


UNIVERSITÀ DELLA CALABRIA

UNIVERSITÀ DELLA CALABRIA

Dipartimento di Matematica e Informatica

Dottorato di Ricerca in Matematica e Informatica

XXVII CICLO

3D WEB-BASED PARALLEL APPLICATIONS FOR THE NUMERICAL MODELING OF NATURAL PHENOMENA

Settore Disciplinare INF/01 – INFORMATICA

Coordinatore: Ch.mo Prof. Nicola Leone

Supervisori: Prof. Donato D'Amborsio

Prof. William Spataro

Dottorando: Dott. Roberto Parise

Alla mia famiglia

Abstract

In this thesis, I designed and implemented three new web applications tailored for the Cellular Automata (CA) simulation models SCIDDICA-k1, SCIARA-fv3 and ABBAMPAU, making use of the Google Web Toolkit framework and WebGL.

Moreover, I have contributed to the optimizations of the numerical models mentioned above and I also developed part of a library, called OpenCAL, for developing CA simulation models in C/C++. In this case, my most significant contribution regarded the support given to the parallelization through the OpenCL standard, in order to facilitate with a few lines of codes, the parallelization for the execution on any device, especially on General Purpose Computation with Graphics Processing Units (GPGPU).

The development of the web applications involved the implementation of strategies so that optimizing the server load in the connections' management and enhancing the real time visualization of maps on devices of any kind, even mobile.

As regards the OpenCAL library, the tests performed on a test models has shown significant performance improvements in terms of speedup, thanks also to the use of some new optimization strategies. In this way, the validity of the use of graphics processing units as alternative to more expensive hardware solutions for the parallelization of CA models has been confirmed.

Sommario

In questo lavoro di tesi ho progettato e implementato tre web application per i modelli di simulazione ad Automi Cellulari SCIDDICA-k1, SCIARA-fv3 e ABBAMPAU, utilizzando il framework Google Web Toolkit e WebGL.

Inoltre, ho contribuito ad alcune ottimizzazioni dei modelli numerici sopra citati e ho sviluppato parte di una libreria, chiamata OpenCAL, per lo sviluppo di modelli di simulazione ad Automi Cellulari in C/C++. Il mio contributo piú significativo ha riguardato la parallelizzazione della libreria in OpenCL per consentire una parallelizzazione semplificata dell'automa cellulare rispetto all'impoego diretto di OpenCL e l'esecuzione su device eterogenei, in particolar modo su schede grafiche per il calcolo general-purpose (General Purpose Computation with Graphics Processing Units - GPGPU).

Lo sviluppo delle web application ha coinvolto l'applicazione di strategie per ottimizzare il carico dei server nella gestione delle connessioni e per rendere piú performante la visualizzazione in tempo reale delle mappe su qualsiasi tipo di dispositivo, anche mobile.

Per quanto riguarda la libreria OpenCAL, gli esperimenti effettuati su alcuni modelli base mostrano significativi miglioramenti nelle performance in termini di speedup, grazie anche all'utilizzo di alcune strategie d'ottimizzazione nuove, confermando la validitá dell'uso di processori grafici come alternativa a soluzioni hardware classiche, generalemente piu‘ costose, per la parallelizzazione di modelli ad Automi Cellulari.

Contents

1 Introduction

From the early days of Computer Science, one of the most important purposes of computers was the simulation of the natural phenomena. With these simulations, man has the total control of the reproduced "world" [37] and he can test his theories and assumptions by comparing it to the reality. Indeed, the execution speed of computers has allowed the use of *numerical simulation* as an instrument to solve complex equations systems, with which scientists can "model" the *complex phenomena* of reality. Nowadays, these simulations gain a particular importance because they are not only used to test theories and assumptions, but also to prevent, in medium and long-term, the evolution over time of a *complex system*; in this context the prevention of natural disasters, weather forecasts, evaluations of financial trends and so on are placed. The application fields of simulations are countless; for example, they can be used for the vocational education of those professionals for which the training in a real environment would lead to safety problems or costs (such as in pilots' training).

However, many real systems are not suitable for an immediate and natural "classical" modeling, based on a first analytical and deductive phase, followed by a later resolution through numerical methods. In complex systems' simulation the computational paradigm of *Cellular Automata* [71] is widely adopted, which represent, in some contexts, an alternative method to systems in which are used differential equations, especially employed in Physics to describe the laws that rule the evolution of a phenomenon. According to the Cellular Automata modeling approach, a system can be considered as

composed of several simple elements, each one evolves following purely local laws. The global evolution of the system then "emerges" from the evolution of all constituent elements. Cellular Automata are successfully used to simulate complex systems in several fields, like Computational Fluid Dynamics, Artificial Life, Molecular Dynamics, Biology, Genetics, Chemistry, Geology, Cryptography, Financial world,Territorial Analysis, modeling of road traffics, image processing.

In the execution of the simulations it is often necessary to perform a considerable quantity of computations. It may then happen that, in some contexts, computers take excessively long processing times, making the simulations useless for practical purposes. For this reason, it is required to speed up the simulations; therefore, it is useful to resort on parallel computers, composed by many processing units able to run many programs simultaneously, in order to solve different problems or execute a single problem taking less time.

However, the use of high-performance parallel computers (supercomputers) is not always possible for who needs to simulate complex phenomena such as certain geological phenomena that are of particular interest in order to prevent disasters. In Italy, for example, it would be desirable to be able to simulate in advance the effects of landslides that could hit the territory, which is particularly exposed to this kind of phenomena.

Usually, softwares allow an interactive simulation and integrate a 2D and/or 3D visualization system. In most cases, however it is sequential software, despite the current technological development can permit to have access to parallel systems with high computational capacity at low cost (compared to the past). This is true especially in the case of General-Purpose Computation with Graphics Processing Units (GPGPU), which uses the graphics cards for computational purposes. Generally, for Desktop Applications computation and visualization are combined in a single software. In some cases, however, it is possible to find examples of client-server applications where the computational part is independent from the graphical interface and the visualization system; the computation can therefore be performed remotely, possibly on supercomputers. A further development is given by the most recent Web technologies, thanks to which it is possible to realize on one hand graphical interfaces comparable, if not better, to classical ones, and on the other hand complete and functional interactive systems for 2D and 3D scientific visualization. Furthermore, web applications improve the level of usability of the applications itself and remove those problems related to software download. Besides, the end user does not have to worry about where the computation is executed, nor the details of the computational model used.

The work done in this dissertation is placed in this latter context, where

supercomputing and Web 2.0 converge in order to realize new applications for computational models for the simulation of complex systems. For this purpose, three Cellular Automata numerical models and three respective web applications have been implemented, each one with an interactive 3D visualization system, implemented using WebGL. The implemented models are: SCIDDICA-k1 with the Swii2 web application for debris flows simulation, SCIARA-fv3 with SciaraWii for lava flows and ABBAMPAU with Awii to display and control the evolution of wildfire. Instead, as regards the aspects related to the increase of performance, a significant contribution to the development of a new library for Cellular Automata has been carried out, especially to its parallelization in OpenCL for the execution on GPUs.

The thesis is organized as described below. The second Chapter focuses on different theoretical arguments and the presentation of some simulation models: the Cellular Automata with their most important theoretical results and some common applications; their specifically application for modelling and simulating some natural complex phenomena; the algorithm for the minimisation of differences [19]; there is also an overview on GPGPU techniques like CUDA, OpenCL and OpenACC; a general description of the innovations introduced by the WEB 2.0; finally, the AJAX development method, its applications and the resultant innovative effects. The third Chapter shows our latest release on lava flows simulations, debris flow simulations and wildfire simulations, SCIARA-fv3, SCIDDICA-k1 and ABBAMPAU, respectively and the structures of SCIDDICA, SCIARA and ABBAMPAU Web Interactive Interfaces are described, with some applications and the main differences among them. Software architecture is described, as well as the 3D visualizations systems based on WebGL. The fourth Chapter focuses on the description of the OpenCAL library and the advantages by using its OpenCL support. Then, some implementations and performance analysis are reported. The last Chapter concludes with general discussions and directions for future work.

2

A brief overview of Cellular Automata, GPGPU and Web 2.0

2.1 Cellular Automata

Nowadays most of natural phenomena are well described and known, thanks to the effort of scientists that studied the basic physic's laws for centuries; for example, the freezing of water or the conduction that are well know and qualitative analyzed. Natural systems are usually composed by many parts that interact in a complex net of causes/consequences that is at least very difficult but most of the times impossible to track and to describe. Even if the single components are each very simple, extremely complex behavior emerge naturally due to the cooperative effect of many components. Much has been discovered about the nature of the components in natural systems, but little is known about the interactions that these components have in order to give the overall complexity observed. Classical theoretical investigations of physical system have been based on mathematical models, like differential equations, that use calculus as tool to solve them, which are able to describe and allow to understand the phenomena, in particular for those that can be described by which are linear differential equation[1] that are easy solvable with calculus. Problems arise when non-linear differential equations come

[1]Some electro-magnetism phenomena, for instance, can be described by linear differential equations.

out from the modellation of the phenomena, like fluid turbulence[2]. Classical approaches usually fails to handle these kind of equations due to the elevated number of components, that make the problem intractable even for a computer based numerical approach. Another approach to describe such systems is to distill only the fundamental and essential mathematical mechanism that yield to the complex behavior and at the same time capture the essence of each component process.

Figure 2.1: A 3D cellular automaton with toroidal cellular space.

Cellular Automata (CA) are a candidate class of such systems and are well suitable for the modellation and simulation of a wide class of systems, in particular those ones constructed from **many identical** components, each (ideally) simple, but together capable of complex behaviour [65] [66]. In literature there are lots applications of Cellular Automata in a wide rage of class problems from gas [25] and fluid turbulence [62] simulation to macroscopic phenomena [30] like epidemic spread [59], snowflakes and lava flow [15] [60]. CA were first investigated by S. Ulam working on growth of crystals using lattice network and at the same time by Von Neumann in order to study self-reproduction [70]; it was not very popular until the 1970 and the famous Conway's game of life [13], then was widely studied on the theoretical viewpoint, computational universality were proved[3] [64] and then mainly utilised, after 1980's, as a parallel model due to its intrinsically parallel nature implemented on parallel computers [46].

[2]Conventionally described by Navier-Stokes differential equations.

[3]CA is capable of simulating a Turing machine, i.e. is capable of computing every computable problems (Church-Turing thesis). For instance, Game of life was proved to be capable of simulating logical gates (with special patterns as *gliders* and *guns*).

2.1.1 Informal Definition

Informally, a cellular automaton is a mathematical model that consists of a discrete lattice of sites and a value, the state, that is updated in a sequence of discrete timestamps (steps) according to some logical rules that depend on a neighbor sites of the cell. Hence CA describe systems whose the overall behavior and evolution of the system may be exclusively described on the basis of local interactions [75]. The most stringent and typical characteristic of the CA-model is the restriction that the local function does not depend on the time t or the place i: a cellular automaton has homogeneous space/time behavior. It is for this reason that CA are sometimes referred to as *shift-dynamical* or *translation invariant* systems. From another point of view we can say that in each lattice site resides a finite state automaton[4] that take as input only the states of the cells in its neighborhood (see figure 2.4).

2.1.1.1 Cellular space dimension and geometry

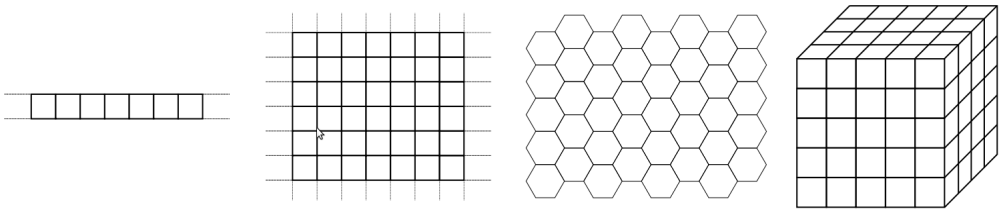
The cellular space is a *discrete* d-dimensional lattice of sites (see figure 2.2). For 1-D automaton the only way to discretize the space is in a one-dimensional grid. For automaton with dimensionality higher than 1 the shape of each cell can be different than squared. In 2D tessellation for example each cell can be hexagonal or triangular instead of squared. Each tessellation present advantages and disadvantages. For instance the squared one does not give any graphical representation problem[5], but present problems of anisotropy for some kind of simulations[6] [25]. An hexagonal tessellation can solve the anisotropy problem [73] but presents obvious graphical issues. Often, to avoid complications due to a boundary, periodic boundary conditions are used, so that a two-dimensional grid is the surface of a torus (see picture 2.1).

[4] A simple and well know computational model. It has inputs, outputs and a finite number of states (hence a finite amount of memory); An automata changes state at regular time-steps.

[5] Each cell could be easily mapped onto a pixel.

[6] The HPP model for fluid simulation was highly anisotropic due to the squared tessellation.

Figure 2.2: Examples of cellular spaces. (a) 1-D, (b) 2-D squared cells, (c) 2-D hexagonal cells, (d) 3-D cubic cells.



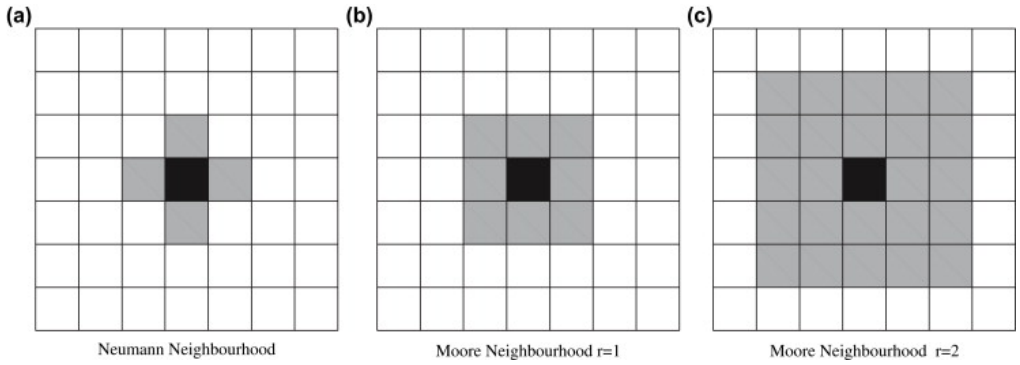
2.1.1.2 Neighborhood

The evolution of a cell's state is function of the states of the neighborhood's cells. The geometry and the number of cells that are part of the neighborhood depends on the tessellation type, but it has to have three fundamental properties:

1. **Locality**. It should involve only a "limited" number of cells.
2. **Invariance**. It should not be changed during the evolution.
3. **Homogeneity**. It has to be the same for each cell of the automaton.

Typically neighborhood "surrounds" the central cell. For 1-D cellular automata its borders are identified with a number r called *radius* [74]. A $r = 2$ identify $n = 2r+1$ cells in a 1D lattice: the central cell plus the right and left cells. Typical 2D cellular space neighborhood are the those of Moore and von Neumann neighborhood. The number of cells in the Moore neighborhood of range r is the odd squares $(2r + 1)^2$, the first few of which are 1, 9, 25, 49, 81, and so on as r is increased. Von Neumann's one consist of the central cell plus the cell at north, south, east, and west of the central cell itself. Moore's $(r = 1)$ one add the farther cells at north-east, south-east, south-west and north-west (see figure 2.3).

Figure 2.3: Examples of different kind of neighborhood with different radius values.



2.1.1.3 Transition Function

The evolution of the cell's state is decided by the transition function that is applied at the same time and on each cell. Usually the transition function is deterministic and defined by a *look-up* table only when the total number of state for each cell is small[7] otherwise is defined by an algorithmic procedure. It may be probabilistic, in the case of stochastic cellular automata.

2.1.2 Formal Definition

Cellular automata are dynamic models that are discrete in time, space and state. A simple cellular automaton A is defined by a lattice of cells each containing a finite state automaton, so we briefly give its definition.

2.1.2.1 Finite State Automaton

Also known as deterministic finite automata (DFAs) or as deterministic finite state machines, ther are one of the most studied and simple known computational models . It is a theoretical model of computation[8] that can be in a finite number of states, only one at a time, the current state. Its state can change in response of inputs taken by a transition function that describe the state change given the current state and the received input of the automata. They are much more restrictive in their capabilities than a Turing machines [9], but they are still capable to solve simpler problems, and hence to recog-

[7]Otherwise the dimension of that table would be enormous because the number of entries is exponential in the number of states.

[8]Language recognition problem solvers.

[9]For example we can show that is not possible for an automaton to determine whether the input consist of a prime number of symbols.

nize simpler languages, like well parenthesized string; More in general they are capable to recognize the so called *Regular languages*[10], but they fail for example in parsing *context-free* languages. More formally a DFA is a 5-tuple:

$$M =< Q, \Sigma, \delta, q_0, F >$$

- Q is a finite, nonempty, set of states.
- Σ is the alphabet
- $\delta : Q \times \Sigma \longmapsto Q$ is the transition function (also called next-state function, may be represented in tabular form (see table 2.1)
- q_0 is the initial (or starting) state : $q_0 \in Q$
- F is the set, possibly empty, of final states : $F \subseteq Q$

A run of DFA on a input string $u = a_0, a_1, \ldots, a_n$ is a sequence of states $q_0, q_1, \ldots, q_n$ s.t. $q_i \overset{a_i}{\longmapsto} q_{i+1}$, $0 \leq i \leq n$. It means that for each couple of state and input the transition function deterministically return the next DFA's state $q_i = \delta(q_{i-1}, a_i)$. For a given word $w \in \Sigma^*$ the DFA has a unique run (it is deterministic), and we say that it **accepts** w if the last state $q_n \in F$. A DFA recognizes the language L(M) consisting of all accepted strings.

δ	a	b	c	d	e
q_0	q_0	q_0	q_2	q_1	q_1
q_1	q_1	q_3	q_1	q_1	q_1
q_2	q_3	q_2	q_2	q_0	q_1
q_3	q_0	q_1	q_1	q_0	q_1

Table 2.1: Tabular representation of a DFM's next-state function

Figure 2.4 is an example of DFA[11]. It accepts the language made up of strings with a number N s.t $N \; mod \; 3 = 0$

- $\Sigma = \{a, b\}$
- $Q = \{t_0, t_1, t_2\}$
- $q_0 = t_0$

[10]Languages defined by regular expressions and generated by regular grammar, Class 3 in Chomsky classification. We can prove that for each language L accepted by a DFA exists a grammar L_G s.t. $L = L_G$

[11]Graph representation is the most common way to define and design DFA. Nodes are the states, and the labelled edges are the possible states transition from a state u to a state w given a certain input. Note that, because the automaton is deterministic is not possible for two edges to point to two different nodes if same labelled.

- $F = \{t_0\}$

If we execute the DFA on an input string S={aaabba} we can see that at time t=0 the DFA is in the initial state t_0 and the first symbol of S is read. The transition function is applied once per each symbol is S (i.e. $|S|$). The only rule that match the current state and input is $\delta = (t_0, a) = t_1$ hence the new state is t_1. The DFA accept the string only if there is not any input left and the current state is the final state q_f[12]. S is not accepted by the DFA defined in the example 2.4 because at the end of the computation the reached state is t_1 that is not a final state.

$$t_0 \overset{\delta(t_0,a)}{\longmapsto} t_1 \overset{\delta(t_1,a)}{\longmapsto} t_2 \overset{\delta(t_2,a)}{\longmapsto} t_0 \overset{\delta(t_0,b)}{\longmapsto} t_0 \overset{\delta(t_0,b)}{\longmapsto} t_0 \overset{\delta(t_0,a)}{\longmapsto} t_1$$

On the input $S^1 = \{ababbb\}$ the DFA accept:

$$t_0 \overset{\delta(t_0,a)}{\longmapsto} t_1 \overset{\delta(t_1,b)}{\longmapsto} t_1 \overset{\delta(t_1,a)}{\longmapsto} t_2 \overset{\delta(t_2,b)}{\longmapsto} t_2 \overset{\delta(t_2,a)}{\longmapsto} t_0 \overset{\delta(t_0,b)}{\longmapsto} t_0 \overset{\delta(t_0,b)}{\longmapsto} \mathbf{t_0}$$

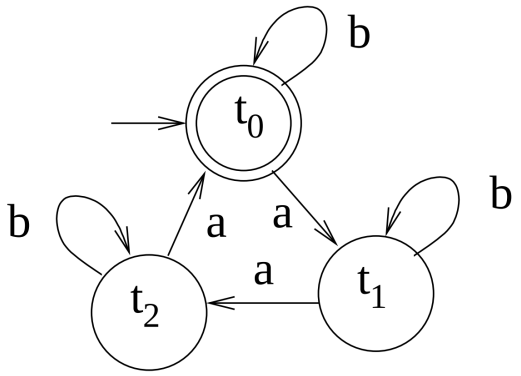


Figure 2.4: Graph representation of a DFA

2.1.3 Homogeneous Cellular Automata

Formally a CA A is a quadruple $A =< Z^d, X, Q, \sigma >$ where:

- $\mathbb{Z}^d = \{i = (i_1, i_1, \ldots, i_d) \mid i_k \in \mathbb{Z}, \forall k = 1, 2, \ldots, d\}$ is the set of cells of the d-dimensional Euclidean space.

[12]Previously we stated that F was a set but we can assume that there is only one final state ($|F| = 1$), because it is easy prove that exist a DFA with only one final state given a generic DFA ($|F| \geq 1$). We add one more state q_f and for each final state $q_i \in F$ we define new rules of the type $\delta(q_i, *) = q_f, * \in I$.

- X is the neighborhood, or neighborhood template; a set of m d-dimensional vectors (one for each neighbor)

$$\xi_j = \{\xi_{j1}, \xi_{j2}, \ldots, \xi_{jd}\} \ , \ 1 \leq j \leq m$$

that defines the set of the neighbors cells of a generic cell $i = (i_1, i_1, \ldots, i_d)$

$$N(X, i) = \{i + \xi_0, i + \xi_2, \ldots, i + \xi_d\}$$

where ξ_0 is the null vector. It means that the cell i is always in its neighborhood and we refer to it cell as *central cell* (see example below).

- Q is the finite set of states of the elementary automaton EA.

- $\sigma = Q^m \rightarrow Q$ is the transition function of the EA. σ must specify $q_k \in Q$ as successor state of the central cell. If there are m cells in the neighborhood of the central cell including itself, then there are $|Q|^m$ possible neighborhood's state configuration. It means that there are $|Q|^{|Q|^m}$ possible transition functions. Plus we can see that the tabular definition of the next-state function is unsuitable for practical purpose. It should have $|\sigma| = |Q|^m$ entries, an exceedingly large number.

- $\tau = C \longrightarrow C \longmapsto \sigma(c(N(X, i)))$ where $C = *cc$: $Z^d \rightarrow Q$ is called the set of the possible configuration and $C(N(X, i)))$ is the set of states of the neighborhood of i.

For example consider a 2D cellular automata with Moore neighborhood and a generic cell c=(10,10) and $|Q| = 5$ possible state for each cell .

$$X = \{\xi_0, \xi_1, \xi_2, \xi_3, \xi_4, \xi_5, \xi_6, \xi_7, \xi_8\} =$$

$$= \{(0,0), (-1,0), (0,-1), (1,0), (0,1), (-1,-1), (1,-1), (1,1), (-1,1)\}$$

Hence the set of the cells belonging to the neighborhood(defined by X) of c=(10,10) is: $V(X, c) = \{(0,0) + c, (-1,0) + c, (0,-1) + c, (1,0) + c, (0,1) + c, (-1,-1) + c, (1,-1) + c, (1,1) + c, (-1,1) + c\}$

$$= \{(10,10), (9,10), (10,9), (11,10), (10,11), (9,9), (11,9), (11,11), (9,11)\}$$

and the total number of entries for the tabular definition of the transition-function is $|Q|^{|X|} = 5^9 = 1953125$ and the total number of possible transition functions is $|Q|^{|Q|^{|X|}} = 5^{5^9} = 5^{1953125}$.

Table 2.2: Encoding of a transition function for a generic elementary CA. On the right the instance 110.

$F(1,1,1) = \{0,1\}$		$F(1,1,1) = 0$
$F(1,1,0) = \{0,1\}$		$F(1,1,0) = 1$
$F(1,0,1) = \{0,1\}$		$F(1,0,1) = 1$
$F(1,0,0) = \{0,1\}$	$\xrightarrow{instance}$	$F(1,0,0) = 0$
$F(0,1,1) = \{0,1\}$		$F(0,1,1) = 1$
$F(0,1,0) = \{0,1\}$		$F(0,1,0) = 1$
$F(0,0,1) = \{0,1\}$		$F(0,0,1) = 1$
$F(0,0,0) = \{0,1\}$		$F(0,0,0) = 0$

2.1.4 Theories and studies

2.1.4.1 Elementary cellular automata

The most simple AC we can imagine is elementary cellular automata [74]. They are one-dimensional periodic N cells array $\{C_i \mid 1 \leq i \leq N, C_i \in \{0,1\}\}$ each with 2 possible state (0,1), and rules that depend only on nearest neighbor value hence a radius r=1 neighborhood with a total number of involved cell $2r+1 = 2 \times 1 + 1 = 3$ (central, right and left cells). Since there are only $2 \times 2 \times 2 \times = 2^{2r+1} = 2^3 = 8$ possible states for the neighborhood of a given cell there are a total of $2^{2^3} = 2^8 = 256$ possible elementary automata (each of which may be identified with a 8-bit binary number [76]).

Wolfram's code

The transition function is $F(C_{i-1}, C_i, C_{i+1})$ is defined by a look-up table of the form stated in table 2.2, and an example of an instance of a function is given (rule 110, an important rule on which [14] proved universal computational power, as Wolfram had conjectured in 1985, and is arguably the simplest Turing complete system [76]) in table 2.2.

More generally Wolfram's code [74, 76] can be calculated conventionally of neighborhoods that are sorted in non-decreasing order , (111=7), (110=6), (101=5) etc., and the may be interpreted as a 8-digit number

$$01101110 = 2^0 \times 0 + 2^1 \times 1 + 2^2 \times 1 + 2^\times 1 + 2^4 \times 0 + 2^5 \times 1 + 2^6 \times 1 + 2^7 \times 0 = 110$$

1. List and sort in decreasing numerical (if interpreted as number) order all the possible configuration of the neighborhood of a given cell.

2. For each configuration, list the state which the given cell will have,

according to this rule, on the next iteration.

3. Interprets the resulting list as binary number and convert it to decimal. That is the Wolfram's code.

Note that it is not possible to understand from a code which is the size or the shape of the neighborhood. It is tacit to suppose that this information is already known.

2.1.4.2 Wolfram's classification

Mathematical analysis of CA may be not so straightforward despite their simple definition. A first attempt to classify CA was attempted by Wolfram [76]. He proposed a set of four classes for CA classification that are the most popular method of CA classification, but they suffer from a degree of subjectivity. Classification is based only on visual valuations, that are obviously subjective. A more rigorous definition of these classes is given in [13] [36]. Here the four Wolfram's classes.

1. these CA have the simplest behavior; almost all initial conditions result in the same uniform initial state (homogeneous state).

2. different initial conditions yield different final patterns, but these different patterns consist of an arrangement of a certain set of structures, which stays the same forever or repeats itself within a few steps(periodic structures).

3. behavior is more complicated and appears random, but some repeated patterns are usually present (often in the form of triangles)(chaotic pattern).

4. in some respects these are the most complicated class; these behave in a manner somewhere in between Class II and III, exhibiting sections both of predictable patterns and of randomness in their pattern formation(complex structures).

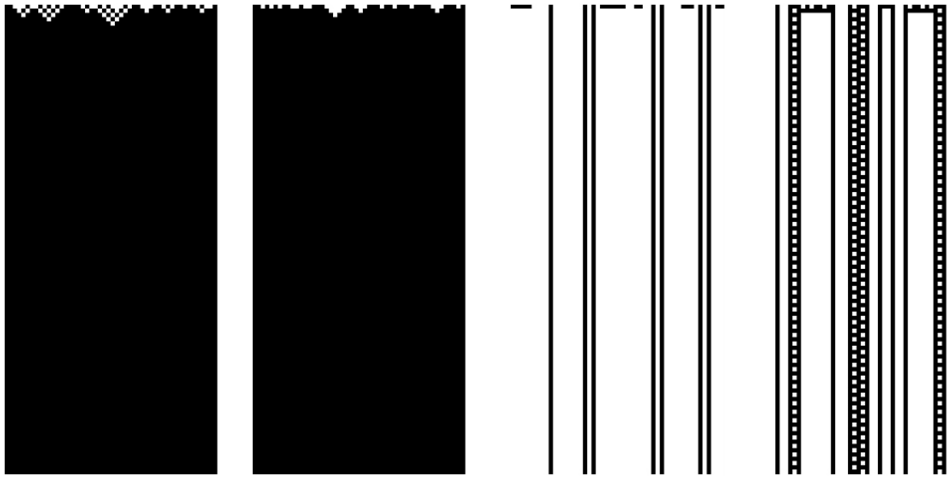
He observed that the behavior of a meaningful class of Cellular Automata by performing computer simulations of the evolution of the automata starting from random configurations. Wolfram suggested that the different behavior

[13] They prove that decide the class(from the wolfram's four one) of membership of a generic CA is an undecidable problem. Is not possible to design an algorithm that solve this problem.

of automata in his classes seems to be related to the presence of different types of attractors.

In figures 2.5 and 2.6 some elementary automata divided in their classes.[14]

Figure 2.5: Class 1 (a,b) and 2 (c,d) elementary cellular automata



(a) Rule 250 (b) Rule 254 (c) Rule 4 (d) Rule 108

We can well see from these examples that automata from class 1 have all cells ending up very quickly with the same value, in a homogeneous state and automata from class 2 with a simple final periodic patterns. Class 3 appear to be chaotic and non-periodic and automata from class 4 have a mixed behaviour, complex-chaotic structures are locally propagated.

2.1.4.3 At the edge of Chaos

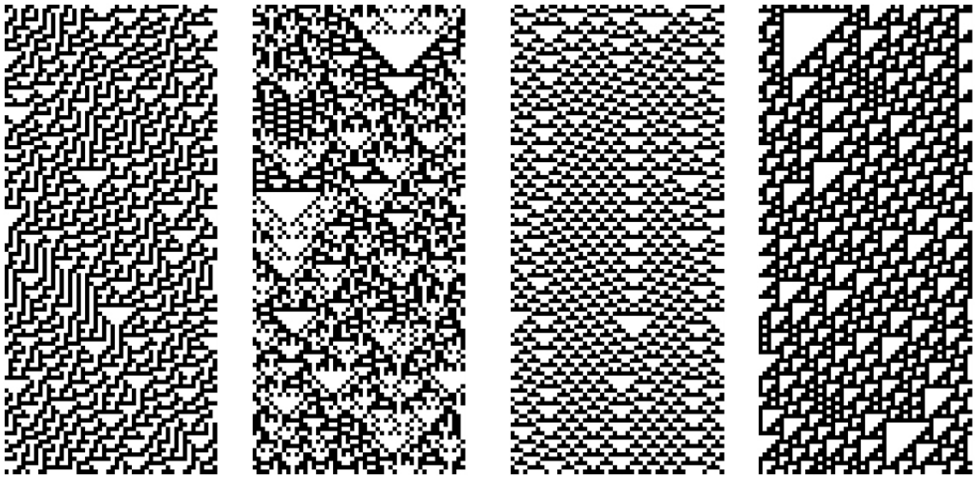
Class 4 automata are at *the edge of chaos* and give a good metaphor for the idea that the *interesting* complexity (like the one exhibit by biological entities and their interactions or analogous to the phase transition between solid and fluid state of the matter, is in equilibrium between stability and chaos [41].

> *Perhaps the most exciting implication (of CA representation of biological phenomena) is the possibility that life had its origin in the vicinity of a phase transition and that evolution reflects the process by which life has gained local control over a successively greater number of environmental parameters affecting its ability*

[14]Images courtesy of `http://plato.stanford.edu/entries/cellular-automata/`

to maintain itself at a critical balance point between order and chaos.
*(**Chris Langton** - Computation at the edge of chaos. Phase transition and emergent computation - pag.13).*

Figure 2.6: Class 3 (a,b) and 4 (c,d) elementary cellular automata



(a) Rule 30 (b) Rule 90 (c) Rule 54 (d) Rule 110

Langton in his famous paper, *Computation at the edge of chaos: phase transition and emergent computation* [41], was able to identify, by simply parametrizing the rule space, the various AC classes, the relation between them and to "couple" them with the classical complexity classes. He introduced the parameter λ [40] that, informally, is simply the fraction of the entries in the transition rule table that are mapped the not-quiescent state.

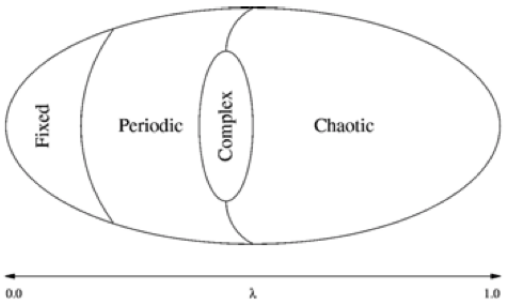
$$\lambda = \frac{K^N - n_q}{K^N}$$

where:

- K is the number of the cell states
- N the arity of the neighborhood
- n_q the number of rules mapped to the quiescent state q_q

Langton's major finding was that a simple measure is correlated with the system behavior: as it goes from 0 to $1 - \frac{1}{K}$(respectively the most homogeneous and the most heterogeneous rules table scenario), the average behavior

Figure 2.7: Relation between lambda parameter and the CA behaviors-Wolfram's classes.



of the system goes from freezing to periodic patterns to chaos and functions with an average value of λ (see [41] for a more general discussion) are being on *on the edge*(see figure 2.7).

He studied a entire family of totalistic CA with $k = 4$ and $N = 5$ with λ varying in $[0, 0.75]$. He was able to determine that values of $\lambda \approx 0.45$ raise up to class 4 cellular automata. A computational system must provide fundamental properties if it is to support computation. Only CA *on the edge* show these properties on manipulating and store information data. Here are the properties that a computational system as to provide:

Storage
: Storage is the ability of the system of preserving information for arbitrarily long times

Transmission
: Transmission is the propagation of the information in the form of signals over arbitrarily long distance

Modification
: Stored and transmitted information is the mutual possible modification of two signals.

Storage is coupled with less entropy of the system, but transmission and modification are not. Few entropy is associated with CA of Class 1 and 2 and high entropy with class 3. Class 4 is something in between, the cells cooperate and are correlate each other, but not too much otherwise they would be overly dependent with one mimicking the other supporting computation in all its aspects and requirements. Moreover class 4 CA are very dependent from the initial configuration opening to the possibility to encode programs in it.

2.1.4.4 Game of life

CA are suitable for representing many physical, biological, social and other human phenomena. But they are a good tool to study under which condition a physical system supports the basic operation constituting the capacity to support computation. The Game of life is a famous 2D cellular automaton of '70s early studied (and perhaps proved) for its universal computation capacity.

Game of life - brief definition

The Game of Life (see figure 2.8) (GOL) [13] is a totalistic CA[15] defined by :

- a 2-D lattice of square cells in an orthogonal grid, ideally infinite
- $Q = \{0, 1\}$ 2 states, and we can picture 1 as meaning alive and 0 dead (those interpretation come from the behaviour of the next-state function).
- X is the Moore neighborhood template.
- σ is the transition function and can be summarized :
  - *Birth*: If the cell is in the state ***dead*** and the number of alive neighbors is ***3***, then the cell state becomes alive (1).
  - *Survival*: If the cell is in the state ***alive*** and the number of alive neighbors is ***2 or 3***, then the cell state is still alive (1).
  - *Dead*: If the cell is in the state ***alive*** and the number of alive neighbors is ***less than 2 or higher than 3***, then the cell state becomes dead (0).

GOL is a class 4 Wolfram's taxonomy, rich complex structures, stable blocks and moving patterns come into existence even starting from a completely random configuration.

[15] A totalistic cellular automaton is a cellular automata in which the rules depend only on the total (or equivalently, the average) of the values of the cells in a neighborhood.

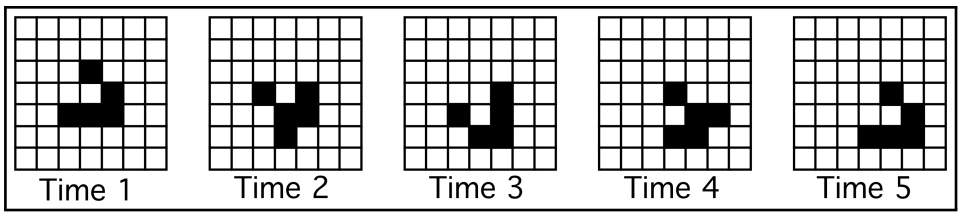
Figure 2.8: GOL execution example.

A famous example block is the *glider* (see picture 2.9) that is a 5-step-period pattern that is capable of moving into the cellular space.

Game of life as a Turing machine

Every CA can be considered a device capable of supporting computation and the initial configuration can encode an input string (a program for example). At some point the current configuration can be interpreted as the result of the computation and decoded in a output string. But as we stated before in subsection 2.1.2.1 not all the computational device have the same computational power. So which is the one of the game of life? Life was proved can compute everything a universal Turing machine can, and under Turing-Church's thesis, everything can be computed by a computer [9].

Figure 2.9: Glider in Conway's game of life.



This raises a computational issue; given the *Halting Theorem*[16] the evolution of *Life* is unpredictable (as all the universal computational systems) so

[16]There can not be any algorithm to decide whether, given an input, a Turing machine will accept or not.

it means that is not possible to use any algorithmically shortcut to anticipate the resulting configuration given an initial input. The most efficient way is to let the system run.

> *Life, like all computationally universal systems, defines the most efficient simulation of its own behavior [33]*

2.1.5 Extension of the Cellular automata model

It is possible to relax some of the assumptions in the general characterization of CA provided in the ordinary CA definitions and get interesting results. Asynchronous updating of the cell, non homogenous lattice with different neighborhood or transition functions.

2.1.5.1 Probabilistic CA

Probabilist CA is are an extension of the common CA paradigm. They share all the basic concept of an ordinary homogeneous CA with an important difference in the transition function. σ is a stochastic-function that choose the next-state according to some probability distributions. They are used in a wide class of problems like in modelling ferromagnetism, statistical mechanics [26] or the cellular Potts model[17]

Cellular Automata as Markov process

Another approach in studying CA, even if it is probably not a practical way to study the CA is to see CA as a Markov process[18]. A Markov process, is a stochastic process that exhibits memorylessness [19] and it means that the future state is conditionally independent[20] of the past. This property of the process means that future probabilities of an event may be determined from the probabilities of events at the current time. More formally if a process has this property following equation holds:

$$\begin{aligned} P(X(t_n) = x \,|X\,(t_1) = x_1, X(t_2) = x_2, \ldots, X(t_{n-1}) = x_{n-1}) \\ = P(X(t_n) = x | X(t_{n-1} = x_{n-1}) \end{aligned}$$

[17]Is a computational lattice-based model to simulate the collective behavior of cellular structures.

[18]Name for the Russian mathematician Andrey Markov best known for his work on stochastic processes.

[19]Also called Markov property.

[20]Two event A and B are independent if $P(AB) = P(A)P(B)$ or in other words that the conditional probability $P(A|B) = P(A)$.

In PCA analysis we are more interested in Markov chain because each cell has a discrete set of possible value for the status variable. In terms of chain a CA is a process that starts in one of these states and moves successively from one state to another. If the chain is currently in state s_i, than it evolve to state s_j at the next step with probability p_{ij}.The changes of state of the system are called transitions, and the probabilities associated with various state changes are called transition probabilities usually represented in the Markov chain transition matrix :

$$M = \begin{pmatrix} p_{11} & p_{12} & p_{13} & \cdots \\ p_{21} & p_{12} & p_{23} & \cdots \\ p_{31} & p_{32} & p_{33} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

This could seems to be a good way to analyze a probabilistic CA but, a 10×10 small grid (common models model use grid 100×100 or larger) identify $2^{10\times10}$ possible states and the resulting matrix dimension is $2^{10\times10} \times 2^{10\times10}$, an indeed very large number.

2.2 GPGPU Technologies

GPGPU, acronym for General-purpose computing on graphics processing units, is a recent phenomenon wich consist in the utilization of a graphics processing unit (GPU[21]), which typically handles computation only for computer graphics and was optimized for a small set of graphic operation, to perform computation in applications traditionally handled by the central processing unit (CPU). Those operations (generation of 3D images) are intrinsically parallel, so, is not surprising if the underlying hardware has evolved into a highly parallel, multithreaded, and many-core processor. The GPU excels at fine grained, data-parallel workloads consisting of thousands of independent threads executing vertex, geometry, and pixel-shader program threads concurrently. Nowadays, the GPUs are not limited to its use as a graphics engine; there is a rapidly growing interest in using these units as parallel computing architecture due to the tremendous performance available in them. Currently, GPUs outperform CPUs on floating point performance and memory bandwidth, both by a factor of roughly 100 [48], easily reaching computational powers in the order of teraFLOPS. GPU works alongside the CPU providing an heterogeneous computation, simply offloading compute-data-intensive portion of program on GPU using it as co-processor highly specialized in parallel tasks. Plus since 2006, date when Nvidia has introduced CUDA, is extremely simple to program these kind of devices for general purpose tasks, although before that date this goal was achieved dealing directly with the graphic API using shaders with all the related constraints such as lack of integers or bit operations.

2.2.1 Why GPU computing?

Traditionally performance improvements in computer architecture have come from cramming ever more functional units onto silicon, increasing clock speeds and transistors number. Moores law [28] states that the number of transistors that can be placed inexpensively on an integrated circuit will double approximately every two years.

[21] Graphic processing unit, term conied by Nvidia in the mid-nineties, and now the most common acronym used.

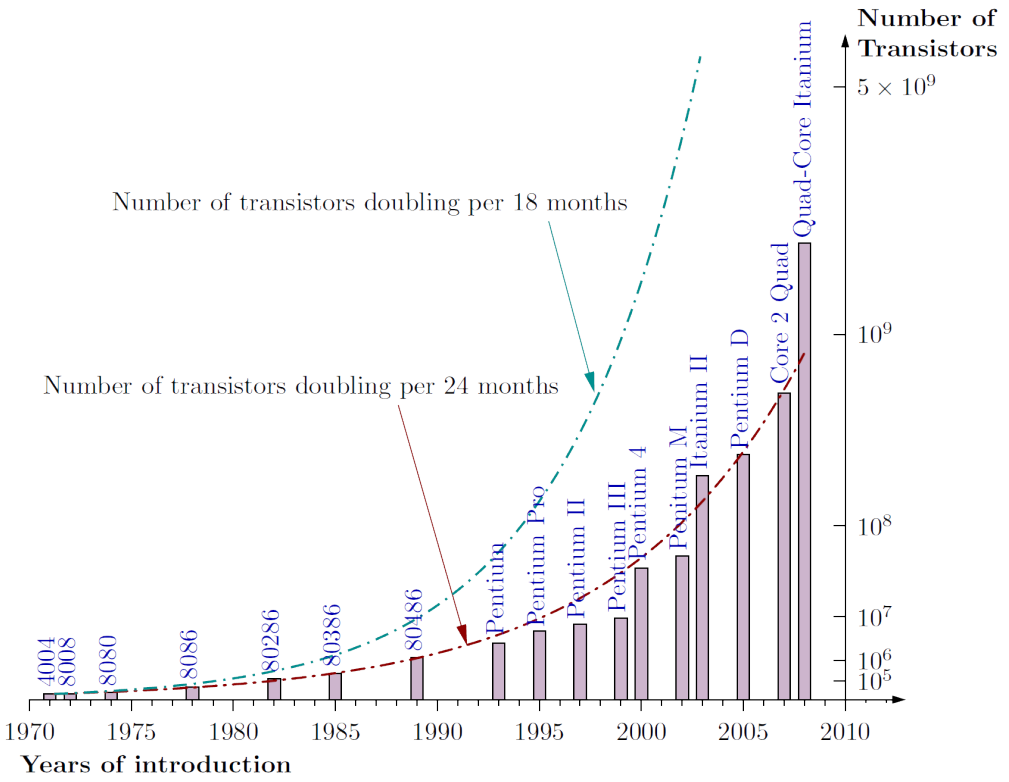


Figure 2.10: Moore's Law and intel family CPU transistors number history.

Coupled with increasing clock speeds CPU performance has until recently scaled likewise. But this trend cannot be sustained indefinitely or forever. Increased clock speed and transistor number require more power and generate more heat. Although the trend for transistor densities has continued to steadily increase, clock speeds began slowing circa 2003 at 3 GHz. If we apply Moore's law type thinking to clock-speed performance, we should be able to buy at least 10 GHz CPUs. However, the fastest CPU available today is 3.80 GHz At same point the performance increase fails to increase proportionally with the added effort in terms of transistors or clock speed because efficient heat dissipation and increasing transistor resolution on a wafer becomes more important and challenging (there will be still the physical limit of dimension for each transistor, the atom). The heat emitted from the modern processor, measured in power density ($\frac{W}{cm^2}$) rivals the heat of a nuclear reactor core [27].

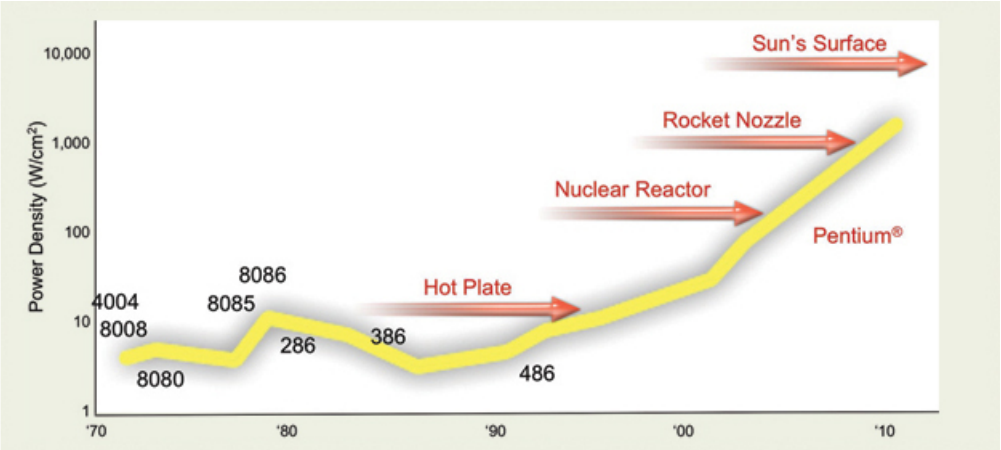


Figure 2.11: Temperature CPUs

But the power demand did not stop in these year, here the necessity of switching on parallel architectures, so today the dominating trend in commodity CPU architectures is multiple processing cores mounted on a single die operating at reduced clock speeds and sharing some resources. Today is normal to use the so-called multi-core (2,4,8,12) CPUs on a desktop PC at home.

2.2.2 From Graphics to General Purpose Computing

The concept of many processor working together in concert in not new in the graphic field of the computer science. Since the demand generated by entertainment started to growth multi-core hardware emerged in order to take advantage of the high parallel task of generating 3D image. In computer graphics, the process of generating a 3D images consist of refreshing pixels at rate of sixty or more Hz. Each pixel to be processed goes through a number of stages, and this process is commonly referred to as the graphic processing pipeline. The peculiarity of this task is that the computation each pixel is independent of the other's so this work is perfectly suitable for distribution over parallel processing elements. To support extremely fast processing of large graphics data sets (vertices and fragments), modern GPUs employ a stream processing model with parallelism. The game industry boosted the development of the GPU, that offer now greater performance than CPUs and are improving faster too (see Figure 2.13 and 2.12). The reason behind the discrepancy in floating-point capability between CPU and GPU is that GPU is designed such that more transistors are devoted to data processing rather than caching and flow control.

The today's Top 500 Supercomputers[22] ranking is dominated by massively parallel computer, built on top of superfast networks and millions of sequential CPUs working in concert but as the industry is developing even more powerful, programmable and capable GPUs in term of GFlops we see that they begin to offer advantages over traditional cluster of computers in terms of economicity and scalability.

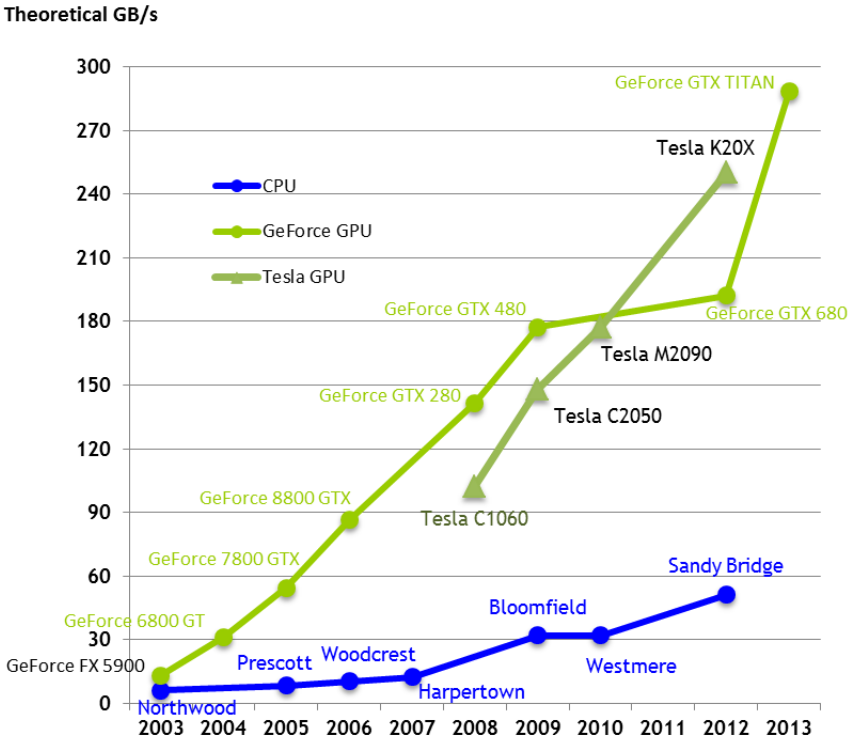


Figure 2.12: Intel CPUs and Nvidia GPUs memory bandwidth chart

2.2.2.1 Traditional Graphics Pipeline

A graphics task such as rendering a 3D scene on the GPU involves a sequence of processing stages (i.e. shaders) that run in parallel and in a prefixed order, known as the graphics hardware pipeline[23] (see Figure 2.14).

The first stage of the pipeline is the vertex processing. The input to this stage is a 3D polygonal mesh. The 3D world coordinates of each vertex of

[22] `http://www.top500.org/statistics/list/`

[23] http://duriansoftware.com/joe/An-intro-to-modern-OpenGL.-Chapter-1:-The-Graphics-Pipeline.html

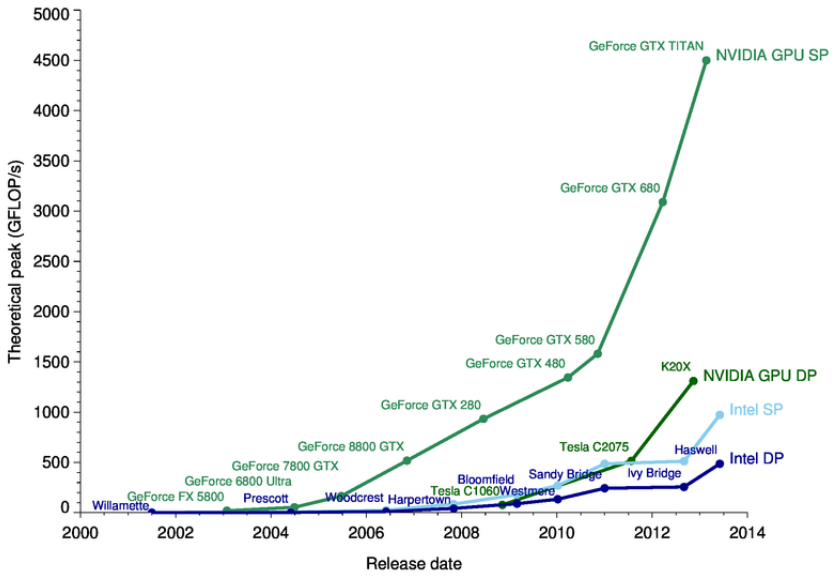
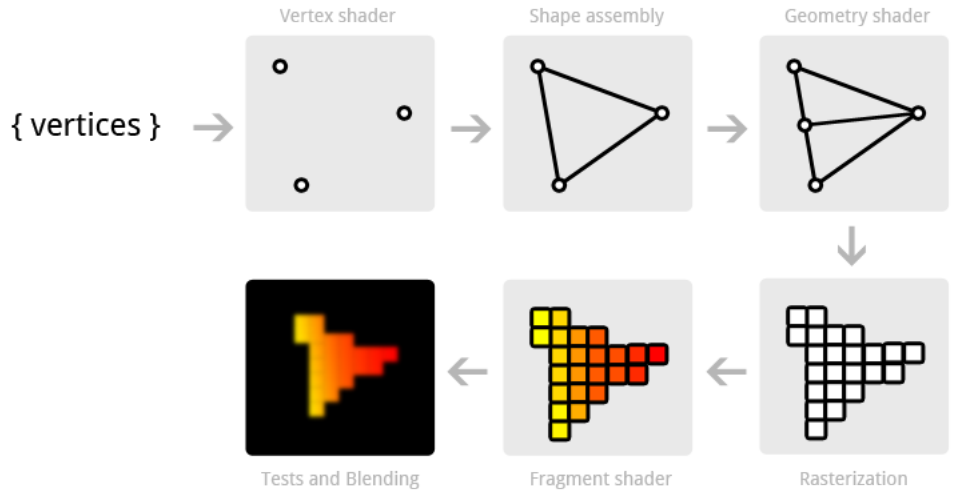


Figure 2.13: Intel CPUs and Nvidia GPUs (single and double precision) Peak G/FLOPS chart

the mesh are transformed to a 2D screen position. Color and texture coordinates associated with each vertex are also evaluated. In the second stage, the transformed vertices are grouped into rendering primitives, such as triangles. Each primitive is scan-converted, generating a set of fragments in screen space. Each fragment stores the state information needed to update a pixel. In the third stage, called the fragment processing, the texture coordinates of each fragment are used to fetch colors of the appropriate texels (texture pixels) from one or more textures. Mathematical operations may also be performed to determine the ultimate color for the fragment. Finally, various tests (e.g., depth and alpha) are conducted to determine whether the fragment should be used to update a pixel in the frame buffer. Each shader in the pipeline performs a basic but specialised operation on the vertices as it passes. In a shader based architecture the individual shader processors exhibit very limited capabilities beyond their specific purpose. Before the advent of CUDA in 2006 most of the techniques for non-graphics computation on the GPU took advantages of the programmable fragment processing stage. The steps involved in mapping a computation on the GPU are as follows:

Figure 2.14: Typical graphic pipeline



1. The data are laid out as texel colors in textures;
2. Each computation step is implemented with a user-defined fragment program. The results are encoded as pixel colors and rendered into a pixel-buffer[24];
3. Results that are to be used in subsequent calculations are copied to textures for temporary storage.

The year 2006 marked a significant turning point in GPU architecture. The G80 was the first NVidia GPU to have a unified architecture whereby the different shader processors were combined into unified stream processors. The resulting stream processors had to be more complex so as to provide all of the functionality of the shader processors they replaced. Although research had been carried out into general purpose programming for GPUs previously, this architectural change opened the door to a far wider range of applications and practitioners. More in detail GPU are well-suited for problems highly data-parallel in wich the same code is executed on many data elements at the same time (SIMD paradigm[25] or more generally as a CRCW PRAM machine[26]).

[24] A buffer in GPU memory which is similar to a frame-buffer.

[25]Single Instruction, Multiple Data: elements of short vectors are processed in parallel. To be clear CUDA paradigm is SIMT: Single Instruction, Multiple Threads

[26]Parallel random-access machine in which each thread can read or write a memory cell.

2.2.3 CUDA

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model created by NVIDIA and implemented by the graphics processing units (GPUs) that they produce. A platform that allow the developers to use an high-level programming language to exploit the parallel power of the hardware in order to solve complex computational problems in a more efficient way than on a CPU. CUDA is attractive because is a complete system(software and hardware model map well onto each other aiding the developer comprehension), from silicon to high-level libraries and a growing experience exists providing a valuable resource to developers.

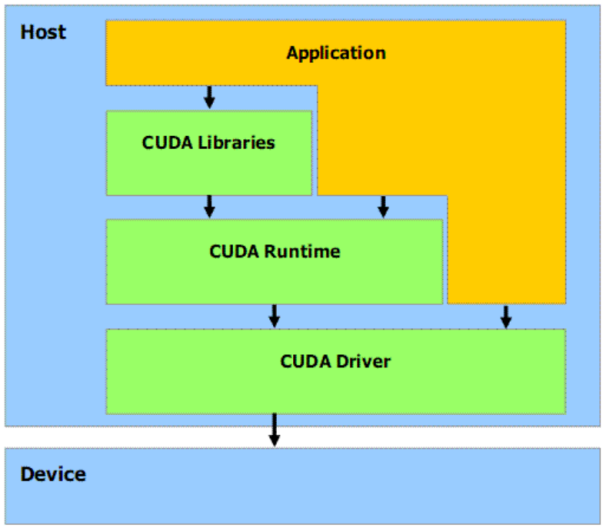


Figure 2.15: Cuda Software Stack

CUDA expose three level of components to an application (See figure 2.15):

1. ***Cuda Driver***:
   - Distinct from graphics driver. The only purpose of this component is to provide the access to the GPU's general purpose functionalities.
2. ***CUDA Runtime***:
   - Built on top of the CUDA Driver, provide an higher level of abstraction making the code less cumbersome especially as far as the complexity of host code for kernel launches is concerned.

3. ***CUDA Libraries***:

   - Built on top of the CUDA Runtime, Is a collection of Libraries (CUBLAS, CUSP, CUFFT, Thrust etc.)[27] providing full-suitable state of the art implementation of algorithms for a wide range of applications.

2.2.3.1 CUDA Programming model

CUDA programming model is designed to fully expose parallel capabilities of NVIDIA GPUs. Even though the language is devoted to general purpose computing, it still requires the programmer to follow a set of paradigms arising from the GPU architecture. CUDA provides a few easily understood abstractions that allow the programmer to focus on algorithmic efficiency and develop scalable parallel applications by expressing the parallelism explicitly. It provides three key abstractions as hierarchy of thread groups, shared memories, and synchronization barrier that provide a clear parallel structure to conventional C code for one thread of the hierarchy.

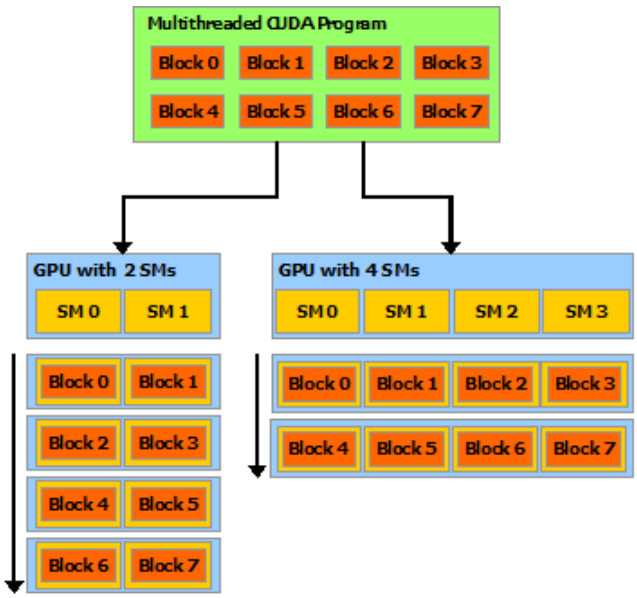


Figure 2.16: Automatic Scalability

The abstractions guide the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel, and then

[27] For example CUFFT provides an interface for computing Fast Fourier Transform up to 10x faster than CPU (`https://developer.nvidia.com/gpu-accelerated-libraries`).

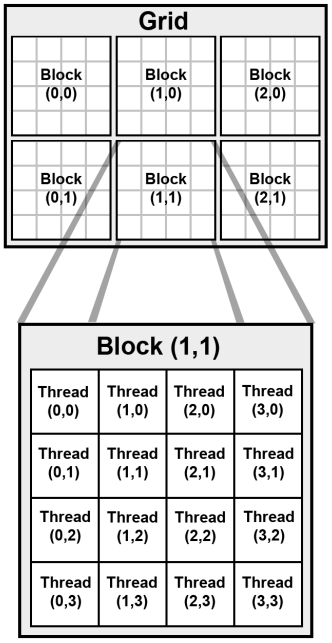


Figure 2.17: Grid of thread blocks

into finer pieces that can be solved cooperatively in parallel. The programming model scales transparently to large numbers of processor cores: a compiled CUDA program executes on any number of processors, and only the runtime system needs to know the physical processor count (See figure 2.16).

2.2.3.2 CUDA Threads and Kernels

A GPU can be seen as a computing device that is capable of executing an elevated number of independent threads in parallel. In addition, it can be thought of as an additional coprocessor of the main CPU (called in the CUDA context Host). In a typical GPU application, data parallel-like portions of the main application are carried out on the device by calling a function (called kernel) that is executed by many threads. Host and device have their own separate DRAM memories, and data is usually copied from one DRAM to the other by means of optimized API calls.

CUDA threads can cooperate together by sharing a common fast shared-memory, implemented using fast DRAM memory similar to first level cache, eventually synchronizing in some points of the kernel, within a so-called thread-block, where each thread is identified by its thread ID as illustrated by Figure 2.17. In order to better exploit the GPU, a thread block usually contains from 64 up to 1024 threads, defined as a three-dimensional array

of type dim3 (containing three integers defining each dimension). A thread can be referred to within a block by means of the built-in global variable *threadIdx*. While the number of threads within a block is limited, it is possible to launch kernels with a larger total number of threads by batching together blocks of threads by means of a grid of blocks, usually defined as a two-dimensional array, which is also of type dim3 (with the third component set to 1). In this case, however, thread cooperation is reduced since threads that belong to different blocks do not share the same memory and thus cannot synchronize and communicate with each other. As for threads, a built-in global variable, *blockIdx*, can be used for accessing the block index within the grid. Threads in a block are synchronized by calling the syncthreads() function: once all threads have reached this point, execution is resumed normally. As previously reported, one of the fundamental concepts in CUDA is the kernel. This is nothing but a C function, which once invoked is performed in parallel by all threads that the programmer has defined. To define a kernel, the programmer uses the *__global__* qualifier before the definition of the function. This function can be executed only by the device and can be only called by the host. To define the dimension of the grid and blocks on which the kernel will be launched on, the user must specify an expression of the form $<<< Grid\_Size, Block\_Size >>>$, placed between the kernel name and the argument list, such as in the following simple example:

```
1   // Kernel definition
2   __global__ void VecAdd(float* A, float* B, float* C)
3   {
4           int i = threadIdx.x;
5           C[i] = A[i] + B[i];
6   }
7   int main()
8   {
9           ...
10          // Kernel invocation with N threads
11          VecAdd<<<1, N>>>(A, B, C);
12  }
```

The above code first defines a kernel called *VectAdd* which will run on all N threads, with the aim to compute in the i-th position of the vector C, the sum of vectors A and B. Assuming that all three vectors have dimension N, each thread in parallel will be the sum of a position. For example, the thread with $ID = 2$ will calculate the sum of $A[2] + B[2]$ and store the result in $C[2]$.

2.2.3.3 Memory hierarchy

In CUDA, threads can access different memory locations during execution. Each thread has its own private memory, each block has a (limited) shared memory that is visible to all threads in the same block and finally all threads have access to global memory. In addition to these memory types, two other read-only, fast on-chip memory types can be defined: texture memory and constant memory. In CUDA, memory usage is crucial for the performance. For example, the shared memory is much faster than the global memory and the use of one rather than the other can dramatically increase or decrease performance. By adopting variable type qualifiers, the programmer can define variables that reside in the global memory space of the device (with *__device__*) or variables that reside in the shared memory space (with *__shared__*) that are accessible only from threads within a block. Typical latency for accessing global memory variables is 200-300 clock cycles, compared with only 2-3 clock cycles for shared memory locations. In addition, global memory suffers from coalesced access problems, meaning that access to data should be performed in a particular fashion in order to fetch (or store) the data in the fewest number of transactions [49]. For these reasons, global memory access should be replaced by shared memory access whenever possible. A CUDA C program can allocate global memory of the device in two different ways: through the linear memory or by means of CUDA arrays. CUDA arrays are types of memory optimized for texture management and were not exploited in this work. The more common adopted linear memory type is allocated using the *cudaMalloc()* function for allocating and *cudaFree()* function for memory de-allocation. Once allocated, it is possible to transfer data from the Host memory to the global device memory, and vice-versa, by means of a special call to the *cudaMemcpy()* function. Specifically, *cudaMemcpy()* takes as parameters four kinds of memory type transfers: Host to Host, Host to Device, Device to Host and Device to Device. Note that all of the previous functions can only be called on the host. Figure 2.18 illustrates the GPU typical memory architecture. As shown, the fast on-chip shared memory is shared by all threads of a block.

As expected, to improve performance, variable access should be carried out in the shared memory rather than global memory, wherever possible. Unfortunately, as Figure 2.18 shows, each variable or data structure allocated in shared memory must first be initialized in the global memory, and afterwards transferred in the shared one. This means that to copy data in the shared memory, global memory access must be first performed. So, the more this type of data is accessed, the more convenient is to use this type of memory, while for few accesses it is evident that shared memory might be somewhat

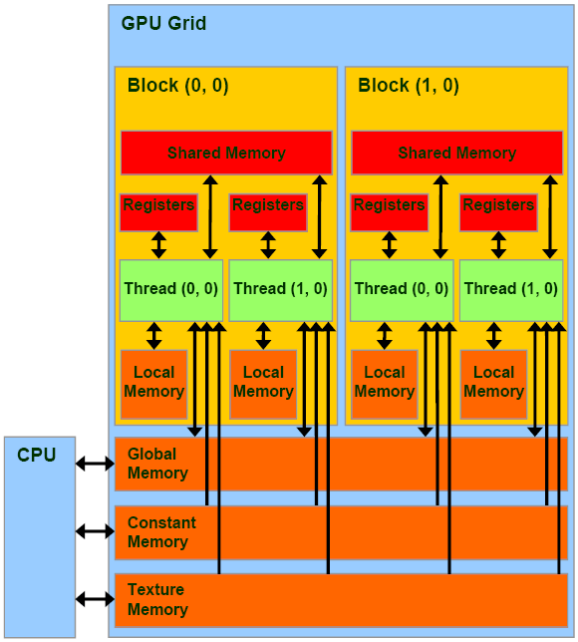


Figure 2.18: Typical memory architecture of a Graphic Processing Unit

degrading. As a consequence, a preliminary analysis of data access of the considered algorithm should be performed in order to evaluate the tradeoff and thus, convenience of using shared memory and how. As reported later in this work, the implementation with a hybrid allocation of variables results in an optimal performance, despite a total shared-memory version as it may be expected.

2.2.3.4 Programming with CUDA C

CUDA C is an extension of C language that permits to write programs for NVIDIA GPUs. With additional constructs and API functions, the programmer is able to allocate and de-allocate memory on the video card (the *device*), transfer the data from the host device (*host*), launch kernels, etc. The CUDA C extension is built on the basis of the CUDA API driver, a low-level library that allows one to perform all the above steps, but which of course is much less user-friendly. On the other hand, the CUDA API driver offers a higher degree of control and is independent of the particular language (e.g., C, Fortran, Java), being written in assembly language. A typical CUDA program can exploit the computing power of both the host (CPU and RAM) and the device (the GPU and memory devices). What follows is a classic pattern of a CUDA application:

1. Allocation and initialization of data structures in RAM memory;
2. Allocation of data structures in the device and transfer of data from RAM to the memory of the device;
3. Definition of the block and thread grids;
4. Performing one or more kernel;
5. Transfer of data from the device memory to Host memory.

In addition, a CUDA application has parts that are normally performed in a serial fashion, and other parts that are performed in parallel.

2.2.4 OpenCL

Released on December 2008 by the Kronos Group[28] OpenCL is an open standard for programming heterogeneous computers built from CPUs, GPUs and other processors that includes a framework to define the platform in terms of a host, one or more compute devices, and a C-based programming language for writing programs for the compute devices (see figure 2.19). One of the first advantages of OpenCL is that it is not restricted to the use of GPUs but it take each resource in the system as computational peer unit, easing the programmer by interfacing with them. Another big advantage is that it is open and free standard and it permit cross-vendor portability[29].

2.2.4.1 Model Architecture

The architecture programming model's follows the CUDA's one but with different names.

[28] A standards consortium.

[29] One of the most important supporter of OpenCL is ATI

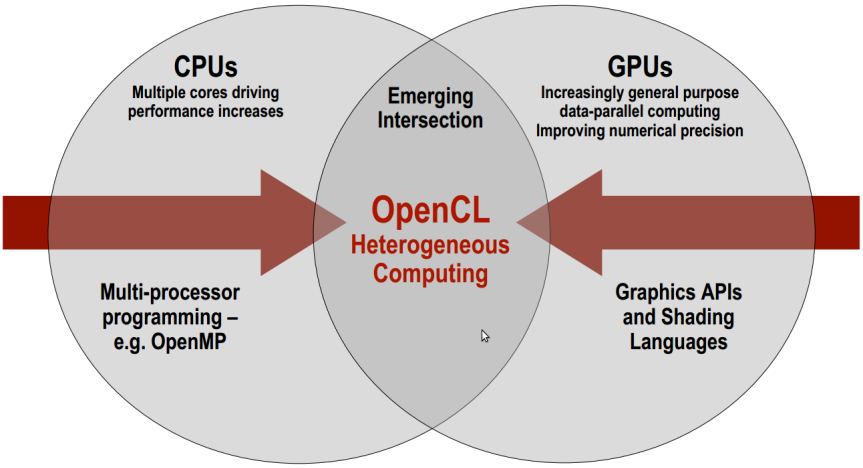


Figure 2.19: OpenCL heterogeneous computing.

Work-items:
are equivalent to the CUDA threads and are the smallest execution entity of the hierarchy. Every time a Kernel is launched, lots of work-items (a number specified by the programmer) are launched, each one executing the same code. Each work-item has an ID, which is accessible from the kernel, and which is used to distinguish the data to be processed by each work-item.

Work-group:
equivalents to CUDA blocks, and their purpose is to permit communication between groups of work-items and reflect how the work is organized (usually organized as N-dimensional grid of work-groups with $N \in \{1, 2, 3\}$). As work-items, they are provided by a unique ID within a kernel. Also the memory model is similar to the CUDA's one. The host has to orchestrate the memory copy to/from the device and explicit;y call the kernel.

A big difference is in how a kernel is queued to execution on the accelerator. Kernels are usually listed in separate files the OpenCL runtime take that source code to create kernel object that can be first decorated with the parameters on which it is going to be executed and then effectively enqueued for execution onto device. Here a brief description of the typical flow of an OpenCL application.

1. Contexts creation: The first step in every OpenCL application is to create a context and associate to it a number of devices, an available

OpenCL platform (there might be present more than one implementation), and then each operation (memory management, kernel compiling and running) is performed within *this* context. In the example 2.2 a context associated with the CPU device and the first finded platform is created.

2. Memory buffers creation: OpenCL buffer Object are created. Those buffer are used to hold data to be computed onto devices.

3. Load and build program: we need to load and build the compute program (the program we intend to run on devices). The purpose of this phase is to create an object ***cl::Program*** that is associable with a context and then proceed building for a particular subset of context's devices. We first query the runtime for the available devices and then load directly source code as string in a ***cl::Program:Source*** OpenCL object (see listing1 2.4).

4. In order a kernel to be executed a *kernel object* must be created. For a given *Program* there would exists more than one entry point (identified by the keyword *__kernel* [30]). We choose one of them for execution specifying in the kernel object constructor

5. We effectively execute the kernel putting it into a *cl::CommandQueue*. Given a cl::CommandQueue queue, kernels can be queued using *queue.-enqueuNDRangeKernel* that queues a kernel on the associated device. Launching a kernel need some parameters (similar to launch configuration in CUDA, see section 2.2.3.2) to specify the work distribution among work-groups and their dimensionality and size of each dimension (see listing 2.1). We can test the status of the execution by querying the associated *event*.

Listing 2.1: OpenCL Queue command, kernel execution

```
1 cl_int err;
2 cl::vector< cl::Platform > platformList;
3 cl::Platform::get(&platformList);
```

[30] Obviously in the same source code one can define more than on kernel.

```
4     checkErr(platformList.size()!=0 ? \\
5                   CL_SUCCESS:-1,"cl::Platform::get");
6     cl_context_properties cprops[3] =
7     {CL_CONTEXT_PLATFORM, (cl_context_properties)(
          platformList[0])(), 0};
8     cl::Context context(CL_DEVICE_TYPE_CPU,cprops,NULL,
          NULL,&err);
9     checkErr(err, "Conext::Context()");
```

Listing 2.2: OpenCL context creation

```
1     cl::Buffer outCL(context,CL_MEM_WRITE_ONLY |
2                                 CL_MEM_USE_HOST_PTR,hw.
                                      length()+1,outH,&err);
3 checkErr(err, "Buffer::Buffer()");
```

Listing 2.3: OpenCL program load and build

```
1     std::ifstream file("pathToSourceCode.cl");
2     checkErr(file.is_open() ? CL_SUCCESS:-1, "
          pathToSourceCode.cl");std::string
3     prog( std::istreambuf_iterator<char>(file),
4     (std::istreambuf_iterator<char>()));
5     cl::Program::Sources source(1,std::make_pair(prog.
          c_str(), prog.length()+1));
6     cl::Program program(context, source);
7     err = program.build(devices,"");
8     checkErr(err, "Program::build()");
```

Listing 2.4: OpenCL program load and build

```
1     cl::CommandQueue queue(context, devices[0], 0, &err);
2     checkErr(err, "CommandQueue::CommandQueue()");cl::
          Event event;
3     err = queue.enqueueNDRangeKernel(kernel,cl::NullRange,
4     cl::NDRange(hw.length()+1),  cl::NDRange(1, 1),NULL,&
          event);
5     checkErr(err, "ComamndQueue::enqueueNDRangeKernel()");
```

2.2.5 OpenACC

OpenACC is a new[31] open parallel programming standard designed to enable to easily to utilize massively parallel coprocessors. It consist of a series of *pragma*[32] pre-compiler annotation that identifies the succeeding block of code or structured loop as a good candidate for parallelization exactly like OpenMP [33] developed by a consortium of companies[34]. The biggest advantage offered by openACC is that the programmer doesn't need to learn a new language as CUDA or OpenCL require and doesn't require a complete transformation of existing code. Pragmas and high-level APIs are designed to provide software functionality. They hide many details of the underlying implementation to free a programmer's attention for other tasks. The compiler is free to ignore any pragma for any reason including: it doesn't support the pragma, syntax errors, code complexity etc. and at the same time it has to provide profiling tool and information about the parallelization(even if it is possible). OpenACC is available both for C/C++ and Fortran. In this document we will concentrate only on C/C++ version. An OpenACC pragma can be identified from the string "#pragma acc" just like an OpenMP pragma can be identified from "#pragma omp". The base concept behind openACC is the *offloading* on the accelerator device. Like CUDA or openCL the execution model is host-directed where the bulk of the application execute on CPU and just the compute intensive region are effectively offloaded on accelerator[35]. The *parallel regions* or *kernel regions*, which typically contains work sharing work such as loops are executed as kernel (concept described in section 2.2.3.2 at page 29). The typical flow of an openACC application is orchestrated by the host that in sequence has to:

- Allocate memory on device.
- Initiate transfer.
- Passing arguments and start kernel execution(a sequence of kernels can be queued).
- Waiting for completion.

[31]Release 1.0 in November 2011.

[32] A pragma is a form of code annotation that informs the compiler of something about the code.

[33]The is a well-known and widely supported standard, born in 1997, that defines pragmas programmers have used since 1997 to parallelize applications on shared memory multicore processor

[34]PGI, Cray, and NVIDIA with support from CAPS

[35]We don't talk of GPU because here, accelerator is referred to the category of accelerating co-processors in general, which the GPU certainly belong to.

- Transfer the result back to the host.
- Deallocate memory.

For each of the action above there is one or more directive that actually implements the directives and a complete set of option permit to tune the parallelization across different kind of accelerators. For instance the *parallel* directive starts a parallel execution of the code above it on the accelerator, constricting *gangs* of workers (once started the execution the number of gangs and workers inside the gangs remain constant for the duration of the *parallel* execution.) The analogy between the CUDA blocks and between workers and cuda threads is clear and permit to easily understand how the work is effectively executed and organized. It has a number of options that permits to for example copy an array on gpu to work on and to copy back the result on the host side.

The syntax of a OpenACC directive is :

- C/C++ : #pragma acc directive-name [clause [[,] clause]...] new-line.
- Fortran : !$acc directive-name [clause [[,] clause]...]

Each clause can be coupled with a number of clauses that modify the behavior of the directive. For example:

- copy(list)Allocates the data in list on the accelerator and copies the data from the host to the accelerator when entering the region, and copies the data from the accelerator to the host when exiting the region.
- copyin(list) Allocates the data in list on the accelerator and copies the data from the host to the accelerator when entering the region.
- copyout(list) Allocates the data in list on the accelerator and copies the data from the accelerator to the host when exiting the region.
- create(list) Allocates the data in list on the accelerator, but doesn't copy data between the host and device.
- present(list) The data in list must be already present on the accelerator, from some containing data region; that accelerator copy is found and used.

2.2.5.1 Wait Directive

The wait directive causes the host program to wait for completion of asynchronous accelerator activities. With no expression, it will wait for all outstanding asynchronous activities.

- C/C++ : #pragma acc wait [(expression)] new-line
- Fortran : !$acc wait [(expression)]

2.2.5.2 Kernel Directive

This construct defines a region of the program that is to be compiled into a sequence of kernels for execution on the accelerator device.

C/C++:
#pragma kernels [clause [[,] clause]...] new-line { structured block }

Fortran:
!$acc kernels [clause [[,] clause]...]
structured block
!$acc end kernels

2.2.5.3 Data Construct

An accelerator data construct defines a region of the program within which data is accessible by the accelerator. It's very useful in order to avoid multiple transfers from host to accelerator or viceversa. If the same pointers are used by multiple directives, a good practice is to declare and allocate those pointers in a *data* construct and use them in parallel or kernel construct with the clause *present* [57].

Description of the clause are taken from the official documentation[36].

2.3 WEB 2.0

In 2001 one of the most important companies of the time in the Internet world went bankrupt, the *Webvan* company. *Webvan* was intended to sell anything to anyone and everywhere; even though it was a company that was born from a business model quite inconsistent, it managed to collect 400 million dollars from venture capitals and following its founder was able to

[36]For a complete list of directive, constructs and pragmas consult the official documentation here : `http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf`

place it on the *Nasdaq*, the American technology stock market. Subsequently, the company was closed almost suddenly.

Webvan is the symbol of the famous Internet Bubble of 2000, which later influenced the development of the whole network.

Since then, there have been many important changes in the world of the Internet and among them the birth of "*Web 2.0*"

This technology has led to the creation of many new companies, some of these are the most important today as *Flickr* or *Youtube* and therefore the mass access has transformed them in business models.

From the social point of view, the "*Web 2.0*" is entering the lives of many people, changing their habits, the way in which to search, consume and especially share information. Many people use *YouTube* to share and view videos, *Wikipedia* for school research, publish their photo album using *Flickr* or *Instagram*, manage and read people's *Blog* and more.

Therefore, *Web 2.0* is the environment where sites and applications that enable the control of the content in the hands of the user were born and continue to emerge. In order to define the *Web 2.0* it is necessary to analyze what the *web* was, and how it has come to the concept of *Web 2.0*.

2.3.1 The dawn of the Web

The first idea of the Web poses its basis on the concept of hypertext accessible by any person through the network. It is formed by simple static HTML pages, articles related to each other or classical *form* and a small group of publishers who create web pages for a wide audience. The result is that many people can get the information by going directly to the source.

However, in a first stage users do not have great possibilities to insert their own content. This phase can be identified with the name of *Web 1.0*, the first "version " of the Web.

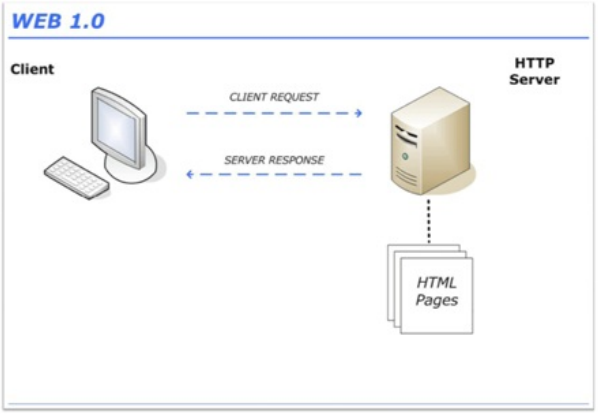


Figure 2.20: Web 1.0 Interaction model.

From this simple picture one can well understand how, client side, you do not have any particular type of processing beyond the simple *graphics rendering* related to the display of the pages. Likewise, the server side is in front of simple collections of pages written in HTML without any element dynamically generated. Thus, a pattern of interaction that is purely static and with limited potentialities emerges, in which the user makes a request via the browser and the server responds by simply sending the document as it was created by the author. In fact, the initial aim was only to apply the concept of hypertext accessible from the network; therefore, at least at first, there were no other needs over the scenario presented.

However, the real success of *dot-com* occurred later, with the introduction of a more dynamic Web. The evolution of technology has in fact led to having *content management* systems and HTML pages with dynamic content created on the fly from *database* content that is constantly updated. These changes allowed on one hand the more active participation of the users and on the other promoted the displacement of many activities on the Web, that were not possible previously. In this way, more and more people have acquired the skills and abilities necessary to be able to write on the Web, and not just to read.

In this case, we can talk of "*Web 1.5*", a first important development of what was the original Web. In this second "version" of the *Web*, there is an important evolution in the interaction model; for the first time, in fact, we can speak of dynamic content and pages that are created on the fly without any content attached in the pages and in the code. This was mainly a server-side revolution since begin to emerge languages specifically created (*Cold-Fusion*, *PHP*, *JSP*, *ASP*, etc.) together with the so-called *Web Server* (*Apache*, *Tomcat*, *IIS*, etc.). Now it is possible to create real applications

that can be uploaded to specific servers and can interact with other entities, which can be other server-side applications or database of various types.

The interaction model is not so revolutionized than before; in fact, facing a *client* request, the *server* always responds by sending an HTML page. The difference is the way in which is obtained this HTML page. While previously, in what had been called "*Web 1.0*", the HTML page was always physically present in the server memory, now it is no longer a certainty.

In its place there can in fact be an application, more or less complex, which in response to a request sent by the client, it processes a well defined information (maybe contained in a database constantly updated), generates one or more pages containing the information required by the client and sends them in a form "understandable" to it, i.e. mainly in HTML.

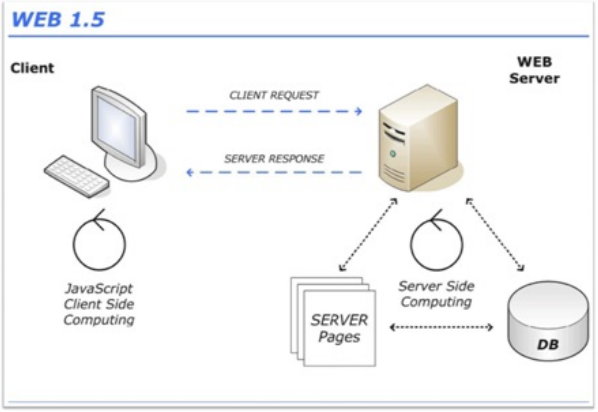


Figure 2.21: Web 1.5 Interaction model.

As you can see from the figure, on the client side you have the possibility to process data through a scripting language specifically created: *JavaScript*.

The fundamental difference between server-side programming languages and *JavaScript* is that the processing done on the client side with the latter is typically limited to local level (form control, etc.), so do not go beyond the client itself, and without involving external entities. Even if the scenario was very simplified, the great power of a model of this type is easy to understand and, at this point, many people have thought that the Web was fully mature, hence seeing in the *Web-Applications* and *Web Services* the highest expression of its potential.

2.3.2 The Web 2.0

With the use of client-side languages that add interactivity to the site and of style sheets that allow you to format the content as best as you want, you can create real *Web applications* that sometimes have nothing inferior to the classic *Desktop Applications*.

The increase in the version of the Web, as mentioned above, does not refer to an update of the technical specifications of the *World Wide Web*, but to a different use of the Web by developers. *Web 2.0* is thus a term used to indicate a state of evolution of the Web. Since 2004 we have witnessed this evolution with the birth of new services and applications which are now integral part of the habit of many people. The term *Web 2.0* was coined at a conference held at *O'Reilly Media* where the Vice-President of *O'Reilly* pointed out that the network was going through a period of renewal and growth that could not be ignored.

Giving a clear and concise definition of what is *Web 2.0*, as repeatedly mentioned, is very difficult for the simple fact that not everyone agrees that it is an innovation but a normal evolution of the Web.

Tim O'Reilly tried to give a compact definition:

> *Web 2.0 is the business revolution in the computer industry caused by the move to the Internet as platform, and an attempt to understand the rules for success on that new platform. Chief among those rules is this: build applications that harness network effects to get better the more people use them.*

From this definition we can understand the heart of the *Web 2.0*, that is to see the Web as a platform exploiting it as much as possible.

With these words is meant the progressive use of web applications which, in turn, always bring a greater continuing presence online of users.

Many claim that the *business revolution* is the result of a *consumer revolution* that is, the user does not want to be passive in front of this means of communication, but want to actively participate in its growth.

Users do not just want to use the Web, but *do* it.

The *Web 2.0* leads to technologies capable of increasing the opportunities for participation; it is a new point of view, namely the willingness to become entrepreneurs conscious of their own opportunities, a community conscious of their power and potential. Seeing this innovation only from the technical point of view is thus a mistake. *Russell Shaw* has a very critical view about it:

> *[...] But Web 2.0 does not exist.*

First of all, Web 2.0 is a marketing slogan.

In support of this, we can not go beyond the definition of Wikipedia:

> *The term "Web 2.0" refers to what some people see as a second phase of develpment of the World Wide Web, including its architecture and its applications. It was coined by Dale Dougherty during a meeting between O'Reilly and Associates (a computer book publisher) and MediaLive International (an event organizer) as a marketable term for a series of conferences.*

Skeptics argue that the term *Web 2.0* does not have a real meaning, as this depends solely on what the proponents agree that it should mean to try to convince the media and investors who are creating something new and better, rather than continue to develop existing technologies (so purely for marketing reasons).

If for them the concept of *Web 2.0* does not exist or does not have a proper meaning, what is it? Many are of the idea that it is the awareness of users and developers to exploit at best an innovative communication tool like the Internet through the active involvement of users, information sharing, and much more.

In this way we can change the concept of use of the network by breaking the hierarchical system formed by administrator and user and innovating the rules in the production of communication patterns.

Users' participation appears to be the core of this innovation on the Web, followed by the possibility of being able to manipulate and transform data from other sites and by the introduction of new technologies such as *AJAX* (*Asynchronous JavaScript and XML*), that have contributed to make the user interfaces faster and more interactive.

In the following, we will see specifically what is AJAX and what it allows to create.

2.4 AJAX

All desktop applications have strengths as wealth, speed and sensitivity, which until a few years ago, seemed to be their prerogative and hence outside of the capabilities and possibilities of the Web. The rapid growth made by the Web in recent years, however, has highlighted the need for web applications to come as close as possible to the desktop applications in terms of efficiency and effectiveness, so as to be able to offer its users the same capabilities provided by desktop applications.

In most cases, all the *Web rich internet applications* in recent years have been linked to the technologies *Macromedia (Flash)* or *Java (Applet)*, unfortunately not always interpretable by all clients and too often used inappropriately for the sole purpose of impressing, without taking care of efficiency and above all of the speed of the application.

In '97, as an alternative to these techniques of *client-server* interaction, was introduced the "*iframe*". Many developers took advantage of this, by changing the source attribute of the page enclosed, thereby simulating a transparent "*refresh*" of part of the contents, although in a rather dirty manner; in practice, there was an asynchronous interaction.

In '98 the *Remote Scripting* appeared, a technology developed by Microsoft with the intent to create a more elegant technique to retrieve different content. After various changes, improvements, and with the aid of an object known as *XMLHttpRequest*, developed by Microsoft itself, this technique became what is now known by the name of *AJAX*

The acronym *AJAX* means *Asynchronous JavaScript And XML*; it was coined and used on February 18, 2005 by *Jesse Garrett*, as the title of a post in his blog.

Today, referring to *AJAX* we refer to the **XMLHttpRequest** object. Depending on the browser we use, it takes different names or is invoked in a different way.

This object allows you to make a request for a resource to a web server independently from the browser you are using. In the request, information may be sent in the form of variables of type GET or POST to send data in a similar way to a traditional form.

The methodology of *AJAX* is partially expressed by the acronym chosen, that is the union of different Web technologies already known, which are used in a completely new way compared to the past. So we are not dealing with a completely new technology to learn, but rather there is a need to rethink the use of certain technologies.

Here is a list of technologies that joined together set up *AJAX*:

- *JavaScript*, which constitutes the connecting point between the various technologies;
- *DOM* (*Document Object Model*) for visualization and dynamic interactions;
- *XML* and *XSLT* for the exchange and manipulation of data;
- `XMLHttpRequest` for asynchronous retrieval of information;
- *CSS* e *XHTML* for standard-based presentation of content. of content.

The main feature is that the request is **asynchronous**, meaning you do not necessarily have to wait until it has been completed before you can perform other operations.

2.4.1 AJAX rich applications

The term "*Rich application*" refers to the type of interaction that characterizes an application. A rich interaction model must be able to support several input methods and be able to respond quickly and simply. This feature is at the basis of every desktop application.

There was much discussion about the possibility of providing web users with "*rich internet application*" with features comparable to those of the desktop applications. Through *AJAX* this is possible.

Currently, the *web-based* services are becoming more robust and powerful but at the same time even heavier and more complicated, thus the traditional web applications are beginning no longer to be able to manage all services offered.

Now we can describe how "classic" web applications work, trying to figure out what's wrong with them.

Generally, the data flow is contained in two steps at a time: user request (*link*, *form*, *refresh*) and server response, then moving on to a new request from the user if necessary.

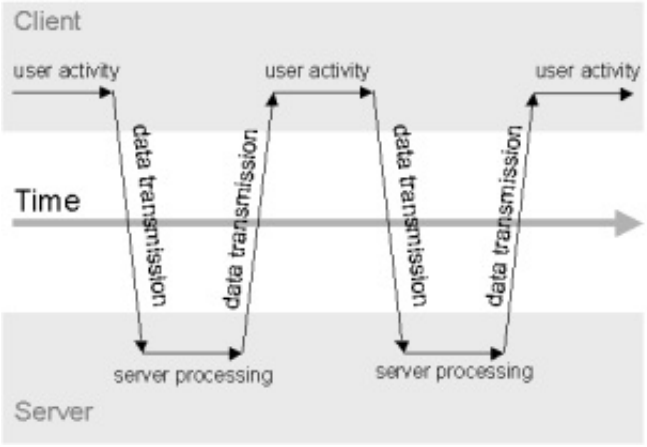


Figure 2.22: Details of a "Classic" Web request.

This is fine for hypertext, but unfortunately does not go as well for software applications. The classical model indeed creates many problems, such as:

- bad performances due to the cycle "*click - wait - refresh*";
- loss of the operating context during the *refresh* of each page;
- excessive use of available bandwidth due to the continuous updating of redundant elements and pages.

In conclusion, everything turns into Web applications that are slow, unreliable, inefficient and with a low productivity. So the goal we are aiming at with *AJAX* is to bring the model of interaction through the *desktop applications* on the Web, that is a spread context.

By using *AJAX* we lose the linearity present in the requests, but while the user is on the same page, the requests to the server can be numerous and completely independent.

Theoretically, nothing prevents us to perform a multitude of simultaneous requests to a server for doing different operations, with or without any control by the user.

AJAX gives the possibility to communicate with the server asynchronously, eliminating the so far inevitable request-response cycle. In fact, with *AJAX* it is possible to use *JavaScript* and *DHTML* to update the page graphics on the fly, and make a request to the server asynchronously. Immediately after having got the response from the server,*JavaScript* can be used again, with the support of *CSS* to redraw the entire page without reloading it.

In particular, *AJAX* introduces in Web applications a *layer*, called *AJAX-Engine*. Instead of loading a webpage, at the beginning of the session the browser loads an engine written in *JavaScript*, that is hidden to the user. This engine is responsible both for the management of the interface rendering, displayed by user, and for client-server communications. The *AJAX-Engine* makes possible the interaction between the client and the application asynchronously. This means that every click made by the user sends a *JavaScript* request to the *AJAX engine* which will then decide if an http request to the server is needed or if it is possible to satisfy the request locally.

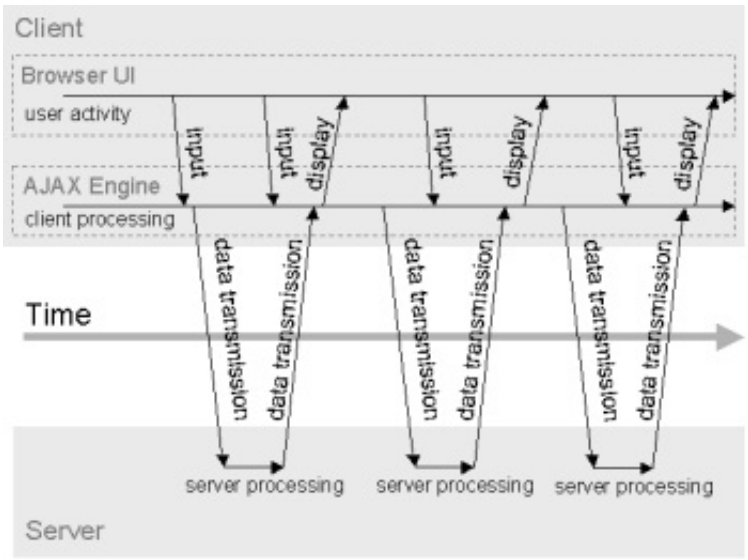


Figure 2.23: Interaction between AJAX application components.

Thus, the waiting time recedes in background and in various types of interaction is almost imperceptible. However, it is necessary to be careful because this time is also one of the biggest problems when *AJAX* is used, both for developers and for users; the former could be in trouble if the asynchronous operation would be forced to wait for a response to complete a series of tasks, while the latter may not have any idea of what is happening to the page, closing it being unaware that they had requested information.

Another consideration is about the type of response that the object expects after a call; indeed, it does not necessarily need to be of type *XML*, but can be simply text, in contrast with the acronym itself but not for this unusual.

2.4.1.1 Benefits and Drawbacks

Although *AJAX* represents a huge innovation for the development of web applications, it must be said that he has limitations.

Starting from the *server-side* you can list a number of very significant advantages.

As we know, the use of external *CSS* files (Cascading Style Sheets), allows the user to save a lot of bandwidth. Indeed, using the *AJAX* the server does not need to transfer the entire page at each interaction, but only the portion needed for the requested operation. This makes the interaction faster for the user and facilitates the bandwidth savings.

A further advantage is related to the calculations to be made. For example, consider an Internet portal full of information. Finding all the information to deal with a traditional interaction may require an excessive workload (interaction with *database*, *web services*, etc), while using *AJAX* the requests are punctual and the server can respond much more efficiently. Sites with a large number of simultaneously connected users, no longer have to operate on all parts of the application, in this way improving the ability to manage a huge number of users.

An important point to emphasize is that a part of calculations can be loaded to the *browser*, in order to exploit the power of the *client-pc* and distribute the workload across all the various users, rather than only on the server. How it can be easily understood, we should not "charge clients too much", if we do not want to get a slowdown on the server with a consequent slowdown on the client.

Figure 2.24: Server load without AJAX.

Figure 2.25: Server load using AJAX.

After talking about some of the benefits resulting from the use of *AJAX*, we can analyze some disadvantages. Filling a page with too many interactions is the first thing that should be avoided for a good use of *AJAX*. Besides, if you do not monitor the transactions on the client, the server is likely to find itself full of requests most likely unnecessary. For example, consider a "*suggest*", which is very useful to every internet user who seeks something; if it is implemented in the wrong way, it becomes a non-trivial problem for the server.

In short, we need to work as much as possible in the economy, in terms of communication with the server. The only solution then is to find the right balance between the need to update something and calculations to be made for the update.

Finally, we can say that the applications must pass many tests to ensure compatibility on different browsers and on different platforms; this aspect is not negligible, especially if we note that practically every browser intends *JavaScript* in its way. More generally we can say that debugging *AJAX* applications is much more complicated than debugging traditional applications.

With *AJAX* unsafe coding potentially becomes easier, that is code that transmits sensitive information without safety precautions. So it is necessary to examine carefully the traffic generated by an *AJAX* application to verify that safety is not compromised.

However, these last two mentioned problems are not lacks of *AJAX*, but only difficulties faced by programmers during the development of a good application.

We can now proceed to analyze "*Client-Side*" benefits and drawbacks.

An immediate aspect that we can note on the client side is the impression to see professional web pages, dynamic and more interactive with a smooth flow of work and without unnecessary interruptions or waitings; of course, this does not mean that the load times are magically eliminated.

To support *AJAX*, there are many libraries, easily integrated, able to impress the users in a way very similar to the graphic plugin of *Adobe-Macromedia*, but with a very considerable saving of resources.

The exchange of information is done in *background*, in an absolutely transparent way to the user who hence does not notice anything. This means that the applications give the user the feeling of completely being local and not being distributed, as they are in reality.

Another worthy note is its almost total compatibility with the most popular modern browsers, and the native or integrable support of the object `XMLHttpRequest` contributes further to the development of *Rich Internet Applications*.

Of course there is also a downside: the development of applications for those users who require assistive technology is not very simple using this approach. The difficulties in implementing usability is probably the biggest flaw of the *AJAX* approach.

So, *AJAX* applications must also prevent these situations by providing users the same functionalities in an alternative way (e.g. through the only use of *HTML* without *JavaScript*), still allowing a proper use of the application.

Another disadvantage is for users with slow connections, which will have to charge a good amount of data represented by the various *JavaScript* files

needed by the *web application* or website. Besides, at worst you may not be able to take advantage of what has been loaded (in case you do not have an updated browser, or in cases of *JavaScript* disabled).

All these minor technical problems can be solved easily, but still a purely practical problem remains, probably the best known and most annoying: the use of the keys "Next" and "Previous" of any *browser*.

Any asynchronous operation gives the feeling of having changed the page and in most cases, if the expected result should not be the one desired by the user, he/she has a habit to go back by pressing the corresponding button in the browser.

Having access to a page with asynchronous operations means that we no longer have the ability to click on the button "backward " to return to the previous state. In fact, by doing this you will be redirected to the previous page, which causes many drawbacks. Even refreshing the page is not necessary to return to the previous state, but only to the initial state.

Therefore, this is a constraint for navigation and also implies a constraint for the indexing or the possibility to notify others of the displayed page. The reason for all this is due to the command managed by *JavaScript* that is not portable like a link. Notifying another user of the page To recommend to another user the page you visited is not possible if not giving precise instructions on what to do to get to the page in question.

A very similar problem is that related to bookmarks. Even in that case, in fact, if the programmer does not take necessary precautions, it may not be possible for the user to perform the *bookmark* of a page.

The solution to these problems is not simple and the reasons are different. First, creating an event handler that can add changes to the browser's history can be a complex task because of different standards or functions present in different browsers. Secondly, the indexing of a single link, for anyone who does not have *JavaScript* or may not make use of this technology, it would become more or less impossible.

As one can see from this quick treatise, also *AJAX* has obvious flaws and limitations with which we must confront. But the efforts made by the developers to overcome these barriers is greatly rewarded by the experience that users may have using the application.

3

Simulation of complex macroscopic natural phenomena and Scientific Web applications

3.1 Cellular Automata application Models

3.1.1 SCIDDICA K1: a cellular automata model to simulate landslides and debris flows.

Cellular Automata (CA) represent a formal frame for dynamical systems, which evolve on the base of local interactions. Some types of landslide, such as debris flows, match well this requirement.

The model SCIDDICA was originally developed for simulating simple cases of flow-like landslides. In its successive releases, higher complexity was essentially managed by progressively adding new local interactions and/or internal transformations to the previous ones: therefore, it could be considered as an "incremental" CA-model.

For CA simulation purposes, landslides can be viewed as a dynamical system, subdivided into elementary parts, whose state evolves as a consequence of local interactions. The cellular space is constituted by squared cells, whose attributes indicated as "substates" describe physical characteristics. For computational reasons, the natural phenomenon is "decomposed"

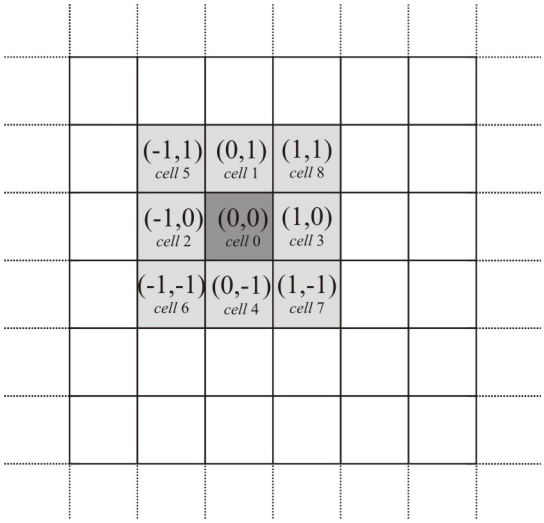


Figure 3.1: Cellular space with squared cells and Moore neighborhood. The central cell (dark grey) has index 0; indexes from 1 to 8 indicate the other adjacent cells (light grey).

into a number of elementary processes, whose proper composition makes up the "transition function" of the CA. By simultaneously applying this function to all the cells, the evolution of the phenomenon can be simulated in terms of modifications of the substates.

Formally, the model SCIDDICA K1 can be defined as:

$$SCIDDICA\ K1 =< R, E, X, Q, P, \tau, \gamma >$$

where:

$R = \{(x, y) \in \mathbb{Z}^2 | -l_x < x < l_x,\ -l_y < y < l_y\}$ identifies the bi-dimensional cellular space with squared cells; $\mathbb{Z}$ is the set of the integer numbers, while l_x and l_y correspond to the limit of the area in which the considered phenomenon evolves.

$E \subset L$ is the set of the cells where landslides are triggered. They can be considered as source cells.

$X = \{(0,0), (0,-1), (-1,0), (1,0), (0,-1), (-1,1), (-1,-1), (1,-1), (1,1)\}$ is the relationship of Moore closeness that defines the neighborhood, given by the central cell and its eight adjacent cells; neighboring cells are indexed from 0 to 8, as shown in 3.1, in order to specify the rules of the transition function;

$S = Q_z \times Q_h \times Q_{h_k} \times Q_d \times Q_f^9$ is the finite set of cell states, given by the Cartesian product of the sets of the considered substates:

- Q_z is the cell altitude;
- Q_h is the thickness of landslide debris;
- Q_{h_k} is the kinetic altitude (or kinetic load) of landslide debris;
- Q_d is the depth of erodible soil cover;
- Q_f^9 represents the six debris outflows from the central cell to the adjacent cells, considered in terms of thickness.

$P = \{p_c, p_{hc_{\theta_0}}, p_{dQ}, p_{mt}, p_{pef}\}$ is the set of the parameters:

- p_c is the cell side;
- $p_{hc_{\theta_0}}$ is the altitude threshold, beyond which a debris column situated on a horizontal plane becomes unstable and starts moving;
- p_{dQ} rules the speed loss following a quadratic mechanism;
- p_{mt} is the energy threshold to trigger the mobilization of the erodible soil cover;
- p_{pef} is the parameter of progressive erosion of the soil cover.

$\tau : Q^9 \to Q$ is the deterministic transition function of the cell, formed by the following elementary processes that are applied in the same order as they are listed below:

1. internal transformation T_1: activation of triggering sources;
2. internal transformation T_2: soil erosion;
3. local interaction I_1: calculation of debris outflows and their kinetic load;
4. local interaction I_2: update of landslide debris thickness and kinetic altitude;
5. internal transformation T_3: speed loss and decrease in kinetic load;
6. internal transformation T_4: calculation of maximum speed and time steps of the cellular automaton.

$\gamma : E \times \mathbb{N} \times Q_d \times Q_h \times Q_{h_k} \to Q_z \times Q_d \times Q_h \times Q_{h_k}$ is the activation sources function for landslides triggering at set steps of the cellular automaton. The main sources are triggered at the first step; other sources can be triggered later. $\mathbb{N}$ is the set of the natural numbers that correspond to the cellular automaton steps.

At the beginning of each simulation (step t = 0), the initial configuration of the system is specified by the states of all the cells. In particular, initial values are assigned as follows:

- $z \in Q_z$ is set equal to the altitude (bedrock elevation plus thickness of erodible bed);
- $d \in Q_d$ is the thickness of the soil cover, which can be eroded by the landslide (in source cells $d = 0$);
- $h \in Q_h$ is zero everywhere, except for the triggering area, where it is equal to d;
- $h_k \in Q_{h_k}$ is zero everywhere;
- $q_f \in Q_f$ is zero everywhere.

The transition function is then applied, step by step, to all the cells and the CA configuration changes: in this way, the evolution of the simulation is obtained.

General considerations

It is important to stress that the CA approach does not explicit velocity in the local context of the cell, since an amount of material (e.g., debris) "moves" from the central cell to an adjacent cell in a CA step, that is a constant time; that implies a constant "velocity". Nevertheless, velocity can be deduced through an analysis of the global behavior of the system. If we look at Hydrodynamics, we can deduce velocity and energy in a CA context.

In Hydrodynamics [43, 56] the kinetic load is defined as

$$h_k = \frac{v^2}{2g}$$

where v is the speed of the flow and g constant of gravity acceleration. Accordingly, if the thickness of the flow is set to h, the height that can be reached by the flow, indicated as the run-up H, can be defined as:

$$H = h + h_k$$

The transition function of SCIDDICA K1

In the following sections, a description of the elementary processes which constitute the transition function of SCIDDICA K1 is presented.

Internal transformation T_1: activation of triggering sources

The internal transformation T_1 causes the activation of triggering sources in E. In fact, for each cell in E is valid:

$$\begin{aligned} {}_nz &= z - d \\ {}_nh &= h + d \\ {}_nd &= 0 \end{aligned}$$

Internal transformation T_2: soil erosion

Determines the soil erosion and its effects. Let $p = vh$ be the product between the debris thickness, h, and its speed, v, this latter calculated as $v = \sqrt{2gh_k}$. The condition so that the erosion of the erodible substrate takes place is: $p > p_{mt}$. In this case, indicating as Δ_t the time corresponding to the CA current calculating step, the amount of eroded soil is:

$$\Delta_d = p \cdot p_{pef} \cdot \Delta_t$$

If $\Delta_d > d$, then $\Delta_d = d$. These new values allow to update, respectively, the substates altitude, depth of erodible soil cover, landslide debris thickness and kinetic load:

$$\begin{aligned} {}_nz &= z - \Delta_d \\ {}_nd &= d - \Delta_d \\ {}_nh &= h + \Delta_d \\ {}_nh_k &= \frac{h \cdot h_k}{h + \Delta_d} \end{aligned}$$

Local interaction I_1: calculation of debris outflows and their kinetic load

The local interaction I_1 determines the debris outflows from the central cell towards its adjacent cells, $f[0,i]$ $i = 0, 1, ..., 8$, and the relative kinetic load, $h_k[0,i]$. It is based on an opportune minimization algorithm, derived from the "minimization of the differences" proposed by Di Gregorio and Serra (1999) [29]. First of all, from the maximum run-up, H_{max}, the damping factor, r, is calculated:

$$r = \frac{h[0] + h_k[0]}{H_{max}}$$

Moreover, for the 8 adjacent cells are set these following values:

$$\begin{aligned} Z[i] &= z[i] \\ r[i] &= r \\ w[i] &= p_c \end{aligned}$$

for cells indexed from 1 to 4, while

$$\begin{aligned} Z[i] &= z[i] - \frac{z[0] - z[i]}{\sqrt{2}} \\ r[i] &= r/\sqrt{2} \\ w[i] &= p_c \cdot \sqrt{2} \end{aligned}$$

for cells indexed from 5 to 8 to consider that these latter are located along diagonal directions of the cellular space, so being at greater distance from the central cell (see figure 3.1). Moreover, for the central cell is imposed:

$$Z[0] = z[0] + h_k[0]$$

Then, follows a phase in which the adjacent cells are eliminated, so they will not able to receive outflows from the central cell. The only cells that will not be eliminated are those cells that satisfy one of these following requirements:

Case 1) $(Z[0] + h[0] > z[i] + h[i]) \wedge (Z[0] > z[i])$

Case 2) $(Z[0] + h[0] > z[i] + h[i]) \wedge \neg (Z[0] > z[i])$

In case 2) we set:

$$Z[i] = z[i] + h[i]$$

For both cases is finally calculated the slope between the central cell and the neighbor cell and the relative angle as:

$$\theta[i] = \arctan \frac{Z[0] + h[0] - Z[i]}{w[i]}$$

The Minimization Algorithm is then applied to the following quantities:

- $q(0) = Z[0] = h[0] + h_k[0]$;
- $m = h[0]$;
- $q(i) = Z[i]$ $(i = 1, 2, \ldots, 8)$;

The Minimization Algorithm eliminates those cells that cannot receive outflows and determines the average altitude, a, of the cells are not eliminated. For cells not eliminated, for which also applies the condition $h[0] > p_{hc_{\theta_0}} \cos\theta[i]$, are calculated outflows and their kinetic loads as follows:

$$f[0,i] = r[i] * (a - Z[i])$$
$$h_k[0,i] = (Z[0] + h[0] - f[0,i]) - Z[i]$$

Local interaction I_2: update of landslide debris thickness and kinetic altitude

The local interaction I_2 determines the update f landslide debris thickness and kinetic altitude.

$$_nh[0] = h[0] - \sum_{i=1}^{8} f[0,i] + \sum_{i=1}^{8} f[i,0]$$
$$_nh_k[0] = \frac{\sum_{i=0}^{8} f[i,0] \cdot h_k[i,0]}{_nh[0]}$$

The new value of landslide debris thickness is obtained by considering debris thickness variations, due to outflows and inflows from/into the central cell.

Internal transformation T_3: speed loss and decrease in kinetic load

The internal transformation T_3 determines the speed loss following a quadratic mechanism and the consequent decrease in kinetic load as:

$$v = \sqrt{2gh_k}$$
$$_nv = v^2 P_{dQ} \Delta_t$$
$$_nh_k = \frac{v^2}{2g}$$

Internal transformation T_4: calculation of maximum speed and time steps of the cellular automaton

Finally, the internal transformation T_4 calculates the maximum speed and the time steps of the cellular automaton as follows:

$$H_{max} = \max_R \{h_k + h\}$$
$$v_{max} = \sqrt{2gH_{max}}$$
$$\Delta_t = p_c/v_{max}$$

3.1.1.1 Applications of the model SCIDDICA K1.

The real event of May 1998 in the Pizzo d'Alvano area

Over the past decades, several studies have attempted to characterize [4,38, 72] and model [5,34,44] debris flows. Debris flows are a mixture of water and sediments that move gaining speed and channeling along the river courses; as a consequence, they can have a destructive strength.

Generally, debris flows developed during intense storms as a result of prolonged rainfall. The mobilization of debris can also occur as a result of dynamic stress, such as those caused by impact, vibration, earthquakes, or in conjunction with volcanic eruptions [47]. Before the event, debris are in a (precarious) balance condition on a side or along a river bank. After the collapse, debris propagate downstream and assume the character of a debris flow also cause of significant values of slope relative to the morphological characters and water availability. In most cases, debris flows originate as slipping surface of soil [39], but they can also be originated for mobilization of sediment previously accumulated along the drainage grating (for rapid erosion along the gullies) or collapse of overlap (natural or artificial) [22,63]. Usually, the development of debris flows is quick and not always preceded by remarkable signals; for this reason, they have often surprised unprepared people. These landslides can reach very high speed (up to tens of meters per second). In correspondence of the end of the path, debris flows decrease in speed quickly and distribute the debris to the mouth of the basin.

A real event occurred on 5 and 6 May 1998: on the side of the massif of Pizzo d'Alvano (Campania, Southern Italy), have been triggered thousands of landslides and debris flows [18]. The flow of the material deeply eroded the debris layer (ranging from a few centimeters to several meters) along the path toward downstream, while the impact with urban areas has caused huge damage and, unfortunately, 161 victims.

In figures 3.2 and 3.3 we can see some of the main landslides occurred on the southern slope of Pizzo d'Alvano. The debris flow of Chiappe di Sarno

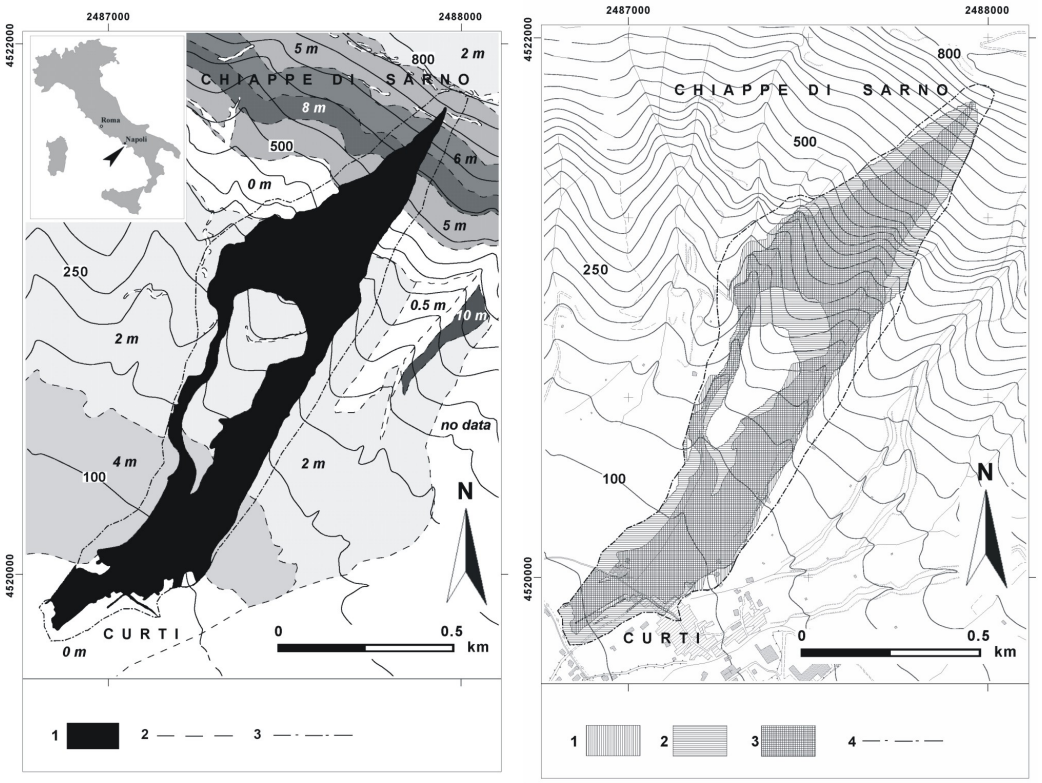
(figure 3.2a) initially propagated along a plain-convex slope, then divided itself into two sections, joined in a successive phase at the base of the massif causing serious damage in the Curti area. Pestello Storto landslide (figure 3.3a) is an example of a well-channeled landslide, even if the flow impacts against a wall downstream, partially reducing its advance.

Simulations of the cases described so far, have been performed for SCIDDICA K1 model calibration purposes.

Application of the model to the real cases of Chiappe di Sarno and Pestello Storto

A preliminary calibration of the CA has allowed to simulate with a well approximation the cases studies considered. The figures 3.2b and 3.3b illustrate the results of the best simulations and comparison with real events, also performed in GIS[1] environment. From a qualitative point of view, the main features of the phenomenon have been played.

[1]GIS (Geographic Information System): a computerized information system composed of a set of software tools to capture, store, extract, transform, analyze and display spatial data from the real world, associating with each geographic feature one or more alphanumeric descriptions.



(a) (b)

Figure 3.2: (a) The landslide of Chiappe di Sarno: (1) identifies the area affected by the landslide; (2) identifies the limit of the areas with a constant thickness of erodible soil (indicated in italics); (3) identifies the border of the analysis performed in GIS environment. (b) Comparison between real and simulated landslide: (1) identifies the areas affected by the real landslide; (2) identifies the areas affected by the simulated landslide; (3) identifies the areas affected by both landslides; (4) identifies the border of the analysis performed in GIS environment.

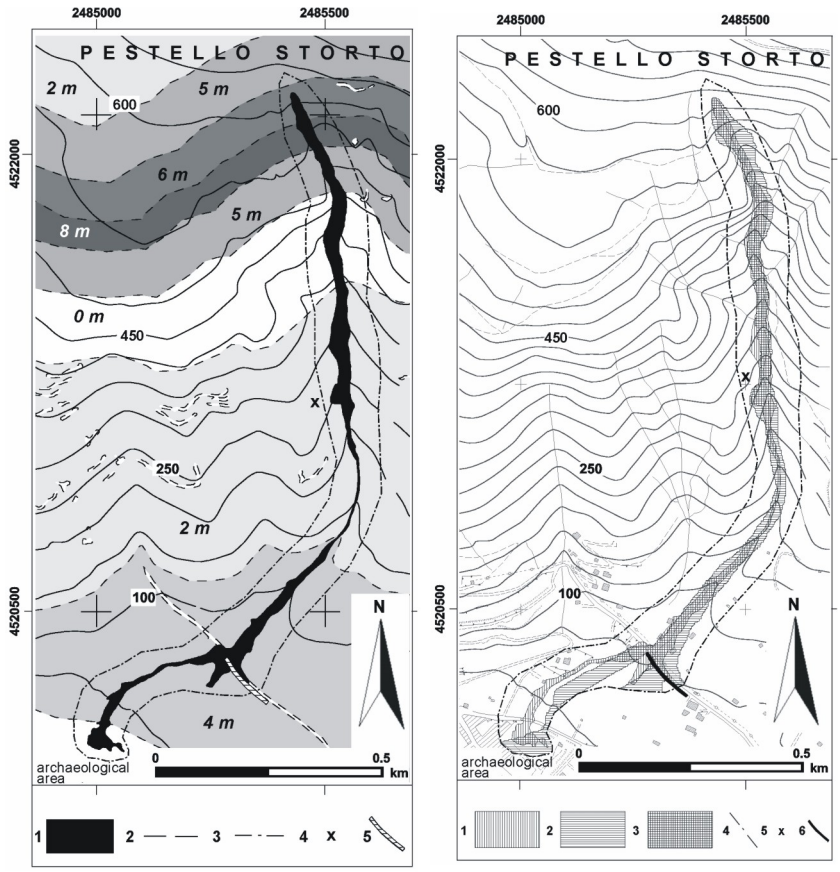


Figure 3.3: (a) The landslide of Pestello Storto: (1) identifies the area affected by the landslide; (2) identifies the limit of the areas with a constant thickness of the erodible soil (indicated in italic); (3) identifies the border of the analysis performed in GIS environment; (4) identifies the sites of secondary ignition; (5) identifies natural or artificial barriers. (b) Comparison between real and simulated landslide: (1) identifies the areas affected by the real landslide; (2) identifies the areas affected by the simulated landslide; (3) identifies the areas affected by both landslides; (4) identifies the border of the analysis performed in GIS environment; (5) identifies the sites of secondary ignition; (6) identifies natural or artificial barriers.

3.1.2 SCIARA-fv3 - Model Formalization

3.1.2.1 Model Overview

SCIARA-fv3 is the latest release of the SCIARA family of Complex Cellular Automata Models for simulating basaltic lava flows. As its predecessor, SCIARA-fv2, it is based on a Bingham-like rheology. However, unlike fv2, it explicitly computes the flow momentum and the time corresponding to the computational step (CA clock). In formal terms, it is defined as:

$$SCIARA - fv3 =< R, X, Q, P, \tau, L, \gamma >$$

where:

1. R is the cellular space, the set of square cells that define the bi-dimensional finite region where the phenomenon evolves.
2. X is the pattern of cells belonging to the Moore neighborhood that influence the cell state change (see fig. 3.4)
3. $Q = Q_z \times Q_h \times Q_T \times Q_{\overrightarrow{p}} \times Q_f^9 \times Q_{\overrightarrow{vf}}^9$ is the finite set of states, considered as Cartesian product of substates. Their meanings are: cell altitude a.s.l., cell lava thickness, cell lava temperature, momentum (both x and y components), lava thickness outflows (from the central cell toward the adjacent cells) and flows velocities (both x and y components), respectively;
4. $P = w, t_0, P_T, P_d, P_{hc}, \delta, \rho, \epsilon, \sigma, c_v$ is the finite set of parameters (invariant in time and space), whose meaning is illustrated in Tab. 3.1; note that P_T, P_d , and P_{hc} are set of parameters;
5. $\tau : Q^9 \longmapsto Q$ is the cell deterministic transition function; it is splitted in *"elementary processes"* which, are described in subsection 3.1.2.2;
6. $L \subseteq R$ specifies the emitted lava thickness from the source cells (i.e. craters);
7. $\gamma : Q_h \times \mathbb{N} \longmapsto Q_h$ specifies the emitted lava thickness from the source cells at each step $k \in \mathbb{N}$

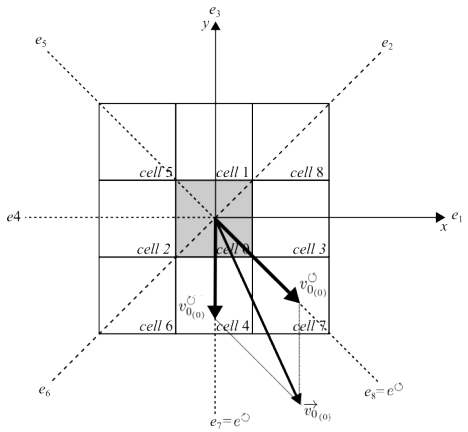


Figure 3.4: Example of Moore neighborhood and decomposition of momentum along the cellular space directions. Cells are indexes from 0 (the central cell, in grey) to 8. Cells integer coordinates are omitted for a better readability.

3.1.2.2 Elementary process

Elementary process τ_1: lava flows computation

The elementary process τ_1 computes lava outflows and their velocities. It is formally defined as:

$$\tau_1 : Q_z^9 \times Q_h^9 \times Q_{\overrightarrow{p}} \rightarrow Q_f^9 \times Q_{\overrightarrow{v_f}}^9$$

Lava flows are computed by a two-step process: the first computes the CA clock, t, i.e. the physical time corresponding to a CA computational step, while the second the effective lava outflows, $h_{(0,i)}$, their velocities $v_{f_{(0,i)}}$ and displacements $s_{(0,i)}$ $(i = 0, 1, ..., 8)$. The elementary process τ_1 is thus executed two times, the first one in "time evaluation mode", the second in "flow computing mode". Both modes compute the so called "minimizing outflows", $\phi_{(0,i)}$, i.e. those which minimize the unbalance conditions within the neighborhood, besides their final velocities and displacements. In "time evaluation mode", t is preliminary set to a large value, $t_{\max}$, and the computed displacement, $s_{(0,i)}$, is compared with the maximum allowed value, $d_{(0,i)}$, which is set to the distance between the central cell and the neighbor that receives the flow. In case of over-displacement, the time t must be opportunely reduced in order to avoid the overflow condition. In case no over-displacement are obtained, t remains unchanged. Eventually, in "flow computing mode", effective lava outflows, $h_{(0,i)}$, are computed by adopting the CA clock obtained

Table 3.1: List of parameters of SCIARA-fv3 with values considered for the simulation of the 2006 Etnean lava flow.

Parameter	Meaning	Unit	Best value
w	Cell side	[m]	10
t_0	Initial CA clock	[s]	1
$t_{\max}$	Upper value for the CA clock	[s]	120
P_T			
T_{sol}	Temperature of solidification	[K]	1143
T_{vent}	Temperature of extrusion	[K]	1360
P_d			
$dP_{T_{sol}}$	Dissipation factor at solidification	-	0.5
$dP_{T_{vent}}$	Dissipation at extrusion	-	0.315
P_{hc}			
$hc_{T_{sol}}$	Critical height at solidification	[m]	23.066
$hc_{T_{vent}}$	Critical height at extrusion	[m]	1.014
r	Relaxation rate	-	0.5
δ	Cooling parameter	-	1.5070
ρ	Lava density	[Kg m^{-3}]	2600
ϵ	Lava emissivity	-	0.9
c_v	Specific heat	[J kg^{-1} K^{-1}]	1150

in "time evaluation mode", by guarantying no overflow condition.

Computation of the minimizing outflows $\phi_{(0,i)}$

The initial velocity of the lava inside the cell, $\overrightarrow{v_0}_{(0)}$, is obtained from the momentum components. In turn, it is decomposed in two components laying over the two directions of the CA cellular space which are the nearest with respect to $\overrightarrow{v_0}_{(0)}$ itself. These latter directions, which will be indicated by $e^{\circlearrowleft}$ and $e^{\circlearrowright}$, can be found by moving in counterclockwise and clockwise directions starting from the direction of $\overrightarrow{v_0}_{(0)}$, respectively, as shown in Fig. 3.4. Thus, if i denotes the i-th direction of the cellular space, $v_{0_{(0)}}^{\circlearrowleft}$ and $v_{0_{(0)}}^{\circlearrowright}$ the modules of the components of $\overrightarrow{v_0}_{(0)}$ along the directions $e^{\circlearrowleft}$ and $e^{\circlearrowright}$, respectively, then the modules of the components of $\overrightarrow{v_0}_{(0)}$ along the directions of the cellular

space can be expressed as:

$$v_{0_{(0,i)}} = \begin{cases} v^{\circlearrowleft}_{0_{(0)}}, & \text{if } i = e^{\circlearrowleft} \\ v^{\circlearrowright}_{0_{(0)}}, & \text{if } i = e^{\circlearrowright} \\ 0, & \text{otherwise} \end{cases}$$

Moreover, let $h_{k(0,i)} = v^2_{0_{(0,i)}}/2g$ denote the kinetic head associated to the i-th component of velocity.

Viscosity effects are modeled in terms of velocity dissipation mechanism, by means of the function dP. It depends on temperature and vary according to a power law of the type $\log dP = a + bT$, where $T \in Q_T$ is the lava temperature and a and b are coefficients determined by solving the system (cf. Tab. 3.1):

$$\begin{cases} \log dP_{T_{sol}} = a + bT_{sol} \\ \log dP_{T_{vent}} = a + bT_{vent} \end{cases}$$

Similarly, the relation between critical height and lava temperature can be described by a power law of the kind $\log hc = c + dT$ whose coefficients are obtained by solving the system (cf. Tab. 3.1):

$$\begin{cases} \log hc_{T_{sol}} = c + dT_{sol} \\ \log hc_{T_{vent}} = c + dT_{vent} \end{cases}$$

Before applying the minimization algorithm of the differences for computing the minimizing outflows, a preliminary control was performed to eliminating cells that cannot receive lava due to their energy conditions. As in [60], a topographic correction is considered for flow symmetry reason. In addition, in SCIARA-fv3 the concepts of effective height, $h_{e_{(0,i)}}$, and apparent height, $h_{a_{(0,i)}}$, was introduced. The first is the part of $h_{(0)}$ that can really flow out of the cell toward its i-th neighborhood, while the second one is the part which is constrained inside the cell due to energy conditions. There are three cases (see Fig. 3.5):

1. if $z_{(0)} + h_{k_{(0,i)}} + h_{(0)} \leq z_{(i)} + h_{(i)}$, then
$$\begin{cases} h_{e_{(0,i)}} = 0 \\ h_{a_{(0,i)}} = h_{(0)} \end{cases}$$
2. if $z_{(0)} + h_{k_{(0,i)}} < z_{(i)} + h_{(i)} < z_{(0)} + hk_{(0,i)} + h_{(0)}$, then
$$\begin{cases} h_{e_{(0,i)}} = (z_{(0)} + h_{k_{(0,i)}} + h_{(0)}) - (z_{(i)} + h_{(i)}) \\ h_{a_{(0,i)}} = h_{(0)} - h_{e_{(0,i)}} \end{cases}$$

3. if $z_{(i)} + h_{(i)} \leq z_{(0)} + h_{k_{(0,i)}}$, then
$$\begin{cases} h_{e_{(0,i)}} = h_{(0)} \\ h_{a_{(0,i)}} = 0 \end{cases}$$

Thus, if denoting with $\theta_{(0,i)} = \arctan((z_{(0)} + h_{a_{(0,i)}} + h_{e_{(0,i)}}/2) - (z_{(i)} + h_{(i)}))$ the slope angle between the central cell and its i-th neighbor (see Fig. 3.5), according to the concept of critical height, the cells for which

$$h_{e_{(0,i)}} \leq hc \cos\theta_i$$

are eliminated and cannot receive flow.

The minimization algorithm of the differences is therefore applied to the following quantities, in order to compute the minimizing outflows:

$u_{(0)} = z_{(0)}$
$m = h_{(0)}$
$u_{(i)} = z_{(i)} + h_{(i)}$

The application of the algorithm determines the computation of the minimizing flows, $\phi_{(0,i)}$, from the central cell to the i-th neighbor, where $\phi_{(0,0)}$ represents the residual flow which does not leave the cell. Eventually, final velocities and displacements are computed. As a first step, final velocities are computed for each outflow $\phi_{(0,i)}$ $(i = 1, 2, \ldots, 8)$, by taking into account dissipation:

$$v_{f_{(0,i)}} = (v_{0_{(0,i)}} + at)(1 - dP)$$

Here, $a = g \sin\theta$ is the acceleration of gravity, and does not take into account dissipation, which is modeled by the function dP. Instead, the final velocity of $\phi_{(0,0)}$ is computed as:

$$v_{f_{(0,0)}} = v_{0_{(0)}}(1 - dP)$$

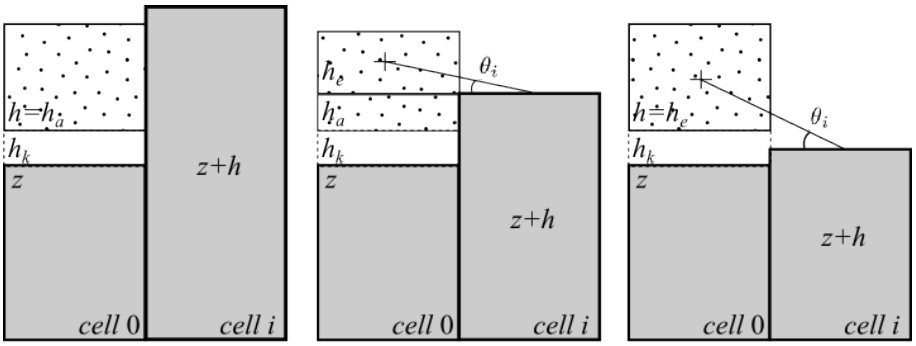
In order to compute the displacement, a mean acceleration is computed, which also takes into account dissipation effects: $\overline{a} = (v_{f_{(0,i)}} - v_{0_{(0,i)}})/t$. Therefore, the displacements $s_{(0,i)}$ $(i = 1, 2, \ldots, 9)$ are computed as:

$$s_{(0,i)} = v_{0_{(0,i)}} t + \frac{1}{2}\overline{a}t^2$$

while, a null displacement is assigned to $\phi_{(0,0)}$:

$$s_{(0,0)} = 0$$

since, even if in the real case a movement can occur, inside the discrete context of the cellular space, it is always located at the center of the cell.



Case 1 *Case* 2 *Case* 2

Figure 3.5: Cases in which the generic neighbor (*cell* i) is eliminated or not eliminated by the minimization algorithm of the difference. If the neighbor is eliminated (*Case* 1), the overall amount of debris inside the central cell is considered as apparent ($h = h_a$), and can not generate an outflow. If the neighbor is not eliminated (*Case* 2 and 3), a part (*Case* 2) or the entire amount of debris (*Case* 3) on the central cell is considered effective ($h \geq h_e$) and can generate outflows. Note that the slope angle θ, considered in the critical height computation, is also shown.

This is a model simplification which is much more correct as the smaller the size of the cell is.

Time evaluation

Once the minimizing outflows are computed, the CA clock can be determined. As stated above, when τ_1 is executed in "time evaluation mode", t is preliminary set to a large value, $t_{\max}$. As a consequence, the computed displacements, $s_{(0,i)}$, can overcome the maximum allowed distance, w, i.e. the distance between the central cell and the neighbor that receive the flow. In case of over-displacement, i.e. $s_{(0,i)} > w$, the time t must be opportunely reduced in order to avoid the overflow. The new value of t is determined as follows:

- for each minimizing flow, $\phi_{(0,i)}$, a new time, $t_{(0,i)}$, is computed by imposing $s_{(0,i)} = w$ and by solving the equation with respect to t:

$$t_{(0,i)} = t = \frac{-v_{0_{(0,i)}} + \sqrt{v_{0_{(0,i)}}^2 + 2\overline{a}w}}{\overline{a}}$$

  so that overflow is avoided between the central cell and its i-th neighbor;

- a new time, t_j, is computed in order to avoid overflow conditions along all the neighborhood as:

$$t_c = \min_{i=1,2,\ldots,8} t_{(0,i)}$$

so that overflow is avoided in all the neighborhood;

- a new minimal time, t_{opt}, is computed as:

$$t_{opt} = \min_{c \in R} t_c$$

in order to avoid overflow conditions over all the cellular space R;

- t_{opt} is multiplied by a relaxation rate factor, $0 < r \leq 1$, for smoothing the phenomenon, and the new CA clock, $\overline{t}$, is obtained:

$$\overline{t} = t_{opt} r$$

Outflows computation

In "flow computing mode", minimizing outflows, $\phi_{(0,i)}$, are re-computed by considering the new CA clock $\overline{t}$. Subsequently, lava outflows, $h_{(0,i)}$, are computed proportionally to the displacement, by simply multiplying the minimizing outflow by the ratio between the actual displacement and the maximum allowed:

$$h_{(0,i)} = \phi_{(0,i)} \frac{s_{(0,i)}}{w}$$

Final velocity and displacement are computed as in subsection 3.1.2.2.

Elementary process τ_2: updating of mass and momentum

The elementary process updates lava thickness and momentum. It is formally defined as:

$$\tau_2 : Q_f^9 \times Q_{\overrightarrow{vf}}^9 \rightarrow Q_h \times Q_{\overrightarrow{p}}$$

Once the outflows $h_{(0,i)}$ are known for each cell $c \in R$, the new lava thickness inside the cell can be obtained by considering the mass balance between inflows and outflows:

$$h_{(0)} = \sum_{i=0}^{9} (h_{(i,0)} - h_{(0,i)})$$

Moreover, also the new value for the momentum can be updated by ac-

cumulating the contributions given by the inflows:

$$\overrightarrow{p}_{(0)} = \sum_{i=0}^{9} h_{(i,0)} \overrightarrow{v_f}_{(i,0)}$$

Elementary process τ_3: temperature variation and lava solidification

$$\tau_3 : Q_f^9 \times Q_T^9 \rightarrow Q_T \times Q_h$$

As in the elementary process τ_1, a two step process determines the new cell lava temperature. In the first one, the temperature is obtained as weighted average of residual lava inside the cell and lava inflows from neighboring ones:

$$\overline{T} = \frac{\sum_{i=0}^{8} h_{(i,0)} T_i}{\sum_{i=0}^{8} h_{(i,0)}}$$

A further step updates the calculated temperature by considering thermal energy loss due to lava surface radiation [51]:

$$T = \frac{\overline{T}}{\sqrt[3]{1 + \frac{3\overline{T}^3 \epsilon \sigma \bar{t} \delta}{\rho c_v w^2 h}}}$$

where ϵ, σ, $\bar{t}$, δ, ρ, c_v, w and h are the lava emissivity, the Stephan-Boltzmann constant, the CA clock, the cooling parameter, the lava density, the specific heat, the cell side and the debris thickness, respectively (see Tab. 3.1). When the lava temperature drops below the threshold T_{sol}, lava solidifies. Consequently, the cell altitude increases by an amount equal to lava thickness and new lava thickness is set to zero.

Lava flows are computed by a two-step process: the first computes the CA clock, t, i.e. the physical time corresponding to a CA computational step, while the second the effective lava outflows, $h_{(0,i)}$, their velocities $v_{f_{(0,i)}}$ and displacements $s_{(0,i)}$ $(i = 0, 1, ..., 8)$. The elementary process τ_1 is thus executed two times, the first one in "time evaluation mode", the second in "flow computing mode". Both modes compute the so called "minimizing outflows", $\phi_{(0,i)}$, i.e. those which minimize the unbalance conditions within the neighborhood, besides their final velocities and displacements. In "time evaluation mode", t is preliminary set to a large value, $t_{\max}$, and the computed displacement, $s_{(0,i)}$, is compared with the maximum allowed value, $d_{(0,i)}$, which is set to the distance between the central cell and the neighbor that receives the flow. In case of over-displacement, the time t must be opportunely reduced

in order to avoid the overflow condition. In case no over-displacement are obtained, t remains unchanged. Eventually, in "flow computing mode", effective lava outflows, $h_{(0,i)}$, are computed by adopting the CA clock obtained in "time evaluation mode", by guarantying no overflow condition.

3.1.3 ABBAMPAU a CA for Wildfire Simulation and Risk Assessment

A classical homogeneous CA can be viewed as a uniform lattice of cells representing identical finite automata, hence endowed with a state and a transition function. Each cell takes as input its state and the states of a set of neighbouring cells defined by a geometrical pattern which is invariant in time and space. Starting from an initial condition, the CA evolves activating on a step by step basis all the identical transition functions simultaneously.

In order to effectively use the CA dynamics for the simulation of spatially complex systems, many extensions of its original definition have been proposed in literature. Typically, the classical CA paradigm was modified to overcome some implicit limits, such as having few states, look-up table transition functions and invariant neighbourhoods [10–12, 20, 67]. The model described in the following, as well as the used formalism, are based on one of such extended CA notions, namely on the Macroscopic Cellular Automata approach introduced in [20] and already exploited for the simulation of many macroscopic phenomena [6, 16, 21, 53, 60, 68].

As in most wildfire spread simulators [23, 32, 42, 52, 69], the approach adopted in this study is based on the Rothermel's fire model [54, 55], which provides the heading rate and direction of spread given the local landscape and wind characteristics. An additional constituent is the commonly assumed elliptical description of the spread under homogeneous conditions (i.e. spatially and temporally constant fuels, wind and topography) [1].

In order to mitigate the accuracy problems that affects most raster-based wildfire simulators [23, 24, 35, 52], the CA adopted in this paper extends the size of the commonly used Moore's neighbourhood and adopts an initial randomization of the spread directions. As shown in [7], such an approach, which does not alter the deterministic nature of the model, provides relevant beneficial effects on the overall accuracy.

More in details, the two-dimensional fire propagation is locally obtained by a growing ellipse having the semi-major axis along the direction of maximum spread, the eccentricity related to the intensity of the so-called *effective wind* and one focus acting as a 'fire source' [52, 68]. At each CA step, the ellipse's size is increased according to both the duration of the time step and

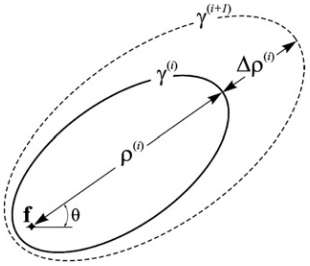


Figure 3.6: Growth of the ellipse γ locally representing the fire front. The symbol ρ denotes the forward spread which is incremented by $\Delta\rho$ at the i-th time step.

maximum rate of spread (see Figure 3.6). Afterwards, a neighbouring cell invaded by the growing ellipse is considered as a candidate to be ignited by the spreading fire. In case of ignition, a new ellipse is generated according to the amount of overlapping between the invading ellipse and the ignited cell. Formally, the model is a two-dimensional CA with square cells defined as:

$$CA = \langle \mathcal{K}, \mathcal{Q}, \mathcal{N}, \mathcal{S}, \mathcal{P}, \eta, \psi, \phi \rangle \tag{3.1}$$

where:

- $\mathcal{K}$ is the set of points in the finite region where the phenomenon evolves. Each point represents the centre of a square cell;
- $\mathcal{Q}$ is a set of *randomized local sources* (RLSs) [7], one point for each cell; they are randomly generated at the beginning of the simulation within an assigned small radius from each of the centres in $\mathcal{K}$. As detailed later, a new ignition in a cell consists of a new ellipse having its rear focus on the local source $\mathbf{q} \in \mathcal{Q}$;
- $\mathcal{N}$ is the set that identifies the pattern of cells influencing the cell state change (i.e. the neighbourhood);
- $\mathcal{S}$ is the finite set of the states of the cell, defined as the Cartesian product of the sets of all the cell's substates;
- $\mathcal{P}$ is the finite set of global parameters, including those that define the fuel bed characteristics according to the standard fuel models used in BEHAVE [3];
- $\eta : \mathcal{S}^{|\mathcal{N}|} \rightarrow \mathcal{S}$ is the transition function accounting for the fire ignition, spread and extinction mechanisms;

– $\psi : \mathcal{S}^{|\mathcal{K}|} \to \mathbb{R}$ is a function that determines the size Δt of each time step according to both the digital terrain model and maximum spread rate among the cells on the current fire front. The value of Δt is then used by the function $\phi : \mathbb{R} \to \mathbb{R}$, for keeping the current time t up to date.

The cell's substates include all the local quantities used by the transition function for modelling the local interactions between the cells (i.e. the fire propagation to neighbouring cells) as well as its internal dynamics (i.e. the fire ignition and growth). In particular, among the substates that define the state of each cell, there are:

– the altitude $z \in \mathbb{R}$ of the cell;

– the fuel model $\mu \in \mathbb{N}$, which is an index referring to one of the mentioned standard models that specify the characteristics of vegetation relevant to Rothermel's equations;

– the combustion state $\sigma \in S_\sigma$, which takes one of the values '*unburnable*', '*burnable*', '*ignited*' and '*burnt*'.

– the accumulated forward spread $\rho \in \mathbb{R}_{\geq 0}$, that is the current distance between the focus $\mathbf{f}$ of the local ellipse and the farthest point on the semi-major axis (see Figure 3.6);

– the angle $\theta \in \mathbb{R}$ (see Figure 3.6), giving the direction of the maximum rate of spread, obtained through the composition of two vectors, namely the so-called *wind effect* and *slope effect* [54];

– the maximum rate of spread $r \in \mathbb{R}_{\geq 0}$, also provided by Rothermel's equations on the basis of the relevant local characteristics [54];

– the eccentricity $\varepsilon \in [0, 1]$ of the ellipse γ representing the local fire front, which is obtained as a function of both the wind and terrain slope through the empirical relation proposed in [2, 23].

In brief, the scheduling of each CA step is organized as follows:

1. first, the global function ψ computes the current duration of the time step Δt and the function ϕ updates the current time t;

2. afterwards, the transition function η is executed for each cell of the automaton. This involves spreading the fire to the neighbouring cells, during the time interval Δt, according to the algorithm described below;

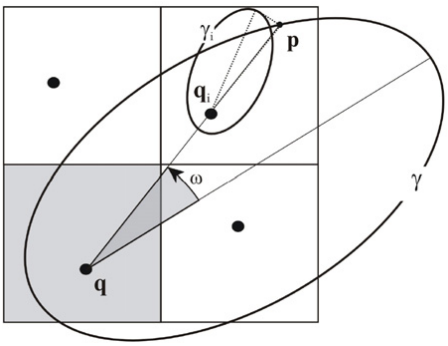


Figure 3.7: The i-th neighbouring cell intersected by the ellipse γ locally representing the fire front.

3. finally, if t is less than a final time t_f a new step is executed, otherwise the simulation ends.

According to the transition function η outlined in Algorithm 1, if the cells are not in the '*burning*' state no further calculation is performed. Otherwise, in case of burning cell, the first step of η consists of checking the condition that triggers the transition to the '*burnt*' state. The latter is verified when none of the neighbouring cells are in the '*burnable*' state, that is when the cell's contribution is no longer necessary to the fire spread mechanism. Then, if the cell still belongs to the fire front, η updates the size of the local ellipse

Algorithm 1: Cell's transition function η.

```
1  if σ = 'burning' then
2      if none of the neighbours are in the 'burnable' state then
3          σ ← 'burned';
4          return;
5      ρ ← ρ + r Δt;
6      foreach cell c_i in the neighbourhood do
7          if the substate σ_i of c_i is 'burnable' then
8              if the cell c_i is reached by γ then
9                  Compute r_i and θ_i;
10                 if r_i > 0 then
11                     σ_i ← 'ignited';
12                     Compute ε_i;
13                     Compute the current local spread ρ_i;
```

γ (line 5).

The next statement of η consists of testing if the fire is spreading towards other cells $\mathbf{c}_i$ of the neighbourhood that are in the '*burnable*' state (lines 6-8). Such a spread test is carried out by checking if γ includes the RLS $\mathbf{q}_i$ of the cell $\mathbf{c}_i$ (see Figure 3.7). If $\mathbf{q}_i$ is inside γ, then a new ellipse γ_i is generated for the cell $\mathbf{c}_i$, having the RLS $\mathbf{q}_i$ assumed as rear focus and r_i, θ_i and ε_i computed through the proper model equations [23, 45, 54]. Further details of the model, including some of the relevant model equations, can be found in [7].

As mentioned above, the CA model is based on the extended Moore's neighbourhood composed of 25 cells represented in Figure 3.8. Also, the use of the RLSs inside each cell allows for obtaining a high number of different spread direction during the fire propagation in a landscape, thus significantly improving the accuracy of the results [7]. However, it is important to note that since the RLSs are generated only once before the beginning of the simulation, the adopted CA model is deterministic.

Clearly a higher number of neighbouring cells corresponds to a higher computational cost of the CA step. Nevertheless, the model can also be used with the standard Moore's neighbourhood composed of 9 cells, still obtaining acceptable accuracies thanks to the adopted RSLs [7].

As stated before, the size Δt of each time-step is determined on a non-local basis by the global function ψ. In general, large values of Δt speed up the simulation because a lower number of steps are required for reaching the final time. However, small values of Δt help to decrease the accumulation of errors during the spread mechanism. Therefore, the main aims of ψ are: (i) to ensure that at least a new ignition will take place during the next step; (ii) to guarantee that the ellipse from the cell having the highest rate of spread does not go beyond any of its neighbouring cells. In practice, at each CA step the function ψ operates as follows. For each burning cell $\mathbf{c}$ (i.e. belonging to the current fire front) and for each of its neighbours $\mathbf{c}_i$ in the '*burnable*' state, the time required by the fire to reach the RLS $\mathbf{q}_i$ of $\mathbf{c_i}$ is computed as:

$$\Delta t_i = \frac{\mid \mathbf{q}_i - \mathbf{q} \mid - \rho_i}{r} \tag{3.2}$$

where $\mathbf{q}$ and r are the RLS and the rate of spread of $\mathbf{c}$, respectively, and ρ_i is the current spread along the vector $\mathbf{q}_i - \mathbf{q}$. Then, to generate at least an ignition at the next step, the value of Δt can be defined as the minimum among all the Δt_i defined by Equation 3.2 [42, 52]. However, the model also allows slightly higher values so that to speed up the simulations with a low impact on the overall accuracy.

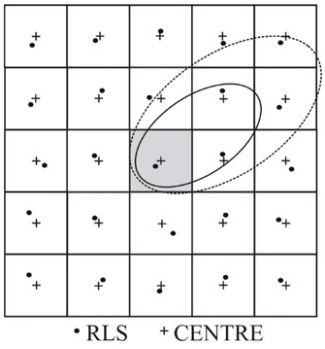


Figure 3.8: The adopted extended neighbourhood $\mathcal{N}$ composed of 25 cells together with an example of RLSs inside each cell.

Note that, to account for a sloping terrain, in Equation 3.2 as well as in Algorithm 1 the altitude of each cell must be considered (i.e. when computing distances, all the involved vectors must be viewed in the three-dimensional Cartesian space).

Many important optimizations of the procedure outlined above can be implemented. The most typical consists of using a suitable dynamic data structure that allows to operate only on the cells belonging to the current fire front (i.e. for which $\sigma =$ '*burning*'). Moreover, in case of repeated simulations with stationary weather conditions and different points of ignition, many substates could be precomputed once (e.g. lines 9 and 12 of the Algorithm 1). Other optimizations could regard the memory usage: for example, it is easy to combine σ and ρ into a single substate. Nevertheless, as shown later in the paper, the same enhancements can be effectively adopted also in the parallel implementation.

3.2 Web applications

In this work, I developed three new scientific Web applications for the simulation of complex macroscopic natural phenomena, namely debris flows, lava flows, and wildfire evolution. The applications are presented below and are called Swii2, SciaraWii and Awii, respectively. The description of all of them is fully complete, even if some repetitions are present. This was done for allowing the reader to jump to the description of a given application without the need to read the other ones.

A Web application is a software that runs in a web browser. It is created in a browser-supported programming language, such as the combination

of JavaScript, HTML and CSS, and relies on a web browser to render the application. The applications here presented were implemented by means of the Google Web Toolkit (GWT), an open source set of tools that allows web developers to create and maintain complex JavaScript front-end applications by means of the Java programming language. Using GWT, developers can develop and debug Ajax applications in Java, by adopting the development environment and tools of their choice. When the application is deployed, the GWT cross-compiler translates the Java application to standalone JavaScript files that are optionally obfuscated and deeply optimized. When needed, JavaScript can also be embedded directly into Java code, using Java comments.

3.2.1 Swii2

Swii2 [50] (SCIDDICA Web Interactive Interface 2) is a web application for debris flows simulation. It is based on the SCIDDICA-k1 CA model, the latest release of the SCIDDICA debris flow Cellular Automata family.

3.2.1.1 The system architecture

As stated above, Swii2 was implemented by means GWT. Following the GWT development approach, the interaction between the user interface and the SCIDDICA-k1 computational model is guarantied by a set of client-side services which are implemented on the server (see e.g. [31]). Multi-client connections are also guarantied. Whenever a user logs in, an asynchronous request is sent to the server in order to establish a connection; here, a servlet binds the client to an individual connection-handler, which allows multiple unambiguous communications through HTTP requests and responses.

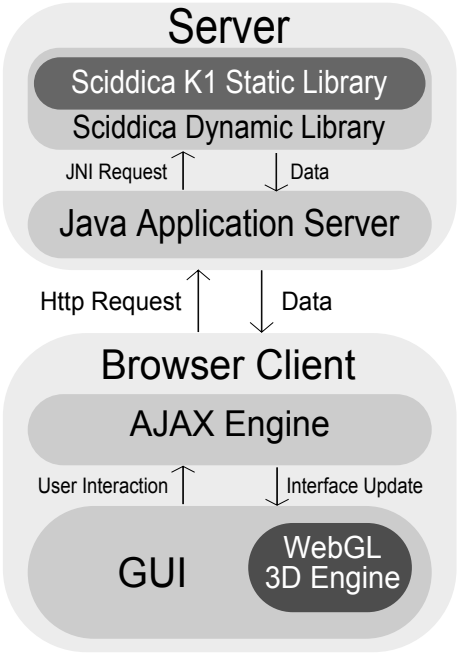


Figure 3.9: The Swii2 system architecture.

Figure 3.9 shows the Swii2 system architecture. Server-side, SCIDDICA-k1 is implemented as a static library, developed in C++ for efficiency reasons. A dynamic-link library (DLL), specifically developed in C++ for permitting the interaction between the simulation model and the Web application, receives requests by Java Native Interface (JNI) methods and provides simulation data to the application server. Data is therefore sent to the client via HTTP and stored into the Web browser cache memory. SCIDDICA-k1 parameters are displayed in GUI controls (in which they can also be modified), while simulation data (e.g. the topographic surface or the simulated debris flow) are visualized by means of the 3D WebGL rendering engine, which runs on a HTML5 `<Canvas>`. Thus, whenever the SCIDDICA-k1 simulation produces a debris flow, it is displayed over the surface and its dynamical behavior can be observed. All the client-server communications are managed by means of asynchronous JavaScript calls, which are able to provide the same usability level of desktop applications to Swii2.

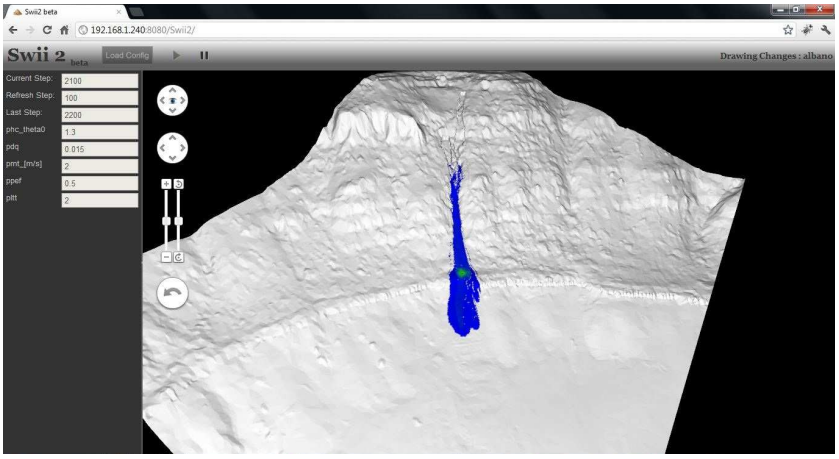


Figure 3.10: A screenshot of Swii2 during a simulation performed by the SCIDDICA-k1 debris flow molel. The left panel allows to view/set both SCIDDICA-k1 and simulation parameters (e.g. current and visualization step).

3.2.1.2 The Swii2 GUI and the visualization system

As previously stated, the Swii2 GUI was mainly written in JavaScript by means of GWT. The graphic layout of some elements has however been modified by means of cascade stile sheets (CSS). Figure 3.10 shows a screenshot of Swii2.

On the upper part of the GUI, a horizontal panel shows the application name/logo and contains the controls which permit to interact with the simulation. A notification area is also present on the right side of the panel. The remaining client area is subdivided in two panels. The left one contains the controls which permits to show the current simulation step, set the graphic update interval and show/edit the SCIDDICA-k1 parameters. The right one contains the graphic output of the simulation. The graphic panel also contains additional controls (in Google Maps style), which allows the user to interact with the 3D model in a very simple way. Buttons have been developed for users which also adopt touch-screen devices. Moreover, application interaction is also possible through mouse based movements. In fact, like most 3D modeling applications, users can move or rotate by dragging the 3D model and zoom it by mouse wheeling. However, as the application is currently under development, the GUI is still oversimplified and some functionalities of the SCIDDICA-k1 simulation model cannot be used through the Swii2 user interface. For instance, it is not possible to upload configuration data to the server directly from Swii2 and the operation must be performed

externally, e.g. through a FTP application, where data must be placed in a specific directory.

As regards the rendering engine, it has been developed by means of the gwt-g3d library[2], which makes easy the integration in GWT. As a matter of fact, it simplifies many development aspects like projection, matrix operations or shader binding. Data meshing has been based on triangle strips. For performance reasons, the function `drawArrays` has been used. In fact, as it represents the vertices following the order in which they appear in the vertices buffer, `drawArrays` guaranties better performance with respect to the alternative WebGL function `drawElements`, which represents the vertices by following the order of a supplementary array of indexes pointing to the vertices.

3.2.1.3 Swii2 preliminary analysis

So far, Swii2 was preliminary tested on a Local Area Network by only considering standard laptops, one acting as a Web server and a maximum of 3 as remote clients accessing simultaneously the former. The level of usability of the GUI resulted more than satisfactory, mainly thanks to the asynchronous communications between client and server. In fact, the activation response of any element of the user interface was practically immediate and comparable with that of desktop applications. Moreover, also data exchange between clients and server, in particular simulation data did not cause a significant slow down. Therefore, the 3D visualization system resulted to be surprisingly efficient, especially if compared with that of the first release of Swii. Moreover, thanks to the low computational requirements of the SCIDDICA-k1 simulation model, also the computational efficiency resulted more than acceptable on the considered server, making Swii2 comparable to standard desktop applications in terms of both efficiency and usability.

3.2.1.4 Cooperative Aspects in Scientific Simulation

As mentioned in Section 3.2.1.1, SCIDDICA-k1 depends on a set of parameters which rule the system evolution. Different sets of parameters are generally needed in order to simulate different types of debris flows. Therefore, if a new debris flow must be simulated, which is different for instance, in rheological terms, with respect to all the other simulated before, SCIDDICA could require a calibration phase, in order to determine a new proper set of parameters to be employed. This phase can be accomplished manually or by means of an automated optimization technique (e.g. by Genetic Algorithms

[2]gwt-g3d homepage: http://code.google.com/p/gwt-g3d/

- see e.g. [17]). However, in both cases, calibration generally requires a large number of trials and thus great computational resources and time. While this could not be an issue in case of a desktop application, it may represent a serious limitation for a client-server system, as the server could not be able to satisfy multiple calibration requests. In addition, the problem could become as greater as the user's community increases. On the other hand, an adequate solution, which does not require high computational resources, is advisable. Without such a solution, the level of the web application's usability could result strongly penalized.

A possible solution could be inspired from the cooperative philosophy of Web 2.0. For instance, depending on some policies and on the computational power of the server(s), calibration experiments could be permitted to a restricted number of users, who are invited to share their results with the community by providing information about the performed experiment, mainly regarding debris flows technical description and calibrated parameters. Such information could then be stored in a intelligent database system for future usage.

After a transition phase, the database should reach a critical size and contain a significant number of SCIDDICA sets of parameters, linked to both the simulated phenomena and the obtained results. In this way, if a new event must be simulated without having a precise idea concerning parameters to be used, the user could query the intelligent database to get a useful starting point. For example, the user could execute a query by using some knowledge on the phenomenon to be simulated as search criterion, and obtain a list of sets of parameters employed in similar cases. A measure of the correspondence between the new case to be simulated and the events in the returned records should be provided, together with information about obtained fitness values.

Such kind of cooperative approach could represent an interesting innovation in the global panorama of scientific applications that, once consolidated, could significantly reduce the employment of optimization techniques. In fact, if the user is facilitated in finding immediately a good set of parameters for the simulation, which can be subsequently further refined, the calibration phase could even become unnecessary. Moreover, this feature should become the more rich and reliable, as its employment increases.

The ideas here just outlined, which can be also applied in the web application contexts presented in the following, will be better formalized and considered in the next release of Swii, as we conjecture that they could represent the basis for a different and innovative way to exploit scientific simulation models.

3.2.2 SciaraWii: the SCIARA-fv3 Web User Interface

Like Swii2, SciaraWii (SCIARA Web Interactive Interactive Interface) is a Web 2.0 application for the SCIARA-fv3 lava flow CA simulation model. SciaraWii allows the user to controls and visualize a SCIARA-fv3 simulation running server-side. The graphical user interface is based on HTML5 and JavaScript, which permits to have a fully portable application. The client is able to control the basic SCIARA-fv3 functionalities thanks to asynchronous callbacks to the server.

SciaraWii was implemented by means GWT, where the interaction between the user interface and the SCIARA-fv3 computational model is performed by a set of client-side services which are implemented on the server (see e.g. [31]). Multi-client connections are also possible: whenever a user logs in, an asynchronous request is sent to the server in order to establish a connection. Here, a servlet binds the client to an individual connection-handler, which allows multiple unambiguous communications through HTTP requests and responses.

3.2.2.1 System architecture

The SciaraWii system architecture is the same as Swii2. The computational model, SCIARA-fv3, is implemented on the server in C++ (for efficiency reasons) as a static library. A dynamic-link library (DLL) receives requests by Java Native Interface (JNI) methods and provides simulation data to the application server. Data is therefore sent to the client via HTTP and stored into the Web browser cache memory. SCIARA-fv3 parameters are displayed in GUI controls (in which they can also be modified), while simulation data such as the topographic surface or the simulated lava flow, are visualized by means of the 3D WebGL rendering engine, which runs on a HTML5 `<Canvas>`. Whenever the simulation produces a lava flow, it is displayed over the surface and its dynamical behavior can be observed. All the client-server communications are managed by means of asynchronous JavaScript calls, which are able to provide the same usability level of desktop applications to SciaraWii. Figure 3.11 shows a screenshot of SciaraWii.

3.2.2.2 Visualization system, Rendering And Decimation

Like Swii2 the rendering engine has been developed by means of the gwt-g3d library, which makes easy the integration in GWT. Data meshing has been based on triangle strips and the function `drawArrays` has been used.

One of the main new features of SciaraWii over Swii2 is the use of *decimation* [58] of triangle meshes. The goal of the decimation algorithm is

to reduce the total number of triangles in a triangle mesh, preserving the original topology and a good approximation to the original geometry.

The decimation algorithm is simple. Multiple passes are made over all vertexes in the mesh. During a pass, each vertex is a candidate for removal and, if it meets the specified decimation criteria, the vertex and all triangles that use the vertex are deleted. The resulting hole in the mesh is patched by forming a local triangulation. The vertex removal process repeats, with possible adjustment of the decimation criteria, until some termination condition is met. Usually the termination criterion is specified as a percent reduction of the original mesh (or equivalent), or as some maximum decimation value. The three steps of the algorithm are:

- characterize the local vertex geometry and topology,
- evaluate the decimation criteria, and
- triangulate the resulting hole.

In SciaraWii the decimation criteria is straightforward. Since maps are generally rectangular and the vertexes are ordered into a matrix, the candidates for removal are individuated by their index. Those with odd index will be updated, their value in the z-axis changed to the average of the values of their neighborhood and those with even index will be removed.

The use of technique has greatly improved the performance of the visualization with large maps.

3.2.2.3 Performance analysis

In order to stress the system's reliability, SciaraWii was tested on a Local Area Network by only considering laptops, with one acting as a Web server and a maximum of 4 as remote clients accessing simultaneously the former. The level of usability of the GUI resulted more than satisfactory, mainly thanks to the asynchronous communications between client and server. Also the 3D visualization system resulted to be surprisingly efficient, especially if compared with that of the first release of SciaraWii, by making SciaraWii comparable to standard desktop applications in terms of both efficiency and usability.

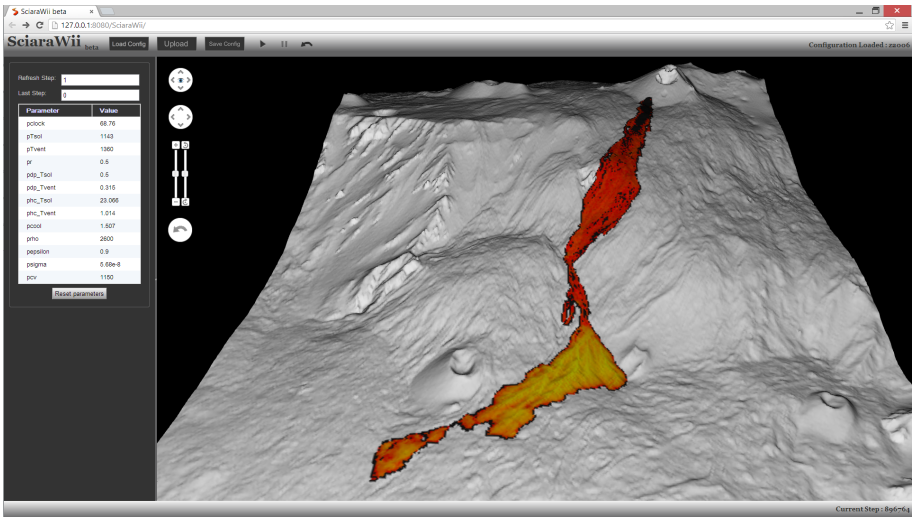


Figure 3.11: A screenshot of the Web user interface for SCIARA-fv3 showing simulation of the 2006 Valle del Bole Etnean lava flow. On the upper part of the application, a horizontal panel shows the name/logo and contains the controls which permit to interact with the simulation. A notification area is also present on the right side of the panel. The remaining client area is subdivided in two panels. The left one contains the controls which permits to show the current simulation step, set the graphic update interval and show/edit SCIARA-fv3 parameters. The right one contains the graphic output of the simulation.

3.2.3 Awii

Like the previous examples, Awii (ABBAMPAU Web Interactive Interface) is a Web 2.0 application used to control and interactively visualize wildfire simulations.

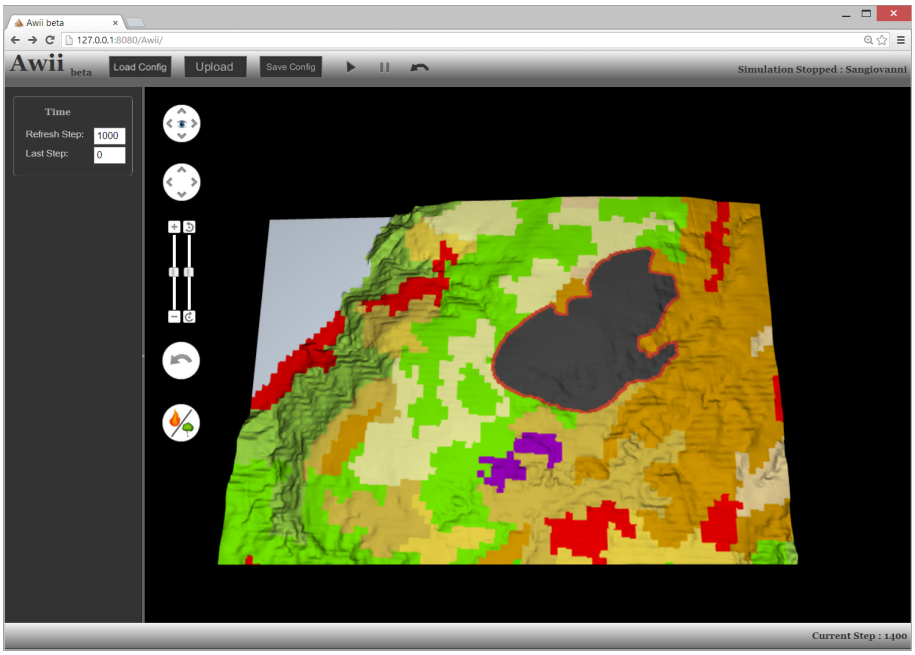


Figure 3.12: 3D simulation of a fire near San Giovanni in Fiore (Italy). On the upper part of the application, a horizontal panel shows the name/logo and contains the controls which permit to interact with the simulation. A notification area is also present on the right side of the panel. The remaining client area is subdivided in two panels. The left one contains the controls which permits to show the current simulation step, set the graphic update interval and set ABBAMPAU simulation computational steps. The right one contains the graphic output of the simulation.

3.2.3.1 System architecture

The graphical user interface is similar to that previously described 3.2.1.2, based on HTML5 and JavaScript (implemented by means GWT), which permits to have a fully portable application. Multi-client connections are still possible: whenever a user logs in, an asynchronous request is sent to the server in order to establish a connection. Here, a servlet binds the client to an individual connection-handler, which allows multiple unambiguous communications through HTTP requests and responses.

The main difference of this application is the simulation model that is not natively integrated, but is present in the form of executable file. This feature increases the independence of the individual simulations.

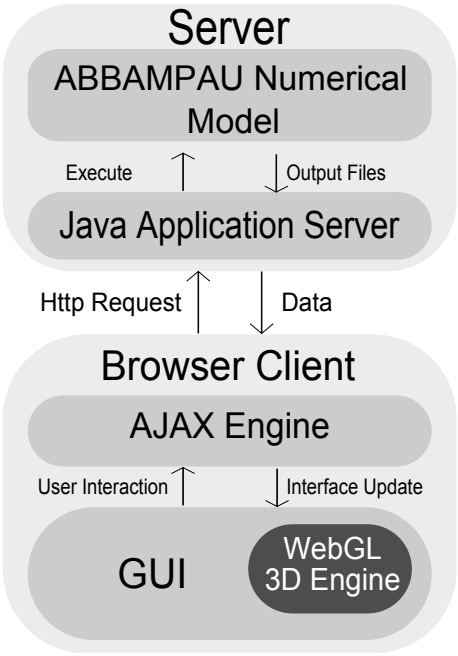


Figure 3.13: The Awii system architecture.

Command Wrapper

As mentioned in subsection 3.2.3.1, the simulation model is contained in an executable file inside the application. This file takes as input the dataset in GIS format and outputs a file of type `dataset_STATE.asc` where the status of the fire is kept. A wrapper in Java has been developed that takes care to integrate this file and all others containing the parameters of the model. The executable is called via the Java method `Runtime.getRuntime().Exec(command);` where there are also specified parameters that change depending on the state in which is located the simulation.

3D Visualization

The application allows 2 ways of viewing the 3D model, related to the coloring of the cells:

1. Representation of soil with colors related to the standard codes of the CORINE program;

2. Representation of soil and fire with orange for the flames and gray 75% for the burnt soil

3.2.3.2 Performance analysis

In order to stress the system's reliabilty, Awii was tested on a Local Area Network by only considering laptops, with one acting as a Web server and a maximum of 10 as remote clients accessing simultaneously the former. Compared to SciaraWii the simultaneous run is more performing because the execution of the simulation is run in a separate process each time, so there is a greater distribution of work for the management of various clients. The level of usability of the GUI resulted is however more than satisfactory.

4
OpenCAL

As mentioned in Chapter 2, Macroscopic Cellular Automata (MCA) represent a parallel computing methodology based on the Cellular Automata paradigm for modelling complex systems at a macroscopic level of description. Well known examples of applications include the simulation of natural phenomena such as lava and debris flows, forest fires, agent based social processes such as pedestrian evacuation and highway traffic problems, besides many others.

Many Cellular Automata software environments and libraries exist. However, when non-trivial modelling is needed, only not open source software are generally available. This is particularly true for Macroscopic Cellular Automata, for which only a significant example of non free software exists, namely the CAMELot Cellular Automata Simulation Environment [61].

In order to fill this deficiency in the world of free software, the `OpenCAL` (Open Cellular Automata Library) software C library has been developed. Currently, only the 2D part of the library has been completed, while the 3D is under development and will not be discussed here. Accordingly, only 2D CA models can be implemented in OpenCAL, among which we can find those considered in this Thesis for the simulation of macroscopic complex natural phenomena.

Similarly to CAMELot, OpenCAL allows for a simple and concise definition of both the transition function and the other characteristics of the cellular automaton definition. Moreover, it allows for both CPU sequential and GPU parallel execution, thanks to the adoption of the Open Computing

Language (OpenCL), one of the most important frameworks for writing programs that can be executed across heterogeneous platforms. It was designed with the aim of greatly simplifying the parallelization of Celluar Automata applications and, for this reason, is capable of hiding the complexity behind the GPU programming. The programmer has only to write the elementary processes that compose the transition function, register substates and set up some parameters (for example, the number of steps). Memory management is completely behind the scene, letting the programmer to concentrate to the modeling processes.

In the following sections, both the sequential and parallel version of the OpenCAL CA library are described by considering some examples of application and by commenting the source code. Moreover, speed-up measurements of the OpenCL parallel implementation are reported, based on the evaluation of the computational times recorded by considering a test debris flow simulation model, namely the SCIDDICA-T CA.

4.1 A brief description of OpenCAL

In order to describe the OpenCAL CA software library, let's start with a simple example of application: Conway's Game of Life. It is one of the most simple yet powerful examples of Cellular Automata, devised by the mathematician John Horton Conway in 1970. Subsequently, a more complex example is presented, concerning a simplified CA numerical model for simulating debris flows, called SCIDDICA-T, which is a former version of SCIDDICA-k1, used for the development of the Swii2 Web 2.0 application.

4.1.1 An OpenCAL implementation of Conway's Game of Life

The Game of Life can be thought as an infinite two-dimensional orthogonal grid of square cells (the cellular space), each of which is in one of two possible states, dead or alive. Every cell interacts with its eight neighbors, which are the cells that are directly horizontally, vertically, or diagonally adjacent to it (the Moore neighborhood). At each time step, one of the following transitions occurs:

1. Any live cell with fewer than two alive neighbors dies, as if by loneliness.
2. Any live cell with more than three alive neighbors dies, as if by overcrowding.

3. Any live cell with two or three alive neighbors lives, unchanged, to the next generation.

4. Any dead cell with exactly three live neighbors comes to life.

The initial configuration of the system specifies the state (dead or alive) of each cell into the cellular space. The evolution of the system is thus obtained by applying the above rules (the CA transition function) simultaneously to every cell in the cellular space, so that each new configuration is a pure function of the previous. The rules continue to be applied repeatedly to create further generations.

The program below shows a simple Game of Life sequential implementation in OpenCAL.

Listing 4.1: Example of OpenCAL sequential implementation of the Conway's game of Life.

```
//--------------------------------------------------------------------------
//   THE LIFE CELLULAR AUTOMATON
//--------------------------------------------------------------------------

#include <cal2D.h>
#include <cal2DIO.h>
#include <cal2DRun.h>
#include <stdlib.h>

//Substate declaration
struct CALSubstate2Di *Q;

void life_transition_function(struct CALModel2D* life, int i, int j)
{
       int sum = 0, n;
       for (n=1; n<life->sizeof_X; n++)
              sum += calGetX2Di(life, Q, i, j, n);

       if ((sum == 3) || (sum == 2 && calGet2Di(life, Q, i, j) == 1))
              calSet2Di(life, Q, i, j, 1);
       else
              calSet2Di(life, Q, i, j, 0);
}

void life_init(struct CALModel2D* life)
{
    //set the whole substate to 0
    calInitSubstate2Di(life, Q, 0);

    //set a glider
        calInit2Di(life, Q, 0, 2, 1);
        calInit2Di(life, Q, 1, 0, 1);
        calInit2Di(life, Q, 1, 2, 1);
        calInit2Di(life, Q, 2, 1, 1);
        calInit2Di(life, Q, 2, 2, 1);
}

```

```
38 void life_finalize(struct CALModel2D* life)
39 {
40         //add cells to the set of active ones
41         calRemoveActiveCell2D(life, 0, 0);
42         calRemoveActiveCell2D(life, 0, 1);
43 
44         //this is needed only if one or more cells are added or eliminated from the
               computationally active cells
45         calUpdateActiveCells2D(life);
46 }
47 
48 //------------------------------------------------------------------------
49 
50 int main()
51 {
52         //cadef and rundef
53         struct CALModel2D* life = calCADef2D (100, 100, CAL_CUSTOM_NEIGHBORHOOD_2D,
               CAL_SPACE_TOROIDAL, CAL_NO_OPT);
54         struct CALRun2D* life_simulation = calRunDef2D(life, 1, 1,
               CAL_UPDATE_IMPLICIT);
55 
56         //add transition function's elementary processes.
57         calAddElementaryProcess2D(life, life_transition_function);
58 
59         //add neighbors of the Moore neighborhood
60         calAddNeighbor2D(life, 0,  0);      //this is the neighbor 0 (central cell)
61         calAddNeighbor2D(life, - 1, 0);     //this is the neighbor 1
62         calAddNeighbor2D(life, 0, - 1);     //this is the neighbor 2
63         calAddNeighbor2D(life, 0, + 1);     //this is the neighbor 3
64         calAddNeighbor2D(life, + 1, 0);     //this is the neighbor 4
65         calAddNeighbor2D(life, - 1, - 1);   //this is the neighbor 5
66         calAddNeighbor2D(life, + 1, - 1);   //this is the neighbor 6
67         calAddNeighbor2D(life, + 1, + 1);   //this is the neighbor 7
68         calAddNeighbor2D(life, - 1, + 1);   //this is the neighbor 8
69 
70         //add substates
71         Q = calAddSubstate2Di(life);
72 
73         //saving configuration
74         calSaveSubstate2Di(life, Q, "./data/life_0000.txt");
75 
76         //simulation run
77         calRunAddInitFunc2D(life_simulation, life_init);
78         calRun2D(life_simulation);
79         calRunFinalize2D(life_simulation);
80 
81 
82         //saving configuration
83         calSaveSubstate2Di(life, Q, "./data/life_LAST.txt");
84 
85         //finalization
86         calFinalize2D(life);
87 
88         return 0;
89 }
90 
91 //------------------------------------------------------------------------
```

All programs that exploit the sequential version of OpenCAL must include the header files `cal2D.h` and `cal2DRun.h`. The first one allows to

define the CA model, while the second to execute a simulation. The header `cal2DIO.h` can be also included in case I/O operation are needed, which is the case of the above example program.

In OpenCAL, a *substate* is a data structure containing two buffers: the *current buffer* and the *next buffer*. Both buffers are represented by linear arrays, even if they correspond to a 2D cellular space. The current buffer is used for reading cell's state values, while the next one for writing the new ones. In order to define a substate, which in the current version of OpenCAL can be of type `CALbyte` (corresponding to the `char` type in C), `CALint` (corresponding to the `int` type in C), or `CALreal` (corresponding to the `double` type in C), it is necessary to declare a pointer to `CALSubstate2D[b,i,r]`, where the letters in brackets refer to the above cited basic types, respectively. For instance, the statement `struct CALSubstate2Di *Q` at line 12 declares a `CALint` substate, i.e. a substate in which each cell can contain values of type `CALint`.

In order to create a CA, an object of type `CALModel2D*` must be defined (line 55), being `CALModel2D` a C structure containing the following data:

- the CA dimension (number of rows and columns of the 2D cellular space);
- the space boundary condition (if toroidal or not);
- the type of optimization considered (active cells or no optimization);
- the cell's neighboring relation (von Neumann, Moore or custom)
- the three arrays of pointers to 2D substates of type `CALbyte`, `CALint`, and `CALreal`;
- the array of function pointers to the transition function's elementary processes.

The function `calCADef2D` (line 55) is used to initialize some CA object data: CA dimension, neighborhood, toroidality of the cellular space, and the optimization used (if any). In order to complete the CA definition, further functions are used. In particular, the arrays of pointers to the CA substates, which are handles to the defined substates (so that they can be automatically updated by OpenCAL after each elementary process has been computed), are updated by means of the function `calAddSubstate2D[b,i,r]`. In fact, the function *registers* the substate provided as argument to the corresponding array. The letters in brackets still refer to the OpenCAL basic types `CALbyte`, `CALint`, and `CALreal`, respectively.

Similarly, CA elementary processes, constituting the CA transition function, are registered by the function `calAddElementaryProcess2D` (line 59). In case more than one elementary process is registered, the order of execution corresponds to the order of registration. The statement at line 59 defines the only callback function `life_transition_function` of the CA, which therefore corresponds to the whole CA transition function. Each elementary process must return `void` and have the same parameters of the function `life_transition_function` (line 15): a pointer to the CA object and the coordinates of a generic cell of the cellular space. OpenCAL will therefore apply the elementary process to each cell of the cellular space transparently to the user.

The elementary process callback function's body contains further calls to OpenCAL functions. In particular, the function `calGetX2Di` at line 19 returns the value of the substate `Q` of the `n`-th neighbor of the cell `(i, j)` of the CA life. Similarly, the function `calGet2Di` returns the value of the substate `Q` of the cell `(i, j)` of the CA. Eventually, `calSet2Di` updates the value of the substate `Q` of the cell `(i, j)` of the CA.

In the considered example, the CA neighboring relation is set to the constant `CAL_CUSTOM_NEIGHBORHOOD_2D` (line 55). In this case, the neighborhood must be explicitly defining by calling the function `calAddNeighbor2D`, as many times as the number of neighbors are (lines 62-70). The function requires the relative coordinates of the neighbor with respect the central cell, which has relative coordinates (0,0). OpenCAL also offers predefined neighborhoods, which can be set by the specifying the following constants in the CA definition function: `CAL_VON_NEUMANN_NEIGHBORHOOD_2D` and `CAL_MOORE_NEIGHBORHOOD_2D`, for the von Neuman and Moore neighborhood, respectively.

At line 76 the function `calSaveSubstate2Di` saves the initial state of the CA, which correspond to the configuration of the only defined substate `Q`, on file located at the path `./data/life_0000.txt`. Such initial configuration has been defined by means of the `life_init` callback function. It belongs to the simulation object and is registered to this object by the function `calRunAddInitFunc2D` (line 79).

At line 80 the function `calRun2D` executes the simulation. The number of computational steps is defined inside the simulation object at the moment of its initialization. In the present example, the simulation will start with the step 1 and will terminate at the step 10 (line 56). The function makes completely transparent the main loop of the simulation, as well as all the updating operation involving the substates's structures.

The call to the function `calRunFinalize2D` (line 81) releases all the memory implicitly allocated by the `life_simulation` object, while the call to the

function `calSaveSubstate2Di` saves the final CA configuration on a file located at the path `./data/life_LAST.txt` (line 85). Eventually, the call to the function `calFinalize2D` releases all the memory implicitly allocated by the `life` CA object.

4.1.2 An OpenCAL implementation of the SCIDDICA-T debris flows model

As already mentioned, SCIDDICA-T is a former, simplified version of the CA SCIDDICA-k1. Therefore, it is an example of Macroscopic Complex Cellular Automata since it requires more than one substate and the cell can assume a great number of different values. Moreover, it depends of some global parameters, which influence the dynamics of the system.

The implementation here presented has a minimal user interface developed in GLUT and a simple visualization system written in OpenGL. Differently to the previous example, the program is organized in three source files, whose content is illustrated below in the listings 4.2, 4.3, and 4.4.

Listing 4.2: An OpenCAL sequential implementation of the SCIDDICA-T CA debris flow model: The source file sciddicaT.h.

```
1  #ifndef sciddicaT_h
2  #define sciddicaT_h
3
4  #include <cal2D.h>
5  #include <cal2DIO.h>
6  #include <cal2DRun.h>
7
8
9  #define ROWS 610
10 #define COLS 496
11 #define P_R 0.5
12 #define P_EPSILON 0.001
13 #define STEPS 4000
14 #define DEM_PATH "./data/dem.txt"
15 #define SOURCE_PATH "./data/source.txt"
16 #define OUTPUT_PATH "./data/width_final.txt"
17
18 //cadef and rundef
19 extern struct CALModel2D* sciddicaT;
20 extern struct CALRun2D* sciddicaTsimulation;
21
22 #define NUMBER_OF_OUTFLOWS 4
23
24 struct sciddicaTSubstates {
25        struct CALSubstate2Dr *z;
26        struct CALSubstate2Dr *h;
27        struct CALSubstate2Dr *f[NUMBER_OF_OUTFLOWS];
28 };
29
```

```
30 struct sciddicaTParameters {
31        CALParameterr epsilon;
32        CALParameterr r;
33 };
34
35 extern struct sciddicaTSubstates Q;
36 extern struct sciddicaTParameters P;
37
38 void sciddicaTCADef();
39 void sciddicaTLoadConfig();
40 void sciddicaTSaveConfig();
41 void sciddicaTExit();
42
43 #endif
```

The header file in listing 4.2 is very simple and contains some definitions and function prototypes, besides two new data structure for aggregating substates and parameters, respectively. By inspecting such data structures, it is simple to observe that SCIDDICA-T substates are `Q.z`, representing the topographic altitude of the cell, `Q.h`, representing the debris thickness, and `Q.f[i] (i=1,...,4)`, representing the 4 debris outflows from the central cell to the neighbors (lines 21-27). Similarly, its possible to observe that model's parameter are P.epsilon and P.r (lines 29-32).

Listing 4.3: An OpenCAL sequential implementation of the SCIDDICA-T CA debris flow model: The source file sciddicaT.c.

```
1  #include "sciddicaT.h"
2  #include <stdlib.h>
3
4  //------------------------------------------------------------------------
5  //        The sciddicaT cellular automaton definition section
6  //------------------------------------------------------------------------
7
8  //global objects declaration
9  struct CALModel2D* sciddicaT; //the cellular automaton
10 struct sciddicaTSubstates Q; //the substates
11 struct sciddicaTParameters P; //the parameters
12 struct CALRun2D* sciddicaTsimulation; //the simulation run
13
14
15 //------------------------------------------------------------------------------
16 //                                  sciddicaT transition function
17 //------------------------------------------------------------------------------
18
19 //first elementary process
20 void sciddicaT_flows_computation(struct CALModel2D* sciddicaT, int i, int j)
21 {
22        CALbyte eliminated_cells[5]={CAL_FALSE,CAL_FALSE,CAL_FALSE,CAL_FALSE,
               CAL_FALSE};
23        CALbyte again;
24        CALint cells_count;
25        CALreal average;
26        CALreal m;
```

```
27          CALreal u[5];
28          CALint n;
29          CALreal z, h;
30
31
32          if (calGet2Dr(sciddicaT, Q.h, i, j) <= P.epsilon)
33                  return;
34
35          m = calGet2Dr(sciddicaT, Q.h, i, j) - P.epsilon;
36          u[0] = calGet2Dr(sciddicaT, Q.z, i, j) + P.epsilon;
37          for (n=1; n<sciddicaT->sizeof_X; n++)
38          {
39                  z = calGetX2Dr(sciddicaT, Q.z, i, j, n);
40                  h = calGetX2Dr(sciddicaT, Q.h, i, j, n);
41                  u[n] = z + h;
42          }
43
44          //computes outflows
45          do{
46                  again = CAL_FALSE;
47                  average = m;
48                  cells_count = 0;
49
50                  for (n=0; n<sciddicaT->sizeof_X; n++)
51                          if (!eliminated_cells[n]){
52                                  average += u[n];
53                                  cells_count++;
54                          }
55
56                          if (cells_count != 0)
57                                  average /= cells_count;
58
59                          for (n=0; n<sciddicaT->sizeof_X; n++)
60                                  if( (average<=u[n]) && (!eliminated_cells[n]) ){
61                                          eliminated_cells[n]=CAL_TRUE;
62                                          again=CAL_TRUE;
63                                  }
64
65          }while (again);
66
67          for (n=1; n<sciddicaT->sizeof_X; n++)
68                  if (eliminated_cells[n])
69                          calSet2Dr(sciddicaT, Q.f[n-1], i, j, 0.0);
70                  else
71                          calSet2Dr(sciddicaT, Q.f[n-1], i, j, (average-u[n])*P.r);
72  }
73
74  //second (and last) elementary process
75  void sciddicaT_width_update(struct CALModel2D* sciddicaT, int i, int j)
76  {
77          CALreal h_next;
78          CALint n;
79
80          h_next = calGet2Dr(sciddicaT, Q.h, i, j);
81          for(n=1; n<sciddicaT->sizeof_X; n++)
82                  h_next += calGetX2Dr(sciddicaT, Q.f[NUMBER_OF_OUTFLOWS - n], i, j, n)
                        - calGet2Dr(sciddicaT, Q.f[n-1], i, j);
83
84          calSet2Dr(sciddicaT, Q.h, i, j, h_next);
85  }
86
87  //------------------------------------------------------------------------
```

```
88  //                                  sciddicaT simulation functions
89  //------------------------------------------------------------------------------
90
91  void sciddicaTSimulationInit(struct CALModel2D* sciddicaT)
92  {
93          CALreal z, h;
94          CALint i, j;
95
96          //initializing substates to 0
97          calInitSubstate2Dr(sciddicaT, Q.f[0], 0);
98          calInitSubstate2Dr(sciddicaT, Q.f[1], 0);
99          calInitSubstate2Dr(sciddicaT, Q.f[2], 0);
100         calInitSubstate2Dr(sciddicaT, Q.f[3], 0);
101
102         //sciddicaT parameters setting
103         P.r = P_R;
104         P.epsilon = P_EPSILON;
105
106         //sciddicaT source initialization
107         for (i=0; i<sciddicaT->rows; i++)
108                 for (j=0; j<sciddicaT->columns; j++)
109                 {
110                         h = calGet2Dr(sciddicaT, Q.h, i, j);
111
112                         if ( h > 0.0 ) {
113                                 z = calGet2Dr(sciddicaT, Q.z, i, j);
114                                 calSet2Dr(sciddicaT, Q.z, i, j, z-h);
115                         }
116                 }
117 }
118
119 void sciddicaTSteering(struct CALModel2D* sciddicaT)
120 {
121     //initializing substates to 0
122         calInitSubstate2Dr(sciddicaT, Q.f[0], 0);
123         calInitSubstate2Dr(sciddicaT, Q.f[1], 0);
124         calInitSubstate2Dr(sciddicaT, Q.f[2], 0);
125         calInitSubstate2Dr(sciddicaT, Q.f[3], 0);
126 }
127
128 CALbyte sciddicaTSimulationStopCondition(struct CALModel2D* sciddicaT)
129 {
130         if (sciddicaTsimulation->step >= STEPS)
131                 return CAL_TRUE;
132         return CAL_FALSE;
133 }
134
135
136 //------------------------------------------------------------------------------
137 //                                  sciddicaT CADef and runDef
138 //------------------------------------------------------------------------------
139
140 void sciddicaTCADef()
141 {
142         //cadef and rundef
143         sciddicaT = calCADef2D (ROWS, COLS, CAL_VON_NEUMANN_NEIGHBORHOOD_2D,
                CAL_SPACE_TOROIDAL, CAL_NO_OPT);
144         sciddicaTsimulation = calRunDef2D(sciddicaT, 1, CAL_RUN_LOOP,
                CAL_UPDATE_IMPLICIT);
145
146         //add transition function's elementary processes
147         calAddElementaryProcess2D(sciddicaT, sciddicaT_flows_computation);
```

```
148         calAddElementaryProcess2D(sciddicaT, sciddicaT_width_update);
149
150         //add substates
151         Q.z = calAddSubstate2Dr(sciddicaT);
152         Q.h = calAddSubstate2Dr(sciddicaT);
153         Q.f[0] = calAddSubstate2Dr(sciddicaT);
154         Q.f[1] = calAddSubstate2Dr(sciddicaT);
155         Q.f[2] = calAddSubstate2Dr(sciddicaT);
156         Q.f[3] = calAddSubstate2Dr(sciddicaT);
157
158         //simulation run setup
159         calRunAddInitFunc2D(sciddicaTsimulation, sciddicaTSimulationInit);
160         calRunAddSteeringFunc2D(sciddicaTsimulation, sciddicaTSteering);
161         calRunAddStopConditionFunc2D(sciddicaTsimulation,
                sciddicaTSimulationStopCondition);
162 }
163
164 //------------------------------------------------------------------------------
165 //                                  sciddicaT I/O functions
166 //------------------------------------------------------------------------------
167
168 //Omissis...
```

Listings 4.3 contains the SCIDDICA-T definition. At the top of the file (lines 9-12), the CA object `sciddicaT`, the set of substates `Q`, the set of parameters `P`, and the CA simulation object `sciddicaTsimulation` are declared. The function `sciddicaTCADef` is used to initialize the above objects and to register substates and elementary processes to the CA object `sciddicaT` (lines 140-162).

Specifically, as regards the initialization of `sciddicaT`, the CA dimensions are provided by some constants defined in the header file `sciddicaT.h`, while the von Neumann neighborhood, a toroidal cellular space and no optimizations are adopted. The simulation object `sciddicaTsimulation` is also initialized with the constant `CAL_RUN_LOOP` as the last computational step specification. In such a case, the simulation does not terminate, unless a given termination criterion is met. This is precisely the case of the considered implementation of SCIDDICA-T, since the OpenCAL function `calRunAddStopConditionFunc2D` (line 161) registers a callback function by means of which a termination criterion is defined. In this case, this termination function was introduced to show how non-trivial termination criteria can be defined in OpenCAL. However, the defined termination criterion is still the same of the previous example of implementation of the Conway's Game of Life (lines 128-133).

Regarding the transition function's elementary processes, they are defined by the callback functions `sciddicaT_flows_computation` (lines 20-72) and `sciddicaT_width_update` (lines 75-85). The first one computes the debris outflows from the central cell towards its neighbors by applying the Minimization Algorithm of the Differences [29], while the second performs the

mass balance by distributing the above computed flows.

Besides the termination function, the initialization and *steering* callbacks are registered (lines 159-160). Specifically, the `sciddicaTSimulationInit` callback is the simulation initialization function (lines 91-117) and is equivalent to that in the Game of Life example. However, note that here an explicit access to each cell of the cellular space (global operation) was performed by simply considering a double `for` loop (lines 107-116). Differently from `sciddicaTSimulationInit`, the `sciddicaTSteering` steering callback is executed automatically and transparently to the user at the end of each computational step and allows to execute global operations. It is used to reset each substates cell to the value zero, by means of the OpenCAL function `calInitSubstate2Dr` (lines 119-126).

Listing 4.4: An OpenCAL sequential implementation of the SCIDDICA-T CA debris flow model: The source file sciddicaTgui.c.

```
1  #include "sciddicaT.h"
2  #include <stdlib.h>
3
4  //Omissis...
5
6  void simulationRun(void)
7  {
8          CALbyte again;
9
10         //simulation main loop
11         sciddicaTsimulation->step++;
12
13     //exectutes the global transition function, the steering function and check for
           the stop condition.
14         again = calRunCAStep2D(sciddicaTsimulation);
15
16         //graphic rendering
17         printf("step: %d; \tactive cells: %d\r", sciddicaTsimulation->step,
               sciddicaTsimulation->ca2D->A.size_current);
18         glutPostRedisplay();
19
20         //check for the stop condition
21         if (!again)
22         {
23                 //breaking the simulation
24                 end_time = time(NULL);
25                 glutIdleFunc(NULL);
26                 printf("\n");
27                 printf("Simulation terminated\n");
28                 printf("Elapsed time: %ds\n", end_time - start_time);
29
30                 //saving configuration
31                 printf("Saving final state to %s\n", OUTPUT_PATH);
32                 sciddicaTSaveConfig();
33         }
34 }
35
```

```
36 int main(int argc, char** argv)
37 {
38         sciddicaTCADef();
39         sciddicaTLoadConfig();
40         sciddicaTComputeExtremes(sciddicaT, Q.z, &z_min, &z_Max);
41
42         glutInit(&argc, argv);
43         glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
44         glutInitWindowSize(640, 480);
45         glutInitWindowPosition(100, 100);
46         glutCreateWindow(argv[0]);
47         glutReshapeFunc(reshape);
48   glutDisplayFunc(display);
49
50     //0missis...
51
52                   glutIdleFunc(simulationRun);
53                   glutMainLoop();
54
55         sciddicaTExit();
56         return 0;
57 }
```

Differently form the Conway's Game of Life example, the simulation loop is managed by a GLUT application, partially shown in listing 4.4. In particular, the single step of the simulation can be found inside the `simulationRun` GLUT idle callback function, which is a function that GLUT calls automatically each time rendering operations are not in execution. Thus, in `simulationRun` the OpenCAL function `calRunCAStep2D` is called, which performs a single simulation step. It returns a CALbyte value that can be `CAL_TRUE` (true) or `CAL_FALSE` (false). In the case the return value is `CAL_FALSE`, that occurs when the stopping criterion is met, the idle callback is set to `NULL`, so that the function is no longer called, and the simulation terminates.

Figure 4.1 shows the graphical output of the simulation of the Tessina (Italy) landslide by means of the SCIDDICA-T debris flow model, as implemented in OpenCAL. Simulation results are in accordance with those obtained by the original implementation of the model, as described in [8].

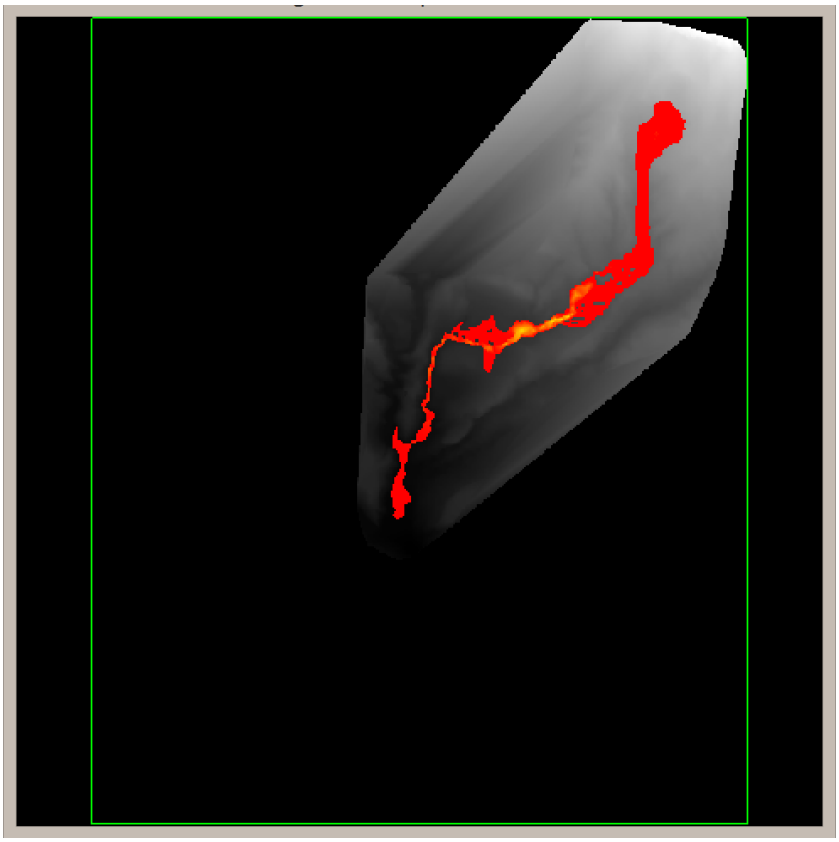


Figure 4.1: Graphical output of the simulation of the Tessina (Italy) landslide by means of the SCIDDICA-T debris flow model, as implemented in OpenCAL.

4.2 A brief description of the OpenCAL parallel OpenCL version

OpenCL programs are composed by two different sections: the host one and the device one. As a consequence, the OpenCL code generally require many lines of code with respect to the equivalent sequential code. In fact, *host-side*, it is necessary to set up the computing devices to be used (for instance the GPU) and, subsequently, write the so called *device-side* kernels in order to exploit the computational power of the configured computing devices. OpenCAL simplifies considerably both of the host and device OpenCL programming in the case of Cellular Automata development. The following sections show the great advantages in using OpenCAL and two examples of

application, namely Conway's Game of Life and SCIDDICA-T.

4.2.1 OpenCAL improvement to OpenCL programming

In order to show the OpenCAL advantages in terms of compactness of source code with respect to the direct programming with OpenCL, the following list shows the "*additional procedures*" needed in most of OpenCL programs:

1. Initialize structures for OpenCL platforms and devices;
2. Find the installed platforms and devices and store those you need;
3. Create a OpenCL context and a program;
4. Load **all** the files where kernels have been stored;
5. Create structures to store kernels and data buffers;
6. Link **all** objects to the own buffer;
7. Set buffers as arguments **for each** kernel;
8. Choose and set the number of workgroups and the dimensions for each of them:
9. Create command queues and launch the kernels;
10. Eventually, handle **any** error generated by the OpenCL functions.

Some of these features are not particularly hard to implement, but others can heavily enlarge the program's dimensions. To prevent this, OpenCAL provides some methods that are able to avoid the need of writing a considerable amount of source code. In the following, some comparative examples are provided in order to evidence the advantage of using OpenCAL for the parallel Cellular Automata implementation instead of OpenCL.

Listing 4.5: Example of OpenCL Platform/Device Initialization

```
1 int main()
2 {
3     cl_platform_id * platform = NULL;
4     cl_platform_id * platforms;
5     cl_device_id **devices, *all_devices;
6     cl_uint err, allPlatSize, allDevicesSize, devicesSize = 0, platVendorLength,
          devNameLength;
7     char* platVendor, devName;
8
```

```
9       //Get all platform
10      err = clGetPlatformIDs(0, NULL, &allPlatSize);
11      platforms = (cl_platform_id*) malloc(sizeof(cl_platform_id) * allPlatSize);
12      err = clGetPlatformIDs(allPlatSize, platforms, NULL);
13
14      //Select platform by vendor
15      for (int i = 0; i < allPlatSize; i++) {
16          err = clGetPlatformInfo(platforms[i], CL_PLATFORM_VENDOR, 0, NULL, &
                platVendorLength);
17          platVendor = (char*) malloc(platVendorLength * sizeof(char));
18          err = clGetPlatformInfo(platforms[i], CL_PLATFORM_VENDOR, sizeof(char) *
                platVendorLength, platVendor, NULL);
19          if (strstr(platVendor, "NVIDIA") != NULL) {
20                  *platform = platforms[i];
21          }
22      }
23      if(platform == NULL){
24          printf("No platform found");
25          return -1;
26      }
27
28      //Get all platform devices
29      err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, 0, NULL, allDevicesSize);
30      *all_devices = (cl_device_id*) malloc(sizeof(cl_device_id) * (*allDevicesSize));
31      err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, *allDevicesSize, *all_devices,
              NULL);
32
33      //Select devices by name
34      for (int i = 0; i < allDevicesSize; i++){
35          err = clGetDeviceInfo(all_devices[i], CL_DEVICE_NAME, 0, NULL, &devNameLength
                );
36          char* devName = (char*) malloc(devNameLength * sizeof(char));
37          err = clGetDeviceInfo(all_devices[i], CL_DEVICE_NAME, sizeof(char) *
                devNameLength, devName, NULL);
38
39          if (strstr(devName, "Tesla") != NULL)
40                  devicesSize++;
41      }
42      int k = 0;
43      *devices = (cl_device_id*) malloc(sizeof(cl_device_id) * devicesSize);
44      for (int i = 0; i < allDevicesSize; i++){
45          err = clGetDeviceInfo(all_devices[i], CL_DEVICE_NAME, 0, NULL, &devNameLength
                );
46          char* devName = (char*) malloc(devNameLength * sizeof(char));
47          err = clGetDeviceInfo(all_devices[i], CL_DEVICE_NAME, sizeof(char) *
                devNameLength, devName, NULL);
48          if (strstr(devName), "Tesla") != NULL) {
49                  *devices[k] = all_devices[i];
50                  k++;
51          }
52      }
53
54      if(err<1){
55          printf("History! Something went wrong");
56          return -1;
57      }
58
59      return 0;
60  }
```

Usually, OpenCL development requires a preliminary phase in which available platforms (vendors, e.g. nVidia, AMD or Intel) and devices are identified. To each of them a specific identifier (id) is automatically assigned by OpenCL. Obviously, platforms and devices can vary from machine to machine and therefore the application must be designed in order to be able to select the proper platforms and computational devices to be used. In listing 4.5 available platforms and devices are analyzed and a *Nvidia Tesla* device selected, if present (otherwise, the program terminates). As it can be seen, the developer is forced to call the same methods multiple times (e.g. `clGetDeviceInfo` at lines 35, 37 and 45) in order to retrieve different platform and device information. As a result, source code can result confused and the developer frustrated. In order to reduce this drawbacks, OpenCAL provides many functions that allows the developer to simplify the platform and devices selection. Listing 4.6 shows exactly the same program of that proposed in Listing 4.5 by using OpenCAL. It is considerably shorter and clearer that of the corresponding version in which OpenCL functions were used directly.

Listing 4.6: Example of OpenCL Platform/Device initialization using OpenCAL

```
1  int main(){
2      CALCLplatform platform;
3      CALCLuint num_devices;
4      CALCLdevice* nvidiaDevices;
5
6      calclGetPlatformByVendor(platform, "NVIDIA");
7      getDevicesByName(platform, nvidiaDevices, num_devices, "Tesla");
8      return 0;
9  }
```

Once one or more computational devices have been chosen, the host must select one or more functions, called kernels, to be placed in a command queue and sent to the devices. Kernel code, unlike serial host code, is executed in parallel and is able to exploit high performance capabilities of the selected computational devices. In order to accomplish such preliminary activities, OpenCL applications must deal with many data structures and operations. For instance, it is necessary to create a program by using the defined kernels, and build it. In this way, developers have to deal with strings and files containing kernels code and then use OpenCL functions to build the program. Moreover, a memory buffer has to be created for each kernel parameter and bound to objects or variables needed for host-device communication.

Listing 4.7: Example of OpenCL kernel load

```
1  //missing code
2  //...
3
4  cl_program program;
5  cl_double argOne, cl_double argTwo;
6  cl_int err;
7  cl_uint num_devices = 1;
8  cl_device_id *devices;
9      .
10     . (get devices)
11     .
12 cl_context context clCreateContext(NULL, num_devices, devices, NULL, NULL, &err)
13
14 //create kernel build command and arguments
15 char* buildArgs = ""; //additional arguments
16 char* build_command;
17 build_command = (char*) malloc(sizeof(char) * (strlen(KERNEL_INCLUDE_DIR) + strlen("
       ␣-I␣") + strlen(buildArgs) + 1));
18 strcpy(build_command, "␣-I␣");
19 strcat(build_command, KERNEL_DIR);
20 strcat(build_command, buildArgs);
21
22 //load file names
23 int num_files = 0;
24 char** filesNames;
25 DIR *dir = opendir(KERNEL_DIR);
26 struct dirent *ent;
27
28 while ((ent = readdir(dir)) != NULL)
29        if (ent->d_name[0] != '.')
30                num_files++;
31 closedir(dir);
32
33 files_names = (char**) malloc(sizeof(char*) * (*num_files));
34 int count = 0;
35
36 dir = opendir(KERNEL_DIR);
37 while ((ent = readdir(dir)) != NULL) {
38        if (ent->d_name[0] != '.') {
39                files_names[count] = (char*) malloc(1 + sizeof(char) * (strlen(
                       KERNEL_DIR) + strlen(ent->d_name)));
40                strcpy((*files_names)[count],KERNEL_DIR);
41                strcat((*files_names)[count], ent->d_name);
42                count++;
43        }
44 }
45 closedir(dir);
46
47 //load files
48 char ** programBuffers = (char**) malloc(sizeof(char*) * num_files);
49 size_t * program_size = (size_t*) malloc(sizeof(size_t) * num_files);
50 for (int i = 0; i < num_files; i++){
51        FILE * file = fopen(filesNames[i], "r");
52        fseek(file, 0, SEEK_END);
53        long fileSize = ftell(file);
54        rewind(file);
55        programBuffers[i] = (char*) malloc(fileSize * sizeof(char));
56        fread(programBuffers[i], sizeof(char), fileSize, file);
57        fclose(file);
58        program_size[i] = fileSize;
59 }
60
```

```
61
62 //loading & building program
63 program = clCreateProgramWithSource(context, num_files, (const char**)
       programBuffers, program_size, &err);
64 err = clBuildProgram(program, num_devices, devices, build_command, NULL, NULL);
65 if (err < 0)
66         ... //print error log
67
68 for (int i = 0; i < num_files; i++)
69         free(filesNames[i]);
70 free(filesNames);
71 free(build_command);
72
73 cl_command_queue queue = clCreateCommandQueue(context, devices[0], (
       cl_command_queue_properties)NULL, &err);
74
75 cl_kernel kernel_one = clCreateKernel(*program, KERNEL_ONE, &err);
76
77 cl_mem bufferArg1 = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR
       , sizeof(cl_double), &argOne, &err);
78 cl_mem bufferArg2 = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR
       , sizeof(cl_double), &argTwo, &err);
79 clSetKernelArg(kernel_one, 0 , sizeof(cl_mem), &bufferArg1);
80 clSetKernelArg(kernel_one, 1 , sizeof(cl_mem), &bufferArg2);
81
82 //missing code
83 //...
```

Listing 4.7 shows a typical OpenCL source code needed in order to allow kernels to run on the device. Without entering in details, it can be seen that it is necessary to create a `cl_context`, which is used by the OpenCL runtime for managing objects like programs, kernels and command queues. Moreover, it is necessary to load kernels code from files and bind them to a `cl_program`. Therefore, the program is built for the specified device and a command queue created. Furthermore, a `cl_kernel` object is defined and bound to the right kernel by means of its function name, which is defined in the kernel code. In most cases, a kernel need some arguments either for input (write) and output (read); in these situations it is necessary to bind the objects that must be passed to the kernel to `cl_mem` objects (generally buffers). Eventually, a buffer is used as kernel argument by means of `clSetKernelArg`.

The set of such operations can be heavily boring for the developer who, however, can still take advantage of using OpenCAL inspite of OpenCL in order to reduce code complexity in the case of Cellular Automata development.

Listing 4.8: Example of OpenCL kernel load using OpenCAL

```
1 //missing code
2 //...
3
```

```
4  struct CALModel2D* model;
5  CALParameterr argOne, argTwo;
6      .
7      . (init model)
8      .
9  CALCLcontext context = calclcreateContext(&device, 1);
10 CALCLprogram program = calclLoadProgramLib2D(context, device, kernelSrc, kernelInc);
11 CALCLToolkit2D * toolkit = calclCreateToolkit2D(model, context, program, device,
       OPTIMIZATION);
12 CALCLkernel kernel_one = calclGetKernelFromProgram(&program, KERNEL_ONE);
13
14 CALCLmem * buffersKernelOne = (CALCLmem *) malloc(sizeof(CALCLmem) * 2);
15 CALCLmem bufferArg1 = calclCreateBuffer(context, &argOne, sizeof(CALParameterr));
16 CALCLmem bufferArg2 = calclCreateBuffer(context, &argTwo, sizeof(CALParameterr));
17 buffersKernelOne[0] = bufferArg1;
18 buffersKernelOne[1] = bufferArg2;
19 calclSetCALKernelArgs2D(&kernel_one, buffersKernelOne, 2);
20
21 calclAddElementaryProcessKernel2D(toolkit, model, &kernel_one);
22
23 //missing code
24 //...
```

In order to show the impact of the new Cellular Automata library here presented on code length and quality, listing 4.8 shows an OpenCAL-based application that is equivalent to that shown in listing 4.7. As it can be seen, the code is significantly more compact (ten lines of code in spite of more than fifty) and readable. One of the main improvements consists in files loading and kernels building, which are now quite transparent to developer thanks to the the adoption of the `calclLoadProgramLib2D` OpenCAL function. Moreover, in listing 4.8 the OpenCAL function `calclCreateToolkit2D` allows to create all the needed structures and buffers.

4.2.2 A simple OpenCAL parallel example of application: The Game of Life

In this section, an example of OpenCAL parallel implementation of a simple cellular automaton is presented by considering the well known Conway's Game of Life, already discussed in this Chapter.

Listing 4.9: Example of OpenCAL usage

```
1  //missing code
2  //...
3  int main()
4  {
5      / ************ OPENCL INIT ******** /
6      CALOpenCL * calOpenCL = calclCreateCALOpenCL();
7      calclInitializePlatforms(calOpenCL);
8      calclInitializeDevices(calOpenCL);
```

```
9
10      // get the first device of the first platform
11      cl_device_id device = calOpenCL->devices[0][0];
12
13      cl_context context = calclcreateContext(1, &device);
14      cl_program program = calclLoadProgramLib(context, device,KERNEL_SRC, NULL);
15
16      / ************ OPENCAL MODEL CREATION AND INIT ******** /
17      CALModel2D * model = calCADef2D(ROWS, COLS, CAL_MOORE_NEIGHBORHOOD_2D,
            CAL_SPACE_TOROIDAL, CAL_NO_OPT);
18
19      // Global CALSubstate2Di pointer used in host-side function
20      CALSubstate2Di * lifeSubstate = calAddSubstate2Di(model);
21      calInitSubstate2Di(model,lifeSubstate,0);
22      calInit2Di(model, lifeSubstate, 0, 2, 1);
23      calInit2Di(model, lifeSubstate, 1, 0, 1);
24      calInit2Di(model, lifeSubstate, 1, 2, 1);
25      calInit2Di(model, lifeSubstate, 2, 1, 1);
26      calInit2Di(model, lifeSubstate, 2, 2, 1);
27
28      / ************ KERNEL INIT AND RUN ******** /
29      CALCLToolkit2D * toolkit = calclCreateToolkit2D();
30      calclInitBuffers2D(toolkit, Model, context, program, CALCL_NO_OPT);
31      cl_kernel elementaryProcess = calclGetKernelFromProgram(program,
            TRANSITION_FUNCTION_KERNEL);
32      calclAddElementaryProcessKernel2D(elementaryProcess, toolkit);
33      cl_command_queue queue = calclCreateCommandQueue(context, device);
34      calclRun2D(toolkit, model, queue, STEPS);
35
36      / ************ FINALIZATION ******** /
37      calSaveSubstate2Di(model, lifeSubstate, SAVE_PATH);
38      calclFinalizeCALOpencl(calOpenCL);
39      calclFinalizeToolkit(toolkit);
40      calFinalize2D(model);
41
42      return 0;
43  }
```

The host-side OpenCAL parallel implementation of the Game of Life is shown in listing 4.9. The first instructions initialize the `CALOpenCL`'s structure, where all platforms and devices installed in the running machine are stored. Subsequently, the device is chosen and the OpenCL context and program are registered. The `calclLoadProgramLib` OpenCAL function takes in input the kernel's directory path. Then, as in the sequential version, the CA model is defined and a substate added. All the structures necessary for the data-transfer between *host* and *device* are created and initialized by means of the `calclCreateToolkit2D` function. Eventually, all the kernel transfer buffers are prepared by means of the function `calclInitBuffers2D`.

Similarly to the sequential version, illustrated and discussed in section 4.1.1, only one elementary process is needed in order to define the finite automaton transition function. The elementary process is registered by means of the OpenCAL `calclAddElementaryProcessKernel2D` function. In case more than one elementary process is needed, their execution order is defined

by the order of registration. After the registration of the elementary process, the OpenCL command queue is created and the simulation executed for a total of `STEPS` iterations by means of the `calclRun2D` OpenCAL function. Eventually, the host-side source code ends with self-explanatory finalization functions.

Listing 4.10: Definition of the Game Of Life transition function using the OpenCAL library and OpenCL.

```
1   __kernel void transitionFunction(MODEL_DEFINITION2D) {
2
3       initThreads();
4
5       int i = getX();
6       int j = getY();
7
8       int sum = 0, n;
9
10      for (n = 1; n < get_neighborhoods_size(); n++)
11          sum += calGetX2Di(MODEL2D, i, j, n, 0);
12
13      if ((sum == 3) || (sum == 2 && calGet2Di(MODEL2D, i, j, 0) == 1))
14          calSet2Di(MODEL2D, 1, i, j, 0);
15      else
16          calSet2Di(MODEL2D, 0, i, j, 0);
17  }
```

As regards the device-side code of the OpenCAL implementation of the Game of Life, the transition function is specified in a separate OpenCL kernel file, shown in listing 4.10. Note that the `transitionFunction` kernel takes `MODEL_DEFINITION2D` as argument. This is a *name* defined in OpenCAL by means of the `#define` directive of the `C` pre-compiler, corresponding to a comma-separated list of couples *type-variable* which defines the link between the CA model and the kernel. Most of these data are those defined by the `calclCreateToolkit2D` initialization function, shown in listing 4.8. Being this data the same for each cellular automaton, the `MODEL_DEFINITION2D` can be used for considerably simplifying the specification of the kernels' parameters. This code, as previously reported, is executed in parallel by each thread of the device. In this version of OpenCAL, each thread refers to a single cell. Accordingly, the statement `initThreads` verifies that a thread is defined for each cell of the cellular automaton, while the `getX()` and `getY()` macros return the global identifiers of the running thread. Finally, the transition function is specified by using OpenCAL functions, like `calGetX2Di` or `calSet2Di`.

4.2.3 A more complex OpenCAL parallel example of application: SCIDDICA-T

In this section, a parallel OpenCAL implementation of the SCIDDICA-T cellular automaton is presented. Similarly to the case of the sequential version, shown in section 4.1.2, source code is illustrated and commented. Specifically, as regards the host-side, the application is subdivided in three source files, while device-side a single file is used for the kernels implementation.

Listing 4.11: An OpenCAL parallel implementation of the SCIDDICA-T CA debris flow model: sciddicaT.h

```
1  //missing code
2  //...
3
4  #define KERNEL_SRC ROOT_DATA_DIR"/kernel/source/"
5  #define KERNEL_INC ROOT_DATA_DIR"/kernel/include/"
6
7  struct SciddicaTMain
8  {
9      struct CALModel2D* M;
10     struct sciddicaTSubstates Q;
11     struct sciddicaTParameters P;
12 };
13
14 void explicitInit(struct SciddicaTMain *);
15
16 //missing code
17 //...
```

The `sciddicaT.h` header file is quite similar to that of the sequential version of SCIDDICA-T, shown in listing 4.2. The only differences are evidenced in listing 4.11, where the definitions of kernels paths and an explicit initialization function were added. Moreover, the `SciddicaTMain` support structure was defined in order to have references to the CA model, substates and parameters.

Listing 4.12: An OpenCAL parallel implementation of the SCIDDICA-T CA debris flow model: sciddicaT.c parallel additions.

```
1  //missing code
2  //...
3
4  //------------------------------------------------------------------------
5  //                                  sciddicaT explicit CADef function
6  //------------------------------------------------------------------------
7
8  void explicitInit(struct SciddicaTMain* s)
9  {
```

```
10          //cadef and rundef
11          s->M = calCADef2D (ROWS, COLS, CAL_VON_NEUMANN_NEIGHBORHOOD_2D,
                CAL_SPACE_TOROIDAL, CAL_OPT_ACTIVE_CELLS);
12
13          //add substates
14          s->Q.z = calAddSubstate2Dr(s->M);
15          s->Q.h = calAddSubstate2Dr(s->M);
16          s->Q.f[0] = calAddSubstate2Dr(s->M);
17          s->Q.f[1] = calAddSubstate2Dr(s->M);
18          s->Q.f[2] = calAddSubstate2Dr(s->M);
19          s->Q.f[3] = calAddSubstate2Dr(s->M);
20
21
22          /****** Only for sequential *******/
23          calAddElementaryProcess2D(s->M, sciddicaT_flows_computation);
24          calAddElementaryProcess2D(s->M, sciddicaT_width_update);
25          calAddElementaryProcess2D(s->M, sciddicaT_remove_inactive_cells);
26
27
28          //cadef and rundef
29          struct CALRun2D* sciddicaTsimulation2 = calRunDef2D(s->M, 1, CAL_RUN_LOOP,
                CAL_UPDATE_IMPLICIT);
30          //simulation run setup
31          calRunAddInitFunc2D(sciddicaTsimulation2, sciddicaTSimulationInit);
32          calRunAddSteeringFunc2D(sciddicaTsimulation2, sciddicaTSteering);
33          calRunAddStopConditionFunc2D(sciddicaTsimulation2,
                sciddicaTSimulationStopCondition);
34      /**********************************/
35
36
37          calLoadSubstate2Dr(s->M, s->Q.z, DEM_PATH);
38          calLoadSubstate2Dr(s->M, s->Q.h, SOURCE_PATH);
39
40          CALreal z, h;
41          CALint i, j;
42
43          //initializing substates to 0
44          calInitSubstate2Dr(s->M, s->Q.f[0], 0);
45          calInitSubstate2Dr(s->M, s->Q.f[1], 0);
46          calInitSubstate2Dr(s->M, s->Q.f[2], 0);
47          calInitSubstate2Dr(s->M, s->Q.f[3], 0);
48
49          //sciddicaT parameters setting
50          s->P.r = P_R;
51          s->P.epsilon = P_EPSILON;
52
53          //sciddicaT source initialization
54          for (i=0; i<s->M->rows; i++)
55                  for (j=0; j<s->M->columns; j++)
56                  {
57                          h = calGet2Dr(s->M, s->Q.h, i, j);
58
59                          if ( h > 0.0 ) {
60                                  z = calGet2Dr(s->M, s->Q.z, i, j);
61                                  calSet2Dr(s->M, s->Q.z, i, j, z-h);
62  #ifdef ACTIVE_CELLS
63                  //adds the cell (i, j) to the set of active ones
64                  calAddActiveCell2D(s->M, i, j);
65                          calUpdateActiveCells2D(s->M);
66  #endif
67                          }
68                  }
```

```
69
70          calUpdate2D(s->M);
71  }
```

The initialization function is shown in listing 4.12 and plays the role of the `sciddicaTCADef` and `sciddicaTSimulationInit` functions of the sequential version, shown in listing 4.3. In fact, it defines the CA model and registers substates and elementary processes. Moreover it defines simulation object and the related callback functions. Eventually, it manages data input (the CA configuration), sets CA parameters and performs the initialization of the debris flow source. The remaining part of the file is the same of that of the sequential version, and therefore is omitted.

Listing 4.13: An OpenCAL parallel implementation of the SCIDDICA-T CA debris flow model: The source file kernel_user.c

```
1   #include <cal2D.h>
2
3   #define Z 0
4   #define H 1
5   #define NUMBER_OF_OUTFLOWS 4
6
7   __kernel void sciddicaT_flows_computation(MODEL_DEFINITION2D, __global CALParameterr
        * epsilon, __global CALParameterr * r) {
8         initThreads2D();
9         __global CALbyte * activeCellsFlags = get_active_cells_flags();
10        CALint cols_ = get_columns();
11
12        int i = getX();
13        int j = getY();
14
15        if (calGetBufferElement2D(activeCellsFlags, cols_, i, j) == CAL_FALSE)
16              return;
17
18        CALbyte eliminated_cells[5] = { CAL_FALSE, CAL_FALSE, CAL_FALSE, CAL_FALSE,
              CAL_FALSE };
19        CALbyte again;
20        CALint cells_count;
21        CALreal average;
22        CALreal m;
23        CALreal u[5];
24        CALint n;
25        CALreal z, h;
26        CALint sizeOfX_ = get_neighborhoods_size();
27        CALParameterr eps = *epsilon;
28
29        if (calGet2Dr(MODEL2D, i, j, H) <= eps)
30              return;
31        m = calGet2Dr(MODEL2D, i, j, H) - eps;
32        u[0] = calGet2Dr(MODEL2D , i, j, Z) + eps;
33        for (n = 1; n < sizeOfX_; n++) {
34              z = calGetX2Dr(MODEL2D, i, j, n,Z);
35              h = calGetX2Dr(MODEL2D , i, j, n, H);
36              u[n] = z + h;
```

```
37          }
38          do {
39                  again = CAL_FALSE;
40                  average = m;
41                  cells_count = 0;
42                  for (n = 0; n < sizeOfX_; n++)
43                          if (!eliminated_cells[n]) {
44                                  average += u[n];
45                                  cells_count++;
46                          }
47                  if (cells_count != 0)
48                          average /= cells_count;
49                  for (n = 0; n < sizeOfX_; n++)
50                          if ((average <= u[n]) && (!eliminated_cells[n])) {
51                                  eliminated_cells[n] = CAL_TRUE;
52                                  again = CAL_TRUE;
53                          }
54          } while (again);
55          for (n = 1; n < sizeOfX_; n++) {
56                  if (eliminated_cells[n])
57                          calSet2Dr(MODEL2D , 0.0, i, j, (n - 1)+2);
58                  else {
59                          calSet2Dr(MODEL2D , (average - u[n]) * (*r), i, j, (n - 1)+2);
60                          calAddActiveCellX2D(MODEL2D,i,j,n);
61                  }
62          }
63  }
64
65  __kernel void sciddicaT_width_update(MODEL_DEFINITION2D) {
66          initThreads2D();
67          __global CALbyte * activeCellsFlags = get_active_cells_flags();
68          CALint cols_ = get_columns();
69
70          int i = getX();
71          int j = getY();
72
73          if (calGetBufferElement2D(activeCellsFlags, cols_, i, j) == CAL_FALSE)
74                  return;
75
76          CALreal h_next;
77          CALint n;
78          h_next = calGet2Dr(MODEL2D, i, j, H);
79
80          for (n = 1; n < get_neighborhoods_size(); n++) {
81                  h_next += (calGetX2Dr(MODEL2D, i, j, n, (NUMBER_OF_OUTFLOWS - n)+2) -
                        calGet2Dr(MODEL2D, i, j, (n-1) +2));
82          }
83          calSet2Dr(MODEL2D, h_next, i, j, H);
84  }
85
86  __kernel void sciddicaTSteering(MODEL_DEFINITION2D) {
87          initThreads2D();
88          __global CALbyte * activeCellsFlags = get_active_cells_flags();
89          CALint cols_ = get_columns();
90          CALint rows_ = get_rows();
91
92          int i = getX();
93          int j = getY();
94
95          if (calGetBufferElement2D(activeCellsFlags, cols_, i, j) == CAL_FALSE)
96                  return;
97
```

```
98          int dim = cols_ * rows_;
99          int s;
100         for (s = 2; s < get_real_substates_num(); ++s)
101                 calInitSubstate2Dr(MODEL2D, 0, i, j, s);
102
103 }
```

As for the Game of Life (see listing 4.10), the device-side of the SCIDDICA-T application is specified in a separate OpenCL kernel file, shown in listing 4.13. In this file, the two elementary processes and the steering function, already present in the sequential version (listing 4.3), are defined.

As for the case of Game of Life, kernels take `MODEL_DEFINITION2D` as first argument, which allows for a great simplification of the function prototype (see section 4.2.2 for further details). Thanks to the adoption of the OpenCAL functions, kernels' source code is almost identical to that of the sequential version. However, kernels are executed in parallel. In particular, the *one thread per cell* strategy has been adopted. Accordingly, the statement `initThreads` verifies that a thread is defined for each cell of the cellular automaton, while the `getX()` and `getY()` macros return the global identifications of the running thread.

Listing 4.14: An OpenCAL parallel implementation of the SCIDDICA-T CA debris flow model: sciddicaTmain.c parallel additions.

```
1  //missing code
2  //...
3
4  int main(int argc, char** argv) {
5
6          int steps = 4000;
7          char * outputPath = "./result";
8          int platformNum = 0;
9          int deviceNum = 0;
10
11         //---------OPENCL INIT-------/
12
13         CALOpenCL * calOpenCL = calclCreateCALOpenCL();
14         calclInitializePlatforms(calOpenCL);
15         calclInitializeDevices(calOpenCL);
16
17         CALCLdevice device = calclGetDevice(calOpenCL, platformNum, deviceNum);
18         CALCLcontext context = calclcreateContext(&device, 1);
19         CALCLprogram program = calclLoadProgramLib2D(context, device, KERNEL_SRC,
               KERNEL_INC);
20
21         //--Parallel CA DEF & INIT---/
22
23         sciddicaTCADef();
24         sciddicaTLoadConfig();
25         sciddicaTComputeExtremes(sciddicaT, Q.z, &z_min, &z_Max);
26         sciddicaTsimulation->init(sciddicaT);
27         calUpdate2D(sciddicaT);
```

```
28
29          CALCLToolkit2D * sciddicaToolkit = NULL;
30
31
32          sciddicaToolkit = calclCreateToolkit2D(sciddicaT, context, program, device,
                CALCL_NO_OPT);
33
34
35          CALCLkernel kernel_elementary_process_one = calclGetKernelFromProgram(&
                program, KER_SCIDDICA_ELEMENTARY_PROCESS_ONE);
36          CALCLkernel kernel_elementary_process_two = calclGetKernelFromProgram(&
                program, KER_SCIDDICA_ELEMENTARY_PROCESS_TWO);
37          CALCLkernel kernel_steering = calclGetKernelFromProgram(&program,
                KER_SCIDDICA_STEERING);
38
39          CALCLmem * buffersKernelOne = (CALCLmem *) malloc(sizeof(CALCLmem) * 2);
40          CALCLmem bufferEpsilonParameter = calclCreateBuffer(context, &P.epsilon,
                sizeof(CALParameterr));
41          CALCLmem bufferRParameter = calclCreateBuffer(context, &P.r, sizeof(
                CALParameterr));
42          buffersKernelOne[0] = bufferEpsilonParameter;
43          buffersKernelOne[1] = bufferRParameter;
44           calclSetCALKernelArgs2D(&kernel_elementary_process_one, buffersKernelOne,2);
45
46          calclAddElementaryProcessKernel2D(sciddicaToolkit, sciddicaT, &
                kernel_elementary_process_one);
47          calclAddElementaryProcessKernel2D(sciddicaToolkit, sciddicaT, &
                kernel_elementary_process_two);
48
49          calclSetSteeringKernel2D(sciddicaToolkit, sciddicaT, &kernel_steering);
50
51          calclRun2D(sciddicaToolkit, sciddicaT, steps);
52
53          sciddicaTSaveConfig(outputPath);
54          sciddicaTExit();
55          calclFinalizeCALOpencl(calOpenCL);
56          calclFinalizeToolkit2D(sciddicaToolkit);
57
58          return 0;
59  }
```

In listing 4.14 the host-side main file of the SCIDDICA-T implementation is shown. Most of this code is equal to that of the examples presented in section 4.2.1, the only difference being that platform and device are chosen by using indexes (cf. `platformNum` and `deviceNum` at line 17). The OpenCL context and program definition, together with source file loading, model initialization, kernels and buffers declaration, are really similar to that in listings 4.6 and 4.8. Note that only the `bufferEpsilonParameter` and the `bufferRParameter` are used as kernels arguments, while nor the `MODEL_DEFINITION2D` name, neither the corresponding parameters, appear explicitly in the source code. As already discussed, the use of the name `MODEL_DEFINITION2D` allows for a considerable reduction of complexity kernels parameters, by also making source code more readable. Eventually, kernels and steering function are registered and the simulation executed by

means of the `calclRun2D` OpenCAL function. Finally, the CA configuration is saved and finalizing functions called.

4.3 OpenCAL parallel computational performance

In order to evaluate the computational performance of the OpenCL parallel implementation of OpenCAL, various experiments were carried out by considering two different GPUs and a 4-core CPU system. A nVidia Tesla K20c, having 2496 cuda cores and a total of 5 GB of GDDR5 RAM, and an AMD SAPPHIRE VAPOR-X R9 280X, having 2048 stream processors and a total of 3 GB of GDDR5 RAM, were used as GPUs accelerators, while a 2.83 GHz dual quad core Xeon E5440 system having 8 GB of RAM chosen as CPU accelerator. This choice was suggested by the fact that a workstation equipped with the above mentioned hardware was available at the time experiments were executed. Table 4.1 lists details of the adopted computational devices.

Device Specs	Intel Xeon E5440	AMD R9 280X	nVidia Tesla K20c
Clock	2.83 GHz	1.07 GHz	0.706 GHz
Cores	4	2048	2496
L1 Cache	128 KB	16 KB	16 KB
L2 Cache	12 MB	2048 KB	1536 KB
Memory Size	8 GB	3.072 GB	5.120 GB
Memory Clock	-	6200 MHz	5200 MHz
Memory Type	-	GDDR5	GDDR5
Memory Bus	-	384 bit	320 bit
Bandwidth	-	298 GB/s	208 GB/s
Peak Performance	-	3,891 GFLOPS	3,524 GFLOPS

Table 4.1: Characteristics of the accelerators used for evaluating the computational performance of the SCIDDICA-T cellular automaton implementation in OpenCAL.

The parallel implementation of SCIDDICA-T, presented in the previous section, was chosen as reference test case and the simulation of the Tessina (Italy) landslide (shown in Figure 4.1) considered for evaluating the different performances among serial and parallel executions. The simulation con-

sisted in a total of 15,000 computational steps, while the dimension of the bi-dimensional cellular space was 610 rows times 496 columns. Serial code was executed on a single core of the E5440 Xeon processor, while different parallel executions were launched on both the dual quad core Xeon CPU and the nVidia and AMD GPUs. Figures 4.2 and 4.3 summarize the obtained results.

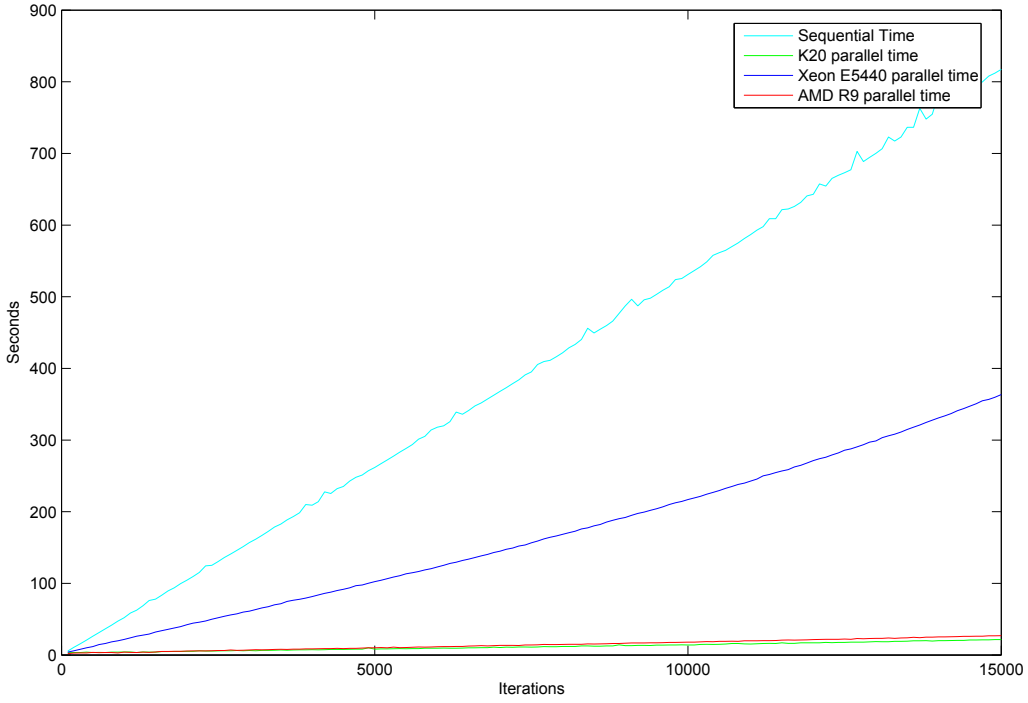


Figure 4.2: Computational performance in terms of elapsed time of the SCIDDICA-T simulation of the Tessina (Italy) landslide.

As reported, a sufficient improvement is obtained in the case of the parallel execution on the CPU based system, which corresponds to about a speedup of 2 (instead of 4) with respect the serial execution on the single core. This is probably due to the overhead coming with the adopted strategy of one thread per cell, which leads to total of 302,560 thread to be processed by 4 cores at each one of the 15,000 computational steps needed to perform the simulation. At the contrary, considerable speedup is obtained by considering the two GPUs. In fact a speedup of about 30 was obtained for the case of the AMD R9 280X, while a speedup of about 40 for the case of the nVidia

Tesla K20c. It seems evident that the high number of threads used greatly favors the computation on the considered GPUs.

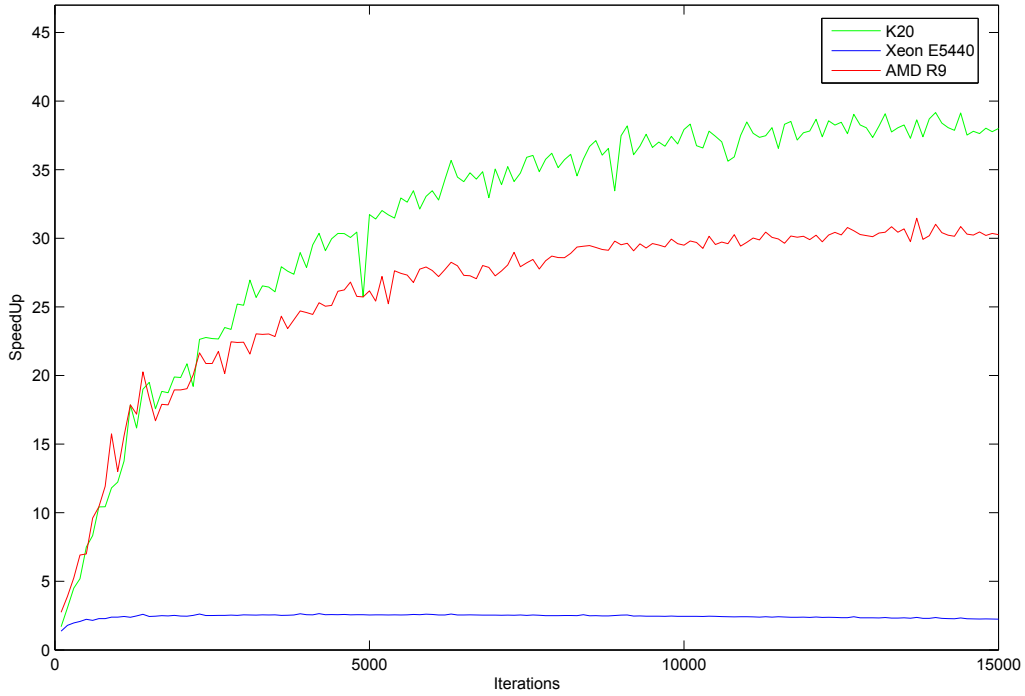


Figure 4.3: Computational performance in terms of speedup of the SCIDDICA-T simulation of the Tessina (Italy) landslide.

As regards the comparison of the adopted GPUs, results of the AMD device overcame the nVidia K20c for about the first 1,000 computational steps, while the nVidia device was faster for all the remaining computational steps. This behavior can be explained by considering the characteristics of the two devices, shown in Table 4.1. In fact, the AMD R9 280X GPU is basically a graphic card used by videogamers, with higher GPU and memory clock and a faster bus. At the contrary, nVidia Tesla K20c has different target: it is a high performance (indeed more expensive) graphic card, specifically designed for high performance computing. As a confirm, it has a greater memory size and more processing units (CUDA cores). Therefore, it is not surprisingly that the AMD GPU was able to go faster in the first computational steps, thanks to its higher single processing unit performance, while the nVidia K20c was able to overcome the performance of the first GPU

when the computational request grew; this is, indeed, exactly the case of the considered simulation, where the number of active cells (i.e. of those cells which are really involved in the computation, having a non null amount of landslide debris to be distributed to its neighbors) grew step by step.

Experiments results show significant performance improvements of the considered cellular automaton with respect the specific case of study, confirming that GPUs can be fruitfully employed for speeding up Cellular Automata, by also representing a cheaper alternative to classic high performance hardware solutions. Moreover, GPUs are generally more ecologic as they require lower power energy with respect to classic high performance computers (e.g. clusters).

As concerns OpenCAL, it allowed for a easy and straightforward parallelization of the considered cellular automata, by also allowing to have very similar serial and parallel codes. Obviously, *ad hoc* and explicit OpenCL implementation can overcome OpenCAL performance, so that programmers has to evaluate the tradeoff of the advantages of having a considerably simplified code development is greater than the possible lack of performance. Nevertheless, further analysis should be performed in order better understand this issue.

5 Conclusions

The work presented in this thesis has concerned the application of Web 2.0 and High-Performance Computing (HPC) technologies for the development of new scientific applications for the simulation of complex macroscopic natural phenomena. Specifically, I have developed three software for the simulation of debris flows, lava flows, and wildfire evolution, respectively.

The adopted numerical models are based on the Cellular Automata (CA) computational paradigm and are already known to the Scientific Community. Specifically, I considered the CA model SCIDDICA-k1 for simulating debris flows, the SCIARA-fv3 CA for simulating lava flows, and ABBAMPAU for simulating wildfire, respectively. The numerical models are based on the empirical method proposed by Di Gregorio and Serra for the simulation of complex macroscopic phenomena and can be considered reliable as they were already successfully applied to the simulation of many real cases of study.

For each of the above cited numerical models, I have developed a novel Web-based rich Graphical User Interface (GUI), which allows to both interact with the underlying numerical model and visualize results in real time, thanks to the adoption of the WebGL application program interface for Web-based 3D interactive computer graphics. Furthermore, with the aim of improving the performance of the computational models, that is often a key-factor in Scientific Computing, I contributed to the parallelization of OpenCAL, a new software library written in the C programming language and developed at the Department of Mathematics and Computer Science of the University of Calabria (Italy) for the implementation of Cellular Automata applications.

At this purpose, the Open Computing Language (OpenCL) was adopted, one of the most important frameworks for writing programs that can be executed across heterogeneous platforms, such as central processing units (CPUs) and graphics processing units (GPUs). OpenCAL allows to implement CA models in a straightforward manner and to execute the simulations on GPUs transparently to the programmer.

By combining the above cited technologies, it was possible to obtain software characterized by many advantages with respect to the classical desktop-based counterpart. For instance, the developed new applications are completely cross-platform and do neither need an installation nor software update process. Starting from Swii, a first and preliminary web application for the simulation of debris flows, the new Swii2 application has been developed by introducing the new landslide CA model SCIDDICA-k1 and, for the first time, WebGL as client-side 3D rendering engine in the specific scientific context. Subsequently, SciaraWii was developed for the simulation of lava flows by means of the CA lava flow model SCIARA-fv3. Significant performance improvements of the visualization system, originally developed for Swii2, were achieved thanks to the introduction of the *decimation technique*. In fact, it was possible to visualize and interact in real time with huge datasets, without the need to resort to expensive hardware. It was also possible to easily display results on mobile devices such as smartphones and tablets. Eventually, Awii was developed for the simulation of wildfires evolution by means of the CA model ABBAMPAU. In Awii, further developments concerned the server side of the software, where a sharp separation between the computational model and the rest of the application was achieved. This contributed to improve the overall efficiency of the system and allowed to adopt it in advanced applications where the simultaneous execution of multiple simulations are required. In order to stress the system's reliability, the web applications were tested by considering a PC as Web server and a maximum of ten other PCs as remote clients, accessing simultaneously the former. Performances results were more than satisfactory, and no system delays observed. The potentialities of the Web 2.0 have therefore been confirmed.

However, in the case of more heavy execution requests, a sensible server slow down could be observed. Furthermore, a possible degradation could also arise in the case the execution of the considered computational models is performed on greater datasets or for longer physical duration (for instance when a simulation of a real phenomenon lasted years is required). In order to prevent possible slow down situations, the numerical simulation could be executed in parallel on more processing elements, for instance on GPUs. At this purpose, it would be possible to adopt OpenCAL. However, at the current stage, OpenCAL has been only used to parallelize test models, such as

a simplified landslide CA simulator, named SCIDDICA-T, and preliminary tests of performance have been executed. Such tests evidenced considerable speedup on GPUs with respect serial execution on CPU on different graphic cards, such as the AMD SAPPHIRE VAPOR-X R9 280X with 3GB GDDR5, 2048 Stream Processor units and a 1070 MHz GPU clock, and the nVidia Tesla K20c with 5GB GDDR5, 2496 CUDA cores and a 706 MHz GPU Clock, respectively. Nevertheless, tests have regarded OpenCAL executions by considering an *Intel Xeon* processor model E5440 at 2.83GHz and 12MB of second level cache as an OpenCL device (i.e., considering it as a parallel hardware). A speedup slightly over 2 was also achieved by the parallel execution on the CPU by using 4 computational cores. This result is probably due to the fact that the parallel algorithm allocates one thread per each cell of the cellular space, that in the considered case is composed by more than 300.000 cells, by producing a great overhead in case of only 4 cores are used for they elaboration. However, a speedup of 30 was obtained by considering the execution of SCIDDICA-T on the AMD GPU, while even of 40 on the Tesla K20c GPU, with respect to the serial execution on a single core of the Xeon processor.

The research presented and discussed in this work cannot certainly be considered definitive, even if the obtained results can be considered satisfactory. Many improvements and additional features can be added to the developed applications and the computational models considered reimplemented by using OpenCAL in order to have the possibility to speed up simulations on GPUs. Moreover, the work related to the development of OpenCAL can be extended and new parallel versions in OpenMP or MPI, implemented in order to exploit multicore processors and cluster of PCs.

Acknowledgments

First of all I would like to express my gratitude to my supervisors **Dr. Donato D'Ambrosio** and **Dr. William Spataro** for having given me guidance, encouragement and motivation throughout my research path. I also wish to express my sincere appreciation to **Giuseppe Filippone** for his advice and inspiration for this work. A special thank goes to my co-workers **Davide Spataro, Alessio De Rango, Maurizio Macrí** (especially for having explained what *the steering* is). I would like to thank my friends which have always been close to me in difficult moments. Thanks to the **Dream Team** and all the boys of football matches. Thanks to my family for their support and love. Finally, a special thanks to **Francesca**, for supporting me and for having been close to me all times.

Bibliography

[1] M.E. Alexander. Estimating the length-to-breadth ratio of elliptical forest fire patterns. In *Proc. 8th Conf. Fire and Forest Meteorology*, pages 287–304, 1985.

[2] H.E. Anderson. Predicting wind-driven wildland fire size and shape. Technical Report INT-305, U.S Department of Agriculture, Forest Service, 1983.

[3] P.L. Andrews. BEHAVE: fire behavior prediction and fuel modeling system - burn subsystem, part 1. Technical Report INT-194, U.S Department of Agriculture, Forest Service, 1986.

[4] A. Armanini. On the dynamic impact of debris flows. In A. Armanini and F. Michiue, editors, *Recent development on debris*, volume 64 of *LNES*, pages 208–226. Springer Verlag, Berlin, 1997.

[5] A. Armanini and L. Fraccarollo. Critical conditions for debris flows. In C.I. Chen, editor, *Debris-flow hazard mitigation: mechanics, prediction, and assessment*, pages 434–443, 1997.

[6] Maria Vittoria Avolio, Gino Mirocle Crisci, Salvatore Di Gregorio, Rocco Rongo, William Spataro, and Donato D'Ambrosio. Pyroclastic flows modelling using Cellular Automata. *Computers & Geosciences*, 32:897–911, 2006.

[7] Maria Vittoria Avolio, Salvatore Di Gregorio, Valeria Lupiano, and Giuseppe A. Trunfio. Simulation of wildfire spread using cellular automata with randomized local sources. In *ACRI 2012*, volume 7495 of *LNCS*, pages 279–288. Springer Berlin / Heidelberg, 2012.

[8] M.V. Avolio, S. DiGregorio, F. Mantovani, A. Pasuto, R. Rongo, S. Silvano, and W. Spataro. Simulation of the 1992 Tessina landslide by a cellular automata model and future hazard scenarios. *International Journal of Applied Earth Observation and Geoinformation*, 2:41–50, 2000.

[9] Conway J. H Berlekamp, E. R. What is life? *Winning Ways for Your Mathematical Plays, Games in Particular*, 2, 1982.

[10] Ivan Blecic, Arnaldo Cecchini, and Giuseppe A. Trunfio. A generalized rapid development environment for cellular automata based simulations. In *Cellular Automata*, volume 3305 of *LNCS*, pages 851–860. Springer Berlin Heidelberg, 2004.

[11] Ivan Blecic, Arnaldo Cecchini, and Giuseppe A. Trunfio. A general-purpose geosimulation infrastructure for spatial decision support. *Transactions on Computational Science*, 6:200–218, 2009.

[12] Claudia Roberta Calidonna, Adele Naddeo, Giuseppe A. Trunfio, and Salvatore Di Gregorio. From classical infinite space-time ca to a hybrid ca model for natural sciences modeling. *Applied Mathematics and Computation*, 218(16):8137–8150, 2012.

[13] J. Conway. The game of life. *Scientific American*, 1970.

[14] Matthew Cook. Universality in elementary cellular automata. *Complex Systems*, 15(1):1–40, 2004.

[15] Gino M. Crisci, Rocco Rongo, Salvatore Di Gregorio, and William Spataro. The simulation model SCIARA: the 1991 and 2001 lava flows at Mount Etna. *Journal of Volcanology and Geothermal Research*, 132(23):253 – 267, 2004.

[16] D. D'Ambrosio, G. Filippone, R. Rongo, W. Spataro, and G.A. Trunfio. Cellular automata and GPGPU: an application to lava flow modeling. *International Journal of Grid and High Performance Computing*, 4(3):30–47, 2012.

[17] Donato D'Ambrosio, William Spataro, and Giulio Iovine. Parallel genetic algorithms for optimising cellular automata models of natural complex phenomena: An application to debris flows. *Computers & Geosciences*, 32(7):861–875, 2006.

[18] M. DelPrete, F.M. Guadagno, and A.B. Hawkins. Preliminary report on the landslides of 5 May 1998, Campania, Southern Italy. *Bulletin of Engineering Geology and the Environment*, 57:113–129, 1998.

[19] Salvatore Di Gregorio and Roberto Serra. An empirical method for modelling and simulating some complex macroscopic phenomena by cellular automata. *Future Generation Computer Systems*, 16(2-3):259–271, 1999.

[20] Salvatore Di Gregorio and Roberto Serra. An empirical method for modelling and simulating some complex macroscopic phenomena by cellular automata. *Future Generation Comp. Syst.*, 16(2-3):259–271, 1999.

[21] Salvatore Di Gregorio, Roberto Serra, and Marco Villani. Applying cellular automata to complex environmental problems: The simulation of the bioremediation of contaminated soils. *Theoretical Computer Science*, 217(1):131–156, 1999.

[22] G.H. Eisbacher and J.J. Clague. *Destructive mass movement in high mountains: hazard and management*, chapter Geological Survey of Canada, pages 84–16. Ottawa, Canada, 1984.

[23] Mark A. Finney. FARSITE: fire area simulator-model development and evaluation. Technical Report RMRS-RP-4, U.S Department of Agriculture, Forest Service, 2004 February 2004.

[24] I.A. French, D.H. Anderson, and E.A. Catchpole. Graphical simulation of bushfire spread. *Mathematical Computer Modelling*, 13:67–71, 1990.

[25] U. Frisch, B. Hasslacher, and Y. Pomeau. Lattice-gas automata for the navier-stokes equation. *Phys. Rev. Lett.*, 56(14):1505–1508, April 1986.

[26] Vichniac G. Simulating physics with cellular automata. *Physica*, D 10:96–115, 1984.

[27] Pat Gelsinger. Intel spring forum. 2004.

[28] Moore Gordon. Cramming more components onto integrated circuits. *Electronics*, 38:114 ff., 1965.

[29] S. Di Gregorio and R. Serra. An empirical method for modelling and simulating some complex macroscopic phenomena by cellular automata. *Fut. Gener. Comp. Sys.*, 16:259–271, 1999.

[30] Salvatore Di Gregorio and Roberto Serra. An empirical method for modelling and simulating some complex macroscopic phenomena by cellular automata. *Future Generation Computer Systems*, 16(23):259 – 271, 1999.

[31] Tacy A. Hanson, R. *GWT in Action.* Manning Pubblications Co, 2007.

[32] Xiaolin Hu and Lewis Ntaimo. Integrated simulation and optimization for wildfire containment. *ACM Transactions on Modeling and Computer Simulation*, 19(4):1–29, 2009.

[33] Andrew Ilachinski. *Cellular Automata: A Discrete Universe*. World Scientific, Singapore, 2001.

[34] R.M. Iverson. The physics of debris flows. *Reviews in Geophysics*, 35:245–296, 1997.

[35] P. Johnston, J. Kelso, and G.J. Milne. Efficient simulation of wildfire spread on an irregular grid. *International Journal of Wildland Fire*, 17:614–627, 2008.

[36] Sheng Yu Karel Culik. *Undecidability of CA Classification Schemes*. 1998.

[37] William J. III Kauffmann and Larry L. Smarr. *Supercomputing and the Transformation of Science.* Scientific American Library, 1993.

[38] D.K. Keefer, R.C. Wilson, R.K. Mark, E.E. Brabb, W.M. Brown III, S.D. Ellen, E.L. Harp, G.F. Wieczorek, C.S. Alger, and R.S. Zatkin. Real-time landslide warning during heavy rainfall. *Science*, 238:921–925, 1987.

[39] J.E. Kesseli. Disintegrating soil slips of the coast ranges of central california. *Journal of Geology*, 51(5):342–352, 1943.

[40] C.G. Langton. Computation at the edge of chaos. Master's thesis, Univeristy of Michigan, 1990.

[41] Chris G. Langton. Computation at the edge of chaos: phase transitions and emergent computation. *Physica D*, 42(1-3):12–37, 1990.

[42] A. M. G. Lopes, M. G. Cruz, and D. X. Viegas. Firestation - an integrated software system for the numerical simulation of fire spread on complex topography. *Environmental Modelling and Software*, 17(3):269–285, 2002.

[43] E. Marchi and A. Rubatta. *Meccanica dei fluidi. Principi e applicazioni.* UTET, Torino, 1981.

[44] R.K. Mark and S.D. Ellen. Statistical and simulation models for mapping debris-flow hazard. In A. Carrara and F. Guzzetti, editors, *Geographical Information Systems in assessing natural hazards*, pages 93–106, 1995.

[45] R.S. McAlpine, B.D. Lawson, and E. Taylor. Fire spread across a slope. In *Proceedings of the 11th Conference on Fire and Forest Meteorology (Society of American Foresters: Bethesda, MD)*, pages 218–225, 1991.

[46] G. Vichniac N. Margolus, T. Toffoli. Cellular automata supercomputers for fluid-dynamics modelling. *Phys. Rev. Lett.*, 56:1694–1696, 1986.

[47] United Nations. *Mudflows. Experience and lessons learned from the management of major disasters.* Departiment of Humanitarian Affairs, Geneva, 1987.

[48] Nvidia. *CUDA C Programming Guide.* 2012.

[49] NVIDIA Corporation. *CUDA C Best Practices Guide.* NVIDIA Corporation, 2701 San Tomas Expressway, Santa Clara 95050, USA, 5.0 edition, October 2012.

[50] Roberto Parise, Donato DAmbrosio, Giuseppe Spingola, Giuseppe Filippone, Rocco Rongo, GiuseppeA. Trunfio, and William Spataro. Swii2, a html5/webgl application for cellular automata debris flows simulation. In GeorgiosCh. Sirakoulis and Stefania Bandini, editors, *Cellular Automata*, volume 7495 of *Lecture Notes in Computer Science*, pages 444–453. Springer Berlin Heidelberg, 2012.

[51] Seung Park and James D. Iversen. Dynamics of lava flow: Thickness growth characteristics of steady two-dimensional flow. *Geophysical Research Letters*, 11(7):641–644, 1984.

[52] Seth H. Peterson, Marco E. Morais, Jean M. Carlson, Philip E. Dennison, Dar A. Roberts, Max A. Moritz, and David R. Weise. Using HFIRE for spatial modeling of fire in shrublands. Technical Report PSW-RP-259, U.S. Department of Agriculture, Forest Service, Pacific Southwest Research Station, Albany, CA, 2009.

[53] Rocco Rongo, William Spataro, Donato D'Ambrosio, Maria Vittoria Avolio, Giuseppe A. Trunfio, and Salvatore Di Gregorio. Lava flow hazard evaluation through cellular automata and genetic algorithms: an application to Mt Etna volcano. *Fundamenta Informaticae*, 87(2):247–267, 2008.

[54] R. C. Rothermel. A mathematical model for predicting fire spread in wildland fuels. Technical Report INT-115, U.S. Department of Agriculture, Forest Service, Intermountain Forest and Range Experiment Station, Ogden, UT, 1972.

[55] R. C. Rothermel. How to predict the spread and intensity of forest and range fires. Technical Report INT-143, U.S. Department of Agriculture, Forest Service, Intermountain Forest and Range Experiment Station, Ogden, UT, 1983.

[56] H. Rouse. *Engineering Hydraulics*. John Wiley & Sons, Chichester, 1950.

[57] Barbara Chapman Rungan Xu, Sunita Chandrasekaran. An openacc code for a c-based heat conduction code. *www.openacc.com*, 2012.

[58] William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen. Decimation of triangle meshes. In *Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1992*, pages 65–70, 1992.

[59] G.Ch Sirakoulis, I Karafyllidis, and A Thanailakis. A cellular automaton model for the effects of population movement and vaccination on epidemic propagation. *Ecological Modelling*, 133(3):209–223, September 2000.

[60] William Spataro, Maria Vittoria Avolio, Valeria Lupiano, Giuseppe A. Trunfio, Rocco Rongo, and Donato D'Ambrosio. The latest release of the lava flows simulation model SCIARA: First application to Mt Etna (Italy) and solution of the anisotropic flow direction problem on an ideal surface. In *International Conference on Computational Science*, pages 17–26, 2010.

[61] Giandomenico Spezzano, Domenico Talia, Salvatore Di Gregorio, Rocco Rongo, and William Spataro. A parallel cellular tool for interactive modeling and simulation. *Computing in Science and Engineering*, 3(3):33–43, 1996.

[62] S. Succi. *Automi cellulari. Una nuova frontiera del calcolo scientifico.* Collana informatica domani / IBM SEMEA. Franco Angeli, 1991.

[63] T. Takahashi. Initiation and flow of various types of debris flow. In G.F. Wieczorek and N.D. Naeser, editors, *Debris-flow hazards mitigation: Mechanics, prediction, and assessment*, Proceedings 2nd International Conference on debris-flow hazard mitigation, pages 15–25, Taipei, Taiwan, August 2000.

[64] J.W. Thatcher. Universality in the von neumann cellular mode. *A. W. Burks (Ed.), Essays on Cellular Automata*, pages 103–131, 1970.

[65] Tommaso Toffoli. Cellular automata as an alternative to (rather than an approximation of) differential equations in modeling physics. *Physica D: Nonlinear Phenomena*, 10(1-2):117–127, January 1984.

[66] Tommaso Toffoli and Norman Margolus. *Cellular automata machines: a new environment for modeling*. MIT Press, Cambridge, MA, USA, 1987.

[67] Paul M. Torrens and Itzhak Benenson. Geographic automata systems. *International Journal of Geographical Information Science*, 19(4):385–412, 2005.

[68] Giuseppe A. Trunfio. Predicting wildfire spreading through a hexagonal cellular automata model. In *Cellular Automata*, volume 3305 of *LNCS*, pages 385–394. Springer Berlin Heidelberg, 2004.

[69] J.M. Vasconcelos, B.P. Zeigler, and J. Pereira. Simulation of fire growth in GIS using discrete event hierarchical modular models. *Advances in Remote Sensing*, 4(3):54–62, 1995.

[70] John Von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, Champaign, IL, USA, 1966.

[71] J. von Neumann (Edited and complete by A. Burks). Theory of self-reproducing automata. *University of Illinois Press*, 1966.

[72] G.F. Wieczorek and N.D. Naeser, editors. *Debris-flow hazards mitigation: mechanics, prediction, and assessment*, Rotterdam, 2000. Proceedings 2nd International Conference on Debris Flow Hazards Mitigation, Balkema.

[73] S. Wolfram. Cellular automaton uids 1: Basics theory. *Journal of Statistical Physics*, 45:471–526, 1986.

[74] Stephen Wolfram. Statistical mechanics of cellular automata. *Reviews of Modern Physics*, 55(3):601–644, 1983.

[75] Stephen Wolfram. Computation theory of cellular automata. *Communications in Mathematical Physics*, 96(1):15–57, 1984.

[76] Stephen Wolfram. *A New Kind of Science*. Wolfram Media, 2002.

List of Figures

List of Tables