . . . . . . 24

4.2 Soak testing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
## Contents

5 Conclusions 28

11 The Biophysical Model 30

6 Theory 31

6.1 Biological background .............................. 31
6.2 Biochemical model ................................. 32
6.3 Locomotion model ................................. 33

7 Methods 35

7.1 Test Setups ....................................... 35
7.1.1 Wave Properties ................................ 35
7.1.2 Excitation-deformation coupling ............... 37
7.1.3 Influence of Noise .............................. 38
7.2 Numerical Methods ................................ 38

8 Results 39

8.1 Wave Properties ................................... 39
8.2 Excitation-deformation coupling ....................... 41
8.3 Influence of noise ................................. 43

9 Conclusions 45

10 Final Remarks 46

A Object Oriented Programming 48

B GPU Programming and nVidia CUDA 52

C Computer Specifications 54

Bibliography 55
## List of Figures

<table>
<thead>
<tr>
<th></th>
<th>Image of <em>Dictyostelium discoideum</em></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>UML Diagram of framework Classes</td>
<td></td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>Self-registering process diagram</td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>Device access to model</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>pState structure</td>
<td></td>
<td>21</td>
</tr>
<tr>
<td>4.1</td>
<td>Framework execution timeline</td>
<td></td>
<td>23</td>
</tr>
<tr>
<td>4.2</td>
<td>Improved execution timeline</td>
<td></td>
<td>23</td>
</tr>
<tr>
<td>4.3</td>
<td>Total time vs Size</td>
<td></td>
<td>26</td>
</tr>
<tr>
<td>4.4</td>
<td>Peak Memory Used (host) vs Time</td>
<td></td>
<td>26</td>
</tr>
<tr>
<td>4.5</td>
<td>Memory Used (device) vs Time</td>
<td></td>
<td>27</td>
</tr>
<tr>
<td>5</td>
<td>Throughput vs Time</td>
<td></td>
<td>27</td>
</tr>
<tr>
<td>6</td>
<td>Schematic of PI3K/PIP3/PIP2/PTEN cycle</td>
<td></td>
<td>32</td>
</tr>
<tr>
<td>8</td>
<td>Plot speed and width for $\alpha$ and $\beta$</td>
<td></td>
<td>40</td>
</tr>
<tr>
<td>8.2</td>
<td>Plot speed and width for $\gamma$ and $S$</td>
<td></td>
<td>40</td>
</tr>
<tr>
<td>8.3</td>
<td>Plot speed and width for $\mu$</td>
<td></td>
<td>41</td>
</tr>
<tr>
<td>8.4</td>
<td>Reference images for excitation-deformation coupling</td>
<td></td>
<td>42</td>
</tr>
<tr>
<td>8.5</td>
<td>Influence of $\tau$</td>
<td></td>
<td>42</td>
</tr>
<tr>
<td>8.6</td>
<td>Influence of $M$</td>
<td></td>
<td>43</td>
</tr>
<tr>
<td>8.7</td>
<td>Influence of $a$</td>
<td></td>
<td>43</td>
</tr>
<tr>
<td>8.8</td>
<td>Images of simulations with different $\sigma$ and plot of the average value of $V$</td>
<td></td>
<td>44</td>
</tr>
</tbody>
</table>
List of Tables

4.1 Time measures for different codes . . . . . . . . . . . . . . . . . . . . . . . 25
8.1 Reference constants and fitted values . . . . . . . . . . . . . . . . . . . . 39
8.2 Wave width and speed figures . . . . . . . . . . . . . . . . . . . . . . . . . 41
8.3 Constants on the $\phi$ equation . . . . . . . . . . . . . . . . . . . . . . . 42
9.1 Final recollection of constants . . . . . . . . . . . . . . . . . . . . . . . . . 45
C.1 Specifications of small station . . . . . . . . . . . . . . . . . . . . . . . . 54
C.2 Specifications of working station . . . . . . . . . . . . . . . . . . . . . . . 54
Dedicated to my grandparents,
who worry about me and ask about my studies and job despite having a hard time understanding it.
Introduction

Ever since Alan Turing proposed his model for the emergence of patterns from an initially homogeneous mixture of two chemicals [1], a great number of discoveries in biology and in cell biology in particular have taken place; putting modelling in a prominent place for its study [2]. This modelling, together with its subsequent simulations, is being of great help in transitioning cell biology into a more quantitative science but it also calls for the existence of tools that are able to help and keep up with it. One such tool is the computer graphics card. Its powerful parallel capabilities make it a great asset for computing most biophysical simulations.

Chemotaxis allows cells to respond to changes in the environment. This way, single cell organisms can move towards food, or flee from unfavorable chemicals. In multicellular organisms, chemotaxis is key in development [3]. e.g. during the embryogenesis, chemotaxis is the process driving the positioning and development of many areas [4], and chemotaxis is also vital for the correct functioning of the mammalian immune system, being key in the trafficking of neutrophils [5],[6],[7]. It has been shown that an organism that shares many similarities with the chemotactic response of the mammalian neutrophile is the Dictyostelium discoideum (shown in Figure 1) [8],[9].

![Figure 1: SEM imaging of Dictyostelium discoideum in different stages of its life cycle. Copyright M.J. Grimson and R.L. Blanton; Biological Sciences Electron Microscopy Laboratory, Texas Tech University.](image)
The goal of this project is twofold:

- The design and implementation of an efficient yet easy to use tool for running biophysical simulations on the gpu. Specifically, the creation of a framework such that users may add their own codes to it and run them in a streamlined manner. The advantages would be: scientists who have no technical knowledge related to gpu programming could still make use of the graphics card to speed up their simulations, scientists would still profit from code reuse, not having to repeat some of the steps involved in the creation of the simulation.

- To improve our understanding of the mechanism underlying the chemotactic response of *D. discoideum* [10] and using experimental data to test whether the model fits or not, and learn which set of parameters leads to realistic behaviour.

The written document is also separated in two parts: a first part dedicated to the framework and a second part focused on the biophysical model.

Before getting into any of the detailed parts comes this introduction.

Then, the first part: the framework, which is broken down to 5 chapters. The first chapter gives an introduction to what is a framework, what are the goals of our framework and how does it roughly work. The second chapter, explains how to make your code so that it can be plugged into the framework; explaining the available interfaces of its different parts. The third chapter, titled ‘Implementation’, goes into some detail explaining how certain parts of the framework are internally done. The fourth chapter explains some performance tests that have been run on the framework and the results obtained. The fifth extracts some conclusions.

The second part, the model, is divided in 4 chapters. It starts with ‘Theory’, where some context is given and the model used is reasoned. Afterwards follows ‘Methods’, explaining what simulations have been done and in what exact manner. It then gives onto ‘Results’, where we show some of the figures obtained and comment on them. The chapter is then closed by ‘Conclusions’.

And at the end a chapter is dedicated to summarizing the ‘Overall Conclusions’.
Part I

The Framework
Chapter 1

Introducing the Framework

The first goal of the project was to provide a useful framework to develop gpu-accelerated simulations of physical systems. By ‘framework’ we mean a program that, even though it owns the flow of control it is still largely incomplete without a user’s extensions: our program will take care of memory usage and management, and it will largely determine what happens when, but it is up to its user to determine how what is done. And by ‘useful’ we mean something that: a) allows scientists without previous knowledge on gpu programming profit from its powerful capabilities; b) simplifies the creation and testing of many different models whether you have some knowledge of gpu programming or not, and c) remains meaningful to those with knowledge of gpu programming by providing a comfortable interface that, while easing their work whenever it can stay simple, does not hinder them from using the most complex and powerful capabilities offered by the graphics card.

The program is a compiled binary named applebeat, that takes 4 arguments when run, i.e.:

```
# ./applebeat <solver> <model> <cfg_file_path> <const_file_path>
```

where `<solver>` is the solver chosen to run, `<model>` the chosen model, `<cfg_file_path>` the path to the configuration file (See Box 1.1), and `<const_file_path>` the path to the constants file (See Box 1.2). If an unknown solver or model is input (we will later explain what those are), an error message will be displayed together with a list of available solvers (or models) before exiting. If an invalid path is given as configuration or constants file, the program will display an error message and crash. If an otherwise invalid file is given (e.g.: badly formatted), the behaviour is undefined.

Let us now look at where the binary comes from. There are two highly differentiated parts in the program:

---

1 The order in which the sequence of instructions is followed.
• **Core code.** This is the framework itself and it will generally suffice to compile it once the first time the binary is built, since these files are not meant to be changed.

• **Extension code.** This part is made by the users. It can be moved in and out as needed. A couple of examples of this code are provided in order to help users in making their own.

Both of them are written in C++ and placed under the ‘src’ directory together. These parts are meant to be built together into the binary via GNU Make using the Makefile provided, which should be conveniently edited to include all extension code.

Finally, it is time to tackle those words, solver and model. Usually, when coding a simulation the code will feature at least the following two different elements: sections dedicated to time stepping, and sections that hold the actual physical properties of the system. When developing many different simulations, the former will very often repeat (e.g. having to write the loop and the time stepping method in each simulation); and there may also be times when the interest is focused on different ways to solve the same system, which will lead to the latter repeating around instead. This idea is one of the main pillars the framework is built upon, and that is why users will find two main objects to manipulate: Solvers (which represent the former) and Models (that stand for the latter). Solvers and Models are the extensions added by the users unto the framework, and they are independent of each other.

---

**Box 1.1: cfg file**

The configuration file provides the simulation’s duration and the simulation’s grid size. Its expected format is: an integer duration, a newline, an integer size. ie:

```
120
400
```

**Box 1.2: const file**

The constant file provides all the constant values needed by the physical simulation. Its expected format is a list of name decimal_value pairs separated by newlines. Comment-like words may be inserted before and/or after any of the pairs if they start with #. An example would be:

```
#setA
alpha 15

#discrete
dt 0.001 #s
dx 0.1 #um
```

---

Well, not exactly. Rather, they are independent of each other’s implementation; that is, of their contents. They are not fully independent since the Model has a public interface that is actively used by the Solver.

Chapter 2

Interface

The first step in order to understand what the framework is and what it does, is taking a look at the interface shared between the core code and the extension code. As we said, there are two kinds of extension code: Solvers and Models. Hence, we shall take a look at one after the other.

2.1 The Solver

The Solver represents the part of the program that takes care of the flow of control and steps over time in our simulation. Examples of solvers would be implementations of the Euler or Runge-Kutta methods. To that effect, it shares a common interface with the framework itself, that will be used to access its utilities:

```cpp
Solver(std::map<std::string, double> &values, int sx, int sy, int sz);
void run(int time);
```

Besides the Solver constructor itself, the only thing that is asked from a solver is a `run` method, whose only input is an integer called `time`, that carries the ‘simulation length’ written in the configuration file. In here you may implement whichever solver you wish and in order to do so, you may make use of the following objects to which every solver has access:

```cpp
Model *pmodel;
Model *pmodel_dev;
```

---

1The constructor is a method that initializes the object; it creates an object of that class, if you will. See Appendix A for a glossary of Object Oriented Programming terms.
dim3 defaultGrid;
dim3 defaultBlock;

The first two are pointers to the Model you are meant to simulate (one in cpu-space and another one in gpu-space). The remaining two are meant to be a help used when running the gpu kernel and should not bother us now.

2.2 The Model

The Model represents the part of the program that holds all the information relevant to the physical model that is being simulated, and only that. Typically this would be a set of partial differential equations with the integrated variables being spatio-temporal fields. Much like the Solver, it shares an interface with the rest of the framework, but lengthier. Thus, we will break it into parts:

2.2.1 New Types

First, we will browse through some types that are defined within this interface, and see what they represent:

- **pExtraData**, is a small package of data that can be passed around. Each Model redefines it and uses it as deemed fit.

- **pFields**, a pointer to the field matrixes. It holds the value of each field in each point of the grid.

- **State**, is a class that represents the simulation data. It has a pFields inside it, among other metadata needed for the working of the simulation (See Box 2.1).

- **pState**, a pointer to a State. In reality, these are the ones that will be directly manipulated.
  - **MemPlace**, an enumeration with values HOST, PINNED and DEVICE. This is used to mark a memory area: DEVICE means gpu, HOST means cpu and PINNED is used for

---

2 See Appendix B for a glossary on concepts related to graphics cards and gpu-oriented programming.

3 Actually, if we take a look at the source code we will see it’s called ‘State’ The reason for that is to prevent errors in the code. This type is not meant to be worked on directly, but through pointers. This way we prevent mistakes like typing ‘State’ instead of ‘pState’ and having it silently work only to crash somewhere else.

4 An enumeration is a user-defined data type that consists of integral constants, each with a name. See Appendix A
Box 2.1: The State

The State is a data structure defined within the Model’s interface, and it is meant to hold the simulation data, separated in ‘fields’, as well as everything else that goes with it. It is defined within the Model’s domain because all other elements in the framework need not know what it is. They will hold it and they will move it around but all the State-related hard work is left to the Model. Also because we need Model to take care of many of the things involved in creating a State, its direct use is strongly discouraged: we shall ask the Model for pointers to a State (pState) and use those instead. Since most of the heavy-lifting is performed by the framework, users are required to know only the following:

- From the Solver point of view, whenever a pState is used remember to go through Model first, that is:
  \[
  \text{Model::pState } s = \ldots
  \]
- From the Model point of view, you can access the data via
  \[
  \text{pState } s;
  \text{s->fields[field_index]}
  \]

2.2.2 Interface visible from a Solver

We shall now take a look at the methods that are visible from a Solver. The methods have been marked with a preceding H when they may be called from host code and with a D when they may be called from device code

H pState getNewEmptyState(MemPlace mp);
DH void setStateToInitial(pState s);
H void deleteState(pState &s);
H void copyState(pState dest, const pState src, bool sync = true);

We start with a set of methods to manage our pStates. In order,

getNewEmptyState Allocates and initializes a State object in the memory area given by mp, then returns a pointer to it.

setStateToInitial Sets the given pState to some defined ‘initial state’.

---

5For a longer explanation see Appendix B
6If the reader does not know what host or device mean in this context, it is recommended he or she reads Appendix B
deleteState

Deallocates all memory used by the given state and sets it to nullptr.

copyState

Copies all data within src to dest. It will wait until the copy is finished before returning, unless sync is set to false.

H void operate(const pState arg1, float arg2, float(&op)(float, float), pState dest);
DH void operate(const pState arg1, float arg2, float(&op)(float, float), pState dest, int pos);
H void operate(const pState arg1, const pState arg2, float(&op)(float, float), pState dest);
DH void operate(const pState arg1, const pState arg2, float(&op)(float, float), pState dest);

Basically, dest = op(arg1, arg2). Being more verbose, the operation op is applied to arg1 and arg2 in that order, on a point-per-point basis and to all fields within the pStates. If the version with pos is used, then it’s all fields but only in that one point.

H void getDerivatives(const pState src, pState dest, float t, pExtraData ed = nullptr);
D void getDerivatives(const pState src, pState dest, float t, int pos, pExtraData ed = nullptr);

Calculates the derivative over time of src and places the result in dest, where src and dest cannot be the same. t is expected to be the time, and ed holds the extraData package that the Model may need for its calculations. The second version does the same but only for the point in pos.

DH bool isOfInterest(const pState s, int pos);
H void fixContour(pState s);

isOfInterest(...) returns true when calculations related to the point at pos must not be skipped, and false otherwise. fixContour(...) will modify the given pState to ‘manually impose’ the boundary conditions in the given pState.

H pExtraData calculateExtraData(pState s, MemPlace memP);
H pExtraData copyExtraDataToDevice(pExtraData ed);
The first one calculates all extra data using the given pState, and places it in the
given memory area, returning the pointer to that place. The second one takes a HOST
pExtraData and copies its contents to DEVICE, returning the pointer to the place it has
been copied to.

```cpp
H void checkStateForAbnormalValues(const pState s, bool negs, bool zero = false,
                                   bool inf = true);
H void checkStateSpecial(const pState s);
H void checkConstants();
```

These methods are offered for debugging and checking purposes. The first one will print
debug information and halt the program at the encounter of an ‘abnormal’ value among
all points and fields in its given state. Abnormal values are NaN and negative values, zero
and infinity as well if negs, zero and inf are set to true respectively. The second one
is similar and will run a test on all fields in the pState, that will have been custom-made
by its Model. The last one provides a way to check for the constants’ values and to
make sure they have been copied to the gpu correctly.

```cpp
H void printState(const pState s, int frame);
H int getSize(int idx);
```

These two are the last ones. printState(...) will print the given pState. frame is
meant to be some indicative of the current frame (as in, a movie frame) or time, since
the Model itself has no information regarding such things. getSize(...) simply returns
the size of the grid in the dimension given by idx.

### 2.2.3 Interface to provide as a Model

```cpp
Model(int n, int m, int l);
Model(std::map<std::string, double> &values, int n, int m, int l);
bool isOfInterest(const pState s, int pos);
void inner_setStateToInitial(pState s);
void inner_getDerivatives(const pState src, pState dest, float f, int pos,
                           pExtraData ed);
void inner_printState(const pState s, int frame);
void inner_fixContour(pState s);
void checkStateByField(const pState s, int field);
std::vector<std::string> getConstNames();
```
So, starting from the top we have:

**Model**
Two constructor methods. **values** is a map[^1] that contains all the constants found in the *nums* file, the other arguments are the grid size in each of the 3 dimensions.

**isOfInterest**
This method should return **true** when the point of the given pState at **pos** must not be skipped in the computations.

**inner_setStateToInitial**
Sets the given pState to some defined ‘initial state’.

**inner_getDerivatives**
Calculates the derivative of the given pState point at **pos** and puts it in the same position, at **dest**.

**inner_printState**
Prints the given pState.

**inner_fixContour**
Modifies the given pState in order to enforce boundary conditions.

**checkStateByField**
In order to debug or make sure values are within some desired limits, this method is provided. Here you may check for whichever conditions each field must comply, and print an error message or issue a halt in case they don’t.

**getConstNames**
This method is expected to return a vector with the names of the constants used, in the order they are stored in memory.

### 2.2.4 Interface provided to the Models

There are some objects to which all models have access:

- **int N_FIELDS**, this constant holds the number of fields in the non-constructor methods.

- **int SIZE[3]**, an array with the size of each dimension.

- **int XY**, a shortcut for **SIZE[0] * SIZE[1]**.

- **size_t bsize**, size of a field in bytes. It should equal to **sizeof(float)*XY*SIZE[2]**.

- **size_t EXDATA_SIZE**, size of the pExtraData packages.

[^1]: A set of pairs (key, value). In a map, the key values are generally used to sort and uniquely identify the elements, which can then be accessed directly by their corresponding key using the bracket operator.
• `int N_CONSTS`, number of constants used.

• `BMPWriter *printer`, a pointer to a BMPWriter object. It is given as a help in case a user wants to print images for a simulation.
Chapter 3

Implementation

We have commented some of the implementation details already but this chapter will go deeper into it. As explained in the previous chapter, the program is written in C++ and nVidia CUDA, and relies heavily in Object Oriented Programming concepts. From this point on, it will be assumed that the reader has previous knowledge of such concepts, see Appendix A if necessary before reading this chapter.

There is a **Solver** as well as a **Model** virtual class, which all user-made models and solvers must inherit. These classes exist not only to create the interface that allows this extended code to be used, but also to take care of many things in the users’ stead. Both virtual superclasses inherit from a **Registrable<T>** template class, that takes care of the process that enables users to create new classes that are unknown to the core code and use them by simply linking all the things together. There is also a **BMPWriter** class that provides methods for translating arrays and matrixes into BMP image files and more specifically, to make images of the **Model::pState**. Since the solvers will be using the models, **Solver** has a pointer to this **Model** object that will be used. Similarly, models have a pointer to a **BMPWriter** object too. All of this is shown in Figure 3.1 together with some details on their methods and attributes.

3.1 Registrable

This class is meant to encapsulate everything necessary for the different classes (i.e.: Solver, Model) to have self-registrable subclasses. Achieving this self-registering mechanic requires some odd tricks, which we shall now explain.

Starting on one end, it is desired to have somewhere our subclasses will register unto, and there should be only one of these for each superclass. This translates into the existence
Figure 3.1: A simple UML Diagram of the classes that make up the framework and their basic relations. Some relations as well as the methods and attributes of Solver and Model have been omitted for the sake of simplicity. White-headed arrows represent inheritance, dashed arrows dependency (an object of the dependee is a parameter variable or local variable of a method of the dependent class) and plain arrows association (an object of a class contains a pointer or reference to an object of the other class).
Box 3.1: The *static* keyword

The *static* keyword can be used when declaring:

- An object (except in a function parameter list).
- A function.
- An anonymous union.

And it behaves slightly differently depending on the context it is used in:

- If used in a declaration of a class member, it declares an *static* member. *Static* members are not bound to class instances: they are independent objects with *static* storage duration or functions defined in namespace scope, only once in the program.
- If used in a declaration of an object, it specifies *static* storage duration (except if accompanied by thread_local).
- If used in a declaration at namespace scope, it specifies internal linkage.

A concept called 'static storage duration' is mentioned. It means that the storage for the object is allocated when the program begins and deallocated when the program ends and only one instance of the object exists. All objects declared at namespace scope (including global namespace) have this storage duration, plus those declared with static or extern storage duration.

![Diagram showing the program flow followed during the self-registering of user subclasses.](image)

**Figure 3.2:** Diagram showing the program flow followed during the self-registering of user subclasses. No standard notation is used, this is just meant to intuitively show the relations between the elements needed for the mechanic to take place.

of a *static Registry* member object (which we called `regs`). Likewise, we will need to have some member method that takes care of the whole registering and is called before any instance is created, hence an *static registerSubclass* member method is needed. Now, on the opposite end, we must notice that it is desired for all the subclasses to be known and registered before the program even starts (ie. before the `main` is started).
different number of things are done before starting to run the code, and among these are the initialization of \textit{namespace} scope objects and \textit{static} class members. That is why, our ‘trigger’ to have subclasses self-register will be an \textit{static} member (called \texttt{earlyBird}) such that it calls \texttt{registerSubclass} in its initialization. Since there is no basic type that behaves in such a convenient way when initialized, a new type is created that does just that, and this new type is called \texttt{Register}. We are however not yet done: the way things are now, our program would not be guaranteed to work. That is so because \texttt{earlyBird} and \texttt{regs} are in different translation units and hence you cannot guarantee that \texttt{regs} will be initialized before \texttt{earlyBird} is (which means, \texttt{earlyBird} might end up using \texttt{regs} before it even exists in memory). That is why instead of having \texttt{regs} be an \textit{static} attribute, an \textit{static} method called \texttt{getRegistry} is created, inside of which \texttt{regs} is placed, as an \textit{static} local variable. This way, all the desired properties (\textit{static} storage duration, uniqueness) are conserved and we make sure that the object will have been initialized when called. Thus, we are finally left with the program flow illustrated in Figure \ref{fig:registering}.

Now that we have grounded a way for the registering to take place, it might be time to try and explain what do we mean by ‘register’. When we say we want the subclasses to ‘register’ themselves, we mean we want them to do two things:

1. Put their name somewhere in such a way that when later asked for a subclass, we know whether it exists or not.
2. Give us some way to instantiate their objects.

This is achieved by having the \texttt{Registry} be (where \texttt{T} is the template type of \texttt{Registrable<T>}):

\begin{verbatim}
typedef
    map<string, function<T *(map<string, double> &, int, int, int)> >
    Registry;
\end{verbatim}

And then creating the methods:

\begin{verbatim}
    // returns 'true' if there is a subclass registered under 'name'.
    static bool isRegistered(string name);

    // returns list of registered subclasses.
    static vector<string> getKnownOnes();
\end{verbatim}
Implementation

// returns a pointer to a new instance of the subclass registered under
// 'name', passing the rest of the arguments to its constructor.
static T *newOne(string name, map<string, double> &values, int sx,
                  int sy, int sz);

3.2 Solver

As it has been stated at the beginning of the chapter, user-made solvers must all inherit
from the core Solver virtual class. This class is actually pretty empty. Besides storing
pointers to the model that will be used (both in cpu and gpu), the only added function-
alitity is that of providing a default setting for the kernel launch. This is done with a
private method called setParallelBlocks(), that queries the properties of the current
gpu and uses them to set the variables defaultBlock and defaultGrid accordingly.

3.3 Model

The Model has mainly two parts of real interest. On one side, some of the model’s
methods might be used in either cpu or gpu code indifferently which poses some problems
whose nature and solution will be considered later on. On the other side, there is the
structure of the pState, and everything that has been built around it.

3.3.1 Device access

In general, if there is an object in host code and it is wanted to be used in device code; it
is possible to pass it as an argument in a kernel launch. Thus, the object will be copied
into device space and its data will be accessible. However, this does not always work as
there are some restrictions, among which we find the following:

- ‘It is not allowed to pass as an argument to a _global_ function an object of a
class derived from virtual base classes.’ (section E.2.10.4 in [12])
- ‘It is not allowed to pass as an argument to a _global_ function an object of a
class with virtual functions.’ (section E.2.10.3 in [12])

This leaves us with two options: either another model is made in device memory from
scratch or a model is copied from host to device before the kernel launch. The former
might be the simplest, but we should bear in mind that the submodel needs to get the
Box 3.2: Creating a new extension

In order to create a new solver and use it within the framework, follow these steps:

1. Create a headers file (extension *.h) and define your class in there. Make sure you:
   - Include solver.h.
   - Have your class extend Solver.
   - Have all required methods for the interface.
   - Have the Register<SubClass> earlyBird member.

2. Create an implementation file (extension *.cpp) and define all methods declared in the headers file. Make sure you:
   - Include your headers file.
   - Initialise the earlyBird, i.e.:
     ```cpp
     Subclass::Register<SubClass> Subclass::earlyBird("my_new_solver");
     ```
   - Don’t forget to write Subclass:: in front of the methods when defining them.

3. Copy both files into ‘src’ and add the name of your .cpp file at the end of the line that starts with EXTRA_SRC.

4. Build the application with
   ```
   # make
   ```

5. Run the application.

In the case of creating a new model, it pretty similar but there are some more requirements related to the subclass’ constructors:

   - In both initialization lists, a call to Model’s constructor is needed. Its signature is the following:
     ```cpp
     __device__ __host__ Model(size_t dataSize, int nconst, int nfields,
     int sx, int sy = 1, int sz = 1);
     ```
     Where dataSize would be the size of the extraData packages (sizeof(Data), if you have defined it; 0 otherwise), nconst the number of constants used, nfields the number of fields in the model and sx, sy, sz, the grid sizes in each dimension.

   - In the constructor with the map, a call to makeDeviceCopy is needed:
     ```cpp
     makeDeviceCopy</*SubClass*/>(&dev_ptr, this, /* nconst */,
     /* sx */ , /* sy */ , /* sz */);
     ```
     Where the commented parts are to be replaced by the actual content (i.e. writing the number of constants instead of ‘nconst’) and the rest should be left as is.

In any case, checking how the existing models and solvers are made is a good idea when making your own.

information from the constants file, and that this information is stored in a map. Maps are defined in the C++ Standard library, they are not an integral part of the language and they cannot be used in device code. This would leave us with the second option, except for the fact that if we were to copy a model object from host to device in a
straight way, the \textit{vpointer}\footnote{When we have an object of a class that overrides virtual methods, a special member is created automatically by the compiler called \textit{vpointer}. It is a pointer that points to a table where the addresses of that object’s methods are stored.} will no longer work, since it is still pointing to an address in the host memory space (basically, this would translate in trouble whenever the virtual methods in \texttt{Model} were called). In the end, a middle ground approach is taken: we create a new submodel in device (that provides us with a valid \textit{vpointer}) and we then copy all the constants into it. This copy is done in one go, via \texttt{cudaMemcpy(\ldots)} and in order for this to work we rely on the section 9.2, clause 13 of the C++ standard\footnote{Nonstatic data members of a (non-union) class with the same access control (Clause 11) are allocated so that later members have higher addresses within a class object. The order of allocation of non-static data members with different access control is unspecified (Clause 11). Implementation alignment requirements might cause two adjacent members not to be allocated immediately after each other; so might requirements for space for managing virtual functions (10.3) and virtual base classes (10.1).} which guarantees that the ordering of the constants will be the same in both objects.\footnote{\texttt{[13, Section 9.2 Clause 13]}} In order to know \textit{where} to copy (that is, where does the block with the constants start), a placeholder member called \texttt{marker} is created in \texttt{Model}, whose address is taken and saved into the publically accessible \texttt{constants\_start}: the copied block will start there and copy according to the number of constants. This relies on non-standard behaviour\footnote{The order in which the base class subobjects are allocated in the most derived object (1.8) is unspecified.} but allows us to do this transparently without asking anything more from the user, who is already burdened with calling the copying method (i.e. \texttt{makeDeviceCopy<Subclass>}) and defining a constructor that does not require a map. Figure 3.3 shows all elements involved in the mentioned process and their relations in the final implementation.

### 3.3.2 pState

The \texttt{pState} is a pointer to \texttt{State} (which we shall refer to as State). A State is meant to hold all the raw data of the simulation as well as metadata that helps us make some sense of it. Currently, this means:

- **\texttt{int nderiv}**, an integer stating the derivative order of the data held in the State.

- **\texttt{MemPos mp}**, which tells us whether this data is actually stored in host memory or device.

- **\texttt{pFields fields}**, which holds the actual raw data. Currently \texttt{pFields} is a typedef of \texttt{float **}, meaning that this is a pointer to an ‘array’ of pointers, each pointing to the ‘array’ of floats that holds the data for each field.

- **\texttt{Model * dev\_ptr}**, a \texttt{nullptr} if the State is stored in cpu, but a pointer to the State structure that provides access to this data from gpu space.
This structure is graphically shown in Figure 3.4. The details of the methods that work with State objects will not be commented here since they are straightforward once the structure of the object is known.
Figure 3.4: Diagram showing the structure of a State. At the top, the representation of a State with mp equal to HOST or PINNED; at the bottom the one for States with mp equal to DEVICE.
Chapter 4

Performance

In order to test its performance, some kind of simulation must be run. So as to show some variety while staying close to the framework’s possible use it was chosen a simple reaction-diffusion system of size 400x400 that requires the calculation of two integral terms, solved using Euler’s method and printing two images every 150 time steps (out of a total of 120 000 time steps). Besides the framework codes, two other dedicated gpu programs have been made to serve as reference. Thus we have the following four applications:

A) Using the framework, a parallel (solver) implementation in gpu.

B) Using the framework, a serial (solver) implementation in cpu that uses the same model as the previous solver. This is meant to give us a figure for the times taken by serial approaches.

C) Dedicated parallel code, a naive approach that mimics A) by doing the same in the same order, but without all the structures. It should help us get an idea of the overhead introduced directly by the use of the framework.

D) Dedicated parallel code, an improved version that uses streams and threading. This one is provided so as to see our framework’s shortcomings and the indirect losses due to optimizations that aren’t possible or are simply harder to make because of using the framework. Note that it hasn’t been deeply optimized, that would be no small task and goes beyond the scope of this project.

All simulations discussed in this chapter have been run on a linux machine with the specifications shown in Table C.1 in the Appendix C.

\footnote{This is actually one of the models used later in Chapter 7}
4.1 Speed

4.1.1 By sections

The test simulations have been timed at various points in order to get an idea of how much time is spent in each section. The results obtained can be found in Table 4.1 and some approximate execution timelines are shown in Figures 4.1 and 4.2.

Looking at the results, several points can come to our attention. Let us start by taking a look at the timeline that corresponds to A) (Figure 4.1) and noticing that, while the matrix operations are indeed done in a parallel fashion (both the computation of the integral terms as well as the solving of the equations), our software is not fully parallel at all: we have two independent computing engines, yet we use them one at a time.

Furthermore, if we add in the fact that most CPUs nowadays are multicore, we realise we are in fact far from taking full advantage of our available hardware. Taking a look at the timeline of D) (Figure 4.2) the improvement in this sense is apparent: while the Engine and GPU are working, the CPU is printing the images.

Indeed in the figure’s caption it is stated that the GPU blocks actually do use CPU too. Note, however, that it also states that their actual duration is really small. This means that, while the CPU isn’t really idle from start to end of that block, it still spends most of that time idle.
Let us now look at the speed differences between our versions in numbers, shown at Table 4.1. Since it is not our goal to go too deep into these, plus the fact that the amount of factors and details in the implementation of our reference codes that would affect these results is quite high; we will only focus on the really big differences. Clearly, the gpu-accelerated framework is close to 5 times faster than the serial, due to being much faster at both computing the integral terms and solving the system. We can also notice that, the overhead introduced by our framework is currently large: our dedicated code is 1.5 times faster. If we take a look at which part of the system is to blame for this drop in speed, it is clearly the time taken to solve the system (there are big differences in the computing of the integral too, of unknown explanation). The reason for that is probably the combination of having many function calls (there are close to three function calls for each system-wide operation performed) together with the graphics card natural slowness when it comes to code jumps. We must also think that the system that is being solved in this particular case is very simple (few and fast operations), which is bound to make the overhead penalty seem larger than it would otherwise. Lastly we see that codes C) and D) perform about equally, but thanks to the advantages of threading the time spent printing is not added to the total time, resulting in an slight speedup that is proportional to the fraction of time that was spent printing to begin with.

4.1.2 Scaling of total time

In order to see how well our framework scales with the system size, a number of simulations with varying grid sizes have been made with all four codes. The result is shown in Figure 4.3. We can see that as would be expected, the scaling of the parallel codes is much smoother than that of the serial implementation. However, it is also apparent that the scaling of our framework is steeper than that of the dedicated codes.

4.2 Soak testing

To check whether memory usage increases over time, which would hint at the existence of memory leaks; a long simulation of about three hours was done, while measuring peak memory usage in the host (Figure 4.4), current memory usage in the device (Figure 4.5) and whether the delay between prints is increased over time(Figure 4.6). As can be seen in the figures, the framework passed all the tests, showing no apparent increase in either one. Memory usage in device is very noisy, most likely due to the workstation performing other tasks at the same time; however since around the end we see an usage of about the same size as the one we had at the very beginning, we can still be certain that there are no leaks.
<table>
<thead>
<tr>
<th>Section</th>
<th>Code C Total (s)</th>
<th>Code C Avg (ms)</th>
<th>Code D Total (s)</th>
<th>Code D Avg (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>0.0091</td>
<td>0.1584</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Start</td>
<td>0.0026</td>
<td>0.0026</td>
<td>0.0026</td>
<td>0.1558</td>
</tr>
<tr>
<td>Init</td>
<td>0.0065</td>
<td>0.0065</td>
<td>0.0065</td>
<td>0.0065</td>
</tr>
<tr>
<td>Simulation</td>
<td>454.6</td>
<td>396.7</td>
<td>454.6</td>
<td>397.9</td>
</tr>
<tr>
<td>State init.</td>
<td>0.1201</td>
<td>0.997</td>
<td>0.1201</td>
<td>0.997</td>
</tr>
<tr>
<td>Printing</td>
<td>39.12</td>
<td>48.9</td>
<td>39.12</td>
<td>41.65</td>
</tr>
<tr>
<td>Copy H2D</td>
<td>119.2</td>
<td>0.99</td>
<td>116.5</td>
<td>0.99</td>
</tr>
<tr>
<td>Integral</td>
<td>108.3</td>
<td>0.90</td>
<td>108.0</td>
<td>0.95</td>
</tr>
<tr>
<td>Solving</td>
<td>57.51</td>
<td>0.48</td>
<td>52.56</td>
<td>0.50</td>
</tr>
<tr>
<td>Copy D2H</td>
<td>115.6</td>
<td>0.96</td>
<td>109.6</td>
<td>0.89</td>
</tr>
<tr>
<td>Boundary</td>
<td>9.525</td>
<td>0.079</td>
<td>10.47</td>
<td>0.090</td>
</tr>
<tr>
<td>Timing</td>
<td>1.144</td>
<td>1.432</td>
<td>1.144</td>
<td>1.432</td>
</tr>
<tr>
<td>Total Time</td>
<td>454.6</td>
<td>397.9</td>
<td>454.6</td>
<td>397.9</td>
</tr>
</tbody>
</table>

Table 4.1: Time measures for the different codes. Each code shows two measures: Total time in seconds spent in each section, and average duration of that section, computed as the total time divided by the number of times that section is run; in milliseconds. Among the sections shown we find ‘Start’, which accounts for the running time before actually starting the main, ‘Init’ that groups the various initializations, ‘CopyH2D’ and ‘CopyD2H’, which account for the time spent copying data from the cpu memory space to the graphics card and vice versa, and ‘Timing’ that offers a lower bound to the overhead introduced by the timers themselves.
Figure 4.3: The plot shows the scaling of total execution time versus the size of the grid, for all of our different code versions.

Figure 4.4: Peak memory used in host vs time.
Figure 4.5: Memory usage vs time.

Figure 4.6: Delay between prints vs time
In the end, a program that helps the making and execution of simulations has been showcased. It divides simulations in two parts called Solver and Model that are written separately, allowing for code reuse and simplicity of execution and it offers flexibility, having been made with ease of use as a priority while sacrificing as little computational efficiency as possible. We have tested its performance and have checked that indeed, faster simulations are obtained by using gpu-acceleration, specially when the grid size is big; and that the overhead introduced by the data structures is for the most part negligible.

However, we have also seen that the current state of the application is far from perfect. It is, in fact, hardly a competitive solution until the following severe errors are addressed:

1) Users should not be expected to edit the Makefile when adding their extensions.

2) All device code gets loaded in the GPU whether it ends up being used or not. While having different models/solvers loaded in the binary is desired (so as to run different simulations without having to recompile, and so on), having all of them loaded at all times is not sustainable: the graphics card gets uselessly clogged and the memory available for the simulation lowers.

3) The overhead when solving the system in parallel is too great, making that whole part of the code 5 times slower than it should.

Solutions for these have already been thought, although there has been no time to deploy them. Problems 1) and 2) could be solved by making a separate, small application that behaves as an interface for the user with the core applebeat binary. It would have a list of ‘available extensions’ as well as ‘loaded extensions’, and the user would be able to load/unload extensions and add/remove them. When told to run the application,
our interface would see what needs to be compiled and/or linked, do it and then run `applebeat` itself. The second problem would need a more detailed analysis but inlining the Model interface would probably allow us to reduce the penalty.

Apart from the aforementioned, other small issues have been observed, loosely ordered from greater to lower importance:

1) The program should be updated to offer threading support and, specifically, ease the use of threads when printing by having some default (but possible to opt out of) behind-the-scenes threading.

2) The program should be updated to allow the use of CUDA streams to those users that want to use them, so that they can take full advantage of their hardware.

3) It should be noted that, since all fields are worked with together as a pack (called `State`), pipelining is not possible or simply much harder. While this is a problem inherent to the application’s design and it is hard to solve without some major complex redesign; it is still a problem worth noting.

4) Expand current debugging interface to offer more information and more control.

5) Create a timing interface to allow users to validate and benchmark their simulations.

6) Expand pFields to allow (ease) the use of adaptive grids.

7) Even though we know our constants will not be modified during the program’s execution, the current implementation of the Model does not take advantage of this fact. Some way to profit from const correctness might be possible and useful.

8) Current behaviour relies on the base class subobject being laid in memory right before the derived class’ members. This is not standard compliant (as seen in [13 Section 10, Clause 5]). A workaround that does not rely on this should be deployed so as to prevent problems in the future.
Part II

The Biophysical Model
Chapter 6

Theory

6.1 Biological background

Some cells are highly motile, and guide their movement direction following the concentration of extracellular chemicals: this process is called chemotaxis. In order to do so, organisms may use the spatial aspect of the chemical gradient, or the temporal signals produced when moving in it. Prokaryotic cells are much too small to process the spatial information, so they are left with the temporal component alone. They follow a process similar to a random walk that tumbles every so often, where the frequency of its tumbling is decreased when they encounter an increase in chemoattractant concentration [14]. Eukaryotic cells are larger, which allows them to process both [5]. Surprisingly sensitive, able of detecting concentration differences as small as 2-10% between front and back of the cell, we find this process to unfold in a very similar ways with mammalian neutrophils and the amoeba Dictyostelium discoideum [8] [9]. Chemotaxis can be broken down to three interrelated mechanisms: pseudopodia formation, polarization and directional sensing [15]. Here we shall concentrate on the mechanics underlying pseudopodia formation as well as polarization.

During the chemotaxis, the translocation from the cytosol to the membrane at the leading edge of the cell of several (fusion) proteins that contain the PH (Pleckstrin homology) domain, phosphatidylinositol-3,4-bisphosphate (PtdIns(3,4)P$_2$, abbreviated PIP2) and phosphatidylinositol (3,4,5)-trisphosphate (PtdIns(3,4,5)P$_3$, abbreviated PIP3) in particular, is shown to be a trigger involved in actin polymerization and pseudopod formation [16]. In D. discoideum, the leading edge of the membrane is rich in F-actin/PIP3/PI3K (phosphoinositide 3-kinase), while the retracting area is enriched in myosin II/PIP2/PTEN (phosphatase and tensine homolog) [17].

31
Figure 6.1: Schematic of reaction steps in the model. PI3K translocates from the cytosol to the membrane and catalyzes phosphorylation of PIP2 to PIP3 \[18\]. The model incorporates a positive feedback loop from PIP3 to PI3K that enhances this reaction, of unclear molecular identity but probably mediated by F-actin \[10\] \[19\]. The reverse reaction from PIP3 to PIP2 is catalyzed by phosphatase PTEN, which also translocates from the cytosol to the membrane by binding to PIP2 \[20\]. Image extracted from \[10\].

6.2 Biochemical model

In \[10\], a mathematical model is proposed to analyse the phosphoinositides’ signaling. It is shown in Equations 6.1–6.4 and it accounts for the following mechanics: how PI3K in the cytoplasm is recruited to the basal membrane and catalyzes PIP2 into PIP3 \[18\], the possible enhancement of said recruitment by the presence of PIP3 \[19\], how PTEN is recruited into the PIP2 sites of the membrane and mediates desphosphorylation of PIP3 into PIP2 \[20\], the constant production and degradation of PIP2 and the constant degradation of PIP3. A diagram of the cycle is shown in Figure 6.1.

\[
\frac{d[\text{PIP}_2]}{dt} = -k_1[\text{PI3K}_m][\text{PIP}_2] + k_2[\text{PTEN}_m][\text{PIP}_3] + k_4 - k_5[\text{PIP}_2] \quad (6.1)
\]

\[
\frac{d[\text{PIP}_3]}{dt} = k_1[\text{PI3K}_m][\text{PIP}_2] - k_2[\text{PTEN}_m][\text{PIP}_3] - k_3[\text{PIP}_3] \quad (6.2)
\]

\[
\frac{d[\text{PI3K}_m]}{dt} = k_6[\text{PIP}_3]^2[\text{PI3K}_c] - k_7[\text{PI3K}_m] \quad (6.3)
\]

\[
\frac{d[\text{PTEN}_m]}{dt} = k_8[\text{PIP}_2]^2[\text{PTEN}_c] - k_9[\text{PTEN}_m] \quad (6.4)
\]

Where the ‘c’ subindex refers to concentration in cytosol and the ‘m’ to concentration in membrane.

Assuming that the total amount of PI3K and PTEN (i.e.: the sum of its cytosolic and membrane fractions) remains constant and that phosphorylation and dephosphorylation
reactions are fast compared to the rest, the following is obtained [10]:

\[ \text{PI3K}_m = \frac{c_K[\text{PIP3}]^2}{k_7/k_6 + A^{-1}\int_\Omega [\text{PIP3}]^2 dx} \]  
(6.5)

\[ \text{PTEN}_m = \frac{c_P[\text{PIP2}]}{k_9/k_8 + A^{-1}\int_\Omega [\text{PIP2}] dx} \]  
(6.6)

Using these together with Equations 6.1-6.4 and including diffusion of phosphatidylinositols in the basal membrane, the following is obtained:

\[
\begin{align*}
\frac{\partial U}{\partial t} &= D_u \nabla^2 U - \frac{\alpha UV^2}{K_k + A^{-1}\int_\Omega V^2 dx} + \frac{\beta UV}{K_p + A^{-1}\int_\Omega U dx} + S - \gamma U \quad (6.7) \\
\frac{\partial V}{\partial t} &= D_v \nabla^2 V - \frac{\alpha UV^2}{K_k + A^{-1}\int_\Omega V^2 dx} + \frac{\beta UV}{K_p + A^{-1}\int_\Omega U dx} - \mu V \quad (6.8)
\end{align*}
\]

Where \( U = [\text{PIP2}], V = [\text{PIP3}], \alpha = c_Kk_1, \beta = c_pk_2, K_k = k_7/k_6, K_p = k_9/k_8, S = k_4, \gamma = k_5, \) and \( \mu = k_3. \)

### 6.3 Locomotion model

We may now proceed coupling these equations to a phase field [21], so that computations with a moving boundary can be performed in a correct and efficient way. The phase field, which shall be represented by the letter \( \phi \), is a field that evaluates as 1 inside the boundary of the cell’s basal membrane and 0 outside of it. The equation for its dynamics (i.e.: for the dynamics of the basal membrane’s shape) is provided by [10]:

\[
\tau \frac{\partial \phi}{\partial t} = \eta (\nabla^2 \phi - \epsilon^{-2} G'(\phi)) - M \left( \int_\Omega \phi dx - A_0 \right) |\nabla \phi| + (aV - bU)|\nabla \phi| \quad (6.9)
\]

Where \( G(\phi) = 18\phi^2(1-\phi)^2, A = \int_\Omega \phi dx \) and \( A_0 \) is the initial area. The first term has two parts, a first part accounting for surface tension, and a second term that stands for \( \phi \)’s bistable nature around 0 and 1. The second term is related to the conservation of the cell area and finally there is the last term accounts for excitation-deformation coupling, i.e.: how \( U \) and \( V \) affect the interface.

Finally, we shall introduce the phase field term into the equations on \( \partial U/\partial t \) and \( \partial V/\partial t \), and here we will deviate from [10, 22]. Firstly, using \( \partial(\phi U)/\partial t = \nabla (D\phi \nabla U) + \phi F_u(u) \) leads to \( \partial U/\partial t = \phi^{-1} \nabla \left( D\phi \nabla U \right) + F_u(u) - U \phi^{-1} \partial \phi/\partial t, \) whose latter term is a source for numerical instabilities. And secondly, the reason for that formulation is the conservation of the quantity \( \left( \int U dx \right) \) inside the area [21], which in our case is neither necessary nor true since when our boundary retracts, the cell is actually lifting that area of membrane from its base and moving it unto the outer surface. Therefore, the amounts of each
of these proteins (PIP2, PIP3...) are not actually conserved in the basal area of the membrane. In the end, this leads to the following equations:

\[
\frac{\partial U}{\partial t} = \nabla (D_u \phi \nabla U) - \phi \left( -\frac{\alpha UV^2}{K_k + A^{-1} \int_{\Omega} \phi V^2 dx} + \frac{\beta UV}{K_p + A^{-1} \int_{\Omega} \phi U dx} + S - \gamma U \right) \tag{6.10}
\]

\[
\frac{\partial V}{\partial t} = \nabla (D_v \phi \nabla V) - \phi \left( \frac{\alpha UV^2}{K_k + A^{-1} \int_{\Omega} \phi V^2 dx} - \frac{\beta UV}{K_p + A^{-1} \int_{\Omega} \phi U dx} - \mu V \right) \tag{6.11}
\]

Finally, to test the model a white gaussian noise term that represents the stochastic nature of the biochemical system will be added into the \( V \) equation, having the same quantity subtracted from the \( U \).
Chapter 7

Methods

As shown at the end of the previous chapter, the model we have worked with is:

\[
\frac{\tau \partial \phi}{\partial t} = \eta (\nabla^2 \phi - \epsilon^{-2}G'(\phi)) - M \left( \int_{\Omega} \phi dx - A_0 \right) |\nabla \phi| + (aV - bU)|\nabla \phi| \quad (7.1)
\]

\[
\frac{\partial U}{\partial t} = \nabla (D_u \phi \nabla U) - \xi(x,y,t) + \phi \left( \frac{\beta U V}{K_p + A^{-1} \int_{\Omega} \phi U dx} - \frac{\alpha U V^2}{K_k + A^{-1} \int_{\Omega} \phi V^2 dx} \right) + S - \gamma U \quad (7.2)
\]

\[
\frac{\partial V}{\partial t} = \nabla (D_u \phi \nabla U) + \xi(x,y,t) + \phi \left( \frac{\alpha U V^2}{K_k + A^{-1} \int_{\Omega} \phi V^2 dx} - \frac{\beta U V}{K_p + A^{-1} \int_{\Omega} \phi U dx} - \mu V \right) \quad (7.3)
\]

With \( G(\phi) = 18\phi^2(1 - \phi)^2 \), \( A = \int_{\Omega} \phi dx \), \( A_0 \) is the initial area and \( \xi \) is a white gaussian uncorrelated noise with zero mean \( < \xi(x,t) > = 0 \) and variance \( < \xi(x,t)\xi(x',t') >= \sigma^2 \delta(x - x')\delta(t - t') \). In the above equations, \( U \) is the concentration of PIP2, \( V \) the concentration of PIP3 and \( \phi \) the phase field.

In order to study its behaviour properly, different test setups have been devised.

7.1 Test Setups

7.1.1 Wave Properties

The first setup we will see is meant to help us understand the behaviour of the PIP3 waves within the cell matrix, having one of them propagate horizontally through a square space. In order to do so, the base model is simplified by having \( \sigma = 0 \) and \( \phi = 1 \) constant. Because the integral terms transform our local problem into a global one, making not
only simulations but also theoretical analysis on the equations much more difficult, a known simplification substitutes them by the local values. In here we will test both and compare them to see how much their behaviour differs. In the end, the following system is obtained:

\[
\frac{\partial U}{\partial t} = D_u \nabla^2 U - \frac{\alpha UV^2}{K_k + F_k(V)} + \frac{\beta UV}{K_p + F_p(U)} + S - \gamma U \quad (7.4)
\]

\[
\frac{\partial V}{\partial t} = D_v \nabla^2 V + \frac{\alpha UV^2}{K_k + F_k(V)} - \frac{\beta UV}{K_p + F_p(U)} - \mu V \quad (7.5)
\]

Where \(F_k(V) = V^2\) and \(F_p(U) = U\) in the case of the simplified model and \(F_k(V) = A^{-1} \int_{\Omega} V^2 dx\) and \(F_p(U) = A^{-1} \int_{\Omega} U dx\) in the other one (\(A\) is the total area of the space).

The initial conditions used for these tests are:

\[
U(x, y) = 7 \quad \forall x, y; \quad V(x, y) = \begin{cases} 
1.5 & x < 15 \\
0 & \text{otherwise} 
\end{cases} \quad (7.6)
\]

Since some parameter fitting and parameter space explorations have been made using this setup, a number of automated tools has been written to facilitate that task. In particular, an automated tool (wave_chars.py) to be used with the image output for the detection of the wavefront and obtaining its average speed and width has been made in Python. It follows the scheme below:

1. Wait for a small space to appear between the leftmost border and the wave.
2. Keep track of wave width and right flank position in each frame.
3. End when the right flank reaches the right border or there are no more frames.
4. Average recorded widths and compute average speed using starting and ending position of the right flank.

An script\(^1\) has been made that makes use of this tool to, given a reference set of constants, fit them so that desired wave width and speed are achieved, following a simple local search. This search is a very basic one, following this scheme:

1. For each constant \((\alpha, \beta, \gamma, S, \mu, K_k, K_p)\):
   1. Run simulations changing its value until either desired width is reached or distance from original parameter is larger than given threshold.
2. Change value in $D_v$ and $D_u$ until desired speed is reached or distance from original parameter is larger than given threshold.

3. Repeat from step 1 until both width and speed match desired values or number of iterations is greater than given limit.

Other scripts have also been written to perform other kinds of parameter exploration.

### 7.1.2 Excitation-deformation coupling

Another test case has been made, to check how different parameters of the model affect the interactions between the PIP2/PIP3 concentration dynamics and the shape and movement of the cell membrane’s interface. To that end, the model introduced is simplified by having $\sigma = 0$; and two different initial conditions are set so that we have a case where four circular waves are created in each direction of each axis, at half radius distance (which we shall refer to as central), and a case where a spiral wave is created:

\[
\phi(x, y) = 0.5 + 0.5 \tanh \frac{R - \sqrt{y^2 + x^2}}{0.25\epsilon}
\]

\[
U(x, y) = 7\phi(x, y)
\]

\[
V(x, y) = \begin{cases} 
1.5\phi(x, y) & \sqrt{(y-y_i)^2 + (x-x_i)^2} < 7.5 \text{ with } i := 1.4 \\
0 & \text{otherwise}
\end{cases}
\]

\[
\phi(x, y) = 0.5 + 0.5 \tanh \frac{R - \sqrt{y^2 + x^2}}{0.25\epsilon}
\]

\[
d = \min(H/2, 30);
\]

\[
U(x, y) = \begin{cases} 
0 & W/2 \leq x < W \text{ and } H/2 - d \leq y < H/2 \\
7\phi(x, y) & \text{otherwise}
\end{cases}
\]

\[
V(x, y) = \begin{cases} 
1.5\phi(x, y) & W/2 \leq x < W \text{ and } H/2 \leq y < H/2 + d \\
0 & \text{otherwise}
\end{cases}
\]

Where $R$ is our cell’s radius, $\epsilon$ the membrane width, $(x_i, y_i)$ the coordinates of the four wave origins and $H, W$ the height and width of our space respectively.
7.1.3 Influence of Noise

Finally, the complete model is used, and we check the influence of noise. In order to do so, the following initial conditions are used:

\[
\begin{align*}
\phi(x, y) &= 0.5 + 0.5 \tanh \frac{R - \sqrt{y^2 + x^2}}{0.25 \epsilon} \\
U(x, y) &= 7\phi(x, y) \\
V(x, y) &= 0
\end{align*}
\]  

(7.9)

Where \( R \) is our cell’s radius and \( \epsilon \) the membrane width. A threshold \( t_1 \) is also used, such that \( \phi(x, y) < t_1 \Rightarrow \sigma = 0 \), to prevent the creation of waves right in the outside of the basal membrane.

7.2 Numerical Methods

All of the above models have been solved using a finite differences approach and Euler’s method, with a discretization of \( \Delta x = 0.1 \mu m \) and \( \Delta t = 0.001 s \). The models that make use of a phase field also set a threshold value on \( \phi \) under which \( \frac{\partial U}{\partial t} \) and \( \frac{\partial V}{\partial t} \) both equal 0 directly, so as to save some time.

When generating the noise in the discrete system, the following is used:

\[
\text{noise} = \sigma \sqrt{\frac{2}{\Delta t \Delta x^2}} N(0,1)
\]  

(7.10)

From \( N(0,1) \) a normal distribution with zero mean and one standard deviation, where \( \sigma \) is a parameter that adjusts the noise intensity.
Chapter 8

Results

8.1 Wave Properties

Constants in Table 8.1 were used as a reference, extracted from [10]. However, simulations with these values lead to waves that qualitatively match those seen in experimental experiments [23] but do not quantitatively match, as shown in Table 8.2. Using a script to perform a local exploration, a set of constants that yield a wave front with the desired properties is obtained (Table 8.1).

Additionally, thousands of simulations have been performed to obtain the dependency of wave speed and width on each constant, with both the complete model (with integral terms) and the simplified one. The results from these simulations are presented in Figures 8.1, 8.2, 8.3.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Reference values [10]</th>
<th>Fitted values (simplified model)</th>
<th>Fitted values (base model)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D_u$</td>
<td>0.005</td>
<td>0.002222</td>
<td>0.002778</td>
</tr>
<tr>
<td>$D_v$</td>
<td>0.02</td>
<td>0.008889</td>
<td>0.011111</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
</tr>
<tr>
<td>$\beta$</td>
<td>0.4</td>
<td>0.4</td>
<td>0.4</td>
</tr>
<tr>
<td>$S$</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>0.015</td>
<td>0.015</td>
<td>0.015</td>
</tr>
<tr>
<td>$\mu$</td>
<td>0.1</td>
<td>0.04972</td>
<td>0.043634</td>
</tr>
<tr>
<td>$K_k$</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>$K_p$</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 8.1: The table shows the reference values extracted from [10] and the final values obtained from modifying the reference values to fit experimental data. All constants use $\mu m$ as distance unit and s for time, i.e.: $D_u$ and $D_v$ are in $\mu m^2/s$; $\alpha$, $\beta$, $\gamma$ and $\mu$ in $s^{-1}$; $S$ in $[s^{-1}]$; $K_p$ in $[]$ and $K_k$ in $[{}^2]$. 

39
Results

Figure 8.1: The values of the obtained waves’ speed and width are shown for different values of $\alpha$ and $\beta$. The blue series is obtained with the full model while the red series results form the simplified model, the red line is set at the value shown in experimental data. All other constants have the values shown in the second column of Table 8.1.

Figure 8.2: The values of the obtained waves’ speed and width are shown for different values of $\gamma$ and $\S$. The blue series is obtained with the full model while the red series results form the simplified model, the red line is set at the value shown in experimental data. All other constants have the values shown in the second column of Table 8.1.
Table 8.2: Wave width and speed figures obtained simulating with the reference constants and the real values shown by experimental data.

<table>
<thead>
<tr>
<th></th>
<th>Speed ($\mu$m/s)</th>
<th>Width ($\mu$m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference [10]</td>
<td>0.157</td>
<td>2.1</td>
</tr>
<tr>
<td>Fitted values (simplified model)</td>
<td>0.128</td>
<td>4.96</td>
</tr>
<tr>
<td>Fitted values (base model)</td>
<td>0.126</td>
<td>5.09</td>
</tr>
<tr>
<td>Experimental</td>
<td>0.125</td>
<td>5</td>
</tr>
</tbody>
</table>

Figure 8.3: The values of the obtained waves’ speed and width are shown for different values of $\mu$. The blue series is obtained with the full model while the red series results form the simplified model, the red line is set at the value shown in experimental data. All other constants have the values shown in the second column of Table 8.1.

8.2 Excitation-deformation coupling

Next we are taking a look at how the parameters on the equation of $\phi$ affect the movement of the interface. The parameters used are extracted from [10] and shown in Table 8.3. Using these given parameters realistic behaviour could not be achieved, that is why the value of $\tau$ used in the end is a different one, shown in Table 8.3 too.

Specifically, this will be a qualitative analysis on the effect of constants $\tau$, $M$ and $a$ and which values make the system behave realistically and how does its behaviour change when increasing or decreasing them. The results are shown on Figures 8.5 to 8.7. We can see $\tau$ affects directly at the interfaces’ willingness to follow the wave of PIP3. At low values, it follows the wave closer, and at higher it becomes more rigid; behaving much like a static field for really high values. $M$ affects how strictly the area is conserved. For
Results

<table>
<thead>
<tr>
<th>Constant</th>
<th>Reference value</th>
<th>Values used</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\eta$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$\tau$</td>
<td>8.3</td>
<td>85</td>
</tr>
<tr>
<td>$M$</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>$a$</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>$b$</td>
<td>0.03</td>
<td>0.03</td>
</tr>
<tr>
<td>$A_0$</td>
<td>78.6433</td>
<td>78.6433</td>
</tr>
<tr>
<td>$\epsilon$</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 8.3: Constants for the $\phi$ equation. Units are as follows: $\eta$ and $A_0$ in $\mu m^2$, $\tau$ in s, $M$ in $\mu m^{-1}$, $a$ and $b$ in $[\mu m]^{-1}$ and $\epsilon$ in $\mu m$.

Figure 8.4: Reference images taken at time $t = 22.5$ s using the constants shown in Table 8.3 under the column ‘Values used’.

Figure 8.5: All images were taken at $t = 22.5$ s of the simulation. Figures 8.5i, 8.5ii were calculated with $\Delta t = 0.0005$, the rest with the standard set of constants. Figures 8.5i, 8.5ii had spiral initial conditions, while the other two used the central initial conditions.

Lower values, we see the interface contracts more than would be appropriate, while for higher values the interface follows the wave too tightly, in its attempt to keep the area constant. On the other hand, the higher the value of $a$, the more weight $V$ will have in determining the shape of the cell. To summarize, all of them affect the rigidity of the interface, but each affects it in slightly different ways.
8.3 Influence of noise

Some simulations have been made, using different values for $\sigma$ and we have also kept a record for the average value of $V$ at each frame for every simulation (Figure 8.8). For high values of $\sigma$ the system stays in a saturated state permanently and for low values, we can see that the noise is not enough to excite the system. It is somewhere in between those two states that wave sources are created, producing a peak in average $V$. We have also checked that for values higher than 0.08 the simulation crashes, most probably due to the proliferation of spots with negative concentrations due to high intensity negative noise spikes.
Results

Figure 8.8: Images of simulations with different $\sigma$ and plot of the average value of $V$. For each value of $\sigma$, images of the simulation at times 0.15s, 150s and 300s are provided together with a plot of the average value of $V$ vs time. Some extra images are also provided, at the times where the plot peaks.
Chapter 9

Conclusions

The use in simulations of the simplified model (i.e.: without the integrals) is discouraged. It has been shown that the properties (wave speed, width) of the obtained system differ from those that would be obtained with the full system. While this can be fixed by using a different constant fitting, it is certainly not recommended to make parameter explorations using the simplified model, since the further the constants are from the point of experimental agreement, the further the behaviour deviates from the one that would be shown by the full model.

There has been no time to run enough simulations with noise and analyze them carefully, so the results shown in that section would be only preliminary and need to be completed.

All in all, the model when used together with the constants shown in Table 9.1 provides a realistic behaviour that agrees with experimental data. The next logical step would be to couple this model for the inner communication together with a suitable model for exterior sensing, so that more complete simulations of the movement of the *D. discoideum* when in an exterior gradient of chemoattractant can be obtained.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D_u$</td>
<td>$0.002778 \mu m^2/s$</td>
<td>$K_p$</td>
<td>$3 \mu m^{-1}$</td>
</tr>
<tr>
<td>$D_v$</td>
<td>$0.011111 \mu m^2/s$</td>
<td>$\eta$</td>
<td>$1 \mu m^2$</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>$1.2 \text{ s}^{-1}$</td>
<td>$\tau$</td>
<td>$85 \text{ s}$</td>
</tr>
<tr>
<td>$\beta$</td>
<td>$0.4 \text{ s}^{-1}$</td>
<td>$M$</td>
<td>$0.5 \mu m^{-1}$</td>
</tr>
<tr>
<td>$S$</td>
<td>$0.1 \text{ [ ] s}^{-1}$</td>
<td>$a$</td>
<td>$12 \text{ [ ]}^{-1} \mu m$</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>$0.015 \text{ s}^{-1}$</td>
<td>$b$</td>
<td>$0.03 \text{ [ ]}^{-1} \mu m$</td>
</tr>
<tr>
<td>$\mu$</td>
<td>$0.043634 \text{ s}^{-1}$</td>
<td>$\epsilon$</td>
<td>$1 \mu m$</td>
</tr>
<tr>
<td>$K_k$</td>
<td>$6 \text{ [ ]}^2$</td>
<td>$\sigma$</td>
<td>$0.004$</td>
</tr>
</tbody>
</table>

Table 9.1: Definitive constants used to obtain realistic simulations.
Chapter 10

Final Remarks

In the end we have created a framework that allows users with knowledge in C++ programming but little or no knowledge in CUDA and gpu-acceleration profit from the speedup offered by the graphics card in their physical simulations. Our framework also separates the physical model from the resolution method, providing a more comfortable environment that allows code reuse and provides a great help when many different yet similar simulations have to be performed.

Overall we feel this is a good asset that has proven itself useful during the resolution of the biophysical model for the chemotactic response of \textit{D. discoideum}. This is not to say that there is no room for improvement. The framework has some limitations: it is slightly slower than would be desired, and some algorithms (most prominently, those that rely on adaptive grids) cannot be implemented within it. Specific suggestions to bring the framework closer to its ideal state have been made, in Chapter 5.

The model used for the simulation of the inner signaling in the chemotactic response of \textit{D. discoideum} behaves in accordance to experimental data. Because the reference set of constants does not produce realistic behaviour, a great amount of time has been spent in their fitting. This, coupled with the fact that it took a long time to be able to get started with the simulations (to have the framework ready for them) means that we could not delve as deep as desired into the simulation results. Therefore, simulations shown here are only a preliminary result, and should be completed in the future.

And even further into the future, the model presented could be coupled to a system that simulates the sensory mechanisms that translate spatio-temporal gradients of chemoattractant into the quantities used in our model (PIP2, PIP3). Before that, however; a more detailed study of its behaviour with the current noise term (longer and more
Final Remarks

simulations) as well as with other kinds of noise (e.g.: spatially correlated) would be necessary.
Appendix A

Object Oriented Programming

Under these lines follows a list of terms and concepts related to object oriented programming and their definitions. These are brief descriptions meant to help readers follow with the first half of the document, for more detailed information looking at the information sources [11], [13] is recommended.

**Translation Unit**  A source file, together with its included files (and so on, recursively).

**Namespace**  A *namespace* is an environment that isolates symbols. This way, large projects may have more than one variable with the same name without having them mistaken for each other.

**Global Namespace**  When no namespace is specified, the *global* namespace is used, which is just another namespace to work in.

**Internal Linkage**  *Internal linkage* refers to symbols that are only accessible throughout one translation unit (the one they are defined in). Its opposite is *external linkage*, which refers to symbols that my be accessed beyond one particular translation unit.

**Integral type**  All of the following are considered fundamental integral types: *bool*, *char* and its extended length types and *int* and its unsigned and long counterparts.
**Enumeration**  Defined using the `enum` keyword, is a user-defined data type that is defined with a list of integral constants, each with a name. Variables of this type may take any of these names as value. E.g.:

```c
enum color { red, green, blue, yellow, black }
color x = red;
```

**Class**  A class is a data structure that may contain function members in addition to its data members. These members are declared within the bounds of an `access specifier` (i.e.: `public`, `protected` or `private`), that modifies the access rights of the members that follow. By default, members declared outside of an access specifier are treated as `private`.

**Struct**  Declared using the `struct` keyword, it is a class that defaults the access of its members to `public`.

**Object**  An object is an instance of a class.

**Access specifiers**  May be used both when declaring a class’ members or during inheritance, it is a keyword that modifies the accessibility of the following members, or of the members of the following classes. There are three kinds: public, protected and private. Public members are accessible from anywhere where the object is visible. Protected members are accessible from members of the same class, from its friends, and from members of derived classes. Finally, private members are accessible only from members of the same class and from their friends.

**Friendship**  The `friend` declaration grants another function or class access to private and protected members of the class where that declaration appears.

**Derived Classes**  Any class may be declared as derived from one or more other `base` classes, which in turn may or may not be derived from other classes, forming an `inheritance hierarchy`. Each base class is present within the derived class as a `base class subobject` and all its (non-private) members are accessible.

**Virtual Functions**  A virtual function is a class member function that can be overridden by its derived classes, so that calls to that function will call the overridden method instead.
**Override** By using the keyword **override**, one ensures the function is virtual and that it is overriding a virtual base class.

**VPointers** VPointers are elements existing in the implementation of the inheritance. When we have an object of a class that overrides virtual methods, a special member is created automatically by the compiler called *vpointer*. It is a pointer that points to a table (*vtable*) where the addresses of that object’s methods are stored, that way when the object is later accessed, maybe from a pointer to one of its base classes, the correct method will still be found in the table.

**Function Template** A *function template* is in fact a family of functions defined using generic types. Even if the function itself is written only once, an instance is created in the compiled files for each types it is used with. E.g.:

```cpp
template<typename T> void f(int x, T a){
    for (int i = 0; i < x; i++) cout << a << endl;
}
f(5,'a'); // Instantiates f<char>(int, char) and calls it
f(7,1.0); // Instantiates f<double>(int, double) and calls it
```

**Class Template** A *class template* is actually a family of generated classes using generic types, working in a way similar to that of function templates.

```cpp
template<typename T>
class vector {
    /* ... */
};

vector<int> v; //code for the vector<int> class will be generated
                //and 'v' will be an instance
vector<char> c;
```

**Static** The **static** keyword can be used when declaring:

- An object (except in a function parameter list).
- A function.
• An anonymous union.

And, depending on the context it is used in:

• If used in a declaration of a class member, it declares an *static* member. *Static* members are not bound to class instances: they are independent objects with *static* storage duration or functions defined in *namespace* scope, only once in the program.

• If used in a declaration of an object, it specifies *static* storage duration (except if accompanied by *thread_local*).

• If used in a declaration at *namespace* scope, it specifies internal linkage.

**Static Storage Duration**  When an object has static storage duration, it is allocated when the program begins and deallocated when the program ends and only one instance of the object exists. All objects declared at *namespace* scope (including *global namespace*) have this storage duration, plus those declared with *static* or *extern*. 
Appendix B

GPU Programming and nVidia CUDA

This introduces some concepts used in GPU Programming and more specifically nVidia CUDA. It is meant as a brief help, but for a more comprehensive explanation see [24], and for more detailed and technical information, [12].

Host/Device Paradigm  When programming GPU-accelerated applications, we have to deal with two (or more) computing units: the cpu (host) and the gpu (device). These two work with different memory spaces, having no vision and little control over each other’s memories. Hence, since our program is bound to start on the host, we will need to copy whatever data we need from this space into the device, and then we can move there and run the device calculations. Once the calculations are done we will need to copy data once more, from the device to the host in order to be able to see it.

Basic structure of a graphics card  Graphics cards are able to do many things in parallel. This parallel structure is not without some organization, the levels are as follows:

- Thread, a sequential set of instructions.

- Warp, a set of threads that run concurrently in the same multiprocessor in a Single Instruction, Multiple Data (SIMD) fashion.

- Block, a set of threads that can synchronize with each other, organized in a 3D matrix.
• Grid, a group of blocks organized in a 3D matrix. No synchronization is possible between blocks.

**Warp divergence** Because threads in a warp run in a SIMD fashion, a problem called *warp divergence* arises. Basically, if there is a divergence in the program flow (for example, there is an `if` clause and as a consequence some threads in the warp run one piece of code and other threads run another), there is a decrease in parallelism.

**Kernels** Functions called from the host but run in the device are called *kernels*. When launching a kernel, a distribution of threads must be given (e.g.: 'how many threads to launch per block? how many blocks?').

**Asynchrony** While the GPU is busy running device code, the CPU keeps running its host code by default, until a function that forces synchronization is reached. Data transfers between host and device are managed by an independent element and can occur at the same time too. Therefore, when working with GPU-accelerated applications, 3 major working devices are used: the CPU, the copy engine and the GPU. Some graphics cards have two copy engines instead of one: one for each direction (that way you can copy from host to device and from device to host simultaneously).

**CUDA Streams** A *stream* is a sequence of operations that is done in device, in the order issued by the host. The order in which operations in different streams occur is not defined. By default, calls to CUDA functions as well as kernel launches are all done in the *default stream*. The *default stream* is a synchronizing stream: no operation in this stream will begin until all operations from all other streams are finished, and an operation in the default stream must complete before any other operation in any stream can continue.
Appendix C

Computer Specifications

<table>
<thead>
<tr>
<th>Processor</th>
<th>Intel (R) Core(TM)2 Duo CPU E7300: 2.66GHz, 2 cpus</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache</td>
<td>32K+32K</td>
</tr>
<tr>
<td>L2 cache</td>
<td>3072K</td>
</tr>
<tr>
<td>RAM</td>
<td>1GiB DIMM + 2GiB DIMM</td>
</tr>
<tr>
<td>Graphics Card</td>
<td>nVidia GeForce GT 630 Rev. 2 (GK208)</td>
</tr>
</tbody>
</table>

Table C.1: Hardware specifications of the small working station used to run the performance checks

<table>
<thead>
<tr>
<th>Processor</th>
<th>Intel (R) Core(TM) i4770k: 3.5GHz, 8 cpus</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache</td>
<td>32K+32K</td>
</tr>
<tr>
<td>L2 cache</td>
<td>256K</td>
</tr>
<tr>
<td>L3 cache</td>
<td>8192K</td>
</tr>
<tr>
<td>RAM</td>
<td>15GiB</td>
</tr>
<tr>
<td>Graphics Card</td>
<td>nVidia Quadro M4000 Rev. a1 (13f1)</td>
</tr>
</tbody>
</table>

Table C.2: Hardware specifications of the working station used to run the simulation batches starred in the second part of the project.
Bibliography


