UNIVERSITY OF CALABRIA

Department of Mathematics and Computer Science
Via P. Bucci
I-87036 Rende, Italy

OpenCAL User Guide

The Open Computing Abstraction Layer

Version 1.0

Donato D'Ambrosio, Alessio De Rango, Davide Spataro, and William Spataro

April, 2016

Acknowledgements

We are gratefull to all OpenCAL developers and supporters.
In particular, we acknowledge Francesca "Chesire Cat" Aceto, Roberto "Psrom" Parise, Luigi Olivella, Paola "AruQ" Arcuri, Carmelo La Gamba, Maurizio Macri', Marco "Pecos" Oliverio, Rocco "Rick" Rongo, Alfonso Senatore, and Mario "Suinos" D'Onghia.

Contents

Chapter 1

Introduction

OpenCAL (Open Computing Abstraction Layer) is a parallel computational software library, developed as an Open Source project at the Department of Mathematics and Computer Science of the University of Calabria (Italy) and released under the LGPL v3.0 license.

OpenCAL allows for the definition of numerical simulation models based on the Exteded Cellular Automata (XCA) general formalism, thus supporting Cellular Automata (CA), the Finite Differences method (FDM) and, in general, all numerical methods based on structured computational grids.

OpenCAL is developed in C for the maximum efficiency, can be used in C/C++ applications, and can run in parallel on both CPUs, thanks to its implementation based on OpenMP, and on GPUs, thanks to its implementation based on OpenCL.

The library has been tested on both CPUs and GPUs by considering different CA examples of application, including the well known Conway's Game of Life and the SciddicaT XCA debris flows simulation model. Results have demonstrated the goodness the library both in terms of usability and performance.

In the present release, 2D and 3D numerical models can be defined. Actually, even 1D models can be defined as a degenerate case of 2D CA. The library also offers diverse facilities (e.g. it provides many predefined cell neighborhoods), allows to make the simulation main loop explicit and provides a built in optimization algorithm to speed up the simulation. Moreover, OpenCAL offers a built in interactive 2D/3D visualization system developed in OpenGL Compatibility Profile, so that it can run everywhere, even on old workstations.

The present manual reports the main usage of the `OpenCAL` library related to the sequential, OpenMP- and OpenCL-based versions, the installation procedure, besides examples of application. In particular, Chapter 2 deals with download and installation, while Chapter 3 introduces the XCA computational paradigm. Chapter 4 is about serial CA and XCA development with OpenCAL, and introduces the different library features by examples. Chapter 5 is about the OpenMP-based parallel version of OpenCAL and also introduces the library by examples. Chapter 6 briefly introduces to General Purpose GPU programming with OpenCL and then presents the OpenCL-based version of OpenCAL, still by examples. OpenCAL-GL is discussed at the end of each of the above Chapters, together with computational performances of some of the implemented CA.

Chapter 2

Installation

The release of `OpenCAL`, here presented, is a collection of four different software libraries. Under the name `OpenCAL` we identify the serial version of the library. It comes together with two different parallel implementations based on OpenMP and OpenCL, namely OpenCAL-OMP and OpenCAL-CL, respectively. Eventually, OpenCAL-GL identifies an OpenGL/GLUT-based visualization library.

The library can be currently obtained only as source code to be compiled from GitHub. Some dependencies must be satisfied, depending on which libraries must be compiled. Eventually, headers and libraries can be installed and, eventually, uninstalled if no longer needed. In the following Sections we will see all the steps needed to obtain the software and make it working.

2.1 Requirements and dependencies

In order to build the different libraries, you need a quite recent ANSI C compiler (e.g. gcc under Linux/Unix systems or Microsoft Visual C++ under Windows[1]), and CMake (to generate the makefiles, or even project files for several IDEs like Eclipse). An OpenCL implementation (e.g. by AMD, Intel or NVIDIA) is also needed to build OpenCAL-CL, as well as GLUT (e.g. freeglut) is needed to build OpenCAL-GL. Some dependencies must also be satisfied, depending on the library to be built:

OpenCAL: A quite recent ANSI C compiler, and CMake version 2.8 or greater.

OpenCAL-OMP: A C compiler supporting at least Open-MP version 2.0[2], and CMake version 2.8 or greater.

Open-CL: A quite recent ANSI C compiler, CMake version 3.1 or greater, and OpenCL version 1.2 or greater.

OpenCAL-GL: A quite recent ANSI C compiler, CMake version 2.8 or grater, OpenGL/GLUT headers and libraries [3]. Moreover, POSIX Threads are required.

[1] The clang C compiler can also be used, taking in mind that it still does not fully support Open-MP natively.

[2] For a list of OpenMP compliant compilers see the following link: http://openmp.org/wp/openmp-compilers/.

[3] For example `freeglut-devel` or `freeglut3-dev` packages on `yum/dnf`- and `apt`-based systems, respectively.

Eventually, Doxygen and Graphviz are required to build the software documentation, which provides specific information about implemented data structures and functions.

Note that Visual Studio users under Microsoft Windows have to use the the freeglut and pthreads provided with OpenCAL. In this case, please copy the libs directory, containing both freeglut and pthreads for Microsoft Visual Studio, in the desired path on the local disk (e.g. in `C:\`). Moreover, paths containing DLLs (e.g. `C:\libs\pthreads\bin` and `C:\libs\freeglut\bin`) have to be added to the Windows PATH environment variable, in order to be found by the loader.

2.2 Obtaining OpenCAL

The stable releases of libraries and examples can be downloaded as source code at the following GitHub urls:

opencal-1.0	https://github.com/OpenCALTeam/opencal/archive/1.0.zip
opencal-examples-1.0	https://github.com/OpenCALTeam/opencal-examples/archive/1.0.zip

Development releases of libraries and examples can also be downloaded (for instance into the `~/git` directory, where the symbol ~ identifies the user's home directory) by cloning the GitHub repositories:

```
user@machine:-$ cd ~/git
user@machine:-$ git clone https://github.com/OpenCALTeam/opencal
user@machine:-$ git clone https://github.com/OpenCALTeam/opencal-examples
```

Note that, Microsoft Windows users have to set their git client such that it does not convert files from the UTF-8 format towards any other file format (e.g. ANSI), since OpenCL kernel must currently be of the former format.

2.3 Build and Install

2.3.1 Generating project files with CMake

In order to generate the Unix makefiles or project files for Microsoft Visual Studio (or other IDEs), needed to compile OpenCAL libraries and examples, the following steps can be carried out:

1. Enter the OpenCAL source tree root directory (e.g. `~/git/opencal-1.0/`);
2. Create a directory for the binaries (e.g. `~/git/opencal-1.0/build/`) and enter into it;
3. Run CMake using the options listed in Table 2.1 to control which features will be enabled in the compiled libraries.

Each CMake option corresponds to a target. If you are not interested in some of them, simply switch off the corresponding option. If you omit a CMake option, the default value will be assumed (cf. Table 2.1). If you want to build everything (serial and parallel libraries, examples and documentation), use the following commands:

Table 2.1: List of CMake options, alongside their default values and effects

CMake option	Default	Effect
BUILD_OPENCAL_SERIAL	ON	Build serial version
BUILD_OPENCAL_OMP	ON	Build OpenMP-based parallel version
BUILD_OPENCAL_CL	OFF	Build OpenCL-based parallel version
BUILD_OPENCAL_GL	OFF	Build OpenCAL-GL visualization library
BUILD_DOCUMENTATION	OFF	Build HTML API documentation

```
user@machine:-$ cd ~git/opencal-1.0/
user@machine:-$ mkdir build && cd build
user@machine:-$ cmake ../ -DBUILD_OPENCAL_SERIAL=ON \
                          -DBUILD_OPENCAL_OMP=ON \
                          -DBUILD_OPENCAL_CL=ON \
                          -DBUILD_OPENCAL_GL=ON \
                          -DBUILD_DOCUMENTATION=ON
```

Under Windows, freeglut and/or pthreads could not be found by CMake modules, depending on the path where they were installed. In this case, it is possible to use the CMake GUI and provide the include path and libraries explicitly. For instance, if the aforementioned dependencies have been satisfied as suggested, pthreads headers could be found in `C:\libs\pthreads\include`, while the library in `C:\libs\pthreads\lib\pthread.lib`. The same holds for the freeglut library.

Custom installation path

Another useful CMake option is `CMAKE_INSTALL_PREFIX:PATH`, that allows to change the default installation directory, that is `/usr/local` under Linux systems and `C:\Program Files (x86)` under Windows. Therefore, to select `/opt` as installation target directory, change the above invocation of CMake into the one below.

```
user@machine:-$ cd ~git/opencal-1.0/
user@machine:-$ mkdir build && cd build
user@machine:-$ cmake ../ -DBUILD_OPENCAL_SERIAL=ON \
                          -DBUILD_OPENCAL_OMP=ON \
                          -DBUILD_OPENCAL_CL=ON \
                          -DBUILD_OPENCAL_GL=ON \
                          -DBUILD_DOCUMENTATION=ON\
                          -DCMAKE_INSTALL_PREFIX:PATH=/opt
```

or, under Microsoft Windows:

```
user@machine:-$ cd C:\git\opencal-1.0\
user@machine:-$ mkdir build && cd build
user@machine:-$ cmake ../ -DBUILD_OPENCAL_SERIAL=ON \
                          -DBUILD_OPENCAL_OMP=ON \
                          -DBUILD_OPENCAL_CL=ON \
                          -DBUILD_OPENCAL_GL=ON \
                          -DBUILD_DOCUMENTATION=ON\
                          -DCMAKE_INSTALL_PREFIX:PATH=C:\libs
```

2.3.2 Compiling

Once makefiles have been produced, everything is set up and ready for compiling. To compile under Linux, use the following command:

```
user@machine:-$ make -j n
```

where n is the number of cores of your machine you want to use for speeding up the compilation process.

Under Windows it is sufficient to open the OpenCAL-1.0.sln Visual Studio solution and press the F7 key to build the libraries.

2.3.3 Install

You can install the compiled libraries, headers and API documentation using the following Linux command:

```
user@machine:-$ sudo make install
```

or equivalently, if the user is not in the `sudoers` list:

```
user@machine:-$ sudo -
root@machine:-$ make install
```

Under Linux systems, files are installed by default in `/usr/local/`. See Table 2.2 for major details.

Table 2.2: Default installation paths for OpenCAL files under Linux systems

Installation path	Installed objects
`/usr/local/opencal-1.0/lib/`	Shared objects
`/usr/local/opencal-1.0/include/`	Header files
`/usr/local/opencal-1.0/include/OpenCAL-CL/kernel`	OpenCL kernels
`/usr/local/opencal-1.0/doc/`	API documentation

Under Windows, it is sufficient to compile the INSTALL project, that can be found under the Visual Studio solution. Note that, in order to install files in the default path, Visual Studio has to be run as Administator. In the case the installation path was set to `C:\libs`, as we suggest, files will be installed as described in Table 2.3. Note that, as for dependencies, also the directory containing OpenCAL DLLs (e.g. `C:\libs\opencal-1.0\bin`) have to be added to the Windows PATH environment variable.

Table 2.3: Installation paths for OpenCAL files under Windows systems.

Installation path	Installed objects
`C:\libs \opencal -1.0\bin`	DLLs
`C:\libs \opencal -1.0\lib`	.lib archives
`C:\libs \opencal -1.0\include`	Header files
`C:\libs \opencal -1.0\include \OpenCAL -CL\kernel`	OpenCL kernels
`C:\libs \opencal -1.0\doc`	API documentation

One more operation is necessary if you want use CMake to compile your own OpenCAL-based applications, i.e. copy the `FindOpenCAL.cmake`, located in `~/git/opencal/cmake`, into the directory containing the other CMake modules. Under Linux systems, such a directory is usually `/usr/share/cmake/Modules`. Therefore, you can simply use the following command to copy the OpenCAL find module into the appropiate directory:

```
user@machine:~$ sudo cp ~/git/opencal/cmake/FindOpenCAL.cmake /usr/share/
    cmake/Modules
```

Under Windows, CMake modules can be generally found in the following `C:\Program Files (x86)\CMake\share\cmake-3.5\Modules` directory (or a similar path, depending on CMake version).

2.3.4 Test the installation

In order to test software installation you can compile and run the `life` example program in `~/git/opencal-examples/OpenCAL/cal_life`. You can use CMake, as we did before, or compile it directly.

If you want to use CMake, follow the steps below:

```
user@machine:~$ cd ~/git/opencal-examples/OpenCAL/cal_life
user@machine:~$ mkdir build && cd build
user@machine:~$ cmake ..
user@machine:~$ cd ..
user@machine:~$ ./bin/cal_life
```

In the latter case you need to specify where include files and shared objects are located and link the required library, as in the following example:

```
user@machine:~$ cd ~/git/opencal-examples/OpenCAL/cal_life
user@machine:~$ gcc source/life.c -o life -I/usr/local/opencal-1.0/include \
                                          -L/usr/local/opencal-1.0/lib \
                                          -lopencal
user@machine:~$ ./life
```

In both cases, the `life_0000.txt` and `life_LAST.txt` files should be generated. If this is the case, congratulations, you installed OpenCAL properly.

Under Windows, the same steps have to be performed, by exploiting the CMake GUI application in case.

2.4 Compiling the examples

The examples can be easily compiled using CMake, all together or one at a time. To compile all the examples at once, follow the steps below (cf. Table 2.4 for available options and their default values):

```
user@machine:~$ cd ~/git/opencal-examples
user@machine:~$ mkdir build
user@machine:~$ cd build
user@machine:~$ cmake .. -DBUILD_OPENCAL_CL=ON -DBUILD_OPENCAL_GL=ON
```

Table 2.4: List of OpenCAL examples CMake options, alongside their default values and effects

CMake option	Default	Effect
`BUILD_OPENCAL_SERIAL`	ON	Build OpenCAL serial examples
`BUILD_OPENCAL_OMP`	ON	Build OpenCAL-OMP examples
`BUILD_OPENCAL_CL`	OFF	Build OpenCAL-CL examples
`BUILD_OPENCAL_GL`	OFF	Build OpenCAL-GL examples

Windows users have to refer the opencal-examples-1.0 solution and compile the examples within the Visual Studio environment.

To compile a particular example, please follow the steps described in Section 2.3.4.

2.5 Uninstall

If you want to uninstall OpenCAL, you can call `make` with the uninstall target, remove the installation direcory and, if installed, remove the FindOpenCAL.cmake module, as in the following example:

```
user@machine:-$ sudo make uninstall &&
                    rm -rf /usr/local/opencal-1.0 && \
                    rm /usr/share/cmake/Modules/FindOpenCAL.cmake
```

Under Visual Studio it is sufficient to compile the UNINSTALL project.

2.6 Web Page and Bug Reporting

The Web page for OpenCAL is at http://opencal.telesio.unical.it and contains up-to-date news and a list of bug reports. OpenCALGitHub homepage is at https://github.com/OpenCALTeam/opencal. For further information or bug reports contact mailto:opencal@telesio.unical.it or use the submit an issue at the following url https://github.com/OpenCALTeam/opencal/issues.

When reporting a bug, please include as much information and documentation as possible. Helpful information would include OpenCAL version, OpenMP/OpenCL implementation and version used, configuration options, type of computer system, problem description, and error message output.

Chapter 3

Cellular Automata

Cellular Automata (CA) are parallel computing models, whose evolution is ruled by local rules. They are largely employed in Science and Engineering for a wide range of problems, especially when classic methods (e.g., based on differential equations) can not be conveniently applied. In this Chapter, CA are briefly introduced, together with a recent extension of them, known as Extended Cellular Automata (XCA), which are widely used for the modeling of physical extended systems.

3.1 Informal Definition of Cellular Automata

A cellular automaton can be thought as a d-dimensional space, called *cellular space*, subdivided in regular cells of uniform shape and size. Each cell embeds a *finite automaton*, one of the most simple and well known computational models, which can assume a finite number of states. At time $t = 0$, cells are in arbitrary states and the CA evolves step by step by changing the states of the cells at discrete time steps, by applying the same local rule of evolution, i.e. the cell's *transition function*, simultaneously (i.e. in parallel) to each cell of the CA. Input for the cell is given by the states of a predefined (usually small) set of neighboring cells, which is assumed invariant in space and time. It is possible to identify an informal definition of cellular automaton by simply listing its main properties:

- It is formed by a d-dimensional space (i.e. the *cellular space*), partitioned into cells of uniform shape and size (e.g. triangles, squares, hexagons, cubes, etc. - see Figure 3.1);
- The number of cell's states is finite;
- The evolution occurs through discrete steps;
- Each cell changes state simultaneously to each other (i.e. they change state concurrently, in parallel) thanks to the application of the cell's *transition function*;
- The cell's state transition depends on the states of a set of neighboring cells;
- The evolving cell is called *central cell* and can belong to its neighborhood;

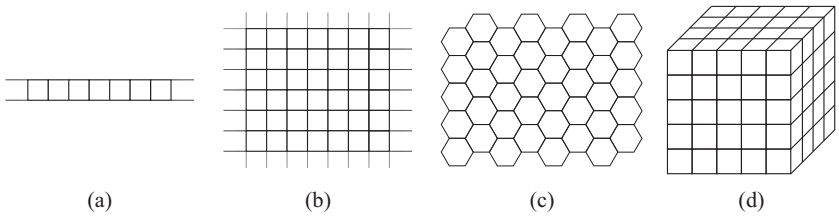


Figure 3.1: Example of cellular spaces: (a) one-dimensional, (b) two-dimensional with square cells, (c) two-dimensional with hexagonal cells, (d) three-dimensional with cubic cells.

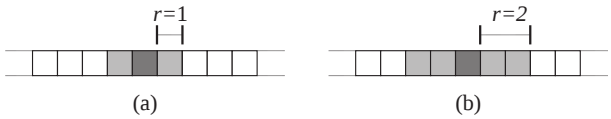


Figure 3.2: Example of neighborhood with radius (a) $r = 1$ and (b) $r = 2$ for one-dimensional Cellular Automata. Central cell is represented in dark gray, while adjacent cells are in light gray. Note that the central cell can optionally belong to its own neighborhood.

- The neighboring relationship is local, uniform and invariant over time (see Figures 3.2, 3.3, and 3.4 for examples of 1D, 2D, and 3D neighborhoods, respectively).

Despite their simple definition, CA can exhibit very interesting complex global behaviors. Moreover, from a computational point of view, they are equivalent to Turing Machines. This means, in principle, that everything that can be computed can also be by means of a cellular automaton (Church Turing thesis). Thanks to their *computational universality*, CA gained a great consideration among the Scientific Community and were employed for solving a great variety of different complex problems.

3.2 Formal Definition of Cellular Automata

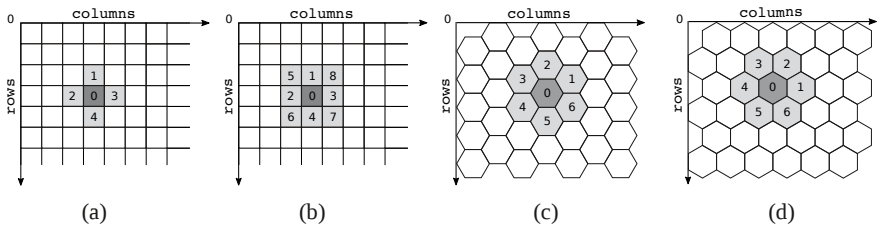
Cellular Automata are formally defined as a quadruple:

$$A =< R, X, Q, \sigma >$$

where:

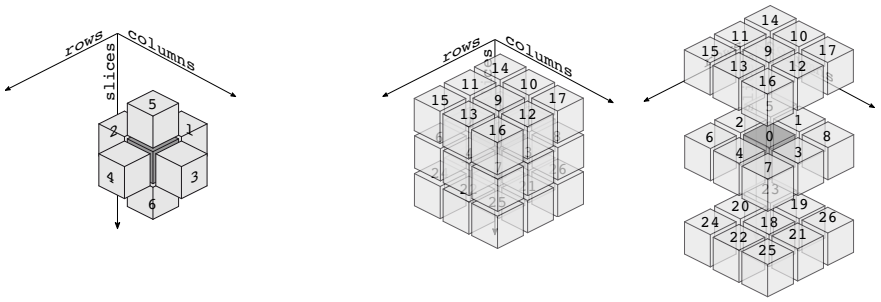
- $R = \{i = (i_0, i_1, ..., i_{d-1}) \mid i_k \in \mathbb{Z} \quad \forall k = 0, 1, ..., d-1\}$ is the set of points, with integer coordinates, which defines the d-dimensional cellular space;
- $X = \{\xi_0, \xi_1 ... \xi_{m-1}\}$ is the finite set of m d-dimensional vectors

$$\xi_j = \{\xi_{j_0}, \xi_{j_1}, ... \xi_{j_{d-1}}\}$$



(a) (b) (c) (d)

Figure 3.3: Examples of von Neumann (a) and Moore (b) neighborhoods for two-dimensional CA with square cells. Examples of Moore neighborhoods are also shown for hexagonal CA, both for the cases of horizontal (c) and vertical (d) orientations. Central cell is represented in dark gray, while adjacent cells are in light gray. A reference system is here considered to evaluate cells coordinates in terms of row and column indices in a matrix-style representation, and a 0-based numerical identifier assigned to each cell in the neighborhood for direct access.



(a) (b) (b')

Figure 3.4: Examples of von Neumann (a) and Moore (b, b′) neighborhoods for three-dimensional CA with cubic cells. Central cell is represented in dark gray, while adjacent cells are in light gray. A reference system is here considered to evaluate cells coordinates in terms of row, column and slice indices in a matrix-style representation, and a 0-based numerical identifier assigned to each cell in the neighborhood for direct access.

that define the set

$$N(X, i) = \{i + \xi_0, i + \xi_1, ..., i + \xi_{m-1}\}$$

of coordinates of cells belonging the the central cell's neighborhood (i.e. the cell with coordinates $i = (i_1, i_2, ...i_d)$). In other words, X represents the geometrical pattern that specifies the neighborhood relationship;

- Q is the finite set of states of the cell;
- $\sigma : Q^m \rightarrow Q$ is the cell's transition function.

3.3 Some Applications of Cellular Automata

CA are particularly suited to model and simulate classes of complex systems characterized by a large number of interacting elementary components. The assumption that if a system behavior is complex, the model that describes it must necessarily be of the same complexity is replaced by the idea that its behavior can be described, at least in some cases, in very simple terms.

Among different fields, fluid-dynamics is one of most important applications for CA and, in this research branch, many different CA-based methods can be found in the literature to simulate fluid flows. For instance, Lattice Gas Automata (LGA) were introduced for describing the motion and collision of *particles* on a grid and it was shown that such models can reproduce the main fluid dynamical properties. The continuum limit of these models leads to the Navier-Stokes equations. Lattice Gas models can be regarded as microscopic models, as they describe the motion of fluid particles which interact by scattering. An advantage of Lattice Gas models is that the simplicity of particles, and of their interactions, allow for the simulation of a large number of them, making it therefore possible to observe the emergence of flow patterns. Furthermore, since they are CA systems, it is possible to easily run simulations in parallel. A different approach to LGA is represented by Lattice Boltzmann models in which the state variables can take continuous values, as they are supposed to represent the density of fluid particles, endowed with certain properties, located in each cell (space and time are also discrete, as in lattice gas models). Both Lattice Gas and Lattice Boltzmann Models have been applied for the description of fluid turbulence.

Since many complex natural phenomena evolve on very large areas, they are therefore difficult to be modeled at a microscopic level of description. Among these, we can find some real flow-type phenomena like debris and lava flows, as well as floods and pyroclastic flows. Besides rheological complex behavior, they generally evolve on complex topographies, that can even change during the phenomenon evolution, and are often characterized by branching and rejoining of the flow. In order to better model such kind of phenomena, an extended notion of Cellular Automata (Extended Cellular Automata, described in the next Section), can represent a valid alternative to classical CA.

3.4 Extended Cellular Automata

As regards the modeling of natural complex phenomena, Prof. Gino Mirocle Crisci and co-workers from University of Calabria (Italy) proposed a method based on

an Extended notion of Cellular Automata (XCA), firstly applied to the simulation of basaltic lava flows in the 80's[1]. It was shown that the approach behind XCA can greatly make more straightforward the modeling of some complex systems. Compared to classical CA, XCA are different because of the following reasons:

- The cell's state is decomposed in *substates*, each of them representing the set of admissible values of a given characteristic assumed to be relevant for the modeled system and its evolution (e.g., lava temperature, lava thickness, etc, in the case of a lava flow model). The set of states for the cell is simply obtained as the Cartesian product of the considered substates.
- As the cell's state can be decomposed in substates, also the transition function can be split into *elementary processes*, each of them representing a particular aspect that rules the dynamic of the considered phenomenon. In turn, *elementary processes* can be split into *local interactions*, which refer to rules that deal with interactions among substates of the cell with neighbor ones (e.g., mass exchange with neighbors) and *internal transformations*, defined as the changes in the values of the substates due only to interactions among substates inside the cell (e.g. the solidification of the lava inside the cell due to the temperature drop).
- A set of *parameters*, commonly used to characterize the dynamic behaviors of the considered phenomenon, can be defined.
- Global operations can also be allowed (e.g. to model external influences that can not easily be described in terms of local interactions, or to perform reductions over the whole, or a subset of, the cellular space). They are often referred as *steering* operations.

3.4.1 Formal Definition of Extended Cellular Automata

Formally, a XCA is defined as a 7-tuple:

$$A =< R, X, Q, P, \sigma, \Gamma, \gamma >$$

where:

- R is the d-dimensional cellular space.
- X is the geometrical pattern that specifies the neighborhood relationship; $m = |X|$ represent the number of elements in the set X, i.e. the number of neighbors for the central cell.
- $Q = Q_0 \times Q_1 \times .... \times Q_{n-1}$ is the set of cell's states, expressed as Cartesian product of the n considered *substates* $Q_0 \times Q_1 \times .... \times Q_{n-1}$.
- $P = p_0, p_1, ..., p_{p-1}$ is the set of CA *parameters*.They can allow a fine tuning of the XCA model, with the purpose of reproducing different dynamical behaviors of the phenomenon of interest.

[1]XCA are also known as Complex Cellular Automata (CCA), Macroscopic Cellular Automata (MCA), and Multicomponent Cellular Automata (MCA)

- $\sigma : Q^m \rightarrow Q$ is the cell's transition function. It is split in *s elementary processes*, $\sigma_0, \sigma_1, ..., \sigma_{s-1}$, each one describing a particular aspect ruling the dynamic of the considered system.
- $\Gamma \subseteq R$ is the region over which steering is applied.
- $\gamma : Q^{|\Gamma|} \rightarrow Q^{|\Gamma|} \times \mathbb{R}$ is the (global) steering function.

In the next Chapters, some examples of XCA will be presented. Their implementations in OpenCAL will also be described, both in serial (Chapter 4) and in parallel (Chapters 5 and 6).

Chapter 4

OpenCAL

This Chapter introduces the serial implementation of OpenCAL by examples. After a brief overview about data types, main structure and adopted conventions, the implementation of a simple 2D cellular automaton is described. Subsequently, four different implementations of a more complex 2D example are illustrated, to show how simulation efficiency can progressively be improved. The implementation of a simple 3D model is also presented for the sake of completeness. The last part of the Chapter deals with OpenCAL-GL and shows how to integrate a basic OpenGL/GLUT visualization system in both 2D and 3D OpenCAL-based applications.

4.1 Statements conventions

The OpenCAL API adopts the following conventions:

- Derived data types are characterised by the `CAL` prefix, followed by a type identifier formed by one or more capitalised keywords, an optional suffix identifying the model dimension (`2D` or `3D`), and an eventual optional suffix specifying the basic scalar type (`b`, `i`, or `r`, for `CALbyte`, `CALint` and `CALreal` derived types, respectivey);
- Constants and enumerals are characterised by the `CAL_` prefix, followed by one or more uppercase keywords separated by the _ character (e.g. the `CAL_TRUE` and `CAL_FALSE` boolean-type values);
- Functions are characterised by the `cal` prefix, followed by at least one capitalized keyword, and end with a suffix specifying the model dimension (`2D` or `3D`) and the basic datatype (`b`, `i`, or `r`, for `CALbyte`, `CALint` and `CALreal` derived types, respectively).

Moreover, the `{arg1|arg2|...|argn}` and `[arg1|arg2|...|argn]` conventions are adopted in the following. The first one identifies a list of n mutually exclusive arguments, where one of the arguments is needed, while the second a list of n non-mutually exclusive optional arguments.

OpenCAL basic type	Basic type alias	C type	Substate type
`CALbyte`	`CALParameterb`	`char`	`CALSubstate{2D|3D}b`
`CALint`	`CALParameteri`	`int`	`CALSubstate{2D|3D}i`
`CALreal`	`CALParameterr`	`double`	`CALSubstate{2D|3D}r`

Table 4.1: OpenCAL basic scalar and substate types.

OpenCAL substate type	Meaning
`CALModel{2D|3D}`	CA data type
`CALRun2D{2D|3D}`	Simulation data type

Table 4.2: OpenCAL types for CA and simulation objects.

4.2 Basic data types and main loop structure

The current version of OpenCAL provides support for three different scalar data types, namely `CALbyte`, `CALint`, and `CALreal`, which redefine the `char`, `int` and `double` C native scalar types, respectively. For each of them, a corresponding substate type is provided, namely `CALSubstate{2D|3D}{b|i|r}`, as well as more complex objects for model and simulation definitions, namely `CALModel{2D|3D}` and `CALRun2D{2D|3D}`, respectively. Supported scalar types and substates are listed in Table 4.1, while Table 4.2 lists model and simulation data types.

The model object essetially allows to define the XCA dimension, the neighbourhood (which can be predefined or custom - for maximum flexibility), the cellular space boundary behaviour and if the active cell optimization has to be employed.

Substates and elementary processes must be registered to the model object to be transparently processed. For this purpose, the `calAddSubstate{2D|3D}{b|i|r}()` and `calAddElementaryProcess{2D|3D}{b|i|r}()` API functions can be adopted. Specifically, substates are composed by two computational layes, namely the *current* and *next* ones. Both of them are implemented by means of linearised arrays, even if they represent 2D and 3D domain data. Nevertheless, a transparent multi indices-based access is guarantied by specific API functions. If unsafe operations are not considered (see below in this Chapter), the current layer is used as a read-only memory, while the next one for updating the velues for the central cell, by guarantying the implicit parallelism. Single-layer substates are also implemented, whcih have the current layer only. This latter can be used for internal transformation processing and, obviusly, do not need to be updated. Instead, elementary processes are defined by means of callback functions. Theese latter must return void and take a pointer to the model object as first parameter, followed by a list of integer parameters representing the coordinates of a generic cell of the cellular space.

The simulation object determines of the system evolution. It essentially allows to define how many steps have to be computed, and the update scheme. This latter, in particular, can be implicit or explicit. In the first case, OpenCAL applies the elementary processes in the same order in which they have been registered to the model object and, after the application of each of them, updates all the registered substates (even if they were left unchanged). In the other case, the model transition function must be overridden, the elemantary processes explicitly applied and the substates explicitly updated, allowing for improving both flexibility and performace. In fact, the elementary processes application order can be changed with respect to the

one chosen at the registration stage, and unneeded substates update can be avoided. Figure 4.1 shows the OpenCAL main implicit simulation loop. Before entering the loop, if defined, the init function is executed once and a substates update performed. Afterwards, while the current step is lower or equal to the final step of computation (or this latter is set to `CAL_RUN_LOOP`, which defines an infinite loop), elementary processes are executed once at a time, in the order they have been registered to the model object, and substates updated after the application of each them. Moreover, just before the end of the computational step, the steering function (if defined) is applied and substates updated again. At the end of the computational step, a stop condition is checked, which can stop the simulation even before the last step is reached.

4.3 Statements conventions

In OpenCAL, besides the three basic supported scalar data types, namely `CALbyte`, `CALint`, and `CALreal`, which redefine the `char`, `int` and `double` C native scalar data types, respectively, the API adopts the following conventions:

- Derived data types are characterised by the `CAL` prefix, followed by a type identifier formed by one or more capitalised keywords, an optional suffix identifying the model dimension (`2D` or `3D`), and an eventual optional suffix specifying the basic scalar type (`b`, `i`, or `r`, for `CALbyte`, `CALint` and `CALreal` derived types, respectivey);
- Constants and enumerals are characterised by the `CAL_` prefix, followed by one or more uppercase keywords separated by the _ character (e.g. the `CAL_TRUE` and `CAL_FALSE` boolean-type values);
- Functions are characterised by the `cal` prefix, followed by at least one capitalized keyword, and end with a suffix specifying the model dimension (`2D` or `3D`) and the basic datatype (`b`, `i`, or `r`, for `CALbyte`, `CALint` and `CALreal` derived types, respectively).

Moreover, the `{arg1|arg2|...|argn}` and `[arg1|arg2|...|argn]` conventions are adopted in the following. The first one identifies a list of n mutually exclusive arguments, where one of the arguments is needed, while the second a list of n non-mutually exclusive optional arguments.

4.4 Conway's Game of Life

In order to introduce OpenCAL, this section starts by implementing the Conway's Game of Life, one of the most simple, yet powerful examples of CA, devised by mathematician John Horton Conway in 1970.

The Game of Life can be thought as an infinite two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, *dead* or *alive*. Every cell interacts with the eight adjacent neighbors belonging to the Moore neighborhood. At each time step, one of the following transitions occur:

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.

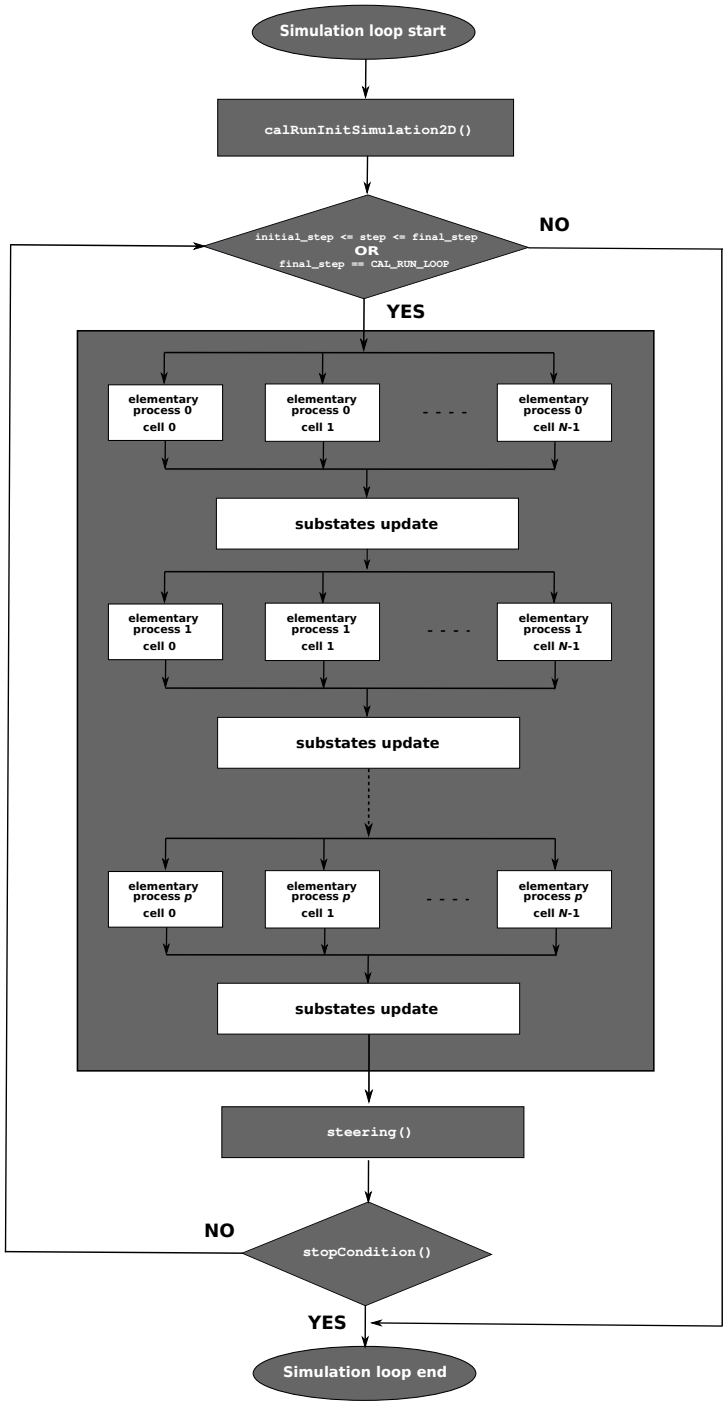


Figure 4.1: OpenCAL main loop chart.

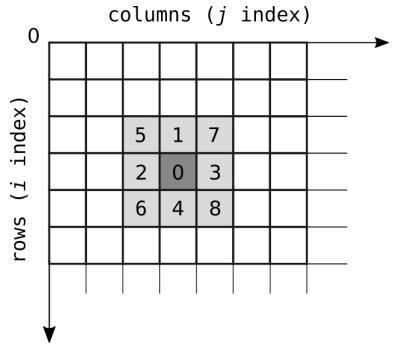


Figure 4.2: OpenCAL 2D cellular space and Moore neighborhood. Cells are individuated by a couple of matrix-style integer coordinates (i, j), where i represents the row and j the column. Cell (0,0) is the one located at the upper-left corner. Moore neighborhood is represented in gray, with the central cell highlighted in dark gray. Neighboring cells can also be indexed by the integer subscripts shown within the cells. Cells indices are implicitly assigned by OpenCAL, both in the case of predefined and custom neighborhoods. In the first case, indices can not be modified, while in the second case, indices are assigned progressively in an automatic way each time a new neighbor is added to the CA by means of the `calAddNeighbor2D()` function.

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 in the cellular space. The evolution of the system is thus obtained by applying the above rules (which define the cell transition function) simultaneously to every cell in the cellular space, so that each new configuration is function of the one at the previous step. The rules continue to be applied repeatedly to create further generations. For more details on the Game of Life, please check Wikipedia at the URL http://en.wikipedia.org/wiki/Conway's_Game_of_Life.

The formal definition of the Life CA is reported below.

$$Life =< R, X, Q, \sigma >$$

where:

- R is the set of points, with integer coordinates, which defines the 2-dimensional cellular space. The generic cell in R is individuated by means of a couple of integer coordinates (i, j), where $0 \leq i < i_{max}$ and $0 \leq j < j_{max}$. The first coordinate, i, represents the row, while the second, j, the column. The cell at coordinates $(0, 0)$ is located at the top-left corner of the computational grid (cf. Figure 4.2).

- $X = \{(0,0), (-1,0), (0,-1), (0,1), (1,0), (-1,-1), (1,-1), (1,1), (-1,1)\}$ is the Moore neighborhood relation, a geometrical pattern which identifies the cells influ-

encing the state transition of the central cell. The neighborhood coordinates of the generic cell of coordinate (i, j) is given by

$$N(X, (i, j)) =$$

$$= \{(i, j) + (0, 0), (i, j) + (-1, 0), \ldots, (i, j) + (-1, 1)\} =$$

$$= \{(i, j), (i - 1, j), \ldots, (i - 1, j + 1)\}$$

Here, a subscript operator can be used to index cells belonging to the neighborhood. Let $|X|$ be the number of elements in X, and $n \in \mathbb{N}$, $0 \leq n < |X|$; the notation

$$N(X, (i, j), n)$$

represents the coordinates of the n^{th} neighborhood of the cell (i, j). Thereby, $N(X, (i, j), 0) = (i, j)$, i.e. the central cell, $N(X, (i, j), 1) = (i - 1, j)$, i.e. the first neighbor, and so on (cf. Figure 4.2).

- $Q = \{0, 1\}$ is the set of cell states.
- $\sigma : Q^9 \rightarrow Q$ is the deterministic cell transition function. It is composed by one elementary process, which implements the aforementioned transition rules.

The program in Listing 4.1 provides a complete OpenCAL-based implementation of Game of Life in just 60 lines of code, by defining both the CA model and the simulation object, needed to let the CA evolve step by step.

Header files at lines 3-5 allow to use the serial implementation of OpenCAL. Specifically, the `cal2D.h` header allows to define the model and substates objects, as well as to define the elementary processes composig the transition function. Instead, the `cal2DRun.h` header allows to define the simulation object. Eventually, the `cal2DIO.h` provides some basic input/output functions for reading/writing substates from/to file. The model object is then declared at line 9, while lines 10 and 11 declare a substate and a simulation object, respectively.

Objects declared at lines 9-11 are defined later in the `main` function. In particular, the *Life* CA object is defined at line 29 by the `calCADef2D()` function. The first 2 parameters define the CA dimensions (the number of rows and columns, respectively), while the third parameter defines the neighborhood pattern. Here, the predefined Moore neighborhood is selected (cf. Figure 4.2), among those provided by OpenCAL. See Listings 4.4 and 4.5 for a list of OpenCAL predefined 2D and 3D neighborhoods, respectively. Custom neighborhoods can also be defined by means of the `calAddNeighbor2D()` function. In both cases, 0-based indices are progressively assigned each time a new cell is added to the neighborhood. The fourth parameter specifies the boundary conditions. In this case, the CA cellular space is considered as a torus, with cyclic conditions at boundaries. The last parameter allows to specify if the model has to use the so called *active cells optimization*, that permits to restrict the computation to only *non-stationary cells*. In this case, no optimization is considered. The complete definition of `calCADef2D()` is provided in Listing 4.2. The `calCADef3D()` 3D CA definition function is also shown in Listing 4.2 for the sake of completeness. The `CALNeighborhood2D` enum type (Listing 4.4) allows to select a predefined or a custom neighborhood to be defined in the application. In particular, `CAL_VON_NEUMANN_NEIGHBORHOOD_2D` corresponds to the von Neumann pattern,

```
1  // Conway's game of Life Cellular Automaton
2
3  #include <OpenCAL/cal2D.h>
4  #include <OpenCAL/cal2DIO.h>
5  #include <OpenCAL/cal2DRun.h>
6  #include <stdlib.h>
7
8  // declare CA, substate and simulation objects
9  struct CALModel2D* life;
10 struct CALSubstate2Di* Q;
11 struct CALRun2D* life_simulation;
12
13 // The cell's transition function
14 void lifeTransitionFunction(struct CALModel2D* life, int i, int j)
15 {
16   int sum = 0, n;
17   for (n=1; n<life->sizeof_X; n++)
18     sum += calGetX2Di(life, Q, i, j, n);
19
20   if ((sum == 3) || (sum == 2 && calGet2Di(life, Q, i, j) == 1))
21     calSet2Di(life, Q, i, j, 1);
22   else
23     calSet2Di(life, Q, i, j, 0);
24 }
25
26 int main()
27 {
28   // define of the life CA and life_simulation simulation objects
29   life = calCADef2D(8, 16, CAL_MOORE_NEIGHBORHOOD_2D, CAL_SPACE_TOROIDAL,
         CAL_NO_OPT);
30   life_simulation = calRunDef2D(life, 1, 1, CAL_UPDATE_IMPLICIT);
31
32   // add the Q substate to the life CA
33   Q = calAddSubstate2Di(life);
34
35   // add transition function's elementary process
36   calAddElementaryProcess2D(life, lifeTransitionFunction);
37
38   // set the whole substate to 0
39   calInitSubstate2Di(life, Q, 0);
40
41   // set a glider
42   calInit2Di(life, Q, 0, 2, 1);
43   calInit2Di(life, Q, 1, 0, 1);
44   calInit2Di(life, Q, 1, 2, 1);
45   calInit2Di(life, Q, 2, 1, 1);
46   calInit2Di(life, Q, 2, 2, 1);
47
48   // save the Q substate to file
49   calSaveSubstate2Di(life, Q, "./life_0000.txt");
50
51   // simulation run
52   calRun2D(life_simulation);
53
54   // save the Q substate to file
55   calSaveSubstate2Di(life, Q, "./life_LAST.txt");
56
57   // finalize simulation and CA objects
58   calRunFinalize2D(life_simulation);
59   calFinalize2D(life);
60
61   return 0;
62 }
```

Listing 4.1: An OpenCAL implementation of the Conway's Game of Life.

```
struct CALModel2D* calCADef2D (
  int rows,
  int columns,
  enum CALNeighborhood2D CAL_NEIGHBORHOOD_2D,
  enum CALSpaceBoundaryCondition CAL_TOROIDALITY,
  enum CALOptimization CAL_OPTIMIZATION
)
```

Listing 4.2: Definition of the calCADef2D() function.

```
struct CALModel3D* calCADef3D(
  int rows,
  int columns,
  int slices,
  enum CALNeighborhood3D CAL_NEIGHBORHOOD_3D,
  enum CALSpaceBoundaryCondition CAL_TOROIDALITY,
  enum CALOptimization CAL_OPTIMIZATION
);
```

Listing 4.3: Definition of the calCADef3D() function.

`CAL_MOORE_NEIGHBORHOOD_2D` to the Moore one, `CAL_HEXAGONAL_NEIGHBORHOOD_2D` and `CAL_HEXAGONAL_NEIGHBORHOOD_ALT_2D` to the hexagonal and alternative hexagonal patterns, respectively (cf. Figure 3.3, Section 3.1). As regards 3D neighborhoods patterns, they are defined by means of the `CALNeighborhood3D` enum type (Listing 4.5). Here, we can find the 3D equivalent versions of the von Neumann and Moore neighborhoods, while hexagonal neighborhoods are (obviously) not defined. Custom neighborhoods will be discussed later in Section 4.5. Similarly, the `CALSpaceBoundaryCondition` enum type (Listing 4.6) allows to set cyclic condition at boundaries. Eventually, the `CALOptimization` enum type (Listing 4.7) allows to consider the *active cells optimization*, discussed later in this Chapter.

The CA simulation object is defined at line 30 by the `calRunDef2D()` function. The first parameter is a pointer to a CA object (`life` in our case), while the second and third parameters specify the initial and last simulation step, respectively. In this case, we just perform one step of computation, being both the first and last step set to 1. The last parameter allows to specify the substate update policy, which can be implicit or explicit. The `CALUpdateMode` enumeration, shown in Listing 4.10, defines possible update policies. In the first case, OpenCAL performs substates update automatically, while in the second the user must explicitly apply the elementary processes composig the transition function and update the involved substates. The complete definition of `calRunDef2D()` is provided in Listing 4.8, while the 3D version of the same function can be found in Listing 4.9. The `CALUpdateMode` type, shown

```
enum CALNeighborhood2D {
  CAL_CUSTOM_NEIGHBORHOOD_2D,
  CAL_VON_NEUMANN_NEIGHBORHOOD_2D,
  CAL_MOORE_NEIGHBORHOOD_2D,
  CAL_HEXAGONAL_NEIGHBORHOOD_2D,
  CAL_HEXAGONAL_NEIGHBORHOOD_ALT_2D
};
```

Listing 4.4: The CALNeighborhood2D enum type.

```
  enum CALNeighborhood3D {
    CAL_CUSTOM_NEIGHBORHOOD_3D,
    CAL_VON_NEUMANN_NEIGHBORHOOD_3D,
    CAL_MOORE_NEIGHBORHOOD_3D
};
```

Listing 4.5: The CALNeighborhood3D enum type.

```
  enum CALSpaceBoundaryCondition{
    CAL_SPACE_FLAT = 0,
    CAL_SPACE_TOROIDAL
  };
```

Listing 4.6: The CALSpaceBoundaryCondition enum type.

```
  enum CALOptimization{
    CAL_NO_OPT = 0,
    CAL_OPT_ACTIVE_CELLS
  };
```

Listing 4.7: The CALOptimization enum type.

```
  struct CALRun2D* calRunDef2D (
    struct CALModel2D* ca2D,
    int initial_step,
    int final_step,
    enum CALUpdateMode UPDATE_MODE
  )
```

Listing 4.8: Definition of the calRunDef2D() function.

```
  struct CALRun3D* calRunDef3D(
    struct CALModel3D* ca3D,
    int initial_step,
    int final_step,
    enum CALUpdateMode UPDATE_MODE
  );
```

Listing 4.9: Definition of the calRunDef3D() function.

```
  enum CALUpdateMode {
    CAL_UPDATE_EXPLICIT = 0,
    CAL_UPDATE_IMPLICIT
  };
```

Listing 4.10: The CALUpdateMode enum type.

in Listing 4.10, enumerates possible update policies.

Line 33 allocates memory and registers the `Q` substate to the `life` CA, while line 36 registers an elementary process to the transition function. The `calAddSubstate2Di()` function is self-explanatory, while `calAddElementaryProcess2D()` must be discussed more in detail. In particular, it takes the handle to the CA model to which the elementary process must be registered and a pointer to a callback function, that defines the elementary process itself. In our example, we specified `lifeTransitionFunction` as second parameter, being it the name of a developer-defined function implementing the transition function roles (cf. lines 14-24). As can be seen, the elementary process callback function returns `void`. Moreover, it takes a pointer to a CA object as first parameter, followed by a couple of integers, representing the coordinates of the generic cell in the CA space. This is the function prototype which is common to each elementary process.

Note that, for each computational step, each elementary process is applied simultaneously to each cell. This characteristic is known as *implicit parallelism* and is obtained in OpenCAL by considering two different working planes, namely *current* and *next*, for each registered substate. The *current* plane is used to read cells state at the current CA step, while the *next* to store updated values. In this manner, the *current* plane remains unchanged for the overall current CA step. As a consequence, even in the case of serial computation, in which cells are updated once at a time, the resulting effect is that all the cells are updated simultaneously on the basis of their sates at the current computational step. At the contrary, if updated states would be stored in the `current` computing plane, such updated values would affect the state change of neighboring cells even in the current computational step, and the computation would result actually serial. When all the cells have been processed and their states updated, computing planes are switched (i.e. *next* becomes the new *current* and *current* the new *next*), and the process is reiterated [1]. Note that, besides specific cases, working planes are completely transparent to the user.

When the user implements an elementary process, by defining its callback function, a set of OpenCAL functions can be used to retrieve the substates values for both the central and the neighboring cells, and to update the central cell state. In the specific case of the Game of Life, the `calGet2Di()` function gets the central cell value of the substate `Q` (note that the central cell is identified by the coordinates (i, j), coming from the parameters of the callback function), the `calGetX2Di()` function gets the value of the `Q` substate for the n-th neighbor, and the `calSet2Di()` function updates the value of the `Q` substate for the central cell. In the Game of Life example, we defined just one elementary process, that therefore represents the whole cell transition function. However, as we will see later, many elementary processes can be defined in OpenCAL by simply calling the `calAddElementaryProcess{2D|3D}()` function many times. If the user defines more than one elementary process, these will be applied in the same order they were registered to the CA.

The `calInitSubstate2Di()` function at line 39 sets the whole `Q` substate to the value 0 (for both the current and next working planes), i.e. the value of the `Q` substate is set to 0 in each cell of the cellular space. Lines 42-46, set the value of the `Q` substate for some cells to 1, in order to define a so called *glider* pattern. In this case, the `calInit2Di()` function, used for this purpose, takes the cells coordinates as the third and fourth parameters, while the value 1 to be set was specified through the

[1] The *implicit parallelism* is also used in the parallel versions of OpenCAL, with the difference that more than one cell can be processed and updated concurrently by exploiting more than one processing unit.

```
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Figure 4.3: Initial configuration of Game of Life, as implemented in Listing 4.1.

```
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Figure 4.4: Final configuration of Game of Life (actually, just one step of computation), as implemented in Listing 4.1.

last parameter. In this way, the initial condition of the system was defined within the `main` function, even if, as we will see later in this Chapter, OpenCAL allows for registering an initialization function, to be executed once before the simulation loop.

The `calSaveSubstate2Di()` function (line 49) saves the `Q` substate to file, while the `calRun2D()` function (line 52) enters the simulation loop (actually, only one computational step in this example), and returns to the `main` function when the simulation is complete. The `calSaveSubstate2Di()` is called again at line 55 to save the new (last) configuration of the CA (represented by the only defined `Q` substate) to file, while the last two functions at lines 58 and 59 release memory previously automatically allocated by OpenCAL for CA, substates (actually, only `Q` in this case) and simulation objects. The `return` statement at line 61 ends the program.

Figures 4.3 and 4.4 show the initial and final configuration of Game of Life, respectively, as implemented in Listing 4.1.

4.5 Custom Neighborhoods

In the Game of Life example, we used the predefined Moore neighborhood. As already stated, OpenCAL also provides other 2D and 3D predefined neighborhoods (cf. Listings 4.4 and 4.5). Furthermore, it allows for the definition of custom neighborhood patterns.

In order to define a custom neighborhood, `CAL_CUSTOM_NEIGHBORHOOD_{2D|3D}` must be used as the third parameter of the `calCADef{2D|3D}()` function. By doing this, you will start with an empty neighboring pattern and can subsequently call the `calAddNeighbor{2D|3D}()` function to add a cell to the pattern (cf. Listings 4.11 and 4.12).

Listing 4.13 shows an example of how a custom neighborhood pattern can be built for the Game of Life CA described above. In particular, the Moore neighborhood is built by using a sequence of nine calls to the `calAddNeighbor2D()` function. The first

```
struct CALCell2D*  calAddNeighbor2D(
    struct CALModel2D* ca2D,          // pointer to a 2D CA object
    int i,                            // row coordinate
    int j                             // column coordinate
);
```

Listing 4.11: The calAddNeighbor2D() functions to define custom neighborhood patterns in 2D CA.

```
struct CALCell3D*  calAddNeighbor3D(
    struct CALModel3D* ca3D,          // pointer to a 3D CA object
    int i,                            // row coordinate
    int j,                            // column coordinate
    int k                             // slice coordinate
);
```

Listing 4.12: The calAddNeighbor3D() function to define custom neighborhood patterns in 3D CA.

```
// ...
int main ()
{
  // define of the life CA and life_simulation simulation objects
  life = calCADef2D (8, 16, CAL_MOORE_NEIGHBORHOOD_2D,
       CAL_CUSTOM_NEIGHBORHOOD_2D , CAL_NO_OPT );
  //...

  //add neighbors of the Moore neighborhood
  calAddNeighbor2D(life,   0,   0); // neighbor 0 (central cell)
  calAddNeighbor2D(life, - 1,   0); // neighbor 1
  calAddNeighbor2D(life,   0, - 1); // neighbor 2
  calAddNeighbor2D(life,   0, + 1); // neighbor 3
  calAddNeighbor2D(life, + 1,   0); // neighbor 4
  calAddNeighbor2D(life, - 1, - 1); // neighbor 5
  calAddNeighbor2D(life, + 1, - 1); // neighbor 6
  calAddNeighbor2D(life, + 1, + 1); // neighbor 7
  calAddNeighbor2D(life, - 1, + 1); // neighbor 8

  //...
}
```

Listing 4.13: Example of custom neighbourhood pattern; the sequence of calls to the calAddNeighbor2D() function defines the Moore neighbourhood for the Game of Life CA.

time the function is called, it adds the relative coordinates (0,0) to the neighboring pattern. This first set of coordinates receives the subscript identifier 0 and therefore can be used later to refer the central cell. For instance, if `calSet2Di(life,Q,i,j,0)` is called, as at line 23 of Listing 4.1, the relative coordinates of the neighbor 0 (specified as last parameters), i.e. (0,0), are added to the coordinates of the cells the elementary process is applying to, i.e. (i,j) (cf. second and third to last parameters), by obtaining the cell (i,j) itself, which corresponds to the central cell by definition. The subsequent calls to `calAddNeighbor2D()` add further couples of coordinates to the neighboring pattern, by progressively assigning an integer subscript handle to each of them.

Eventually, note that it is possible to define custom neighborhoods starting from a predefined one. For instance, by considering the above Game of Life example, it is possible to specify the value `CAL_MOORE_NEIGHBORHOOD_2D` as parameter for the `calCADef2D` function and then add further neighbors by calling `calAddNeighbor2D()` as many times the as the number of cells have to be added.

4.6 SciddicaT

In Section 4.4 we illustrated an OpenCAL implementation of a simple cellular automaton, namely the Conways Game of Life. Here, we will deal with a more complex example, concerning the implementations of one of the simplest versions of the Sciddica landslide simulation models, namely SciddicaT. Different versions of SciddicaT will be considered, ranging from a naive to a fully optimized implementation.

4.6.1 SciddicaT naive implementation

The first, naive, version of SciddicaT here considered is formally defined as:

$$SciddicaT_{naive} =< R, X, Q, P, \sigma >$$

where:

- R is the set of points, with integer coordinates, which defines the 2-dimensional cellular space over which the phenomenon evolves. The generic cell in R is individuated by means of a couple of integer coordinates (i, j), where $0 \leq i < i_{max}$ and $0 \leq j < j_{max}$. The first coordinate, i, represents the row, while the second, j, the column. The cell at coordinates $(0, 0)$ is located at the top-left corner of the computational grid.

- $X = \{(0, 0), (-1, 0), (0, -1), (0, 1), (1, 0)\}$ is the von Neumann neighborhood relation (cf. Figure 3.3), a geometrical pattern which identifies the cells influencing the state transition of the central cell. The neighborhood of the generic cell of coordinate (i, j) is given by

$$N(X, (i, j)) =$$
$$= \{(i, j) + (0, 0), (i, j) + (-1, 0), (i, j) + (0, -1), (i, j) + (0, 1), (i, j) + (1, 0)\} =$$
$$= \{(i, j), (i - 1, j), (i, j - 1), (i, j + 1), (i + 1, j)\}$$

  Here, a subscript operator can be used to index cells belonging to the neighborhood. Let $|X|$ be the number of elements in X, and $n \in \mathbb{N}$, $0 \leq n < |X|$; the notation

$$N(X, (i, j), n)$$

represents the n^{th} neighborhood of the cell (i, j). Thereby, $N(X, (i, j), 0) = (i, j)$, i.e. the central cell, $N(X, (i, j), 1) = (i - 1, j)$, i.e. the first neighbor, and so on.

- Q is the set of cell states. It is subdivided in the following substates:
  - Q_z is the set of values representing the topographic altitude (i.e. elevation);
  - Q_h is the set of values representing the debris thickness;
  - Q_o^4 are the sets of values representing the debris outflows from the central cell to the neighboring ones.

  The Cartesian product of the substates defines the overall set of states Q:

  $$Q = Q_z \times Q_h \times Q_o^4$$

  so that the cell state is specified by the following sextuplet:

  $$q = (q_z, q_h, q_{o_0}, q_{o_1}, q_{o_2}, q_{o_3})$$

  In particular, q_{o_0} represents the outflows from the central cell towards the neighbor 1, q_{o_1} the outflow towards the neighbor 2, and so on.

- P is set of parameters ruling the CA dynamics:
  - p_ϵ is the parameter which specifies the thickness of the debris that cannot leave the cell due to the effect of adherence;
  - p_r is the relaxation rate parameter, which affects the size of outflows (cf. section above).

- $\sigma : Q^5 \to Q$ is the deterministic cell transition function. It is composed by two elementary processes, listed below in the same order they are applied:
  - $\sigma_1 : (Q_z \times Q_h)^5 \times p_\epsilon \times p_r \to Q_o^4$ determines the outflows from the central cell to the neighboring ones by applying the *minimization algorithm of the differences*. In brief, a preliminary control avoids outflows computation for those cells in which the amount of debris is smaller or equal to p_ϵ, acting as a simplification of the adherence effect. Thus, by means of the minimization algorithm, outflows $q_o(0, m)$ $(m = 0, \ldots, 3)$ from the central cell towards its four adjacent cells are evaluated, and the Q_o^4 substates accordingly updated. Note that, $q_o(0, 0)$ represents the outflow from the central cell towards the neighbor 1, $q_o(0, 1)$ the outflow towards the neighbor 2, and so on. In general, $q_o(0, m)$ represents the outflows from the central cell towards the $n = (m + 1)^{th}$ neighboring cell. Eventually, a relaxation rate factor, $p_r \in]0, 1]$, is considered in order to obtain the local equilibrium condition in more than one CA step. This can significantly improve the realism of model as, in general, more than one step may be needed to displace the proper amount of debris from a cell towards the adjacent ones. In this case, if $f(0, m)$ $(i = 0, \ldots, 3)$ represent the outgoing

flows towards the 4 adjacent cells, as computed by the minimization algorithm, the resulting outflows are given by $q_o(0,m) = f(0,m) \cdot p_r$ $(i = 0,\ldots,3)$.

– $\sigma_2 : Q_h \times (Q_o^4)^4 \rightarrow Q_h$ determines the value of debris thickness inside the cell by considering mass exchange in the cell neighborhood: $h'(0) = h(0) + \sum_{m=0}^{3}(q_o(0,m) - q_o(m,0))$. Here, $h'(0)$ is the new debris thickness inside the cell, while $q_o(m,0)$ represents the inflow from the $n = (m+1)^{th}$ neighboring cell. No parameters are involved in this elementary process.

In the following Listing 4.14, an OpenCAL implementation of SciddicaT is shown.

```
1  // The SciddicaT debris flows XCA simulation model
2
3  #include <OpenCAL/cal2D.h>
4  #include <OpenCAL/cal2DIO.h>
5  #include <OpenCAL/cal2DRun.h>
6  #include <stdlib.h>
7  #include <time.h>
8
9  // Some definitions...
10 #define ROWS 610
11 #define COLS 496
12 #define P_R 0.5
13 #define P_EPSILON 0.001
14 #define STEPS 4000
15 #define DEM_PATH "./data/dem.txt"
16 #define SOURCE_PATH "./data/source.txt"
17 #define OUTPUT_PATH "./data/width_final.txt"
18 #define NUMBER_OF_OUTFLOWS 4
19
20 // Declare XCA model (sciddicaT), substates (Q), parameters (P),
21 // and simulation object (sciddicaT_simulation)
22 struct CALModel2D* sciddicaT;
23
24 struct sciddicaTSubstates {
25   struct CALSubstate2Dr *z;
26   struct CALSubstate2Dr *h;
27   struct CALSubstate2Dr *f[NUMBER_OF_OUTFLOWS];
28 } Q;
29
30 struct sciddicaTParameters {
31   CALParameterr epsilon;
32   CALParameterr r;
33 } P;
34
35 struct CALRun2D* sciddicaT_simulation;
36
37 // The sigma_1 elementary process
38 void sciddicaTFlowsComputation(struct CALModel2D* sciddicaT, int i, int j)
39 {
40   CALbyte eliminated_cells[5]={CAL_FALSE,CAL_FALSE,CAL_FALSE,CAL_FALSE,
         CAL_FALSE};
41   CALbyte again;
42   CALint cells_count;
43   CALreal average;
44   CALreal m;
45   CALreal u[5];
46   CALint n;
47   CALreal z, h;
48
49   if (calGet2Dr(sciddicaT, Q.h, i, j) <= P.epsilon)
50     return;
51
```

```
52    m = calGet2Dr(sciddicaT, Q.h, i, j) - P.epsilon;
53    u[0] = calGet2Dr(sciddicaT, Q.z, i, j) + P.epsilon;
54    for (n=1; n<sciddicaT->sizeof_X; n++)
55    {
56      z = calGetX2Dr(sciddicaT, Q.z, i, j, n);
57      h = calGetX2Dr(sciddicaT, Q.h, i, j, n);
58      u[n] = z + h;
59    }
60
61    //computes outflows
62    do{
63      again = CAL_FALSE;
64      average = m;
65      cells_count = 0;
66
67      for (n=0; n<sciddicaT->sizeof_X; n++)
68        if (!eliminated_cells[n]){
69          average += u[n];
70          cells_count++;
71        }
72
73        if (cells_count != 0)
74          average /= cells_count;
75
76        for (n=0; n<sciddicaT->sizeof_X; n++)
77          if( (average<=u[n]) && (!eliminated_cells[n]) ){
78            eliminated_cells[n]=CAL_TRUE;
79            again=CAL_TRUE;
80          }
81    }while (again);
82
83    for (n=1; n<sciddicaT->sizeof_X; n++)
84      if (eliminated_cells[n])
85        calSet2Dr(sciddicaT, Q.f[n-1], i, j, 0.0);
86      else
87        calSet2Dr(sciddicaT, Q.f[n-1], i, j, (average-u[n])*P.r);
88  }
89
90  // The sigma_2 elementary process
91  void sciddicaTWidthUpdate(struct CALModel2D* sciddicaT, int i, int j)
92  {
93    CALreal h_next;
94    CALint n;
95
96    h_next = calGet2Dr(sciddicaT, Q.h, i, j);
97    for(n=1; n<sciddicaT->sizeof_X; n++)
98      h_next +=  calGetX2Dr(sciddicaT, Q.f[NUMBER_OF_OUTFLOWS - n], i, j, n) -
           calGet2Dr(sciddicaT, Q.f[n-1], i, j);
99
100   calSet2Dr(sciddicaT, Q.h, i, j, h_next);
101 }
102
103 // SciddicaT simulation init function
104 void sciddicaTSimulationInit(struct CALModel2D* sciddicaT)
105 {
106   CALreal z, h;
107   CALint i, j;
108
109   //initializing substates to 0
110   calInitSubstate2Dr(sciddicaT, Q.f[0], 0);
111   calInitSubstate2Dr(sciddicaT, Q.f[1], 0);
112   calInitSubstate2Dr(sciddicaT, Q.f[2], 0);
113   calInitSubstate2Dr(sciddicaT, Q.f[3], 0);
114
115   //sciddicaT parameters setting
116   P.r = P_R;
117   P.epsilon = P_EPSILON;
118
```

```
119   //sciddicaT source initialization
120   for (i=0; i<sciddicaT->rows; i++)
121     for (j=0; j<sciddicaT->columns; j++)
122     {
123       h = calGet2Dr(sciddicaT, Q.h, i, j);
124
125       if ( h > 0.0 ) {
126         z = calGet2Dr(sciddicaT, Q.z, i, j);
127         calSet2Dr(sciddicaT, Q.z, i, j, z-h);
128       }
129     }
130 }
131
132 // SciddicaT steering function
133 void sciddicaTSteering(struct CALModel2D* sciddicaT)
134 {
135   // set flow to 0 everywhere
136   calInitSubstate2Dr(sciddicaT, Q.f[0], 0);
137   calInitSubstate2Dr(sciddicaT, Q.f[1], 0);
138   calInitSubstate2Dr(sciddicaT, Q.f[2], 0);
139   calInitSubstate2Dr(sciddicaT, Q.f[3], 0);
140 }
141
142 int main()
143 {
144   time_t start_time, end_time;
145
146   // define of the sciddicaT CA and sciddicaT_simulation simulation objects
147   sciddicaT = calCADef2D (ROWS, COLS, CAL_VON_NEUMANN_NEIGHBORHOOD_2D,
          CAL_SPACE_TOROIDAL, CAL_NO_OPT);
148   sciddicaT_simulation = calRunDef2D(sciddicaT, 1, STEPS, CAL_UPDATE_IMPLICIT)
          ;
149
150   // add transition function's sigma_1 and sigma_2 elementary processes
151   calAddElementaryProcess2D(sciddicaT, sciddicaTFlowsComputation);
152   calAddElementaryProcess2D(sciddicaT, sciddicaTWidthUpdate);
153
154   // add substates
155   Q.z = calAddSubstate2Dr(sciddicaT);
156   Q.h = calAddSubstate2Dr(sciddicaT);
157   Q.f[0] = calAddSubstate2Dr(sciddicaT);
158   Q.f[1] = calAddSubstate2Dr(sciddicaT);
159   Q.f[2] = calAddSubstate2Dr(sciddicaT);
160   Q.f[3] = calAddSubstate2Dr(sciddicaT);
161
162   // load configuration
163   calLoadSubstate2Dr(sciddicaT, Q.z, DEM_PATH);
164   calLoadSubstate2Dr(sciddicaT, Q.h, SOURCE_PATH);
165
166   // simulation run
167   calRunAddInitFunc2D(sciddicaT_simulation, sciddicaTSimulationInit);
168   calRunAddSteeringFunc2D(sciddicaT_simulation, sciddicaTSteering);
169   printf ("Starting simulation...\n");
170   start_time = time(NULL);
171   calRun2D(sciddicaT_simulation);
172   end_time = time(NULL);
173   printf ("Simulation terminated.\nElapsed time: %lds\n", end_time-start_time)
          ;
174
175   // saving configuration
176   calSaveSubstate2Dr(sciddicaT, Q.h, OUTPUT_PATH);
177
178   // finalizations
179   calRunFinalize2D(sciddicaT_simulation);
180   calFinalize2D(sciddicaT);
181
182   return 0;
```

```
183 }
```

Listing 4.14: An OpenCAL implementation of the SciddicaT debris flows simulation model.

As for the case of Game of Life, the XCA model and the simulation objects are declared at lines 22 and 35, respectively, and defined later into the main function at lines 147 and 148, respectively. The 2D cellular space results in a structured grid of of `ROWS` rows times `COLS` columns square cells, corresponding to i_{max} and j_{max} of the formal definition, respectively (cf. lines 10-11), and the von Neumann neighborhood is adopted. The cellular space is still toroidal, as in *Life*, and no optimization is considered. Regarding the simulation object, a total of `STEPS` steps (i.e. 4000 steps - cf. line 14) are set, and implicit substates updating considered.

Substates and parameters are grouped into two different C structures (lines 24-28 and 30-33, respectively). Substates are therefore bound to the CA context by means of the `calAddSubstate2Dr()` function (lines 155-160), as well as elementary processes are defined as callback functions and registered to the model object by means of the `calAddElementaryProcess2D()` function (lines 151-152).

The topographic altitude and debris thickness substates are initialized from files through the `calLoadSubstate2Dr()` function (lines 163-164), while the remaining initial state of the model is set by means of the `calRunAddInitFunc2D()` function. It registers the `sciddicaTSimulationInit()` callback, which is executed once before the execution of the simulation loop, in which the elementary processes are applied to the whole set of cells of the cellular space. Such callback function must return void and take a pointer to a simulation object as parameter. Differently to an elementary process, which can only access state values of cells belonging to the neighborhood, this function can perform global operations over the whole cellular space. In the specific case of the $SciddicaT_{naive}$ model, the `sciddicaTSimulationInit()` function (lines 104-130) sets the values of all the outflows from the central cell to its neighbors to zero, by means of the function `calInitSubstate2Dr()` (lines 110-113). Moreover, it sets the values of the P.r and P.epsilon parameters (lines 116-117) and initializes the debris flow source by simply subtracting the source debris thickness to the topographic altitude. For this purpose, a nested double `for` is executed to check the debris thickness in each cell of the cellular space. Here, the `sciddicaT->rows` and `sciddicaT->cols` members of the CA object are used, which represent the cellular space values of rows and columns, respectively. Still, the `calGet2Dr()` and `calSet2Dr()` functions are here employed to read/update substates values inside the cells. Alternatively to the double `for` loop, it would be possible to define a fictitious elementary process in which otflows are locally set to zero, and thus call the `calApplyElementaryProcess2D()` to apply it to the whole computational domain.

Line 168 registers a *steering* callback by means of the `calRunAddSteeringFunc2D()` function. Steering is executed at the end of each computational step (i.e. after all the elementary processes have been applied to each cell of the cellular space), and can perform global operations over the whole cellular space. The steering callback prototype must return void and take a pointer to a simulation onject, as for the `sciddicaTSteering()` callback (lines 132-140). In the specific case of $SciddicaT_{naive}$, the steering callback simply sets to zero outflows everywhere through the `calInitSubstate2Dr()` function. Again, as for the case of the init callback, a fictitious elementary process could be considered for this purpose.

The function `calRun2D()` (line 171) enters the OpenCAL simulation loop, which

executes a total of 4000 steps (cf. lines 14 and 148). Eventually, the final debris flow path is saved to file by means of the `calSaveSubstate2Dr()` function (line 176) and previously allocated memory is released (lines 179-180).

As regards the elementary processes, the first one, σ_1, is defined at lines 38-88, while the second, σ_2, at lines 91-101. In both cases, the `calGet2Dr()` `calGetX2Dr()` functions are employed to get values for the central cell and its neighbors, respectively. Moreover, the `calSet2Dr()` function, updates the central cell state.

Figure 4.5 shows the *SciddicaT*$_{naive}$ simulation of the 1992 Tessina (Italy) landslide. Both the initial landslide source and the final flow path configuration are shown.

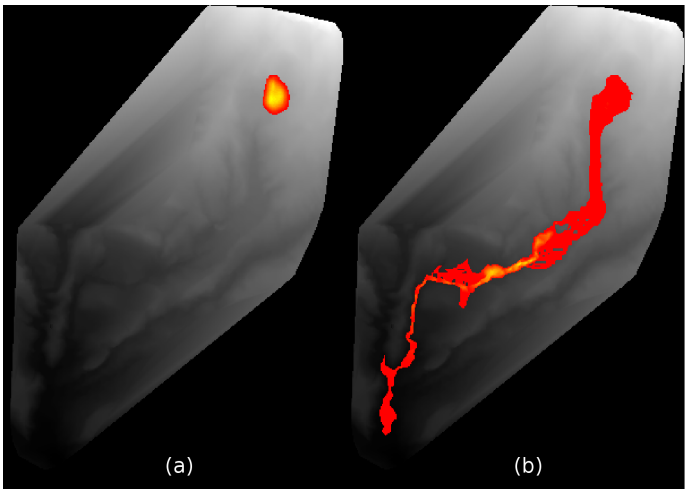


Figure 4.5: *SciddicaT*$_{naive}$ simulation of the 1992 Tessina (Italy) landslide. Topographic altitudes are represented in gray scale (black for lower altitude, white for the highest elevation). Debris thickness is represented with colors ranging from red (lower values) to yellow (higher values). (a) Initial configuration. (b) Final debris flow path. Note that the graphic output was generated by using the `cal_sciddicaT-glut` application, that implements the *SciddicaT*$_{naive}$ model and provides a minimal visualization system. You can find `cal_sciddicaT-glut` in the examples directory.

As regards computational performance, the simulation shown in Figure 4.5 was executed on an Intel Core i7-4702HQ CPU @ 2.20GHz by exploiting only a single core. The simulation lasted a total of 240 seconds and considered a total of 4000 computational steps.

4.6.2 SciddicaT with active cells optimization

In this section we present a computationally improved version of SciddicaT, which takes advantage of the built-in OpenCAL active cells optimization. As stated above, this optimization is able to restrict computation to a subset of cells which are actually involved in computation, by neglecting those cells that will not change state to the next step (stationary cells).

In the case of SciddicaT, only cells containing debris and their neighbors can change state to the next step, as they can be interested in mass variation due to

outflows and inflows. At the beginning of the simulation, we can simply initialize the set of active cells to those cells containing debris (i.e. those cells forming the initial landslide source). Moreover, we can add to this set new cells or remove some ones from it. Specifically, if an outflow is computed from an active cell towards a neighboring non-active cell, this latter can be added to the set of active cells and considered for state change by the remaining elementary processes in the current computation step [2] (if any), or by the next computational step. Similarly, if a given active cell looses a sufficient amount of debris, it can be eliminated from the set of active cells. In the case of SciddicaT, this happens when its thickness becomes lower than or equal to a given threshold (i.e. p_ϵ).

In order to account for these processes, we have to slightly revise the SciddicaT definition. In particular we have to add the set of active cells, A. The optimized SciddicaT model is now defined as

$$SciddicaT_{ac} =< R, A, X, Q, P, \sigma >$$

where $A \subseteq R$ is the set of active cells, while the other components are defined as before. The transition function is now defined as:

$$\sigma : A \times Q^5 \rightarrow Q \times A$$

denoting that it is applied to only the cells in A and that it can add/remove active cells. More in detail, the σ_1 elementary process has to be modified, as it can activate new cells. Moreover, a new elementary process, σ_3, has to be added in order to remove cells that cannot produce outflows during the next computational step due to the fact that their debris thickness is negligible. The new sequence of elementary processes is listed below, in the same order they are applied.

- $\sigma_1 : A \times (Q_z \times Q_h)^5 \times p_\epsilon \times p_r \rightarrow Q_o^4 \times A$ determines the outflows from the central cell to the neighboring ones, as before. In addition, each time an outflow is computed, the neighbor receiving the flow is added to the set of active cells.
- $\sigma_2 : A \times Q_h \times (Q_o^4)^4 \rightarrow Q_h$ determines the value of debris thickness inside the cell by considering mass exchange in the cell neighborhood. This elementary process does not change with respect to the $SciddicaT_{naive}$ version.
- $\sigma_3 : A \times Q_h \times p_\epsilon \rightarrow A$ removes a cell from A if its debris thickness is lower than or equal to the p_ϵ threshold.

In order to implement the $SciddicaT_{ac}$ debris flows model in OpenCAL, we have to change the definition of the CA object, by also adding the third σ_3 elementary process. Moreover, the σ_1 elementary process has to be changed. A complete implementation of the active cells optimized version of SciddicaT is shown in Listing 4.15 for the sake of completeness, even if only the differences with respect to the original implementation are commented.

```
1  // The SciddicaT debris flows model with the active cells optimization
2
3  #include <OpenCAL/cal2D.h>
4  #include <OpenCAL/cal2DIO.h>
5  #include <OpenCAL/cal2DRun.h>
6  #include <OpenCAL/cal2DUnsafe.h>
7  #include <stdlib.h>
```

[2] Remember that, by default, substates are updated after the application of each elementary process.

```
8  #include <time.h>
9
10 // Some definitions...
11 #define ROWS 610
12 #define COLS 496
13 #define P_R 0.5
14 #define P_EPSILON 0.001
15 #define STEPS 4000
16 #define DEM_PATH "./data/dem.txt"
17 #define SOURCE_PATH "./data/source.txt"
18 #define OUTPUT_PATH "./data/width_final.txt"
19 #define NUMBER_OF_OUTFLOWS 4
20
21 // declare CCA model (sciddicaT), substates (Q), parameters (P),
22 // and simulation object (sciddicaT_simulation)
23 struct sciddicaTSubstates {
24   struct CALSubstate2Dr *z;
25   struct CALSubstate2Dr *h;
26   struct CALSubstate2Dr *f[NUMBER_OF_OUTFLOWS];
27 } Q;
28
29 struct sciddicaTParameters {
30   CALParameterr epsilon;
31   CALParameterr r;
32 } P;
33
34
35 // The sigma_1 elementary process
36 void sciddicaTFlowsComputation(struct CALModel2D* sciddicaT, int i, int j)
37 {
38   CALbyte eliminated_cells[5]={CAL_FALSE,CAL_FALSE,CAL_FALSE,CAL_FALSE,
         CAL_FALSE};
39   CALbyte again;
40   CALint cells_count;
41   CALreal average;
42   CALreal m;
43   CALreal u[5];
44   CALint n;
45   CALreal z, h;
46   CALreal f;
47
48
49   m = calGet2Dr(sciddicaT, Q.h, i, j) - P.epsilon;
50   u[0] = calGet2Dr(sciddicaT, Q.z, i, j) + P.epsilon;
51   for (n=1; n<sciddicaT->sizeof_X; n++)
52   {
53     z = calGetX2Dr(sciddicaT, Q.z, i, j, n);
54     h = calGetX2Dr(sciddicaT, Q.h, i, j, n);
55     u[n] = z + h;
56   }
57
58   //computes outflows and updates debris thickness
59   do{
60     again = CAL_FALSE;
61     average = m;
62     cells_count = 0;
63
64     for (n=0; n<sciddicaT->sizeof_X; n++)
65       if (!eliminated_cells[n]){
66         average += u[n];
67         cells_count++;
68       }
69
70       if (cells_count != 0)
71         average /= cells_count;
72
73       for (n=0; n<sciddicaT->sizeof_X; n++)
74         if( (average<=u[n]) && (!eliminated_cells[n]) ){
```

```
75            eliminated_cells[n]=CAL_TRUE;
76            again=CAL_TRUE;
77          }
78
79    }while (again);
80
81    for (n=1; n<sciddicaT->sizeof_X; n++)
82      if (eliminated_cells[n])
83        calSet2Dr(sciddicaT, Q.f[n-1], i, j, 0.0);
84      else
85      {
86        calSet2Dr(sciddicaT, Q.f[n-1], i, j, (average-u[n])*P.r);
87        calAddActiveCellX2D(sciddicaT, i, j, n);
88      }
89  }
90
91  // The sigma_2 elementary process
92  void sciddicaTWidthUpdate(struct CALModel2D* sciddicaT, int i, int j)
93  {
94    CALreal h_next;
95    CALint n;
96
97    h_next = calGet2Dr(sciddicaT, Q.h, i, j);
98    for(n=1; n<sciddicaT->sizeof_X; n++)
99      h_next +=  calGetX2Dr(sciddicaT, Q.f[NUMBER_OF_OUTFLOWS - n], i, j, n) -
           calGet2Dr(sciddicaT, Q.f[n-1], i, j);
100
101   calSet2Dr(sciddicaT, Q.h, i, j, h_next);
102 }
103
104 // The sigma_3 elementary process
105 void sciddicaTRemoveInactiveCells(struct CALModel2D* sciddicaT, int i, int j)
106 {
107   if (calGet2Dr(sciddicaT, Q.h, i, j) <= P.epsilon)
108     calRemoveActiveCell2D(sciddicaT,i,j);
109 }
110
111
112 void sciddicaTSimulationInit(struct CALModel2D* sciddicaT)
113 {
114   CALreal z, h;
115   CALint i, j;
116
117   //sciddicaT parameters setting
118   P.r = P_R;
119   P.epsilon = P_EPSILON;
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   //sciddicaT source initialization
128   for (i=0; i<sciddicaT->rows; i++)
129     for (j=0; j<sciddicaT->columns; j++)
130     {
131       h = calGet2Dr(sciddicaT, Q.h, i, j);
132
133       if ( h > 0.0 ) {
134         z = calGet2Dr(sciddicaT, Q.z, i, j);
135         calSetCurrent2Dr(sciddicaT, Q.z, i, j, z-h);
136
137         //adds the cell (i, j) to the set of active ones
138         calAddActiveCell2D(sciddicaT, i, j);
139       }
140     }
141 }
```

```
142
143 // SciddicaT steering function
144 void sciddicaTSteering(struct CALModel2D* sciddicaT)
145 {
146   // set flow to 0 everywhere
147   calInitSubstate2Dr(sciddicaT, Q.f[0], 0);
148   calInitSubstate2Dr(sciddicaT, Q.f[1], 0);
149   calInitSubstate2Dr(sciddicaT, Q.f[2], 0);
150   calInitSubstate2Dr(sciddicaT, Q.f[3], 0);
151 }
152
153
154 int main()
155 {
156   time_t start_time, end_time;
157
158   // define of the sciddicaT CA and sciddicaT_simulation simulation objects
159   struct CALModel2D* sciddicaT = calCADef2D (ROWS, COLS,
          CAL_VON_NEUMANN_NEIGHBORHOOD_2D, CAL_SPACE_TOROIDAL,
          CAL_OPT_ACTIVE_CELLS);
160   struct CALRun2D* sciddicaT_simulation = calRunDef2D(sciddicaT, 1, STEPS,
          CAL_UPDATE_IMPLICIT);
161
162   // add transition function's sigma_1 and sigma_2 elementary processes
163   calAddElementaryProcess2D(sciddicaT, sciddicaTFlowsComputation);
164   calAddElementaryProcess2D(sciddicaT, sciddicaTWidthUpdate);
165   calAddElementaryProcess2D(sciddicaT, sciddicaTRemoveInactiveCells);
166
167   // add substates
168   Q.z = calAddSingleLayerSubstate2Dr(sciddicaT);
169   Q.h = calAddSubstate2Dr(sciddicaT);
170   Q.f[0] = calAddSubstate2Dr(sciddicaT);
171   Q.f[1] = calAddSubstate2Dr(sciddicaT);
172   Q.f[2] = calAddSubstate2Dr(sciddicaT);
173   Q.f[3] = calAddSubstate2Dr(sciddicaT);
174
175   // load configuration
176   calLoadSubstate2Dr(sciddicaT, Q.z, DEM_PATH);
177   calLoadSubstate2Dr(sciddicaT, Q.h, SOURCE_PATH);
178
179   // simulation run
180   calRunAddInitFunc2D(sciddicaT_simulation, sciddicaTSimulationInit);
181   calRunAddSteeringFunc2D(sciddicaT_simulation, sciddicaTSteering);
182   printf ("Starting simulation...\n");
183   start_time = time(NULL);
184   calRun2D(sciddicaT_simulation);
185   end_time = time(NULL);
186   printf ("Simulation terminated.\nElapsed time: %lds\n", end_time-start_time)
          ;
187
188   // saving configuration
189   calSaveSubstate2Dr(sciddicaT, Q.h, OUTPUT_PATH);
190
191   // finalizations
192   calRunFinalize2D(sciddicaT_simulation);
193   calFinalize2D(sciddicaT);
194
195   return 0;
196 }
```

Listing 4.15: An OpenCAL implementation of the $SciddicaT_{ac}$ debris flows simulation model with the active cells optimization.

As noted, few modifications to the original source code are needed to add the active cells optimization. In particular, the active cells support is enabled by means of the `CAL_OPT_ACTIVE_CELLS` parameter at line 159, while the third elementary

process added at line 165. As regards the elementary process σ_1, it is the same of that adopted in $SciddicaT_{naive}$, with the exception that when an outflow is generated, the cell receiving the flow is added to the set A of the active cells (line 87). Note that the `calAddActiveCellX2D()` can be considered as an unsafe function, as it affects the activation state of a neighboring cell. For this reason, it is important to maintain the activation and deactivation phases completely disjoint, in order to avoid possible race conditions and thus logical errors. As a matter of fact, the σ_1 elementary process only add cells to A, while is the σ_3 one that removes cells from A when they become inactive (lines 107-108). According to the formal definition of $SciddicaT_{ac}$, this occurs in the case the debris thickness becomes lower than or equal to the p_ϵ threshold parameter. Since, in general, the cell state activation/deactivation is threshold-dependent, the active cells optimization is also known as *quantization*.

Regarding the computational performance, the same simulation shown in Figure 4.5 was executed using the $SciddicaT_{ac}$ implementation. Still, only a single core of the Intel Core i7-4702HQ CPU was used. The simulation lasted a total of 23 seconds, versus 240 seconds obtained for the naive version, which is about 10.5 times faster. Indeed, this can be considered a very good result and can be easily obtained. In general, simulations which involve only a subset of the whole computational domain can take advantage by the active cells optimization.

4.6.3 SciddicaT with direct neighbors update

OpenCAL allows for a further optimization by means of the so called *unsafe operations*, i.e. operations that are not strictly permitted by the formal definition of Cellular Automata. Obviously, in order to be well defined, a CA exploiting unsafe operations must be equivalent to a CA that does not use them.

In the case of SciddicaT, we will permit the transition function to update the state of the neighboring cells, while the CA formally only allows for state change for the central one. When an outflow is computed from the central cell towards a neighbor, the flow can be immediately subtracted from the central cell and added to the neighbor. This does not change the state of the system at the current step, which is defined by the *current* computational plane, since updated values are written to the *next* plane. As a result, the *current* plane is not corrupted by the unsafe operation, and the *next* plane is used for progressively accounting mass variation inside the cells. By introducing such feature, outflows do not need to be saved into additional substates anymore, as they are used to account mass exchange directly during outflows computation. As figured out, this can give rise to a further performance improvement for the application. The $SciddicaT_{ac+dnu}$ CA exploiting both the active cells optimization and unsafe operations is formally defined as:

$$SciddicaT_{ac+dnu} =< R, A, X, Q, P, \sigma >$$

where:

- R is the set of points, with integer coordinates, which defines the 2-dimensional cellular space over which the phenomenon evolves. The generic cell in R is individuated by means of a couple of integer coordinates (i, j), where $0 \leq i < i_{max}$ and $0 \leq j < j_{max}$. The first coordinate, i, represents the row, while the second, j, the column. The cell at coordinates $(0, 0)$ is located at the top-left corner of the computational grid.

- $A \subseteq R$ is the set of active cells, i.e. those cells actually involved in computation.
- $X = \{(0,0),(-1,0),(0,-1),(0,1),(1,0)\}$ is the von Neumann neighborhood relation (cf. Figure 3.3), a geometrical pattern which identifies the cells influencing the state transition of the central cell. The neighborhood of the generic cell of coordinate (i, j) is given by

$$N(X,(i,j)) =$$
$$= \{(i,j)+(0,0),(i,j)+(-1,0),(i,j)+(0,-1),(i,j)+(0,1),(i,j)+(1,0)\} =$$
$$= \{(i,j),(i-1,j),(i,j-1),(i,j+1),(i+1,j)\}$$

  Here, a subscript operator can be used to index cells belonging to the neighborhood. Let $|X|$ be the number of elements in X, and $n \in \mathbb{N}$, $0 \leq n < |X|$; the notation

$$N(X,(i,j),n)$$

  represents the n^{th} neighborhood of the cell (i, j). Thereby, $N(X,(i,j),0) = (i,j)$, i.e. the central cell, $N(X,(i,j),1) = (i-1,j)$, i.e. the first neighbor, and so on (cf. Figure 4.2).

- Q is the set of cell states; it is subdivided in the following substates:
  - Q_z is the set of values representing the topographic altitude (i.e. elevation);
  - Q_h is the set of values representing the debris thickness;

  The Cartesian product of the substates defines the overall set of state Q:

$$Q = Q_z \times Q_h$$

  so that the cell state is specified by:

$$q = (q_z, q_h)$$

- P is set of parameters ruling the CA dynamics:
  - p_ϵ is the parameter which specifies the thickness of the debris that cannot leave the cell due to the effect of adherence;
  - p_r is the relaxation rate parameter, which affects the size of outflows (cf. section above).

- $\sigma : A \times Q^5 \to Q$ is the deterministic cell transition function. It is composed by two elementary processes:
  - $\sigma_1 : A \times (Q_z \times Q_h)^5 \times p_\epsilon \times p_r \to (A \times Q_h)^5$ determines the outflows from the central cell to the neighboring ones and updates debris thickness inside the central cell and its neighbors accordingly. It also adds the neighboring cells receiving a flow to the set A of the active cells.

– $\sigma_2 : A \times Q_h \times p_\epsilon \rightarrow A$ removes the cell from the set A of the active cells if the debris thickness inside the cell is lower than or equal to the p_ϵ threshold.

Note that, only the topographic altitude and the debris thickness are now considered as model substates, as the four outflows substates are no longer needed. Moreover, the number of elementary process now considered is two, instead of three for the previous versions of SciddicaT. The OpenCAL implementation of $SciddicaT_{ac+dnu}$ is shown in Listing 4.16.

```
1  // The SciddicaT further optimized CCA debris flows model
2
3  #include <OpenCAL/cal2D.h>
4  #include <OpenCAL/cal2DIO.h>
5  #include <OpenCAL/cal2DRun.h>
6  #include <OpenCAL/cal2DUnsafe.h>
7  #include <stdlib.h>
8  #include <time.h>
9
10 // Some definitions...
11 #define ROWS 610
12 #define COLS 496
13 #define P_R 0.5
14 #define P_EPSILON 0.001
15 #define STEPS 4000
16 #define DEM_PATH "./data/dem.txt"
17 #define SOURCE_PATH "./data/source.txt"
18 #define OUTPUT_PATH "./data/width_final.txt"
19 #define NUMBER_OF_OUTFLOWS 4
20
21 // declare CCA model (sciddicaT), substates (Q), parameters (P),
22 // and simulation object (sciddicaT_simulation)
23 struct sciddicaTSubstates {
24   struct CALSubstate2Dr *z;
25   struct CALSubstate2Dr *h;
26 } Q;
27
28 struct sciddicaTParameters {
29   CALParameterr epsilon;
30   CALParameterr r;
31 } P;
32
33
34 // The sciddicaT transition function
35 void sciddicaTFlowsComputation(struct CALModel2D* sciddicaT, int i, int j)
36 {
37   CALbyte eliminated_cells[5]={CAL_FALSE,CAL_FALSE,CAL_FALSE,CAL_FALSE,
         CAL_FALSE};
38   CALbyte again;
39   CALint cells_count;
40   CALreal average;
41   CALreal m;
42   CALreal u[5];
43   CALint n;
44   CALreal z, h;
45   CALreal f;
46
47
48   m = calGet2Dr(sciddicaT, Q.h, i, j) - P.epsilon;
49   u[0] = calGet2Dr(sciddicaT, Q.z, i, j) + P.epsilon;
50   for (n=1; n<sciddicaT->sizeof_X; n++)
51   {
52     z = calGetX2Dr(sciddicaT, Q.z, i, j, n);
53     h = calGetX2Dr(sciddicaT, Q.h, i, j, n);
54     u[n] = z + h;
```

```
55    }
56
57    //computes outflows and updates debris thickness
58    do{
59      again = CAL_FALSE;
60      average = m;
61      cells_count = 0;
62
63      for (n=0; n<sciddicaT->sizeof_X; n++)
64        if (!eliminated_cells[n]){
65          average += u[n];
66          cells_count++;
67        }
68
69        if (cells_count != 0)
70          average /= cells_count;
71
72        for (n=0; n<sciddicaT->sizeof_X; n++)
73          if( (average<=u[n]) && (!eliminated_cells[n]) ){
74            eliminated_cells[n]=CAL_TRUE;
75            again=CAL_TRUE;
76          }
77
78    }while (again);
79
80    for (n=1; n<sciddicaT->sizeof_X; n++)
81      if (!eliminated_cells[n])
82      {
83        f = (average-u[n])*P.r;
84        calSet2Dr (sciddicaT,Q.h,i,j,   calGetNext2Dr (sciddicaT,Q.h,i,j)   - f
              );
85        calSetX2Dr(sciddicaT,Q.h,i,j,n, calGetNextX2Dr(sciddicaT,Q.h,i,j,n) + f
              );
86
87        //adds the cell (i, j, n) to the set of active ones
88        calAddActiveCellX2D(sciddicaT, i, j, n);
89      }
90  }
91
92
93  void sciddicaTRemoveInactiveCells(struct CALModel2D* sciddicaT, int i, int j)
94  {
95    if (calGet2Dr(sciddicaT, Q.h, i, j) <= P.epsilon)
96      calRemoveActiveCell2D(sciddicaT,i,j);
97  }
98
99
100 void sciddicaTSimulationInit(struct CALModel2D* sciddicaT)
101 {
102   CALreal z, h;
103   CALint i, j;
104
105   //sciddicaT parameters setting
106   P.r = P_R;
107   P.epsilon = P_EPSILON;
108
109   //sciddicaT source initialization
110   for (i=0; i<sciddicaT->rows; i++)
111     for (j=0; j<sciddicaT->columns; j++)
112     {
113       h = calGet2Dr(sciddicaT, Q.h, i, j);
114
115       if ( h > 0.0 ) {
116         z = calGet2Dr(sciddicaT, Q.z, i, j);
117         calSetCurrent2Dr(sciddicaT, Q.z, i, j, z-h);
118
119         //adds the cell (i, j) to the set of active ones
120         calAddActiveCell2D(sciddicaT, i, j);
```

```
121         }
122       }
123   }
124
125
126   int main()
127   {
128     time_t start_time, end_time;
129
130     // define of the sciddicaT CA and sciddicaT_simulation simulation objects
131     struct CALModel2D* sciddicaT = calCADef2D (ROWS, COLS,
            CAL_VON_NEUMANN_NEIGHBORHOOD_2D, CAL_SPACE_TOROIDAL,
            CAL_OPT_ACTIVE_CELLS);
132     struct CALRun2D* sciddicaT_simulation = calRunDef2D(sciddicaT, 1, STEPS,
            CAL_UPDATE_IMPLICIT);
133
134     // add transition function's sigma_1 and sigma_2 elementary processes
135     calAddElementaryProcess2D(sciddicaT, sciddicaTFlowsComputation);
136     calAddElementaryProcess2D(sciddicaT, sciddicaTRemoveInactiveCells);
137
138     // add substates
139     Q.z = calAddSingleLayerSubstate2Dr(sciddicaT);
140     Q.h = calAddSubstate2Dr(sciddicaT);
141
142     // load configuration
143     calLoadSubstate2Dr(sciddicaT, Q.z, DEM_PATH);
144     calLoadSubstate2Dr(sciddicaT, Q.h, SOURCE_PATH);
145
146     // simulation run
147     calRunAddInitFunc2D(sciddicaT_simulation, sciddicaTSimulationInit);
148     printf ("Starting simulation...\n");
149     start_time = time(NULL);
150     calRun2D(sciddicaT_simulation);
151     end_time = time(NULL);
152     printf ("Simulation terminated.\nElapsed time: %lds\n", end_time-start_time)
            ;
153
154     // saving configuration
155     calSaveSubstate2Dr(sciddicaT, Q.h, OUTPUT_PATH);
156
157     // finalizations
158     calRunFinalize2D(sciddicaT_simulation);
159     calFinalize2D(sciddicaT);
160
161     return 0;
162   }
```

Listing 4.16: An OpenCAL implementation of the SciddicaT debris flows simulation model with both the active cells optimization and the direct neighbors updddate unsafe operation.

As noted, the definitions of the model and the simulation objects does not change from the previous implementation (lines 131-132), while only two elementary processes are considered (lines 135-136). In particular, the first call to `calAddElementaryProcess2D()` registers the callback function implementing the σ_1 elementary process. It computes outflows from the (active) central cell to its neighbors (line 83) and updates the debris thickness in both the central cell and the neighboring cell receiving a flow (lines 84-85). Moreover, neighboring cells receiving a flow are added to the set A of active cells (line 88) and therefore will be considered for elaboration by the subsequent elementary process (σ_2) in the current step of computation[3]. In particular, the `calSetX2Dr()` *unsafe* function is used

[3]This is due to the fact that a substates update is performed after the first elementary process has been

to update the debris thickness of the neighboring cells receiving a flow, while the `calAddActiveCellX2D()` function is used to add a neighboring cells receiving a flow to the set A of active cells. The σ_2 elementary process, simply removes inactive cells from A (lines 95-86), as in the previous example.

Substates are added to the CA at lines 139-140. Here, the first substate, Q_z, is added by means of the `calAddSingleLayerSubstate2Dr()` function, here considered to allocate memory only for the *current* computing plane. In fact, topographic altitude only changes at the simulation initialization stage (cf. lines 147 and 117), while it remains unchanged during computation as its value is never updated by the transition function. This allows for memory space allocation optimization and possibly for computational performance improvements. Note that, at line 117 we used the `calSetCurrent2Dr()` function, instead of the usual `calSet2Dr()`. The `calSetCurrent2Dr()` function allows for updating the *current* computational plane (the only present in the Q_z substate), while `calSet2Dr()` would update the *next* computational plane, by producing an access violation error.

Regarding the computational performance, the same simulation shown in Figure 4.5 was executed by considering $SciddicaT_{ac+dnu}$ on a single core of the same Intel core i7 processor. The simulation lasted a total of 13 seconds, versus 240 seconds obtained by $SciddicaT_{naive}$, corresposnding to a speed up of about 18.5.

4.6.4 SciddicaT with explicit simulation loop

Even if results obtained so far can be considered more than satisfying, it is further possible to improve computational performance of SciddicaT by avoiding unnecessary substates updating. In fact, in some cases, elementary processes do not affect one or more model substates and therefore their updating can be avoided.

As we stated above, when we use the implicit `calRun2D()` simulation loop, an update of all the defined substates is executed at the end of each elementary process. However, this behavior can be modified by making the OpenCAL simulation loop explicit.

In the specific case of the above implementation of SciddicaT, the second elementary process, σ_2, just removes cells that become inactive from the set A of active cells and does not affect the model substates[4]. As a consequence, no substates updating is needed after the application of σ_2. Being substates updating a time consuming operation, this can further speed up your simulation.

The new $SciddicaT_{ac+dnu+esl}$ OpenCAL implementation of SciddicaT, which take advantage of an explicit simulation loop, avoiding unnecessary substate updating, is presented in Listing 4.17. It also shows how a generic stopping criterion can be defined.

```
1  // The SciddicaT further optimized CCA debris flows model
2
3  #include <OpenCAL/cal2D.h>
4  #include <OpenCAL/cal2DIO.h>
5  #include <OpenCAL/cal2DRun.h>
6  #include <OpenCAL/cal2DUnsafe.h>
7  #include <stdlib.h>
8  #include <time.h>
9
10 // Some definitions...
```

applied to all the (active) cells of the cellular space. This behavior is set by means of the `CAL_UPDATE_IMPLICIT` parameter used in the definition of the simulation object at line 132 of Listing 4.16.

[4] Actually, only $Q.h$ can be updated by the transition function, since $Q.z$ is a single-layered substate.

```
11  #define ROWS 610
12  #define COLS 496
13  #define P_R 0.5
14  #define P_EPSILON 0.001
15  #define STEPS 4000
16  #define DEM_PATH "./data/dem.txt"
17  #define SOURCE_PATH "./data/source.txt"
18  #define OUTPUT_PATH "./data/width_final.txt"
19  #define NUMBER_OF_OUTFLOWS 4
20
21  // declare CCA model (sciddicaT), substates (Q), parameters (P),
22  // and simulation object (sciddicaT_simulation)
23  struct CALModel2D* sciddicaT;
24
25  struct sciddicaTSubstates {
26    struct CALSubstate2Dr *z;
27    struct CALSubstate2Dr *h;
28  } Q;
29
30  struct sciddicaTParameters {
31    CALParameterr epsilon;
32    CALParameterr r;
33  } P;
34
35  struct CALRun2D* sciddicaT_simulation;
36
37
38  // The sciddicaT transition function
39  void sciddicaTFlowsComputation(struct CALModel2D* sciddicaT, int i, int j)
40  {
41    CALbyte eliminated_cells[5]={CAL_FALSE,CAL_FALSE,CAL_FALSE,CAL_FALSE,
          CAL_FALSE};
42    CALbyte again;
43    CALint cells_count;
44    CALreal average;
45    CALreal m;
46    CALreal u[5];
47    CALint n;
48    CALreal z, h;
49    CALreal f;
50
51
52    m = calGet2Dr(sciddicaT, Q.h, i, j) - P.epsilon;
53    u[0] = calGet2Dr(sciddicaT, Q.z, i, j) + P.epsilon;
54    for (n=1; n<sciddicaT->sizeof_X; n++)
55    {
56      z = calGetX2Dr(sciddicaT, Q.z, i, j, n);
57      h = calGetX2Dr(sciddicaT, Q.h, i, j, n);
58      u[n] = z + h;
59    }
60
61    //computes outflows and updates debris thickness
62    do{
63      again = CAL_FALSE;
64      average = m;
65      cells_count = 0;
66
67      for (n=0; n<sciddicaT->sizeof_X; n++)
68        if (!eliminated_cells[n]){
69          average += u[n];
70          cells_count++;
71        }
72
73        if (cells_count != 0)
74          average /= cells_count;
75
76        for (n=0; n<sciddicaT->sizeof_X; n++)
77          if( (average<=u[n]) && (!eliminated_cells[n]) ){
```

```
78            eliminated_cells[n]=CAL_TRUE;
79            again=CAL_TRUE;
80          }
81
82    }while (again);
83
84    for (n=1; n<sciddicaT->sizeof_X; n++)
85      if (!eliminated_cells[n])
86      {
87        f = (average-u[n])*P.r;
88        calSet2Dr (sciddicaT,Q.h,i,j,   calGetNext2Dr (sciddicaT,Q.h,i,j)   - f
              );
89        calSetX2Dr(sciddicaT,Q.h,i,j,n, calGetNextX2Dr(sciddicaT,Q.h,i,j,n) + f
              );
90
91        //adds the cell (i, j, n) to the set of active ones
92        calAddActiveCellX2D(sciddicaT, i, j, n);
93      }
94  }
95
96
97  void sciddicaTRemoveInactiveCells(struct CALModel2D* sciddicaT, int i, int j)
98  {
99    if (calGet2Dr(sciddicaT, Q.h, i, j) <= P.epsilon)
100     calRemoveActiveCell2D(sciddicaT,i,j);
101 }
102
103
104 void sciddicaTSimulationInit(struct CALModel2D* sciddicaT)
105 {
106   CALreal z, h;
107   CALint i, j;
108
109   //sciddicaT parameters setting
110   P.r = P_R;
111   P.epsilon = P_EPSILON;
112
113   //sciddicaT source initialization
114   for (i=0; i<sciddicaT->rows; i++)
115     for (j=0; j<sciddicaT->columns; j++)
116     {
117       h = calGet2Dr(sciddicaT, Q.h, i, j);
118
119       if ( h > 0.0 ) {
120         z = calGet2Dr(sciddicaT, Q.z, i, j);
121         calSetCurrent2Dr(sciddicaT, Q.z, i, j, z-h);
122
123         //adds the cell (i, j) to the set of active ones
124         calAddActiveCell2D(sciddicaT, i, j);
125       }
126     }
127
128   calUpdate2D(sciddicaT);
129 }
130
131
132 void sciddicaTransitionFunction(struct CALModel2D* sciddicaT)
133 {
134   // active cells must be updated first becouse outflows
135   // have already been sent to (pheraps inactive) the neighbours
136   calApplyElementaryProcess2D(sciddicaT, sciddicaTFlowsComputation);
137     calUpdateActiveCells2D(sciddicaT);
138     calUpdateSubstate2Dr(sciddicaT, Q.h);
139
140   // here you don't need to update Q.h
141   calApplyElementaryProcess2D(sciddicaT, sciddicaTRemoveInactiveCells);
142   calUpdateActiveCells2D(sciddicaT);
143 }
```

```
144
145
146 CALbyte sciddicaTSimulationStopCondition(struct CALModel2D* sciddicaT)
147 {
148   if (sciddicaT_simulation->step >= STEPS)
149     return CAL_TRUE;
150   return CAL_FALSE;
151 }
152
153
154 int main()
155 {
156   CALbyte again;
157   time_t start_time, end_time;
158
159   // define of the sciddicaT CA and sciddicaT_simulation simulation objects
160   sciddicaT = calCADef2D (ROWS, COLS, CAL_VON_NEUMANN_NEIGHBORHOOD_2D,
          CAL_SPACE_TOROIDAL, CAL_OPT_ACTIVE_CELLS);
161   sciddicaT_simulation = calRunDef2D(sciddicaT, 1, CAL_RUN_LOOP,
          CAL_UPDATE_EXPLICIT);
162
163   // add transition function's sigma_1 and sigma_2 elementary processes
164   calAddElementaryProcess2D(sciddicaT, sciddicaTFlowsComputation);
165   calAddElementaryProcess2D(sciddicaT, sciddicaTRemoveInactiveCells);
166
167   // add substates
168   Q.z = calAddSingleLayerSubstate2Dr(sciddicaT);
169   Q.h = calAddSubstate2Dr(sciddicaT);
170
171   // load configuration
172   calLoadSubstate2Dr(sciddicaT, Q.z, DEM_PATH);
173   calLoadSubstate2Dr(sciddicaT, Q.h, SOURCE_PATH);
174
175   // simulation run
176   calRunAddInitFunc2D(sciddicaT_simulation, sciddicaTSimulationInit);
177   calRunAddGlobalTransitionFunc2D(sciddicaT_simulation,
          sciddicaTransitionFunction);
178   calRunAddStopConditionFunc2D(sciddicaT_simulation,
          sciddicaTSimulationStopCondition);
179
180   printf ("Starting simulation...\n");
181   start_time = time(NULL);
182   // applies the callback init func registered by calRunAddInitFunc2D()
183   calRunInitSimulation2D(sciddicaT_simulation);
184   // the do-while explicitates the calRun2D() implicit looop
185   do{
186       again = calRunCAStep2D(sciddicaT_simulation);
187       sciddicaT_simulation->step++;
188   } while (again);
189   calRunFinalizeSimulation2D(sciddicaT_simulation);
190   end_time = time(NULL);
191   printf ("Simulation terminated.\nElapsed time: %lds\n", end_time-start_time)
          ;
192
193   // saving configuration
194   calSaveSubstate2Dr(sciddicaT, Q.h, OUTPUT_PATH);
195
196   // finalizations
197   calRunFinalize2D(sciddicaT_simulation);
198   calFinalize2D(sciddicaT);
199
200   return 0;
201 }
```

Listing 4.17: An OpenCAL implementation of the $SciddicaT_{ac+dnu+esl}$ debris flows simulation model with the active cells optimization

CA model	Elapsed time [s]	Speedup
$SciddicaT_{naive}$	240	1
$SciddicaT_{ac}$	23	10.5
$SciddicaT_{ac+dnu}$	13	18.5
$SciddicaT_{ac+dnu+esl}$	12	20.0

Table 4.3: Computational performance of four different implementations of the SciddicaT debris flows model.

As noted, the `calRunAddGlobalTransitionFunc2D()` function is called to register a custom transition function callback (line 177). In the specific case of SciddicaT, the `sciddicaTransitionFunction()` callback (lines 132-143) is used to make the elementary processes application and the substates update explicit. Here, the elementary processes are applied in the same order they are defined by means of the `calAddElementaryProcess2D()` function (which is the default behavior of OpenCAL), even if you are free to re-order the call sequence within the explicit transition function callback. In particular, the `sciddicaTFlowsComputation()` elementary process is applied to each (active) cell into the computational domain by means of the `calApplyElementaryProcess2D()`. Then, the set A of the active cells and the Q_h substate are updated by `calUpdateActiveCells2D()` and `calUpdateSubstate2Dr()`, respectively[5]. Eventually, the `sciddicaTRemoveInactiveCells()` elementary process is applied, which only removes cells that become inactive during the current computational step, and the set A is accordingly updated.

As regards the computational performance, execution time of this further optimized version lasted 12 seconds to complete the 4000 steps required by the simulation on a single core of the same Intel Core i7 processor used before. Table 5.1 resumes computational performance of all the above illustrated SciddicaT implementations.

4.7 A three-dimensional example

In order to introduce three-dimensional structured grid-based model development with OpenCAL, this section describes the implementation of a simple 3D model, namely the *mod2* 3D CA. In this model, cells can be in one of two different states, 0 or 1, as in Game of Life. The cellular space is a parallelepiped made by cubic cells, while the cell neighborhood is the 3D Moore one, consisting of the central cell and its adjacent cells. The transition function simply evaluates the quantity s as the number of neighboring cells which are in the state 1 and sets the new state for the central cell as $s\%2$ (i.e. the remainder of s divided by 2). This simple example of 3D CA is formally defined as:

$$mod2 =< R, X, Q, \sigma >$$

where:

- R is the set of points, with integer coordinates, which defines the 3-dimensional cellular space. The generic cell in R is individuated by means of a triple of

[5]Note that active cells are updated first otherwise the subsequent substate update could neglect some cells that have become active during the current step. For instance, inactive cells can receive a flow and become active at the current step of computation. If the set of active cells is not updated before any other substates, those new cells will still be considered inactive during the current step and their value will not be updated, by losing debris flow mass.

integer coordinates (i, j, k), where $0 \leq i < i_{max}$, $0 \leq j < j_{max}$, and $0 \leq k < k_{max}$. The first coordinate, i, represents the row, the second, j, the column, while the third coordinate represents the slice. The cell at coordinates $(0, 0, 0)$ is located at the top-left-far corner of the computational grid.

- $X = \{(0,0,0), \ldots, (-1,1,0), (0,0,-1), \ldots, (-1,1,-1), (0,0,1), \ldots, (-1,1,1)\}$ is the Moore neighborhood relation, a geometrical pattern which identifies the cells influencing the state transition of the central cell. The neighborhood of the generic cell of coordinate (i, j) is given by

$$N(X, (i,j,k)) =$$
$$= \{(i,j,k) + (0,0,0), \ldots, (i,j,k) + (-1,1,-1)\} =$$
$$= \{(i,j,k), \ldots, (i-1, j+1, k-1)\}$$

  Here, a subscript operator can be used to index cells belonging to the neighborhood. Let $|X|$ be the number of elements in X, and $n \in \mathbb{N}$, $0 \leq n < |X|$; the notation

$$N(X, (i,j,k), n)$$

  represents the n^{th} neighborhood of the cell (i, j, k). Thereby, $N(X, (i, j, k), 0) = (i, j, k)$, i.e. the central cell, $N(X, (i, j, k), 1) = (i - 1, j, k)$, i.e. the first neighbor, and so on.

- $Q = \{0, 1\}$ is the set of cell states.

- $\sigma : Q^{27} \to Q$ is the deterministic cell transition function. It is composed by one elementary process, which implements the previously described transition rules.

As imagined, the OpenCAL implementation of the *mod2* 3D CA is very simple as the Conway's game of Life. The complete source code is shown in Listing 4.18.

```
1  // mod2 3D Cellular Automaton
2
3  #include <OpenCAL/cal3D.h>
4  #include <OpenCAL/cal3DIO.h>
5  #include <OpenCAL/cal3DRun.h>
6
7  #define ROWS 5
8  #define COLS 7
9  #define LAYERS 3
10
11 // declare CA, substate and simulation objects
12 struct CALModel3D* mod2;
13 struct CALSubstate3Db* Q;
14 struct CALRun3D* mod2_simulation;
15
16 // The cell's transition function
17 void mod2TransitionFunction(struct CALModel3D* ca, int i, int j, int k)
18 {
19   int sum = 0, n;
20
21   for (n=0; n<ca->sizeof_X; n++)
22     sum += calGetX3Db(ca, Q, i, j, k, n);
23
24   calSet3Db(ca, Q, i, j, k, sum%2);
25 }
```

```
26
27 int main()
28 {
29   // define of the mod2 CA and mod2_simulation simulation objects
30   mod2 = calCADef3D(ROWS, COLS, LAYERS, CAL_MOORE_NEIGHBORHOOD_3D,
         CAL_SPACE_TOROIDAL, CAL_NO_OPT);
31   mod2_simulation = calRunDef3D(mod2, 1, 1, CAL_UPDATE_IMPLICIT);
32
33   // add the Q substate to the mod2 CA
34   Q = calAddSubstate3Db(mod2);
35
36   // add transition function's elementary process
37   calAddElementaryProcess3D(mod2, mod2TransitionFunction);
38
39   // set the whole substate to 0
40   calInitSubstate3Db(mod2, Q, 0);
41
42   // set a seed at position (2, 3, 1)
43   calInit3Db(mod2, Q, 2, 3, 1, 1);
44
45   // save the Q substate to file
46   calSaveSubstate3Db(mod2, Q, "./mod2_0000.txt");
47
48   // simulation run
49   calRun3D(mod2_simulation);
50
51   // save the Q substate to file
52   calSaveSubstate3Db(mod2, Q, "./mod2_LAST.txt");
53
54   // finalize simulation and CA objects
55   calRunFinalize3D(mod2_simulation);
56   calFinalize3D(mod2);
57
58   return 0;
59 }
```

Listing 4.18: An OpenCAL implementation of the *mod2* 3D CA.

Even if Listing 4.18 is concise, it completely defines the *mod2* 3D CA and performs a simulation (actually, only one step in this example). Lines 3-5 include some header files for the 3D version of OpenCAL, while lines 12-14 declare CA, substate and simulation objects. These are therefore defined later in the main function. In particular, line 30 defines the CA as a parallelepiped having `ROWS` rows, `COLS` columns and `SLICES` slices (cf. lines 7-9). Moreover, the 3D Moore neighborhood is here set (see Figure 3.4, Chapter 3) as well as cyclic conditions at boundaries. Eventually, no optimizations are considered. The complete definition of `calCADef3D()` is provided in Listing 4.3. Line 31 defines the simulation object by setting just one step of computation and implicit substate update. The complete definition of `calRunDef3D` is provided in Listing 4.9. Finally, the only substate, Q, is defined at line 34. Note that, since it was defined by means of the `calInitSubstate3Db()` function, each element $q \in Q$ results to be of the CALbyte type. Line 37 registers the one and only CA elementary process, namely the `mod2TransitionFunction()` function, which is then implemented at lines 17-25. Line 43 initializes the cell substate Q at coordinates (2, 3, 1) to the state 1. The obtained initial configuration is then saved at line 46, and the simulation executed at line 49. The final configuration is therefore saved at line 52 and, eventually, memory previously and implicitly allocated is released at lines 55-56. Note that, differently to the previous examples, almost all the OpenCAL functions come in the 3D flower. For instance, this is the case of the `alGetX3Db()` and `calSet3Db()` functions at lines 22 and 24, respectively, which take `k` as third cell coordinate, identifying the cellular space slice.

```
0  0  0  0  0  0  0
0  0  0  0  0  0  0
0  0  0  0  0  0  0
0  0  0  0  0  0  0
0  0  0  0  0  0  0

0  0  0  0  0  0  0
0  0  0  0  0  0  0
0  0  0  1  0  0  0
0  0  0  0  0  0  0
0  0  0  0  0  0  0

0  0  0  0  0  0  0
0  0  0  0  0  0  0
0  0  0  0  0  0  0
0  0  0  0  0  0  0
0  0  0  0  0  0  0
```

Figure 4.6: Initial configuration of mod2 3D CA, as implemented in Listing 4.18.

Figures 4.6 and 4.7 show the initial and final configuration of *mod2* 3D CA as implemented in Listing 4.18, respectively. A graphical representation after 77 computational step is shown in Figure 4.8.

4.8 Combining OpenCAL and OpenCAL-GL

In this section, we will show how to combine the OpenCAL and OpenCAL-GL libraries to obtain an immediate graphic output for your simulation. For this purpose, here we re-propose two of the examples presented above, namely the SciddicaT debris flow model and the *mod2* CA, by adding to them an OpenGL-based viewer in a straightforward manner. Note that the graphic outputs for the SciddicaT and *mod2* CA shown above were instead obtained by means of non-integrated *ad hoc* GLUT applications, which can be found in the OpenCAL examples, together with others, ending with the `-glut` suffix.

4.8.1 Implementing SciddicaT in OpenCAL and OpenCAL-GL

A new implementation of SciddicaT is presented in Listing 4.19, which integrates a simple 2D and 3D multi-view visualization system based on OpenCAL-GL. The complete implementation is shown for the sake of completeness. A screenshot is shown in Figure 4.9.

```
1  // The SciddicaT debris flows CCA simulation model width_final
2  // a 3D graphic visualizer in OpenCAL-GL
3
4  #include <OpenCAL/cal2D.h>
5  #include <OpenCAL/cal2DIO.h>
6  #include <OpenCAL/cal2DRun.h>
7  #include <OpenCAL-GL/calgl2D.h>
8  #include <OpenCAL-GL/calgl2DWindow.h>
9  #include <stdlib.h>
10
```

```
0  0  0  0  0  0  0
0  0  1  1  1  0  0
0  0  1  1  1  0  0
0  0  1  1  1  0  0
0  0  0  0  0  0  0

0  0  0  0  0  0  0
0  0  1  1  1  0  0
0  0  1  1  1  0  0
0  0  1  1  1  0  0
0  0  0  0  0  0  0

0  0  0  0  0  0  0
0  0  1  1  1  0  0
0  0  1  1  1  0  0
0  0  1  1  1  0  0
0  0  0  0  0  0  0
```

Figure 4.7: Final configuration of mod2 3D CA (actually, just one step of computation), as implemented in Listing 4.18.

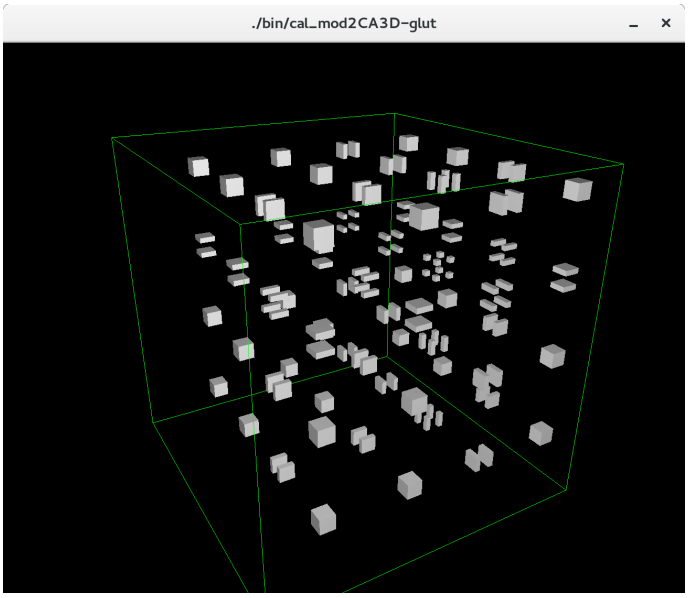


Figure 4.8: Graphical representation of the mod2 3D CA after 77 computational steps, as implemented in Listing 4.18. CA dimensions were set to (rows, cols, slices) = (65, 65, 65), while the initial seed located at coordinates (12, 12, 12). Cells in black are in the state 0, cell in white are in the state 1.

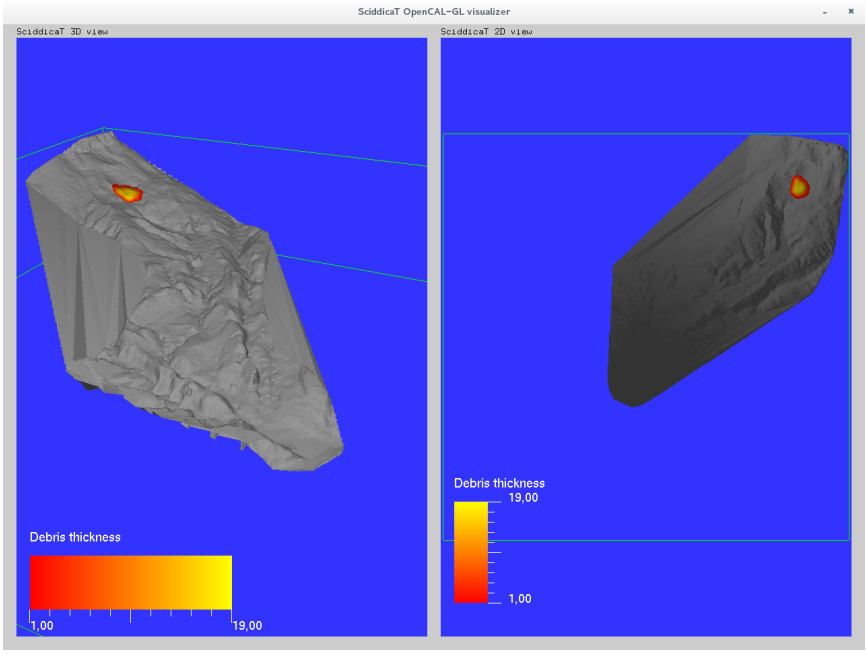


Figure 4.9: Screenshot of the SciddicaT debris flow model with a multi-view 2D and 3D visualization system based on OpenCAL-GL.

```
11 // Some definitions...
12 #define P_R 0.5
13 #define P_EPSILON 0.001
14 #define NUMBER_OF_OUTFLOWS 4
15 #define DEM "./data/dem.txt"
16 #define SOURCE "./data/source.txt"
17 #define FINAL "./data/width_final.txt"
18 #define ROWS 610
19 #define COLUMNS 496
20 #define STEPS 4000
21
22 // declare CCA model (sciddicaT), substates (Q), parameters (P),
23 // and simulation object (sciddicaT_simulation)
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 struct CALModel2D* sciddicaT;           //the cellular automaton
36 struct sciddicaTSubstates Q;            //the substates
37 struct sciddicaTParameters P;           //the parameters
38 struct CALRun2D* sciddicaT_simulation;        //the simulartion run
39
40 // The sigma_1 elementary process
41 void sciddicaTFlowsComputation(struct CALModel2D* sciddicaT, int i, int j) {
42   CALbyte eliminated_cells[5] = {CAL_FALSE, CAL_FALSE, CAL_FALSE, CAL_FALSE,
         CAL_FALSE};
43   CALbyte again;
44   CALint cells_count;
45   CALreal average;
46   CALreal m;
47   CALreal u[5];
48   CALint n;
49   CALreal z, h;
50
51   if(calGet2Dr(sciddicaT, Q.h, i, j)<=P.epsilon)
52     return;
53
54   m = calGet2Dr(sciddicaT, Q.h, i, j)-P.epsilon;
55   u[0] = calGet2Dr(sciddicaT, Q.z, i, j)+P.epsilon;
56   for(n = 1; n<sciddicaT->sizeof_X; n++) {
57     z = calGetX2Dr(sciddicaT, Q.z, i, j, n);
58     h = calGetX2Dr(sciddicaT, Q.h, i, j, n);
59     u[n] = z+h;
60   }
61
62   //computes outflows
63   do {
64     again = CAL_FALSE;
65     average = m;
66     cells_count = 0;
67
68     for(n = 0; n<sciddicaT->sizeof_X; n++)
69       if(!eliminated_cells[n]) {
70         average += u[n];
71         cells_count++;
72       }
73
74     if(cells_count!=0)
75       average /= cells_count;
76
77     for(n = 0; n<sciddicaT->sizeof_X; n++)
```

```
78        if((average<=u[n])&&(!eliminated_cells[n])) {
79          eliminated_cells[n] = CAL_TRUE;
80          again = CAL_TRUE;
81        }
82
83    } while(again);
84
85    for(n = 1; n<sciddicaT->sizeof_X; n++)
86      if(eliminated_cells[n])
87        calSet2Dr(sciddicaT, Q.f[n-1], i, j, 0.0);
88      else
89        calSet2Dr(sciddicaT, Q.f[n-1], i, j, (average-u[n])*P.r);
90  }
91
92  // The sigma_2 elementary process
93  void sciddicaTWidthUpdate(struct CALModel2D* sciddicaT, int i, int j) {
94    CALreal h_next;
95    CALint n;
96
97    h_next = calGet2Dr(sciddicaT, Q.h, i, j);
98    for(n = 1; n<sciddicaT->sizeof_X; n++)
99      h_next += calGetX2Dr(sciddicaT, Q.f[NUMBER_OF_OUTFLOWS-n], i, j, n)-
           calGet2Dr(sciddicaT, Q.f[n-1], i, j);
100
101   calSet2Dr(sciddicaT, Q.h, i, j, h_next);
102 }
103
104 // SciddicaT simulation init function
105 void sciddicaTSimulationInit(struct CALModel2D* sciddicaT) {
106   CALreal z, h;
107   CALint i, j;
108
109   //initializing substates to 0
110   calInitSubstate2Dr(sciddicaT, Q.f[0], 0);
111   calInitSubstate2Dr(sciddicaT, Q.f[1], 0);
112   calInitSubstate2Dr(sciddicaT, Q.f[2], 0);
113   calInitSubstate2Dr(sciddicaT, Q.f[3], 0);
114
115   //sciddicaT parameters setting
116   P.r = P_R;
117   P.epsilon = P_EPSILON;
118
119   //sciddicaT source initialization
120   for(i = 0; i<sciddicaT->rows; i++)
121     for(j = 0; j<sciddicaT->columns; j++) {
122       h = calGet2Dr(sciddicaT, Q.h, i, j);
123
124       if(h>0.0) {
125         z = calGet2Dr(sciddicaT, Q.z, i, j);
126         calSet2Dr(sciddicaT, Q.z, i, j, z-h);
127       }
128     }
129 }
130
131 // SciddicaT steering function
132 void sciddicaTSteering(struct CALModel2D* sciddicaT) {
133   CALreal value = 0;
134
135   //initializing substates to 0
136   calInitSubstate2Dr(sciddicaT, Q.f[0], 0);
137   calInitSubstate2Dr(sciddicaT, Q.f[1], 0);
138   calInitSubstate2Dr(sciddicaT, Q.f[2], 0);
139   calInitSubstate2Dr(sciddicaT, Q.f[3], 0);
140 }
141
142 // SciddicaT stop condition function
143 CALbyte sciddicaTSimulationStopCondition(struct CALModel2D* sciddicaT) {
144   if(sciddicaT_simulation->step>=STEPS)
```

```
145     return CAL_TRUE;
146   return CAL_FALSE;
147 }
148
149 // SciddicaT exit function
150 void exitFunction(void) {
151   // saving configuration
152   calSaveSubstate2Dr(sciddicaT, Q.h, FINAL);
153
154   // finalizations
155   calRunFinalize2D(sciddicaT_simulation);
156   calFinalize2D(sciddicaT);
157 }
158
159
160 int main(int argc, char** argv) {
161   struct CALGLDrawModel2D* draw_model3D = NULL;
162   struct CALGLDrawModel2D* draw_model2D;
163
164   atexit(exitFunction);
165
166   calglInitViewer("SciddicaT OpenCAL-GL visualizer", 5, 800, 600, 10, 10,
          CAL_TRUE, 0);
167
168   //cadef and rundef
169   sciddicaT = calCADef2D(ROWS, COLUMNS, CAL_VON_NEUMANN_NEIGHBORHOOD_2D,
          CAL_SPACE_TOROIDAL, CAL_NO_OPT);
170   sciddicaT_simulation = calRunDef2D(sciddicaT, 1, CAL_RUN_LOOP,
          CAL_UPDATE_IMPLICIT);
171   //add substates
172   Q.z = calAddSubstate2Dr(sciddicaT);
173   Q.h = calAddSubstate2Dr(sciddicaT);
174   Q.f[0] = calAddSubstate2Dr(sciddicaT);
175   Q.f[1] = calAddSubstate2Dr(sciddicaT);
176   Q.f[2] = calAddSubstate2Dr(sciddicaT);
177   Q.f[3] = calAddSubstate2Dr(sciddicaT);
178   //add transition function's elementary processes
179   calAddElementaryProcess2D(sciddicaT, sciddicaTFlowsComputation);
180   calAddElementaryProcess2D(sciddicaT, sciddicaTWidthUpdate);
181   //load configuration
182   calLoadSubstate2Dr(sciddicaT, Q.z, DEM);
183   calLoadSubstate2Dr(sciddicaT, Q.h, SOURCE);
184   //simulation run setup
185   calRunAddInitFunc2D(sciddicaT_simulation, sciddicaTSimulationInit);
186   calRunInitSimulation2D(sciddicaT_simulation);
187   calRunAddSteeringFunc2D(sciddicaT_simulation, sciddicaTSteering);
188   calRunAddStopConditionFunc2D(sciddicaT_simulation,
          sciddicaTSimulationStopCondition);
189
190   // draw_model3D definition
191   draw_model3D = calglDefDrawModel2D(CALGL_DRAW_MODE_SURFACE, "SciddicaT 3D
          view", sciddicaT, sciddicaT_simulation);
192   // Add nodes
193   calglAdd2Dr(draw_model3D, NULL, &Q.z, CALGL_TYPE_INFO_VERTEX_DATA,
          CALGL_TYPE_INFO_USE_NO_COLOR, CALGL_DATA_TYPE_STATIC);
194   calglColor2D(draw_model3D, 0.5, 0.5, 0.5, 1.0);
195   calglAdd2Dr(draw_model3D, Q.z, &Q.z, CALGL_TYPE_INFO_COLOR_DATA,
          CALGL_TYPE_INFO_USE_CURRENT_COLOR, CALGL_DATA_TYPE_DYNAMIC);
196   calglAdd2Dr(draw_model3D, Q.z, &Q.z, CALGL_TYPE_INFO_NORMAL_DATA,
          CALGL_TYPE_INFO_USE_NO_COLOR, CALGL_DATA_TYPE_DYNAMIC);
197   calglAdd2Dr(draw_model3D, Q.z, &Q.h, CALGL_TYPE_INFO_VERTEX_DATA,
          CALGL_TYPE_INFO_USE_NO_COLOR, CALGL_DATA_TYPE_DYNAMIC);
198   calglAdd2Dr(draw_model3D, Q.h, &Q.h, CALGL_TYPE_INFO_COLOR_DATA,
          CALGL_TYPE_INFO_USE_RED_YELLOW_SCALE, CALGL_DATA_TYPE_DYNAMIC);
199   calglAdd2Dr(draw_model3D, Q.h, &Q.h, CALGL_TYPE_INFO_NORMAL_DATA,
          CALGL_TYPE_INFO_USE_NO_COLOR, CALGL_DATA_TYPE_DYNAMIC);
200   // InfoBar
```

Drawing type	Meaning
CALGL_DRAW_MODE_FLAT	Basic substate representation
CALGL_DRAW_MODE_SURFACE	DEM-like representation

Table 4.4: Possible OpenCAL-GL drawing types. DEM is the acronym of Digital Elevation Model.

```
201   //calglRelativeInfoBar2Dr(draw_model3D, Q.h, "Debris thickness",
          CALGL_TYPE_INFO_USE_RED_SCALE, CALGL_INFO_BAR_ORIENTATION_VERTICAL);
202   calglInfoBar2Dr(draw_model3D, Q.h, "Debris thickness",
          CALGL_TYPE_INFO_USE_RED_SCALE, 20, 120, 300, 40);
203
204   // Set offset between substates
205   calglSetHeightOffset2D(draw_model3D, 0.0f);
206
207   // Hide/display intervals of cells
208   //  calglHideDrawJBound2D(draw_model3D, 0, draw_model3D->calModel->columns);
209   //  calglDisplayDrawJBound2D(draw_model3D, 300, draw_model3D->calModel->
          columns);
210   //  calglHideDrawIBound2D(draw_model3D, 100, 150);
211
212   draw_model2D = calglDefDrawModel2D(CALGL_DRAW_MODE_FLAT, "SciddicaT 2D view"
          , sciddicaT, sciddicaT_simulation);
213   draw_model2D->realModel = draw_model3D->realModel;
214   calglInfoBar2Dr(draw_model2D, Q.h, "Debris thickness",
          CALGL_TYPE_INFO_USE_RED_SCALE, 20, 200, 50, 150);
215
216   calglSetLayoutOrientation2D(CALGL_LAYOUT_ORIENTATION_VERTICAL);
217
218   calglSetDisplayStep(100);
219
220   calglMainLoop2D(argc, argv);
221
222   return 0;
223 }
```

Listing 4.19: An OpenCAL implementation of the sciddicaT CA with openCAL-GL graphic output.

As noticed, the `OpenCAL-GL/calgl2D.h` and `OpenCAL-GL/calgl2DWindow.h` headers are included to use OpenCAL-GL at lines 7-8, and two visualization objects (of type `CALDrawModel2D`) are declared into the `main()` function at lines 170-171. Each of them allows to define a particular view for the CA inside a graphic window.

The `calglDefDrawModel2D()` function (lines 200, 218) is used to initialize graphic objects. The first parameter specifies the type of drawing to be considered and can assume the values listed in Table 4.4. If the *basic* (or *flat*) *visualization model* is selected, substates are represented by considering their actual dimensions. Accordingly, 2D substates are represented as flat planes, as well as 3D substates as parallelepipeds. At the contrary, if the *surface model* is selected, 2D substates will be represented in 3D by interpreting data inside cells as altitude values (to be used along the z coordinate). Note that the surface visualization model is valid for 2D CA only. The second parameter is a label, representing the title of the view within the graphic window. Eventually, the third parameter is a pointer to the CA object to be visualized, while the last one is a pointer to the corresponding simulation object.

To build a specific view, we can use the `calglAdd2Dr()` function (lines 204-208). It adds a node into a tree-based representation of the graphic view, where each node has a link to a CA substate and specifies how this substate has to be considered for representation. Specifically, the first parameter is a pointer to a OpenCAL-GL

Substate data interpretation	Meaning
CALGL_TYPE_INFO_VERTEX_DATA	Vertex data
CALGL_TYPE_INFO_NORMAL_DATA	Normal data
CALGL_TYPE_INFO_COLOR_DATA	Color data

Table 4.5: Possible OpenCAL-GL graphic interpretation for substate data.

Substate color specification	Meaning
CALGL_TYPE_INFO_USE_CONST_VALUE	Color is set by using the calglColor2D() function
CALGL_TYPE_INFO_USE_GRAY_SCALE	Gray gradient (white for the highest value)
CALGL_TYPE_INFO_USE_RED_SCALE	Red gradient (red for the highest value)
CALGL_TYPE_INFO_USE_GREEN_SCALE	Green gradient (green for the highest value)
CALGL_TYPE_INFO_USE_BLUE_SCALE	Blue gradient (blue for the highest value)

Table 4.6: Possible OpenCAL-GL substate color settings.

visualization object, while the second and third parameters are pointers to the parent node and the new node to be created, respectively. The fourth parameter specifies how substate data have to be interpreted, if as vertices (to be used to geometrically represent the data), normal vectors[6] (to be used for light calculation), or colors (to be used for shading purposes - cf. Table 4.5). The fifth parameter specifies the color scheme to be used. Possible values are listed in Table 4.6. Eventually, the last parameter specifies if the substate has to be considered static (i.e. it is not updated during the simulation) or dynamic (i.e. values are modified by the simulation). Please refer to Table 4.7 for possible values. In general, however, visualization of static substates is more efficient.

In the considered example (Listing 4.19), the `draw_model3D` object defines a 3D view for the CA, since the first parameter of the `calglDefDrawModel2D()` function is set to `CALGL_DRAW_MODE_SURFACE` (line 200) and, therefore, the substate is interpreted as a 3D surface in which each cell contains an altitude. The tree-based graphic representation is initialized at line 202, where the root node is defined and linked to the `Q.z` substate. It is assumed as the root node because its parent node is set to `NULL`. Moreover, the function specifies that the `Q.z` data has to be used as vertices, to be used to build a mesh representation of the substate itself. Lines 204-205 attach two new *information nodes* to the parent, which corresponds to the root node in this case, both of them having a link to the `Q.z` substate. In particular, line 204 sets the color to be used for representing the root node (i.e. the topographic surface defined by `Q.z`). In this specific case, it is the one previously defined at line 203. Instead, line 205 specifies that `Q.z` have also to be used for normal vectors calculation. Line 206 creates a further new node and attaches it to the root again. In particular, this new node links the `Q.h` substate and sets its data to be used as vertices. The hierarchy thus obtained makes that `Q.h` is represented above `Q.z`, such as two layers. Note that, in some cases, due to approximations of the graphic rendering, layers can intersect each other. In

[6]Normal vectors are obtained by scalar values representing vertices by means of cross product and normalization operations. They are only used for light calculation and are not displayed.

Substate updating type	Meaning
CALGL_DATA_TYPE_DYNAMIC	Dynamic data
CALGL_DATA_TYPE_STATIC	Static data

Table 4.7: Possible substate data updating type.

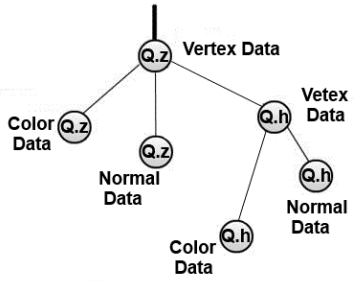


Figure 4.10: Tree-based representation of the 3D graphic view defined in Listing 4.19 and shown in Figure 4.9.

Possible values for view layouts	Meaning
`CALGL_LAYOUT_ORIENTATION_HORIZONTAL`	Horizontal layout
`CALGL_LAYOUT_ORIENTATION_VERTICAL`	Vertical layout
`CALGL_LAYOUT_ORIENTATION_UNKNOW`	Automatic layout

Table 4.8: Possible OpenCAL-GL graphic interpretation for substate data.

this cases, you can move up a layer by means of the `calglSetHeightOffset2D()` function, as done at line 205.

Lines 207-208 define other information nodes for the `Q.h` substate, specifically for color and normal data. Figure 4.10 show the tree obtained by considering the statements at lines 200-208 of Listing 4.19.

The function `calglInfoBar2Dr()` defines a legend for the view. The first parameter is a pointer to the OpenCAL-GL view object, while the second a pointer to a reference substate to associate the resulting scalar-bar. The third parameter is a caption, while the fourth specifies the color scheme to be used, that must be set accordingly to the one used for the substate representation. The last four parameters specify the position (in x and y coordinates with respect to the lower-left corner of the graphic view within the window) and extension for the legend (in pixels). In the considered example, a legend for the `Q.h` substate is defined and attached to the `draw_model3D` object at line 211.

Lines 218-220 of Listing 4.19 define a further view for the same CA. In particular, the first parameter of the `calglDefDrawModel2D()` function, which is set to `CALGL_DRAW_MODE_FLAT`, specifies a 2D (i.e. flat) representation, while the other view specifications are inherited from the previously defined `draw_model3D` object (line 219). A legend is also defined at line 220.

Line 222 sets a vertical layout for arranging the two defined views within the window, by means of the `calglSetLayoutOrientation2D()` function. Allowed layouts are listed in Table 4.8. If no layouts are specified, OpenCAL-GL automatically arranges the defined views inside the graphic window in a matrix layout.

The `calglMainLoop2D()` function (line 226) enters the application main loop. It corresponds to the GLUT main loop and executes both the simulation steps and the graphic rendering. As a result, the application execution flow never returns to the `main()` function and therefore, to manage memory deallocation and other

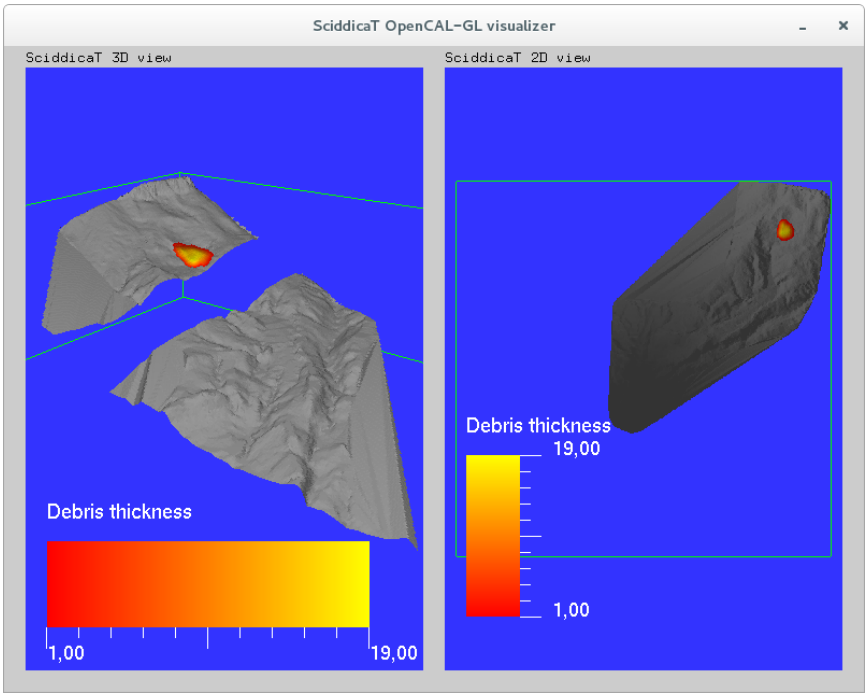


Figure 4.11: Screenshot of the SciddicaT debris flow model with a multi-view 2D and 3D visualization system based on OpenCAL-GL. First view is obtained by cutting off some rows and columns of the cellular space from representation.

possible operations, a callback function is defined and executed just before program termination, namely `exitFunction()` (lines 157-165). It was registered at line 173 through the `atexit()` function. Note that, if you do not want to visualize all the computational steps, you can call the `calglSetDisplayStep()` function to specify a refreshing interval in steps (cf. line 224).

Eventually, if you remove the comments at lines 214-216, you will obtain a partial view for the CA which, for instance, can be used to represent cross sections. In the specific case of SciddicaT, you will obtain the result shown in Figure 4.11. Calls at lines 214-215 only set the column index j to vary from 300 and the number of columns of the CA model by firstly removing all the indices form visualization (line 214) and then adding the interval [300, `draw_model3D->calModel->columns`]. Similarly, line 216 acts on the row index, i, by hiding from visualization the indices belonging to the interval [100, 150].

4.8.2 Implementing the *mod2* CA in OpenCAL and OpenCAL-GL

As for the previous example about SciddicaT, here we re-propose the *mod2* CA application to which we added an OpenCAL-GL viewer. The complete source code is shown in Listing 4.20 for the sake of completeness, while Figure 6.3 shows a

screenshot of the application.

```
1  // mod2 3D Cellular Automaton
2
3  #include <OpenCAL-GL/calgl3D.h>
4  #include <OpenCAL-GL/calgl3DWindow.h>
5  #include <stdio.h>
6  #include <stdlib.h>
7
8  #define ROWS 25
9  #define COLS 25
10 #define LAYERS 25
11 #define STEPS 4000
12
13 // declare CA, substate and simulation objects
14 struct CALModel3D* mod2;              //the cellular automaton
15 struct CALSubstate3Db *Q;             //the substate Q
16 struct CALRun3D* mod2_simulation;     //the simulartion run
17
18
19 // The cell's transition function (first and only elementary process)
20 void mod2TransitionFunction(struct CALModel3D* ca, int i, int j, int k)
21 {
22   int sum = 0, n;
23
24   for (n=0; n<ca->sizeof_X; n++)
25     sum += calGetX3Db(ca, Q, i, j, k, n);
26
27   calSet3Db(ca, Q, i, j, k, sum%2);
28 }
29
30 // Simulation init callback function used to set a seed at position (24, 0, 0)
31 void mod2SimulationInit(struct CALModel3D* ca)
32 {
33   //initializing substate to 0
34   calInitSubstate3Db(ca, Q, 0);
35   //setting a specific cell
36   calSet3Db(ca, Q, 24, 0, 0, 1);
37 }
38
39 // Stop condition callback function
40 CALbyte mod2SimulationStopCondition(struct CALModel3D* mod2)
41 {
42   if (mod2_simulation->step >= STEPS)
43     return CAL_TRUE;
44   return CAL_FALSE;
45 }
46
47 // Callback unction called just before program termination
48 void exitFunction(void)
49 {
50   //finalizations
51   calRunFinalize3D(mod2_simulation);
52   calFinalize3D(mod2);
53 }
54
55 // The main() function
56 int main(int argc, char** argv)
57 {
58   // Declare a viewer object
59   struct CALGLDrawModel3D* drawModel;
60
61   atexit(exitFunction);
62
63   //cadef and rundef
64   mod2 = calCADef3D(ROWS, COLS, LAYERS, CAL_MOORE_NEIGHBORHOOD_3D,
         CAL_SPACE_TOROIDAL, CAL_NO_OPT);
65   mod2_simulation = calRunDef3D(mod2, 1, 4001, CAL_UPDATE_IMPLICIT);
```

```
66    //add substates
67    Q = calAddSubstate3Db(mod2);
68    //add transition function's elementary processes
69    calAddElementaryProcess3D(mod2, mod2TransitionFunction);
70
71    //simulation run setup
72    calRunAddInitFunc3D(mod2_simulation, mod2SimulationInit);
73    calRunInitSimulation3D(mod2_simulation);  //It is required in the case the
          simulation main loop is explicitated; similarly for
          calRunFinalizeSimulation3D
74    calRunAddStopConditionFunc3D(mod2_simulation, mod2SimulationStopCondition);
75
76    // Initialize the viewer
77    calglInitViewer("mod2 3D CA viewer", 1.0f, 400, 400, 40, 40, CAL_TRUE, 1);
78    //drawModel definition
79    drawModel = calglDefDrawModel3D(CALGL_DRAW_MODE_FLAT, "3D view", mod2,
          mod2_simulation);
80    calglAdd3Db(drawModel, NULL, &Q, CALGL_TYPE_INFO_VERTEX_DATA,
          CALGL_TYPE_INFO_USE_NO_COLOR, CALGL_DATA_TYPE_DYNAMIC);
81    calglColor3D(drawModel, 0.5f, 0.5f, 0.5f, 1.0f);
82    calglAdd3Db(drawModel, Q, &Q, CALGL_TYPE_INFO_COLOR_DATA,
          CALGL_TYPE_INFO_USE_CURRENT_COLOR, CALGL_DATA_TYPE_DYNAMIC);
83    calglAdd3Db(drawModel, Q, &Q, CALGL_TYPE_INFO_NORMAL_DATA,
          CALGL_TYPE_INFO_USE_NO_COLOR, CALGL_DATA_TYPE_DYNAMIC);
84
85    // New functions for hide/display intervals of cells
86    //calglHideDrawKBound3D(drawModel, 0, drawModel->calModel->slices);
87    //calglDisplayDrawKBound3D(drawModel, 4, 10);
88    //calglDisplayDrawKBound3D(drawModel, 20, 25);
89    //calglHideDrawJBound3D(drawModel, 0, drawModel->calModel->columns);
90    //calglDisplayDrawJBound3D(drawModel, 2, 6);
91    //calglDisplayDrawJBound3D(drawModel, 18, 21);
92
93    calglMainLoop3D(argc, argv);
94
95    return 0;
96 }
```

Listing 4.20: An OpenCAL implementation of the mod2 3D CA with openCAL-GL graphic output.

As seen, here we consider the 3D implementation of OpenCAL-GL. In fact, the `OpenCAL-GL/calgl3D.h` and `OpenCAL-GL/calgl3DWindow.h` headers are included. Moreover, the OpenCAL-GL functions, already seen in the 2D version in previous sections, are here in their 3D form (see, e.g., the `calglAdd3Db()` function at line 80). They are completely equivalents to the 2D versions and therefore they will not be commented. Eventually, note that statements at lines 86-91 are commented. If you remove the comments, you will end with some parts of the cellular space cut down from visualization.

4.9 Reduction operations

OpenCAL comes with some functions you can use to perform global operations over the cellular space, which essentially are reduction functions. Note that such functions should not be used within elementary processes, but only in the initialization, steering and stop condition callback functions. The proper use of global functions is your own responsibility.

In order to use global functions, simply include the `cal2DReduction.h` header file for 2D CA, or `cal3DReduction.h` for 3D CA. Table 4.9 lists the OpenCAL global reduction functions. All of these functions accept a pointer to a CA object and a

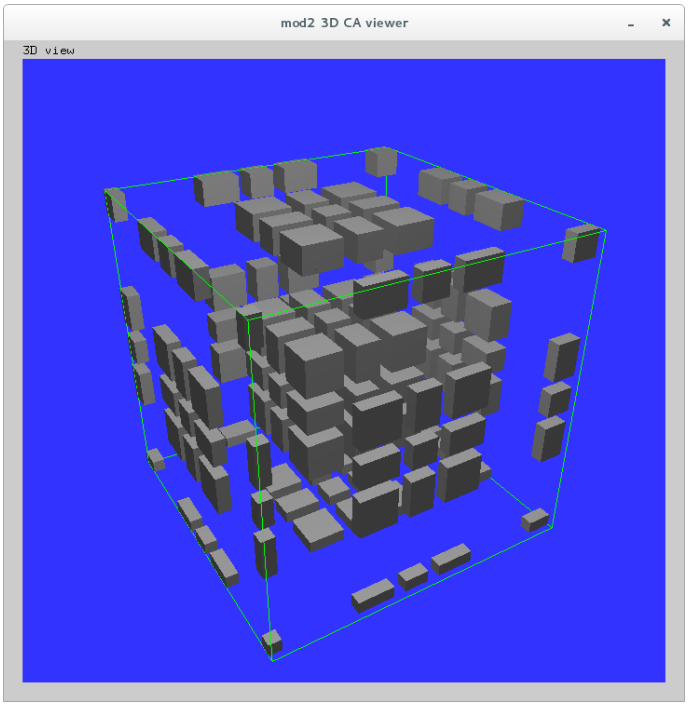


Figure 4.12: Screenshot of the *mod2* 3D CA viewer based on OpenCAL-GL.

Reduction Functions	Meaning
`calReductionComputeMax2Dr()`	Return the maximum value
`calReductionComputeMin2Dr()`	Return the minimum value
`calReductionComputeSum2Dr()`	Return the sum of all substate elements
`calReductionComputeProd2Dr()`	Return the product of all substate elements
`calReductionComputeLogicalAnd2Dr`	Return the logical AND of all substate elements
`calReductionComputeBinaryAnd2Dr()`	Return the binary AND of all substate elements
`calReductionComputeLogicalOr2Dr()`	Return the logical OR of all substate elements
`calReductionComputeBinaryOr2Dr()`	Return the binary OR of all substate elements
`alReductionComputeLogicalXor2Dr()`	Return the logical AND of all substate elements
`calReductionComputeBinaryXor2Dr()`	Return the binary AND of all substate elements

Table 4.9: OpenCAL global reduction functions for 2D CA and substates containing floating point values. In order to obtain the corresponding functions for 3D CA, you need to change 2D in 3D into the functions suffix, while to obtain the equivalent versions for the other supported basic data types, change the last suffix character from r to b or i, for CALbyte- or CALint-based substates, respectively.

```
// ...
#include <cal2DReduction.h>

// ...
void sciddicaTSteering(struct CALModel2D* sciddicaT)
{
  CALreal max_width;
  max_width = calReductionComputeMax2Dr(sciddicaT, Q.h);
  //...
}

// ...
```

Listing 4.21: An example of global reduction operation.

pointer to a substate over which execute the global operation as parameters, and return the value corresponding to the performed reduction. For instance, if you want to know the maximum value of the substate `Q.h` of the SciddicaT CA, you can write something like in Listing 4.21.

Chapter 5

OpenCAL OpenMP version

OpenCAL-OMP is the parallel OpenMP-based implementation OpenCAL, able to expoit all the processing elements on a shared memory machine. OpenCAL-OMP main structure and statements convention remain unchanged with respect to OpenCAL. Moreover, similarly to the serial version, OpenCAL-OMP allows for some *unsafe operations*, which can significantly speed up the application. However, the utmost attention must be paid to avoid*race condition* issues when unsafe operations are considered[1]. In the following Sections we will introduce OpenCAL-OMP by examples, highlighting differences with respect to the serial implementations presented in Chapter 4. In particular, Game of Life is firstly implemented. Subsequently, four different implementations of SciddicaT are illustrated, to show how simulation efficiency can be improved. The implementation of a simple 3D CA is also presented. The last part of the Chapter deals with OpenCAL-GL and shows how to integrate a basic OpenGL/GLUT visualization system in both 2D and 3D applications based on OpenCAL-OMP.

5.1 Conway's Game of Life in OpenCAL-OMP

In Section 4.4, we shown a possible OpenCAL implementation of Conway's Game of Life. Here, we present an OpenCAL-OMP implementation of the same cellular automaton, by discussing the differences with respect its serial implementation. The complete source code can be found in Listing 5.1.

As can be seen, the OpenCAL-OMP implementation of $Life$ is almost identical to the serial one thanks to the seamless parallelization adopted by the library. The only differences can be found at lines 3-5 where, instead of including the OpenCAL header files, the OpenCAL-OMP headers can be found. All the remaining source code is unchanged. In this specific case, besides considering the OpenCAL-OMP header files instead of the OpenCAL ones, no differences do exist between serial and parallel source codes.

[1] For instance, when many threads perform concurrent operations on the same memory locations and such operations are made by more than one atomic machine instruction, it can happen that they can interleave, giving rise to wrong (i.e., non consistent) results. Furthermore, even in the case of atomic operations, the logic order of execution could not be respected. Thus, for instance, a read-write logic sequence of atomic operations can actually become a write-read (wrong) sequence due to the fact that the thread performing the write operation is executed first.

```
1  // Conway's game of Life Cellular Automaton
2
3  #include <OpenCAL-OMP/cal2D.h>
4  #include <OpenCAL-OMP/cal2DIO.h>
5  #include <OpenCAL-OMP/cal2DRun.h>
6
7  // declare CA, substate and simulation objects
8  struct CALModel2D* life;
9  struct CALSubstate2Di* Q;
10 struct CALRun2D* life_simulation;
11
12 // The cell's transition function
13 void lifeTransitionFunction(struct CALModel2D* life, int i, int j)
14 {
15   int sum = 0, n;
16   for (n=1; n<life->sizeof_X; n++)
17     sum += calGetX2Di(life, Q, i, j, n);
18
19   if ((sum == 3) || (sum == 2 && calGet2Di(life, Q, i, j) == 1))
20     calSet2Di(life, Q, i, j, 1);
21   else
22     calSet2Di(life, Q, i, j, 0);
23 }
24
25 int main()
26 {
27   // define of the life CA and life_simulation simulation objects
28   life = calCADef2D(8, 16, CAL_MOORE_NEIGHBORHOOD_2D, CAL_SPACE_TOROIDAL,
         CAL_NO_OPT);
29   life_simulation = calRunDef2D(life, 1, 1, CAL_UPDATE_IMPLICIT);
30
31     //put OpenCAL - OMP in unsafe state execution(to allow unsafe operation to
           be used)
32   calSetUnsafe2D(life);
33
34   // add the Q substate to the life CA
35   Q = calAddSubstate2Di(life);
36
37   // add transition function's elementary process
38   calAddElementaryProcess2D(life, lifeTransitionFunction);
39
40   // set the whole substate to 0
41   calInitSubstate2Di(life, Q, 0);
42
43   // set a glider
44   calInit2Di(life, Q, 0, 2, 1);
45   calInit2Di(life, Q, 1, 0, 1);
46   calInit2Di(life, Q, 1, 2, 1);
47   calInit2Di(life, Q, 2, 1, 1);
48   calInit2Di(life, Q, 2, 2, 1);
49
50   // save the Q substate to file
51   calSaveSubstate2Di(life, Q, "./life_0000.txt");
52
53   // simulation run
54   calRun2D(life_simulation);
55
56   // save the Q substate to file
57   calSaveSubstate2Di(life, Q, "./life_LAST.txt");
58
59   // finalize simulation and CA objects
60   calRunFinalize2D(life_simulation);
61   calFinalize2D(life);
62
63   return 0;
64 }
```

Listing 5.1: An OpenCAL-OMP implementation of the Conway's game of Life.

5.2 SciddicaT

In this Section, the OpenCAL-OMP implementations of the four SciddicaT versions presented in Chapter 4 are shown.

5.2.1 SciddicaT naive implementation

As for the case of Conway's Game of Life, even the OpenCAL-OMP naive implementation of the SciddicaT cellular automaton, namely $SciddicaT_{naive}$, shown in Listing 5.2, does not significantly differ from the serial implementation (cf. Section 4.6). As for the *Life* example, differences only regard the included headers (lines 3-5), while the remaining source code is unchanged.

```
1  // The SciddicaT debris flows CCA simulation model
2
3  #include <OpenCAL-OMP/cal2D.h>
4  #include <OpenCAL-OMP/cal2DIO.h>
5  #include <OpenCAL-OMP/cal2DRun.h>
6  #include <stdlib.h>
7  #include <time.h>
8
9  // Some definitions...
10 #define ROWS 610
11 #define COLS 496
12 #define P_R 0.5
13 #define P_EPSILON 0.001
14 #define STEPS 4000
15 #define DEM_PATH "./data/dem.txt"
16 #define SOURCE_PATH "./data/source.txt"
17 #define OUTPUT_PATH "./data/width_final.txt"
18 #define NUMBER_OF_OUTFLOWS 4
19
20 // declare CCA model (sciddicaT), substates (Q), parameters (P),
21 // and simulation object (sciddicaT_simulation)
22 struct CALModel2D* sciddicaT;
23
24 struct sciddicaTSubstates {
25   struct CALSubstate2Dr *z;
26   struct CALSubstate2Dr *h;
27   struct CALSubstate2Dr *f[NUMBER_OF_OUTFLOWS];
28 } Q;
29
30 struct sciddicaTParameters {
31   CALParameterr epsilon;
32   CALParameterr r;
33 } P;
34
35 struct CALRun2D* sciddicaT_simulation;
36
37 // The sigma_1 elementary process
38 void sciddicaTFlowsComputation(struct CALModel2D* sciddicaT, int i, int j)
39 {
40   CALbyte eliminated_cells[5]={CAL_FALSE,CAL_FALSE,CAL_FALSE,CAL_FALSE,
         CAL_FALSE};
41   CALbyte again;
42   CALint cells_count;
43   CALreal average;
44   CALreal m;
45   CALreal u[5];
46   CALint n;
47   CALreal z, h;
48
49   if (calGet2Dr(sciddicaT, Q.h, i, j) <= P.epsilon)
50     return;
```

```
51
52    m = calGet2Dr(sciddicaT, Q.h, i, j) - P.epsilon;
53    u[0] = calGet2Dr(sciddicaT, Q.z, i, j) + P.epsilon;
54    for (n=1; n<sciddicaT->sizeof_X; n++)
55    {
56      z = calGetX2Dr(sciddicaT, Q.z, i, j, n);
57      h = calGetX2Dr(sciddicaT, Q.h, i, j, n);
58      u[n] = z + h;
59    }
60
61    //computes outflows
62    do{
63      again = CAL_FALSE;
64      average = m;
65      cells_count = 0;
66
67      for (n=0; n<sciddicaT->sizeof_X; n++)
68        if (!eliminated_cells[n]){
69          average += u[n];
70          cells_count++;
71        }
72
73        if (cells_count != 0)
74          average /= cells_count;
75
76        for (n=0; n<sciddicaT->sizeof_X; n++)
77          if( (average<=u[n]) && (!eliminated_cells[n]) ){
78            eliminated_cells[n]=CAL_TRUE;
79            again=CAL_TRUE;
80          }
81    }while (again);
82
83    for (n=1; n<sciddicaT->sizeof_X; n++)
84      if (eliminated_cells[n])
85        calSet2Dr(sciddicaT, Q.f[n-1], i, j, 0.0);
86      else
87        calSet2Dr(sciddicaT, Q.f[n-1], i, j, (average-u[n])*P.r);
88  }
89
90  // The sigma_2 elementary process
91  void sciddicaTWidthUpdate(struct CALModel2D* sciddicaT, int i, int j)
92  {
93    CALreal h_next;
94    CALint n;
95
96    h_next = calGet2Dr(sciddicaT, Q.h, i, j);
97    for(n=1; n<sciddicaT->sizeof_X; n++)
98      h_next +=  calGetX2Dr(sciddicaT, Q.f[NUMBER_OF_OUTFLOWS - n], i, j, n) -
           calGet2Dr(sciddicaT, Q.f[n-1], i, j);
99
100   calSet2Dr(sciddicaT, Q.h, i, j, h_next);
101 }
102
103 // SciddicaT simulation init function
104 void sciddicaTSimulationInit(struct CALModel2D* sciddicaT)
105 {
106   CALreal z, h;
107   CALint i, j;
108
109   //initializing substates to 0
110   calInitSubstate2Dr(sciddicaT, Q.f[0], 0);
111   calInitSubstate2Dr(sciddicaT, Q.f[1], 0);
112   calInitSubstate2Dr(sciddicaT, Q.f[2], 0);
113   calInitSubstate2Dr(sciddicaT, Q.f[3], 0);
114
115   //sciddicaT parameters setting
116   P.r = P_R;
117   P.epsilon = P_EPSILON;
```

```
118
119   //sciddicaT source initialization
120   for (i=0; i<sciddicaT->rows; i++)
121     for (j=0; j<sciddicaT->columns; j++)
122     {
123       h = calGet2Dr(sciddicaT, Q.h, i, j);
124
125       if ( h > 0.0 ) {
126         z = calGet2Dr(sciddicaT, Q.z, i, j);
127         calSet2Dr(sciddicaT, Q.z, i, j, z-h);
128       }
129     }
130 }
131
132 // SciddicaT steering function
133 void sciddicaTSteering(struct CALModel2D* sciddicaT)
134 {
135     //initializing substates to 0
136   calInitSubstate2Dr(sciddicaT, Q.f[0], 0);
137   calInitSubstate2Dr(sciddicaT, Q.f[1], 0);
138   calInitSubstate2Dr(sciddicaT, Q.f[2], 0);
139   calInitSubstate2Dr(sciddicaT, Q.f[3], 0);
140 }
141
142 int main()
143 {
144   time_t start_time, end_time;
145
146   //cadef and rundef
147   sciddicaT = calCADef2D (ROWS, COLS, CAL_VON_NEUMANN_NEIGHBORHOOD_2D,
          CAL_SPACE_TOROIDAL, CAL_NO_OPT);
148   sciddicaT_simulation = calRunDef2D(sciddicaT, 1, STEPS, CAL_UPDATE_IMPLICIT)
          ;
149
150   //add transition function's elementary processes
151   calAddElementaryProcess2D(sciddicaT, sciddicaTFlowsComputation);
152   calAddElementaryProcess2D(sciddicaT, sciddicaTWidthUpdate);
153
154   //add substates
155   Q.z = calAddSubstate2Dr(sciddicaT);
156   Q.h = calAddSubstate2Dr(sciddicaT);
157   Q.f[0] = calAddSubstate2Dr(sciddicaT);
158   Q.f[1] = calAddSubstate2Dr(sciddicaT);
159   Q.f[2] = calAddSubstate2Dr(sciddicaT);
160   Q.f[3] = calAddSubstate2Dr(sciddicaT);
161
162   //load configuration
163   calLoadSubstate2Dr(sciddicaT, Q.z, DEM_PATH);
164   calLoadSubstate2Dr(sciddicaT, Q.h, SOURCE_PATH);
165
166   //simulation run
167   calRunAddInitFunc2D(sciddicaT_simulation, sciddicaTSimulationInit);
168   calRunAddSteeringFunc2D(sciddicaT_simulation, sciddicaTSteering);
169   printf ("Starting simulation...\n");
170   start_time = time(NULL);
171   calRun2D(sciddicaT_simulation);
172   end_time = time(NULL);
173   printf ("Simulation terminated.\nElapsed time: %lds\n", end_time-start_time)
          ;
174
175   //saving configuration
176   calSaveSubstate2Dr(sciddicaT, Q.h, OUTPUT_PATH);
177
178   //finalizations
179   calRunFinalize2D(sciddicaT_simulation);
180   calFinalize2D(sciddicaT);
181
182   return 0;
```

```
183 }
```

Listing 5.2: An OpenCAL-OMP implementation of the $SciddicaT_{naive}$ debris flows simulation model.

5.2.2 SciddicaT with active cells optimization

The OpenCAL-OMP implementation of $SciddicaT_{ac}$, which takes advantage of the built-in OpenCAL active cells optimization, can be found in Listing 5.3. The corresponding serial implementation can be found in Section 4.6.2.

```
1  // The SciddicaT debris flows model with the active cells optimization
2
3  #include <OpenCAL-OMP/cal2D.h>
4  #include <OpenCAL-OMP/cal2DIO.h>
5  #include <OpenCAL-OMP/cal2DRun.h>
6  #include <OpenCAL-OMP/cal2DUnsafe.h>
7  #include <stdlib.h>
8  #include <time.h>
9
10 // Some definitions...
11 #define ROWS 610
12 #define COLS 496
13 #define P_R 0.5
14 #define P_EPSILON 0.001
15 #define STEPS 4000
16 #define DEM_PATH "./data/dem.txt"
17 #define SOURCE_PATH "./data/source.txt"
18 #define OUTPUT_PATH "./data/width_final.txt"
19 #define NUMBER_OF_OUTFLOWS 4
20
21 // declare CCA model (sciddicaT), substates (Q), parameters (P),
22 // and simulation object (sciddicaT_simulation)
23 struct sciddicaTSubstates {
24   struct CALSubstate2Dr *z;
25   struct CALSubstate2Dr *h;
26   struct CALSubstate2Dr *f[NUMBER_OF_OUTFLOWS];
27 } Q;
28
29 struct sciddicaTParameters {
30   CALParameterr epsilon;
31   CALParameterr r;
32 } P;
33
34
35 // The sigma_1 elementary process
36 void sciddicaTFlowsComputation(struct CALModel2D* sciddicaT, int i, int j)
37 {
38   CALbyte eliminated_cells[5]={CAL_FALSE,CAL_FALSE,CAL_FALSE,CAL_FALSE,
         CAL_FALSE};
39   CALbyte again;
40   CALint cells_count;
41   CALreal average;
42   CALreal m;
43   CALreal u[5];
44   CALint n;
45   CALreal z, h;
46   CALreal f;
47
48
49   m = calGet2Dr(sciddicaT, Q.h, i, j) - P.epsilon;
50   u[0] = calGet2Dr(sciddicaT, Q.z, i, j) + P.epsilon;
51   for (n=1; n<sciddicaT->sizeof_X; n++)
52   {
53     z = calGetX2Dr(sciddicaT, Q.z, i, j, n);
```

```
54      h = calGetX2Dr(sciddicaT, Q.h, i, j, n);
55      u[n] = z + h;
56    }
57
58    //computes outflows and updates debris thickness
59    do{
60      again = CAL_FALSE;
61      average = m;
62      cells_count = 0;
63
64      for (n=0; n<sciddicaT->sizeof_X; n++)
65        if (!eliminated_cells[n]){
66          average += u[n];
67          cells_count++;
68        }
69
70        if (cells_count != 0)
71          average /= cells_count;
72
73        for (n=0; n<sciddicaT->sizeof_X; n++)
74          if( (average<=u[n]) && (!eliminated_cells[n]) ){
75            eliminated_cells[n]=CAL_TRUE;
76            again=CAL_TRUE;
77          }
78
79    }while (again);
80
81    for (n=1; n<sciddicaT->sizeof_X; n++)
82      if (eliminated_cells[n])
83        calSet2Dr(sciddicaT, Q.f[n-1], i, j, 0.0);
84      else
85      {
86        calSet2Dr(sciddicaT, Q.f[n-1], i, j, (average-u[n])*P.r);
87        calAddActiveCellX2D(sciddicaT, i, j, n);
88      }
89  }
90
91  // The sigma_2 elementary process
92  void sciddicaTWidthUpdate(struct CALModel2D* sciddicaT, int i, int j)
93  {
94    CALreal h_next;
95    CALint n;
96
97    h_next = calGet2Dr(sciddicaT, Q.h, i, j);
98    for(n=1; n<sciddicaT->sizeof_X; n++)
99      h_next +=  calGetX2Dr(sciddicaT, Q.f[NUMBER_OF_OUTFLOWS - n], i, j, n) -
           calGet2Dr(sciddicaT, Q.f[n-1], i, j);
100
101   calSet2Dr(sciddicaT, Q.h, i, j, h_next);
102 }
103
104 // The sigma_3 elementary process
105 void sciddicaTRemoveInactiveCells(struct CALModel2D* sciddicaT, int i, int j)
106 {
107   if (calGet2Dr(sciddicaT, Q.h, i, j) <= P.epsilon)
108     calRemoveActiveCell2D(sciddicaT,i,j);
109 }
110
111
112 void sciddicaTSimulationInit(struct CALModel2D* sciddicaT)
113 {
114   CALreal z, h;
115   CALint i, j;
116
117   //sciddicaT parameters setting
118   P.r = P_R;
119   P.epsilon = P_EPSILON;
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   //sciddicaT source initialization
128   for (i=0; i<sciddicaT->rows; i++)
129     for (j=0; j<sciddicaT->columns; j++)
130     {
131       h = calGet2Dr(sciddicaT, Q.h, i, j);
132
133       if ( h > 0.0 ) {
134         z = calGet2Dr(sciddicaT, Q.z, i, j);
135         calSetCurrent2Dr(sciddicaT, Q.z, i, j, z-h);
136
137         //adds the cell (i, j) to the set of active ones
138         calAddActiveCell2D(sciddicaT, i, j);
139       }
140     }
141 }
142
143 // SciddicaT steering function
144 void sciddicaTSteering(struct CALModel2D* sciddicaT)
145 {
146   // set flow to 0 everywhere
147   calInitSubstate2Dr(sciddicaT, Q.f[0], 0);
148   calInitSubstate2Dr(sciddicaT, Q.f[1], 0);
149   calInitSubstate2Dr(sciddicaT, Q.f[2], 0);
150   calInitSubstate2Dr(sciddicaT, Q.f[3], 0);
151 }
152
153
154 int main()
155 {
156   time_t start_time, end_time;
157
158   // define of the sciddicaT CA and sciddicaT_simulation simulation objects
159   struct CALModel2D* sciddicaT = calCADef2D (ROWS, COLS,
          CAL_VON_NEUMANN_NEIGHBORHOOD_2D, CAL_SPACE_TOROIDAL,
          CAL_OPT_ACTIVE_CELLS);
160   struct CALRun2D* sciddicaT_simulation = calRunDef2D(sciddicaT, 1, STEPS,
          CAL_UPDATE_IMPLICIT);
161
162   // add transition function's sigma_1 and sigma_2 elementary processes
163   calAddElementaryProcess2D(sciddicaT, sciddicaTFlowsComputation);
164   calAddElementaryProcess2D(sciddicaT, sciddicaTWidthUpdate);
165   calAddElementaryProcess2D(sciddicaT, sciddicaTRemoveInactiveCells);
166
167   // add substates
168   Q.z = calAddSingleLayerSubstate2Dr(sciddicaT);
169   Q.h = calAddSubstate2Dr(sciddicaT);
170   Q.f[0] = calAddSubstate2Dr(sciddicaT);
171   Q.f[1] = calAddSubstate2Dr(sciddicaT);
172   Q.f[2] = calAddSubstate2Dr(sciddicaT);
173   Q.f[3] = calAddSubstate2Dr(sciddicaT);
174
175   // load configuration
176   calLoadSubstate2Dr(sciddicaT, Q.z, DEM_PATH);
177   calLoadSubstate2Dr(sciddicaT, Q.h, SOURCE_PATH);
178
179   // simulation run
180   calRunAddInitFunc2D(sciddicaT_simulation, sciddicaTSimulationInit);
181   calRunAddSteeringFunc2D(sciddicaT_simulation, sciddicaTSteering);
182   printf ("Starting simulation...\n");
183   start_time = time(NULL);
184   calRun2D(sciddicaT_simulation);
185   end_time = time(NULL);
```

```
186   printf ("Simulation terminated.\nElapsed time: %lds\n", end_time-start_time)
          ;
187
188   // saving configuration
189   calSaveSubstate2Dr(sciddicaT, Q.h, OUTPUT_PATH);
190
191   // finalizations
192   calRunFinalize2D(sciddicaT_simulation);
193   calFinalize2D(sciddicaT);
194
195   return 0;
196 }
```

Listing 5.3: An OpenCAL-OMP implementation of the SciddicaT debris flows simulation model with the active cells optimization.

Differently from the $SciddicaT_{naive}$ implementation shown in Listing 5.2, unsafe operations are here exploited due to the active cells optimization. Indeed, the `calAddActiveCellX2D()` function, called at line 87, adds a cell belonging to the neighborhood to the set A of active cells. As evident, more threads can try to add the same cell to the same location of the set A at the same time, by giving rise to a race condition[2]. In order to avoid race condition issues, OpenCAL-OMP is able to *lock* memory locations involved in concurrent non-atomic operations so that each thread can complete its own task without the risk other threads interfere. In order to do this, it is sufficient to place OpenCAL-OMP in *unsafe* state, by calling the `calSetUnsafe2D()` function, as done at line 163. Unsafe functions are provided by the `cal2DUnsafe.h` OpenCAL-OMP header file (line 6). No other modifications to the serial source code are required.

5.2.3 SciddicaT with direct neighbors update

The $SciddicaT_{ac+dnu}$ OpenCAL-OMP implementation of SciddicaT, which takes advantage of both the active cells optimization and the direct neighbors update, is shown in Listing 5.4. The corresponding serial implementation can be found in Section 4.6.3.

```
1  // The SciddicaT further optimized CCA debris flows model
2
3  #include <OpenCAL-OMP/cal2D.h>
4  #include <OpenCAL-OMP/cal2DIO.h>
5  #include <OpenCAL-OMP/cal2DRun.h>
6  #include <OpenCAL-OMP/cal2DUnsafe.h>
7  #include <stdlib.h>
8  #include <time.h>
9
10 // Some definitions...
11 #define ROWS 610
12 #define COLS 496
13 #define P_R 0.5
14 #define P_EPSILON 0.001
15 #define STEPS 4000
16 #define DEM_PATH "./data/dem.txt"
17 #define SOURCE_PATH "./data/source.txt"
18 #define OUTPUT_PATH "./data/width_final.txt"
19 #define NUMBER_OF_OUTFLOWS 4
20
```

[2] Actually, a thread-safe implementation could also be considered for $SciddicaT_{ac}$, becouse the active cells add/remove operations update 8 bit-long flag values in a working array transparently managed by OpenCAL-OMP, which are atomic operations. Unsefe operations are anyway here introduced for illustrative purposes.

```
21 // declare CCA model (sciddicaT), substates (Q), parameters (P),
22 // and simulation object (sciddicaT_simulation)
23 struct CALModel2D* sciddicaT;
24
25 struct sciddicaTSubstates {
26   struct CALSubstate2Dr *z;
27   struct CALSubstate2Dr *h;
28 } Q;
29
30 struct sciddicaTParameters {
31   CALParameterr epsilon;
32   CALParameterr r;
33 } P;
34
35 struct CALRun2D* sciddicaT_simulation;
36
37
38 // The sciddicaT transition function
39 void sciddicaTFlowsComputation(struct CALModel2D* sciddicaT, int i, int j)
40 {
41   CALbyte eliminated_cells[5]={CAL_FALSE,CAL_FALSE,CAL_FALSE,CAL_FALSE,
         CAL_FALSE};
42   CALbyte again;
43   CALint cells_count;
44   CALreal average;
45   CALreal m;
46   CALreal u[5];
47   CALint n;
48   CALreal z, h;
49   CALreal f;
50
51
52   m = calGet2Dr(sciddicaT, Q.h, i, j) - P.epsilon;
53   u[0] = calGet2Dr(sciddicaT, Q.z, i, j) + P.epsilon;
54   for (n=1; n<sciddicaT->sizeof_X; n++)
55   {
56     z = calGetX2Dr(sciddicaT, Q.z, i, j, n);
57     h = calGetX2Dr(sciddicaT, Q.h, i, j, n);
58     u[n] = z + h;
59   }
60
61   //computes outflows and updates debris thickness
62   do{
63     again = CAL_FALSE;
64     average = m;
65     cells_count = 0;
66
67     for (n=0; n<sciddicaT->sizeof_X; n++)
68       if (!eliminated_cells[n]){
69         average += u[n];
70         cells_count++;
71       }
72
73       if (cells_count != 0)
74         average /= cells_count;
75
76       for (n=0; n<sciddicaT->sizeof_X; n++)
77         if( (average<=u[n]) && (!eliminated_cells[n]) ){
78           eliminated_cells[n]=CAL_TRUE;
79           again=CAL_TRUE;
80         }
81
82   }while (again);
83
84   for (n=1; n<sciddicaT->sizeof_X; n++)
85     if (!eliminated_cells[n])
86     {
87       f = (average-u[n])*P.r;
```

```
88        calAddNext2Dr(sciddicaT,Q.h,i,j,-f);
89        calAddNextX2Dr(sciddicaT,Q.h,i,j,n,f);
90
91        //adds the cell (i, j, n) to the set of active ones
92        calAddActiveCellX2D(sciddicaT, i, j, n);
93      }
94  }
95
96
97  void sciddicaTRemoveInactiveCells(struct CALModel2D* sciddicaT, int i, int j)
98  {
99    if (calGet2Dr(sciddicaT, Q.h, i, j) <= P.epsilon)
100     calRemoveActiveCell2D(sciddicaT,i,j);
101 }
102
103
104 void sciddicaTSimulationInit(struct CALModel2D* sciddicaT)
105 {
106   CALreal z, h;
107   CALint i, j;
108
109   //sciddicaT parameters setting
110   P.r = P_R;
111   P.epsilon = P_EPSILON;
112
113   //sciddicaT source initialization
114   for (i=0; i<sciddicaT->rows; i++)
115     for (j=0; j<sciddicaT->columns; j++)
116     {
117       h = calGet2Dr(sciddicaT, Q.h, i, j);
118
119       if ( h > 0.0 ) {
120         z = calGet2Dr(sciddicaT, Q.z, i, j);
121         calSetCurrent2Dr(sciddicaT, Q.z, i, j, z-h);
122
123         //adds the cell (i, j) to the set of active ones
124         calAddActiveCell2D(sciddicaT, i, j);
125       }
126     }
127 }
128
129
130 int main()
131 {
132   time_t start_time, end_time;
133
134   // define of the sciddicaT CA and sciddicaT_simulation simulation objects
135   sciddicaT = calCADef2D (ROWS, COLS, CAL_VON_NEUMANN_NEIGHBORHOOD_2D,
          CAL_SPACE_TOROIDAL, CAL_OPT_ACTIVE_CELLS);
136   sciddicaT_simulation = calRunDef2D(sciddicaT, 1, STEPS, CAL_UPDATE_IMPLICIT)
          ;
137
138   //put OpenCAL - OMP in unsafe state execution(to allow unsafe operation to
          be used)
139   calSetUnsafe2D(sciddicaT);
140
141
142   // add transition function's sigma_1 and sigma_2 elementary processes
143   calAddElementaryProcess2D(sciddicaT, sciddicaTFlowsComputation);
144   calAddElementaryProcess2D(sciddicaT, sciddicaTRemoveInactiveCells);
145
146   // add substates
147   Q.z = calAddSingleLayerSubstate2Dr(sciddicaT);
148   Q.h = calAddSubstate2Dr(sciddicaT);
149
150   // load configuration
151   calLoadSubstate2Dr(sciddicaT, Q.z, DEM_PATH);
152   calLoadSubstate2Dr(sciddicaT, Q.h, SOURCE_PATH);
```

```
153
154   // simulation run
155   calRunAddInitFunc2D(sciddicaT_simulation, sciddicaTSimulationInit);
156   printf ("Starting simulation...\n");
157   start_time = time(NULL);
158   calRun2D(sciddicaT_simulation);
159   end_time = time(NULL);
160   printf ("Simulation terminated.\nElapsed time: %lds\n", end_time-start_time)
          ;
161
162   // saving configuration
163   calSaveSubstate2Dr(sciddicaT, Q.h, OUTPUT_PATH);
164
165   // finalizations
166   calRunFinalize2D(sciddicaT_simulation);
167   calFinalize2D(sciddicaT);
168
169   return 0;
170 }
```

Listing 5.4: An OpenCAL-OMP implementation of the $SciddicaT_{ac+dnu}$ debris flows XCA simulation model with direct neighbors update.

As for the serial implementation, only topographic altitude and debris thickness are considered as substates (lines 25-28, 147-148), since outflows substates are no longer needed. In fact, mass balance is here obtained by adding outflows to the neighbors and subtracting them from the central cell, as soon as they have been computed, in the *next* computing plane, which therefore acts as an accumulation layer. Moreover, only two elementary processes are now necessary (cf. lines 143-144), instead of the three considered for the previous versions of SciddicaT.

Besides the active cells optimization, even the direct neighbors update requires unsafe opetations. For this purpose, the `cal2DUnsafe.h` header file is included at line 6, and the `calSetUnsafe2D()` function called at line 139 to places OpenCAL-OMP in unsafe mode and avoid race condition issues. Besides the already discussed `calAddActiveCellX2D()` function, the `calAddNext2Dr()` and `calAddNextX2Dr()` unsafe functions are here employed (lines 88-89), in place of the combination of get-set operations, as done in the corresponding serial implementation (Listing 4.16, lines 84-85). In fact, let's consider the source code snippet in Listing 5.5 (checked out by Listing 4.16). As it can be seen, for each not-eliminated cell, the algorithm computes a flow, f (line 5) and then subtracts it from the central cell (line 6), adding it to the corresponding neighbour (line 7), in order to accomplish mass balance. In both cases (flow subtraction and adding), a flavor of `calGet` function is called to read the current value of the Q_h substate from the next working plane. Subsequently, a flavor of the `calSet` function is used to update the previously read value. When a single thread is used to perform such operations, no race conditions can obviously occur. At the contrary, even in the case of two concurrent threads, different undesirable situations can take place, which give rise to a race condition and therefore to a wrong result. For instance, let's suppose both threads read the value first, and then write their updated values; in this case, the resulting value will correspond to the one written by the thread that writes the value for last, and the contribution of the other thread is lost. In order to avoid such kind of problems when dealing with more threads, the above mentioned `calAddNext2Dr()` and `calAddNextX2Dr()` functions have to be used, since they lock the cell under consideration and then perform the get-set operations without the risk that other threads can interfere. In this way, no race conditions can be triggered. Obviously, there is a side-effect in terms of computational performance. In fact, as expected, locks can slow down threads

```
1   // <snip>
2   for (n=1; n<sciddicaT->sizeof_X; n++)
3   if (!eliminated_cells[n])
4   {
5     f = (average-u[n])*P.r;
6     calSet2Dr (sciddicaT,Q.h,i,j,  calGetNext2Dr (sciddicaT,Q.h,i,j)  -f);
7     calSetX2Dr(sciddicaT,Q.h,i,j,n,calGetNextX2Dr(sciddicaT,Q.h,i,j,n)+f);
8     // <snip>
9   }
10  // <snip>
```

Listing 5.5: Example of non atomic operation made of a combination of get-set calls.

execution and therefore the entire simulation.

5.2.4 SciddicaT with explicit simulation loop

As for the serial version, also for the OpenMP based release of OpenCAL it is further possible to improve computational performances of SciddicaT by avoiding unnecessary substates updating. As already reported, the `calRun2D()` function used so far to run the simulation loop updates all the defined substates at the end of each elementary process. However, in the specific case of the SciddicaT model, no substates updating should be executed after the application of the second elementary process, as this just removes inactive cells from the set A.

Listing 5.6 presents the $SciddicaT_{ac+dnu+esl}$ OpenCAL-OMP implementation of SciddicaT, based on an explicit global transition function, besides active cells optimization and direct neighbors update. The explicit global transition function is defined by means of `calRunAddGlobalTransitionFunc2D()`. It registers a callback function within which it can be used to both reorder the sequence of elementary processes to be applied in the generic computational step, and also to perform only necessary substates updates. The $SciddicaT_{ac+dnu+esl}$ implementation presented in Listing 5.6 also makes the simulation loop explicit and defines a stopping criterion for the simulation termination.

```
1   // The SciddicaT further optimized CCA debris flows model
2
3   #include <OpenCAL-OMP/cal2D.h>
4   #include <OpenCAL-OMP/cal2DIO.h>
5   #include <OpenCAL-OMP/cal2DRun.h>
6   #include <OpenCAL-OMP/cal2DUnsafe.h>
7   #include <stdlib.h>
8   #include <time.h>
9
10  // Some definitions...
11  #define ROWS 610
12  #define COLS 496
13  #define P_R 0.5
14  #define P_EPSILON 0.001
15  #define STEPS 4000
16  #define DEM_PATH "./data/dem.txt"
17  #define SOURCE_PATH "./data/source.txt"
18  #define OUTPUT_PATH "./data/width_final.txt"
19  #define NUMBER_OF_OUTFLOWS 4
20
21  // declare CCA model (sciddicaT), substates (Q), parameters (P),
22  // and simulation object (sciddicaT_simulation)
23  struct CALModel2D* sciddicaT;
24
```

```
25  struct sciddicaTSubstates {
26    struct CALSubstate2Dr *z;
27    struct CALSubstate2Dr *h;
28  } Q;
29
30  struct sciddicaTParameters {
31    CALParameterr epsilon;
32    CALParameterr r;
33  } P;
34
35  struct CALRun2D* sciddicaT_simulation;
36
37
38  // The sciddicaT transition function
39  void sciddicaTFlowsComputation(struct CALModel2D* sciddicaT, int i, int j)
40  {
41    CALbyte eliminated_cells[5]={CAL_FALSE,CAL_FALSE,CAL_FALSE,CAL_FALSE,
          CAL_FALSE};
42    CALbyte again;
43    CALint cells_count;
44    CALreal average;
45    CALreal m;
46    CALreal u[5];
47    CALint n;
48    CALreal z, h;
49    CALreal f;
50
51
52    m = calGet2Dr(sciddicaT, Q.h, i, j) - P.epsilon;
53    u[0] = calGet2Dr(sciddicaT, Q.z, i, j) + P.epsilon;
54    for (n=1; n<sciddicaT->sizeof_X; n++)
55    {
56      z = calGetX2Dr(sciddicaT, Q.z, i, j, n);
57      h = calGetX2Dr(sciddicaT, Q.h, i, j, n);
58      u[n] = z + h;
59    }
60
61    //computes outflows and updates debris thickness
62    do{
63      again = CAL_FALSE;
64      average = m;
65      cells_count = 0;
66
67      for (n=0; n<sciddicaT->sizeof_X; n++)
68        if (!eliminated_cells[n]){
69          average += u[n];
70          cells_count++;
71        }
72
73        if (cells_count != 0)
74          average /= cells_count;
75
76        for (n=0; n<sciddicaT->sizeof_X; n++)
77          if( (average<=u[n]) && (!eliminated_cells[n]) ){
78            eliminated_cells[n]=CAL_TRUE;
79            again=CAL_TRUE;
80          }
81
82    }while (again);
83
84    for (n=1; n<sciddicaT->sizeof_X; n++)
85      if (!eliminated_cells[n])
86      {
87        f = (average-u[n])*P.r;
88        calAddNext2Dr(sciddicaT,Q.h,i,j,-f);
89        calAddNextX2Dr(sciddicaT,Q.h,i,j,n,f);
90
91        //adds the cell (i, j, n) to the set of active ones
```

```
92        calAddActiveCellX2D(sciddicaT, i, j, n);
93      }
94  }
95
96
97  void sciddicaTRemoveInactiveCells(struct CALModel2D* sciddicaT, int i, int j)
98  {
99    if (calGet2Dr(sciddicaT, Q.h, i, j) <= P.epsilon)
100     calRemoveActiveCell2D(sciddicaT,i,j);
101 }
102
103
104 void sciddicaTSimulationInit(struct CALModel2D* sciddicaT)
105 {
106   CALreal z, h;
107   CALint i, j;
108
109   //sciddicaT parameters setting
110   P.r = P_R;
111   P.epsilon = P_EPSILON;
112
113   //sciddicaT source initialization
114   for (i=0; i<sciddicaT->rows; i++)
115     for (j=0; j<sciddicaT->columns; j++)
116     {
117       h = calGet2Dr(sciddicaT, Q.h, i, j);
118
119       if ( h > 0.0 ) {
120         z = calGet2Dr(sciddicaT, Q.z, i, j);
121         calSetCurrent2Dr(sciddicaT, Q.z, i, j, z-h);
122
123         //adds the cell (i, j) to the set of active ones
124         calAddActiveCell2D(sciddicaT, i, j);
125       }
126     }
127
128   calUpdateActiveCells2D(sciddicaT);
129 }
130
131
132 void sciddicaTransitionFunction(struct CALModel2D* sciddicaT)
133 {
134   // active cells must be updated first becouse outflows
135   // have already been sent to (pheraps inactive) the neighbours
136   calApplyElementaryProcess2D(sciddicaT, sciddicaTFlowsComputation);
137     calUpdateActiveCells2D(sciddicaT);
138     calUpdateSubstate2Dr(sciddicaT, Q.h);
139
140   // here you don't need to update Q.h
141   calApplyElementaryProcess2D(sciddicaT, sciddicaTRemoveInactiveCells);
142   calUpdateActiveCells2D(sciddicaT);
143 }
144
145
146 CALbyte sciddicaTSimulationStopCondition(struct CALModel2D* sciddicaT)
147 {
148   if (sciddicaT_simulation->step >= STEPS)
149     return CAL_TRUE;
150   return CAL_FALSE;
151 }
152
153 void run(struct CALRun2D* simulation)
154 {
155   CALbyte again;
156
157   calRunInitSimulation2D(simulation);
158
159   do{
```

```
160       again = calRunCAStep2D(simulation);
161       simulation->step++;
162     } while (again);
163
164     calRunFinalizeSimulation2D(simulation);
165   }
166
167   int main()
168   {
169     time_t start_time, end_time;
170
171     // define of the sciddicaT CA and sciddicaT_simulation simulation objects
172     sciddicaT = calCADef2D (ROWS, COLS, CAL_VON_NEUMANN_NEIGHBORHOOD_2D,
            CAL_SPACE_TOROIDAL, CAL_OPT_ACTIVE_CELLS);
173     sciddicaT_simulation = calRunDef2D(sciddicaT, 1, CAL_RUN_LOOP,
            CAL_UPDATE_EXPLICIT);
174
175     //put OpenCAL - OMP in unsafe state execution(to allow unsafe operation to
            be used)
176     calSetUnsafe2D(sciddicaT);
177
178     // add transition function's sigma_1 and sigma_2 elementary processes
179     calAddElementaryProcess2D(sciddicaT, sciddicaTFlowsComputation);
180     calAddElementaryProcess2D(sciddicaT, sciddicaTRemoveInactiveCells);
181
182     // add substates
183     Q.z = calAddSingleLayerSubstate2Dr(sciddicaT);
184     Q.h = calAddSubstate2Dr(sciddicaT);
185
186     // load configuration
187     calLoadSubstate2Dr(sciddicaT, Q.z, DEM_PATH);
188     calLoadSubstate2Dr(sciddicaT, Q.h, SOURCE_PATH);
189
190     // simulation run
191     calRunAddInitFunc2D(sciddicaT_simulation, sciddicaTSimulationInit);
192     calRunAddGlobalTransitionFunc2D(sciddicaT_simulation,
            sciddicaTransitionFunction);
193     calRunAddStopConditionFunc2D(sciddicaT_simulation,
            sciddicaTSimulationStopCondition);
194
195     printf ("Starting simulation...\n");
196     start_time = time(NULL);
197     run(sciddicaT_simulation);
198     end_time = time(NULL);
199     printf ("Simulation terminated.\nElapsed time: %lds\n", end_time-start_time)
            ;
200
201     // saving configuration
202     calSaveSubstate2Dr(sciddicaT, Q.h, OUTPUT_PATH);
203
204     // finalizations
205     calRunFinalize2D(sciddicaT_simulation);
206     calFinalize2D(sciddicaT);
207
208     return 0;
209   }
```

Listing 5.6: An OpenCAL-OMP implementation of the SciddicaT debris flows simulation model with explicit simulation loop.

5.2.5 SciddicaT computational performance

Table 5.1 resumes computational performance of all the above illustrated SciddicaT implementations as implemented in OpenCAL-OMP. The considered case of study is the simulation of the Tessina landslide shown in Figure 4.5, which required a

Version	Serial [s]	1thr	2thr	4thr	6thr	8thr
$SciddicaT_{naive}$	240s	0.82 (293s)	1.22 (196s)	1.53 (157s)	1.64 (146s)	1.6 (150s)
$SciddicaT_{ac}$	23s	0.77 (30s)	1.36 (17s)	1.77 (13s)	2.09 (11s)	2.3 (10s)
$SciddicaT_{ac+dnu}$	13s	0.77 (17s)	1.86 (7s)	2.6 (5s)	2.17 (6s)	2.6 (5s)
$SciddicaT_{ac+dnu+esl}$	12s	0.75 (16s)	1.2 (10s)	2.4 (5s)	2.4 (5s)	3.0 (4s)

Table 5.1: Speedups and elapsed times (in brackets) of the four different OpenCAL-OMP implementations of the SciddicaT debris flows model. Elapsed times of the corresponding serial versions are reported in the second column for comparison.

total of 4000 computational steps. The adopted CPU is a Intel Core i7-4702HQ @ 2.20GHz 4 cores (8 threads) processor, already considered for the performance evaluation of the corresponding serial SciddicaT implementations described in Chapter 4. Results are provided both in terms of elapsed time and speedup with respect to the corresponding serial version. Elapsed times of the serial simulations are also reported.

As noted, results are quite good. In particular, the better results in terms of speed up were obtained for the fully optimized SciddicaT implementation (i.e., with the explicit substate updating feature), which runs 3 times faster than the corresponding serial version when executed over 8 threads. Nevertheless, consider that the SciddicaT simulation model here adopted is quite simple and better performance in terms of speed up can certainly be obtained for CA models with more complex transition functions and more extended computational domains.

Eventually, note how progressive optimizations can considerably reduce the overall execution time. In fact, if for the naive (i.e., non optimized at all) serial implementation the elapsed time was 240s, for the fully optimized parallel version the simulation lasted only 3 seconds, corresponding to a speed up value of 80, i.e. the fully optimized parallel version runs 80 times faster than the serial naive implementation.

5.3 A three-dimensional example

In Section 4.7, we described the *mod2* 3D CA and shown a possible OpenCAL implementation. Here, we briefly present an OpenCAL-OMP implementation of the same cellular automaton in Listing 5.1, by discussing the differences with respect the corresponding serial implementation.

As seen, the *mod2* OpenCAL-OMP implementation is almost identical to the serial one. As for the case of Game of Life, the only differences can be found at lines 3-5 where OpenCAL-OMP headers can be found instead of the OpenCAL onces. All the remaining source code is unchanged.

5.4 OpenCAL-GL and global functions

As for OpenCAL, it is possible to exploit OpenCAL-GL to have a simple visualization system by adding few lines of code to your application. Combining OpenCAL-OMP and OpenCAL-GL does not differ from what we have done in Section 4.8 for OpenCAL and OpenCAL-GL. Therefore, please refer to this section for major details.

Similarly, you can also use the same global reduction functions described in Section 4.9 in OpenCAL-OMP, by considering the OpenCAL-OMP `cal2DReduction.h`

```
1  // mod2 3D Cellular Automaton
2
3  #include <OpenCAL-OMP/cal3D.h>
4  #include <OpenCAL-OMP/cal3DIO.h>
5  #include <OpenCAL-OMP/cal3DRun.h>
6
7  #define ROWS 5
8  #define COLS 7
9  #define LAYERS 3
10
11 // declare CA, substate and simulation objects
12 struct CALModel3D* mod2;
13 struct CALSubstate3Db* Q;
14 struct CALRun3D* mod2_simulation;
15
16 // The cell's transition function
17 void mod2TransitionFunction(struct CALModel3D* ca, int i, int j, int k)
18 {
19   int sum = 0, n;
20
21   for (n=0; n<ca->sizeof_X; n++)
22     sum += calGetX3Db(ca, Q, i, j, k, n);
23
24   calSet3Db(ca, Q, i, j, k, sum%2);
25 }
26
27 int main()
28 {
29   // define of the mod2 CA and mod2_simulation simulation objects
30   mod2 = calCADef3D(ROWS, COLS, LAYERS, CAL_MOORE_NEIGHBORHOOD_3D,
         CAL_SPACE_TOROIDAL, CAL_NO_OPT);
31   mod2_simulation = calRunDef3D(mod2, 1, 1, CAL_UPDATE_IMPLICIT);
32
33   // add the Q substate to the mod2 CA
34   Q = calAddSubstate3Db(mod2);
35
36   // add transition function's elementary process
37   calAddElementaryProcess3D(mod2, mod2TransitionFunction);
38
39   // set the whole substate to 0
40   calInitSubstate3Db(mod2, Q, 0);
41
42   // set a seed at position (2, 3, 1)
43   calInit3Db(mod2, Q, 2, 3, 1, 1);
44
45   // save the Q substate to file
46   calSaveSubstate3Db(mod2, Q, "./mod2_0000.txt");
47
48   // simulation run
49   calRun3D(mod2_simulation);
50
51   // save the Q substate to file
52   calSaveSubstate3Db(mod2, Q, "./mod2_LAST.txt");
53
54   // finalize simulation and CA objects
55   calRunFinalize3D(mod2_simulation);
56   calFinalize3D(mod2);
57
58   return 0;
59 }
```

Listing 5.7: An OpenCAL-OMP implementation of the mod2 CA.

and `cal3DReduction.h` specific header for 2D and 3D CA, respectively. Please refer to this section for further details.

Chapter 6

OpenCAL OpenCL version

This chapter introduces OpenCAL-CL, a porting of OpenCAL in OpenCL. OpenCL is an open standard for parallel programming defined by the Khronos Group, that also defines other open standards like OpenGL and Vulkan. Besides computational efficiency, one of the main advantages of OpenCL is portability. In fact, OpenCL allows to run programs across heterogeneous processors, like Central Processing Units (CPUs), Graphics Processing Units (GPUs), Digital Signal Processors (DSPs), and Field-Programmable Gate Arrays (FPGAs).

OpenCAL-CL inherits many OpenCAL features, by also adding parallel computation capability thanks to the adoption of OpenCL. Statements convention is similar to the one adopted in OpenCAL and OpenCAL-OMP, with the peculiarity that the `calcl` and `CALCL` prefixes are adopted for functions and data types, respectively, while the `CALCL_` prefix is used for constants. The active cells optimization is fully supported, while a main difference with respect to OpenCAL and OpenCAL-OMP is that OpenCAL-CL does not support unsafe operations in the current version. Another main difference is that the application is subdivided in two parts, one running on the CPU and one running in parallel on a so called compliant device, like the ones cited above. However, parallelism is almost transparent to the user.

This Chapter introduces OpenCAL-CL by examples. After a brief introduction to both OpenCL and OpenCAL-CL, in which both host and device-side CA development is outlined, the implementation of the $Life$ and $SciddicaT_{naive}$ 2D CA is presented. The implementation of the *mod*2 3D CA is also shown. The last two examples also deal with OpenCAL-GL and show how to integrate a basic OpenGL/GLUT visualization system in both 2D and 3D OpenCAL-CL applications.

6.1 A brief introduction to OpenCL and OpenCAL-CL

In order to better understand how OpenCAL-CL works, it is necessary to introduce some OpenCL concepts. In OpenCL, data exchange and kernels execution are managed thanks to an OpenCL *context*. In particular, the host application links kernels into one or more containers, called *programs*. The program therefore connects kernels with the data to be processed and dispatches them to a special OpenCL structure called *command queue*. This is necessary because only enqueued kernels are actually executed. Figure 6.1 shows the general structure of an OpenCL application. The context contains all the devices, command queues and kernels; each device

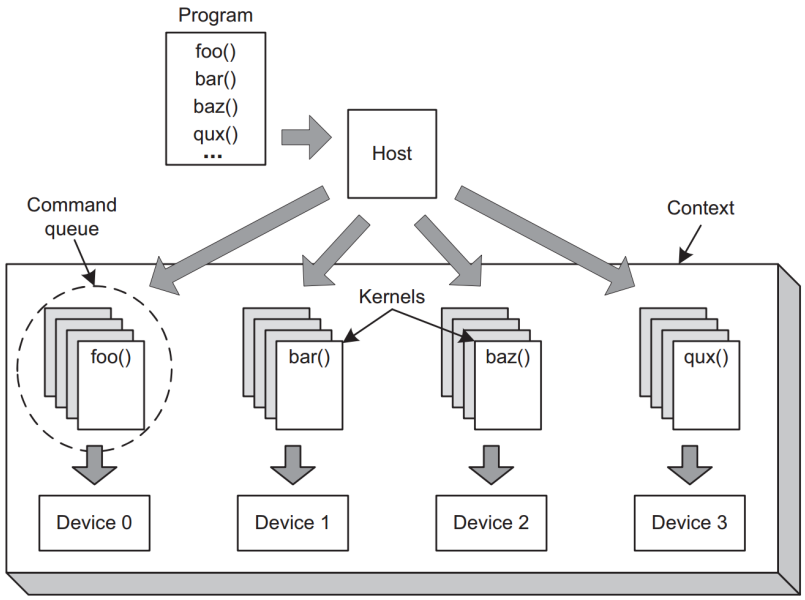


Figure 6.1: General structure of a OpenCL program, from *OpenCL in Action: How to accelerate graphics and computations*, Matthew Scarpino, Manning Pubblication, ISBN-13: 860-1400825129.

has its own command queue and each command queue contains the kernels to be executed on the corresponding device.

As already stated, OpenCAL-CL, as well as OpenCL based applications, are subdivided in two parts: the *host application*, running on the CPU, and the *device application*, running on a compliant computational device (e.g. a Nvidia or AMD GPU). Two CA objects are therefore considered, one allocated host-side, while the other on the compliant device. The host CA object is defined as in OpenCAL and OpenCAL-OMP, while elementary processes, and possibly other global functions like init, steering, and stop condition functions, are registered to the device CA, to be executed in parallel. Accordingly, they must be defined as OpenCL *kernels*. As a consequence, developers have to be able to write some minimal OpenCL code to implement them. However, OpenCAL-CL hides lots of parallel aspects to the user (e.g., the simulation loop is internally managed by the library) and also simplifies data exchange between host and device.

As regards computational aspects, a number of threads equal to the number of CA cells are executed in parallel for each computational step. Each thread applies the CA elementary processes on a specific cell, in the order they have been registered to the device-side CA. In other words, OpenCAL-CL adopts the so called one thread-one cell execution policy. According to OpenCL, threads are grouped into workgroups. Threads within a workgroup can share information on a local memory and also synchronize each other. Moreover, global data can be shared thanks to the device global memory, which however is slower than the local one. Eventually, threads can

be globally synchronized when kernels execution terminates and the control returns to the host application.

In OpenCAL-CL many of the above aspects are completely transparent to the user. In particular, the number of threads to be executed is set by the library, as well as threads grouping into workgroups. Eventually, even data exchange between host and device is completely transparent to the user, with the exception that kernels have to receive data which was not registered into the host-side CA model.

6.1.1 OpenCAL-CL device-side Programming

As already mentioned, functions that can be executed on an OpenCL compliant device are called kernels. In order to explain how to write an OpenCAL-CL based kernel, let's consider the example in Listing 6.1.

First of all, the `calcl2D.h` header file must be included (line 1). According to OpenCL, each kernel definition must start with the `__kernel` keyword (line 8). However, differently to OpenCL, where a kernel can have no parameters, each OpenCAL-CL kernel must have at least the `__CALCL_MODEL_2D` parameter. Actually, it is a macro-like C object defining a list of pre-fixed typed parameters, needed to let the kernel to know any data about the CA model, like substates and the neighbourhood, beside others. Kernels can also take other parameters, as in the example in Listing 6.2. In the specific case, the `Pepsilon` parameter is allocated in the global memory of the compliant device. It is also possible to allocate kernel parameters in the faster local memory, which is shared by threads belonging to the same work group, by means of the OpenCL `__local` keyword. Eventually, kernel parameters can also be private to the current thread by means of the `__private` keyword, as shown in Listing 6.3. The same holds for variables defined inside the kernel body where the above cited `__global`, `__local` and `_private` memory level qualifiers can be used. Note that both kernel parameters and variables declared without any memory level qualifier are implicitly considered as private, as in the case of variables at lines 15 and 16 of Listing 6.1.

The `calclThreadCheck2D()` function at line 11 of Listing 6.1 must be the first to be called into the kernel, as it prevents the execution of a number of threads exceeding the number of CA cells (Listing 6.1, line 11). This is due to the fact that OpenCAL-CL can execute more threads than the number of CA cells for better performance purposes. In the case the active cells optimization is considered, the `calclActiveThreadCheck2D()` must be called instead of `calclThreadCheck2D()`, to prevent the execution of a number of threads exceeding the number of CA cells actually involved in computation. The `calclGlobalRow()` and `calclGlobalColumn()` functions (Listing 6.1, lines 15-16) are used to get the global cell coordinates within the cellular space. The `calclSet2Dr()` function at line 23 updates the substate value for the central cell. It is the kernel-side counterpart of the OpenCAL `calSet2Dr()` cell update functions. A complete list of functions that can be used within OpenCAL-CL kernels are listed in Table 6.1.

Note that, functions that in OpenCAL were requiring a pointer to a CA object, here take the `MODEL_2D` macro-like C object in its place, which implicitly defines a list of prefixed parameters needed by the function (see e.g. line 23 of Listing 6.1). Moreover, substates are accessed by means of numerical handles (cf. the second parameter of the `calclSet2Dr()` function at line 23), which have to be previously defined in the kernel (cf. line 6). The criterion to be adopted is very simple: handles are zero-based IDs. That is, zero (0) is used to link the first host-side substate, i.e. the

```
#include <OpenCAL-CL/calcl2D.h>

// Define handles to CA substates
//...
#define Z 4
#define H 5

__kernel void calcl_kernel_example (__CALCL_MODEL_2D)
{
  // Prevent the execution of more threads than the CA dimension
  calclThreadCheck2D();
  //...

  // Get the cell coordinates back
  CALint i = calclGlobalRow();
  CALint j = calclGlobalColumn();
  //...

  // Set a new value for the substate
  // whose handle is defined by H.
  // Please, note the usage of the
  // MODEL_2D macro-like object
  calclSet2Dr(MODEL_2D, H, i, j, h_next);
  //...
}
```

Listing 6.1: Example of OpenCAL-CL kernel.

Function	Brief description
`calclGlobalRow()`	Get the row coordinate for the current cell
`calclGlobalColumn()`	Get the column coordinate for the current cell
`calclGlobalSlice()`	Get the slice coordinate for the current cell
`calclLocalRow()`	Get the local cell row coordinate within the OpenCL work group
`calclLocalColumn()`	Get the local cell column coordinate within the OpenCL work group
`calclLocalSlice()`	Get the local cell slice coordinate within the OpenCL work group
`calclGet{2D|3D}{b|i|r}()`	Get the substate value for the central cell
`calclGetX{2D|3D}{b|i|r}()`	Get the substate value for a neighboring cell
`calclGetRows()`	Get the number of CA rows
`calclGetColumns()`	Get the number of CA columns
`calclGetSlices()`	Get the number of CA slices
`calclgGetByteSubstatesNum()`	Get the number of substates of `CALbyte` type
`calclGetIntSubstatesNum()`	Get the number of substates of `CALint` type
`calclGetRealSubstatesNum()`	Get the number of substates of `CALreal` type
`calclGetCurrentByteSubstates()`	Get a pointer to the set of current layer of `CALbyte` substates
`calclGetCurrentIntSubstates()`	Get a pointer to the set of current layer of `CALint` substates
`calclGetCurrentRealSubstates()`	Get a pointer to the set of current layer of `CALreal` substates
`calclGetNextByteSubstates()`	Get a pointer to the set of next layer of `CALbyte` substates
`calclGetNextIntSubstates()`	Get a pointer to the set of next layer of `CALint` substates
`calclGetNnextRealSubstates()`	Get a pointer to the set of next layer of `CALreal` substates
`calclGetNeighborhood()`	Get a pointer to the set of neighboring cells
`calclGetNeighborhoodID()`	Get the neighbourhood ID (e.g. von Neumann, Moore, etc.)
`calclGetNeighborhoodSize()`	Get the neighbourhood size
`calclGetBoundaryCondition()`	Get the boundary condition (simple or cyclic)
`calclRunStop()`	Stops the simulation
`calclSet{2D|3D}{b|i|r}()`	Set the substate value for the central cell
`calclThreadCheck[2D|3D]()`	Prevents unnecessary thread execution for 3D CA
`calclActiveThreadCheck[2D|3D]()`	Prevents unnecessary thread execution for 2D CA with active cells

Table 6.1: Some OpenCAL-CL utility kernel functions.

```
1  #include <calcl2D.h>
2  //...
3
4  __kernel void another_calcl_kernel_example (__CALCL_MODEL_2D, __global
       CALParameterr * Pepsilon)
5  {
6
7    //...
8  }
```

Listing 6.2: Another example of OpenCAL-CL kernel, with an additional global parameter.

```
1  #include <calcl2D.h>
2  //...
3
4  __kernel void another_calcl_kernel_example (__CALCL_MODEL_2D, __private
       CALParameterr Pepsilon)
5  {
6
7    //...
8  }
```

Listing 6.3: Another example of OpenCAL-CL kernel, with an additional private parameter.

first substate added to the host-side CA, one (1) is used to link the second substate added to the host-side CA, and so on.

6.1.2 OpenCAL-CL host-side Programming

An OpenCAL-CL host application is typically subdivided in the following parts, which are described in the following sections:

- Definition of the host-side CA model;
- Selection of the OpenCL compliant device;
- Kernels reading and program generation;
- Definition of the device-side CA model;
- Kernels enqueuing;
- Simulation execution on the previously selected compliant device.

Definition of the host-side CA model

The OpenCAL-CL host-side CA definition does not differ from what done for the cases of OpenCAL (cf. Chapter 4). Indeed, in Listing 6.4, a 2D host-side CA object is declared by using the `CALModel2D` OpenCAL data type (line 4), and then initialized by means of the `calCADef2D()` function (line 11), exactly as in the serial version of OpenCAL. Note that the `calcl2D.h` OpenCAL-CL specific header file is included at line 1. This, in turn, includes the `cal2D.h` header, so that it is possible to use OpenCAL data types and functions from an OpenCAL-CL host application.

```
1  #include <OpenCAL-CL/calcl2D.h>
2  //...
3
4  struct CALModel2D* hostCA;
5  //...
6
7  int main(int argc, char** argv)
8  {
9    //...
10
11   hostCA = calCADef2D(ROWS, COLS, CAL_VON_NEUMANN_NEIGHBORHOOD_2D,
         CAL_SPACE_TOROIDAL, CAL_OPT_ACTIVE_CELLS);
12   //...
13
14 }
```

Listing 6.4: An example of OpenCAL-CL host-side application.

Selection of the OpenCL compliant device

OpenCAL-CL provides the `CALCLManager` structure that, together with other utility functions, considerably simplifies platform, device, and context management with respect to the native OpenCL API. Listing 6.5 shows how to select a compliant device in OpenCAL-CL.

Line 7 declares a pointer to the `CALCLManager` OpenCAL-CL data type, and initializes it through the `calclCreateManager()` function. This object, `calcl_manager`, is then used as parameter for the `calclInitializePlatforms()` function (line 10), which fills the object with the platforms available on the machine. Line 13 calls the `calclInitializeDevices()` function, that initializes the available devices, while line 20 selects one of them for kernel execution. Specifically, an object of type `CALCLdevice` is declared and initialized by the function `calclGetDevice()`. This latter takes a pointer to a `CALCLManager` object as first parameter, while the second and third parameters specify the platform and device to be selected, respectively. Since both platforms and devices are identified by a 0-based index[1], statement at line 20 selects the first device belonging to the first platform (e.g. a GTX 980 belonging to the NVIDIA CUDA platform). If system platforms and devices are unknown, the `calclGetPlatformAndDeviceFromStdIn()` function can be used alternatively to `calclGetDevice()`. It prints all the available platforms and devices to standard output and permits for their interactive selection from standard input. Eventually, line 23 creates an OpenCL context, based on the device previously selected. For this purpose, an object of `CALCLcontext` type is declared and defined by means of the `calclCreateContext()` function.

Kernels reading and program generation

After the compliant device has been selected and elementary processes (and possibly other functions) implemented as kernels, these latter can automatically be read and compiled through the `calclLoadProgram{2D|3D}()` functions (cf. Listing 6.6). They

[1] In OpenCAL-CL platforms and devices are stored in a matrix where rows represent platforms and columns devices. Thus, to choose which platform and device to use for the computation, it is necessary to specify their indexes within the matrix. For example, at lines 15, we chose the platform number 0 and the device number 0. If we have a system with 3 NVIDIA GPUs and 3 AMD GPUs, the library will have a 2×3 size matrix, where 2 are the vendors (i.e., the platforms NVIDIA and AMD) and 3 are the GPUs for each platforms. If we want to run the program using the third AMD GPU, we can specify 1 and 2 as indices.

```
#include <OpenCAL-CL/calcl2D.h>
//...

int main (int argc, char** argv)
{
  // Initilize a pointer to the CALCLManager structure
  CALCLManager * calcl_manager = calclCreateManager ();

  // get all available platforms
  calclInitializePlatforms ( calcl_manager );

  // Initialize the devices
  calclInitializeDevices ( calcl_manager );

  // Uncomment if platforms and devices are unknown
  //calclGetPlatformAndDeviceFromStdIn();

  // get the first device on the first platform
  // this call is unnecessary if calclGetPlatformsAndDeviceFromStandardInput
      () is used
  CALCLdevice device = calclGetDevice ( calcl_manager , 0, 0 );

  // create a context
  CALCLcontext context = calclCreateContext ( &device );

  // ...
}
```

Listing 6.5: Access to platform and devices.

```
CALCLprogram calclLoadProgram[2D|3D] (
  CALCLcontext context ,
  CALCLdevice device ,
  char * kernel_source_directory ,
  char * kernel_include_directory
)
```

Listing 6.6: The calclLoadProgram function. It loads and compiles kernels by returning an OpenCL program.

take both the context and device, and also the paths to directories containing the user defined kernels and related headers (if any) as parameters, and returns an OpenCL program[2]. All the files in the kernel source directory are automatically considered, independently from their name. Note that, since kernel headers are optional, the last parameter can be `NULL`.

Definition of the device-side CA model

OpenCAL-CL allows for having a counterpart of the host-side CA to device-side. Such a device-side CA is declared as a `CALCLModel{2D|3D}` object and, beside managing all the CA components device-side, also provides simulation execution features. Note that, this is a main difference with respect to OpenCAL serial version, where the simulation execution is managed by a `CALRun{2D|3D}` complementary object.

To initialize the device-side CA object, the `calclCADef{2D|3D}` function can be used (cf. Listings 6.7 and 6.8 for the 2D and 3D versions, respectively). The first

[2] `CALCLprogram` redefines the `cl_program` OpenCL type.

```
CALCLModel2D * calclCADef2D(
                  struct CALModel2D * host-model,
                  CALCLcontext context,
                  CALCLprogram program,
                  CALCLdevice device
)
```

Listing 6.7: The calclCADef2D function.

```
CALCLModel3D * calclCADef3D(
                  struct CALModel3D * host-model,
                  CALCLcontext context,
                  CALCLprogram program,
                  CALCLdevice device
)
```

Listing 6.8: The calclCADef3D function.

parameter is a pointer to a `CALModel{2D|3D}` object, defined host side as in the case of the serial implementation of OpenCAL (cf. Chapter 4). The second one is an OpenCL context, as well as the third and fourth parameters are an OpenCL program and compliant device, respectively.

Kernels enqueuing

Once compiled and grouped in a program, kernels can be extracted in order to be attached to a command queue for execution. The `calclGetKernelFromProgram()` function can be used to extract a kernel from an OpenCL program. It takes a pointer to the program and the kernel name (cf. Listing 6.9).

The function `calclAddElementaryProcess{2D|3D}` adds a new kernel to the execution queue (cf. Listings 6.10 and 6.11 for the 2D and 3D versions, respectively), in a transparent manner to the user. The function takes both a pointer to host and device-side CA as parameter and also a pointer to an OpenCL kernel.

Note that, if the kernel to be added has one or more parameters (beside the mandatory `__CALCL_MODEL_2D` one), they must be passed to the device-side application. Passing a parameter to a kernel is a operation which depends on how the parameter is declared, i.e., whether if as a pointer or not.

Let as consider the example in Listing 6.2, where a parameter is declared as a pointer to be stored into the device global memory. A double step procedure must be used in this case. First, the `calclCreateBuffer()` function must be called. It takes the OpenCL context, the address of the host parameter to be passed to the kernel, and the dimension of the host parameter type, and returns an object of type `CALCLmem`, which is a buffer containing the value of the host-side parameter.

```
CALCLkernel calclGetKernelFromProgram (
  CALCLprogram * program,
  char * kernelName
)
```

Listing 6.9: The calclGetKernelFromProgram kernel extraction function.

```
void calclAddElementaryProcess2D(
             CALCLModel2D * deviceCA,
             struct CALModel2D * hostCA,
             CALCLkernel * kernel
);
```

Listing 6.10: The calclAddElementaryProcess2D() function.

```
void calclAddElementaryProcess3D(
             CALCLModel3D * deviceCA,
             struct CALModel3D * hostCA,
             CALCLkernel * kernel
);
```

Listing 6.11: The calclAddElementaryProcess3D() function.

This buffer is therefore used by the `calclSetKernelArg[2D|3D]()` functions, which actually perform the data transmission to the device. They take the kernel to which send the parameter, the parameter position within the kernel parameter list (0 for the first one after the mandatory `__CALCL_MODEL_2D` parameter, 1 for the second, and so on), the `CALCLmem` dimension, and the address of the buffer containing the value for the kernel parameter (cf. Listing 6.12). Listing 6.13 shows an example of how parameters can be passed to a kernel with the above described method.

At the contrary, if the kernel parameter is not declared as a pointer, like in the example in Listing 6.3, the above double step process is no longer necessary, since it is sufficient to call the `calclSetKernelArg{2D|3D}()` by specifying the dimension of the actual type of the variable to be passed to the kernel as third parameter .

Simulation execution on the compliant device

The `calclRun{2D|3D}()` function (cf. Listings 6.14 and 6.14 for the 2D and 3D versions, respectively) runs the CA simulation by executing all the defined kernels on the selected compliant device. The first two parameters are pointers to a device and host CA, respectively, while the last two are the initial and final step for the simulation execution. If the last parameter is set to `CAL_RUN_LOOP`, simulation never ends. In this case, to stop the simulation, the user must define a stop condition criterion through the `calclAddStopConditionFunc{2D|3D}()` function (cf. Listings 6.16 and 6.17 for the 2D and 3D versions, respectively). Note that, since the stop condition must be evaluated host side, it must be implemented as a kernel. For this reason, the third parameter is a pointer to an OpenCL kernel. Other kernels can be registered to the device-side CA. Table 6.2 list the OpenCAL-CL that can be used to register kernels to the device-side CA.

```
int calclSetKernelArg[2D|3D](CALCLkernel* kernel,
                             cl_uint arg_index,
                             size_t arg_size,
                             const void *arg_value
)
```

Listing 6.12: The calclSetKernelArg2D() function.

```
#include <OpenCAL-CL/calcl2D.h>
//...

struct sciddicaTParameters {
  CALParameterr epsilon;
  //...
} P;

//...

int main(int argc, char** argv)
{
  //...
  CALCLmem bufferEpsilonParameter;
  //...

  bufferEpsilonParameter = calclCreateBuffer(context, &P.epsilon, sizeof(
      CALParameterr));
  calclSetKernelArg2D(&kernel_elem_proc_flow_computation, 0, sizeof(CALCLmem
      ), &bufferEpsilonParameter);
  //...
}
```

Listing 6.13: Passing parametrs to kernel.

Function	Brief description
`calclAddInitFunc{2D|3D}`	register a global kernel to be executed before simulation loop
`calclAddElementaryProcess{2D|3D}()`	register an elementary process
`calclAddSteeringFunc{2D|3D}`	register a global kernel to be executed after each step
`calclAddStopConditionFunc{2D|3D}()`	register a global stop condition kernel callback

Table 6.2: OpenCAL-CL host-side functions used to register elementary processes and global functions to the device-side CA.

```
void calclRun2D(CALCLModel2D* deviceCA,
                struct CALModel2D * hostCA,
                unsigned int initialStep,
                unsigned maxStep
);
```

Listing 6.14: The calclRun2D() function.

```
void calclRun3D(CALCLModel3D* deviceCA,
                struct CALModel3D * hostCA,
                unsigned int initialStep,
                unsigned maxStep
);
```

Listing 6.15: The calclRun3D() function.

```
void calclAddStopConditionFunc2D(
               CALCLModel2D * deviceCA,
               struct CALModel2D * hostCA,
               CALCLkernel * kernel
);
```

Listing 6.16: The calclAddStopConditionFunc2D function.

```
void calclAddStopConditionFunc3D(
                CALCLModel3D * deviceCA,
                struct CALModel3D * hostCA,
                CALCLkernel * kernel
);
```

Listing 6.17: The calclAddStopConditionFunc3D function.

6.2 Conway's Game of Life with OpenCAL-CL

As already done in previous Chapters about OpenCAL and OpenCAL-OMP, here we introduce OpenCAL-CL programming by implementing Conway's Game of Life. The application is subdivided in two parts, one for the host and one for the compliant device.

The host-side application, running on the CPU and controlling the computation on the compliant device (e.g. a GPU), is shown in Listing 6.18.

```
1  // Conway's game of Life Cellular Automaton
2
3  #include <OpenCAL-CL/calcl2D.h>
4  #include <OpenCAL/cal2DIO.h>
5
6  // Paths and kernel name definition
7  #define KERNEL_SRC "./kernel/source/"
8  #define KERNEL_LIFE_TRANSITION_FUNCTION "lifeTransitionFunction"
9  #define PLATFORM_NUM 0
10 #define DEVICE_NUM 0
11 #define DEVICE_Q 0
12
13 int main()
14 {
15   // Select a compliant device
16   struct CALCLDeviceManager * calcl_device_manager = calclCreateManager();
17   calclPrintPlatformsAndDevices(calcl_device_manager);
18   CALCLdevice device = calclGetDevice(calcl_device_manager, PLATFORM_NUM,
         DEVICE_NUM);
19   CALCLcontext context = calclCreateContext(&device);
20
21   // Load kernels and return a compiled program
22   CALCLprogram program = calclLoadProgram2D(context, device, KERNEL_SRC, NULL)
         ;
23
24   // Define a host-side CA and declare a substate
25     struct CALModel2D* host_CA = calCADef2D(8, 16, CAL_MOORE_NEIGHBORHOOD_2D,
           CAL_SPACE_TOROIDAL, CAL_OPT_ACTIVE_CELLS_NAIVE);
26   struct CALSubstate2Di* Q;
27
28   // Register the substate to the host CA
29   Q = calAddSubstate2Di(host_CA);
30
31   // Initialize the substate to 0 everywhere
32   calInitSubstate2Di(host_CA, Q, 0);
33
34   // Set a glider
35   calInit2Di(host_CA, Q, 0, 2, 1);
36   calInit2Di(host_CA, Q, 1, 0, 1);
37   calInit2Di(host_CA, Q, 1, 2, 1);
38   calInit2Di(host_CA, Q, 2, 1, 1);
39   calInit2Di(host_CA, Q, 2, 2, 1);
40
41   // Define a device-side CA
42     struct CALCLModel2D * device_CA = calclCADef2D(host_CA, context, program,
           device);
```

```
43
44    // Extract a kernel from program
45    CALCLkernel kernel_life_transition_function = calclGetKernelFromProgram(&
          program, KERNEL_LIFE_TRANSITION_FUNCTION);
46
47    // Register a transition function's elementary process kernel
48    calclAddElementaryProcess2D(device_CA, &kernel_life_transition_function);
49
50    // Save the substate to file
51    calSaveSubstate2Di(host_CA, Q, "./life_0000.txt");
52
53    // Run the simulation (actually, only one computational step)
54    calclRun2D(device_CA, 1, 1);
55
56    // Save the substate to file
57    calSaveSubstate2Di(host_CA, Q, "./life_LAST.txt");
58
59    // Finalizations
60    calclFinalizeManager(calcl_device_manager);
61    calclFinalize2D(device_CA);
62    calFinalize2D(host_CA);
63
64    return 0;
65 }
```

Listing 6.18: An OpenCAL-CL host-side implementation of the Conway's Game of Life.

Differently from the serial implementation, discussed in Section 4.4, the OpenCAL-CL `calcl2D.h` header file is include at line 3. It, in turn, includes the `cal2D.h` header, needed to define the host-side CA model. The OpenCAL `cal2DIO.h` header is also included at line 4 for I/O operations.

At line 7, the path containing the kernels to be executed in parallel on the compliant device is defined, while the name of the single kernel considered in this example is defined at line 8. Lines 9-10 define the IDs of the OpenCL platform and device to be considered. For the sake of simplicity, in this example the first device belonging to the first platform is considered.

Lines 15-18 are needed to select the compliant device and to create an OpenCL context. These statements widely simplify the device management and can be considered as a kind of template to be used in each OpenCAL-CL application. Indeed, they will also be adopted in the subsequent examples.

Line 21 reads kernels (actually, just one in this example) from file (contained in the directory specified at line 7), compile and groups them into an OpenCL program, to be used later to extract kernels for execution.

Lines 24-38 are equivalent to the serial implementation of *Life*. The `host_CA` host-side object is defined at line 24 and the `Q` substate declared at line 25. This latter is therefore registered to the host-side CA at line 28. Eventually, a glider is defined at lines 34-38.

Line 41 defines the `device_CA` device-side object. The `calclCADef2D()` function initializes the device-side CA, by performing data transfer from host to device, in a transparent way to the user. Note that this function implicitly registers each host-side defined substate to the device object. However, to access the substate device-side, the user have to define a numeric handle, as discussed below.

In order to register an elementary process to the device-side CA, where computation will take place, a preliminary operation must be performed: the elementary process, which actually is an OpenCL kernel, must be extracted from the previously compiled program. This operation is done at line 44 by means of the

`calclGetKernelFromProgram()`. It returns an OpenCL kernel which is subsequently registered to the device-side CA by means of the `calclAddElementaryProcess2D()` function at line 47.

Lines 50 an 56 are used to save the CA state (represented by the single `Q` substate) before and after simulation execution, respectively.

The CA simulation is executed by means of the `calclRun2D()` function at line 5. In this example, the only defined elementary process is executed in parallel on the compliant device, in a transparently way to the user.

Eventually, lines 59-61 perform memory deallocation for the previously defined objects.

```
1  // Conway's game of Life transition function kernel
2
3  #include <OpenCAL-CL/calcl2D.h>
4
5  #define DEVICE_Q 0
6
7  __kernel void lifeTransitionFunction(__CALCL_MODEL_2D)
8  {
9    calclThreadCheck2D();
10
11   int i = calclGlobalRow();
12   int j = calclGlobalColumn();
13
14   CALint sizeOfX_ = calclGetNeighborhoodSize();
15
16   int sum = 0, n;
17
18   for (n=1; n<sizeOfX_; n++)
19     sum += calclGetX2Di(MODEL_2D, DEVICE_Q, i, j, n);
20
21   if ((sum == 3) || (sum == 2 && calclGet2Di(MODEL_2D, DEVICE_Q, i, j) == 1))
22     calclSet2Di(MODEL_2D, DEVICE_Q, i, j, 1);
23   else
24     calclSet2Di(MODEL_2D, DEVICE_Q, i, j, 0);
25 }
```

Listing 6.19: The OpenCAL-CL kernel implementing the Conway's Game of Life elementary process.

Listing 6.19 shows the OpenCAL-CL kernel implementing the Conway's Game of Life transition function.

The `calcl2D.h` is included at line 3, while the numeric `Q` handle is defined at line 5 to access the the `Q` substate, registered to the host CA and, implicitly, to the device CA at line 50 of Listing 6.18. In this case, the handle takes the value 0, being `Q` the only considered substate. If more substates would be considered, other numeric handles would be defined, taking increasing values.

The elementary process is implemented within the `lifeTransitionFunction()` kernel, defined at lines 7-25. As previously evidenced, it takes the `__CALCL_MODEL_2D` macro, which implicitly defines a set of parameters for the kernel.

Line 9 assures the execution of a number of concurrent threads equal to the number of cells in the CA cellular space, while lines 11-12 get the indexes corresponding to the integer coordinates of the cell the kernel is going to process. In this way, the one thread-one cell policy is simply obtained.

Line 14 gets the neighborhood size, while line 16 declares some variables to be used later.

Lines 18-24 implement the *life* transition rules. Note that, the `calclGetX2Di()` and `calclSet2Di()` functions are here used to access the substate values for the

neighbouring cells and to set the new state for the central cell. They correspond to the `calGetX2Di()` and `calSet2Di()` OpenCAL functions, with the difference that, in spite of taking a CA model as parameter, they take the `MODEL_2D` OpenCAL-CL macro.

Figures 4.3 and 4.4 in Chapter 4 show the initial and final configuration of Game of Life as implemented in Listing 6.18, respectively.

6.3 SciddicaT

In the previous section we illustrated an OpenCAL-CL implementation of a simple cellular automaton, namely the Conways Game of Life. Here, we will deal with a more complex example, concerning the the $SciddicaT_{naive}$ Cellular Automata model for landslide simulation, already presented in Section 4.6 together with its serial implementation. Eventually, we will show how to combine the OpenCAL-CL and OpenCAL-GL libraries to integrate an OpenGL/GLUT visualization system.

```
1  // The SciddicaT debris flows XCA simulation model with
2  // a 3D graphic viewer in OpenCAL-GL
3
4  #include <OpenCAL/cal2DIO.h>
5  #include <OpenCAL-CL/calcl2D.h>
6  #include <OpenCAL-CL/calgl2DRunCL.h>
7  #include <OpenCAL-GL/calgl2D.h>
8  #include <OpenCAL-GL/calgl2DWindow.h>
9  #include <stdlib.h>
10
11 // Some definitions...
12 #define ROWS 610
13 #define COLUMNS 496
14 #define P_R 0.5
15 #define P_EPSILON 0.001
16 #define NUMBER_OF_OUTFLOWS 4
17 #define STEPS 4000
18 #define DEM_PATH "./data/dem.txt"
19 #define SOURCE_PATH "./data/source.txt"
20 #define OUTPUT_PATH "./data/width_final.txt"
21 #define GRAPHIC_UPDATE_INTERVAL 100
22
23 // kernels' names definitions
24 #define ACTIVE_CELLS
25 #define KERNEL_SRC "./kernel/source/"
26 #define KERNEL_INC "./kernel/include/"
27 #define KERNEL_SRC_AC "./kernelActive/source/"
28 #define KERNEL_INC_AC "./kernelActive/include/"
29 #define KERNEL_ELEM_PROC_FLOW_COMPUTATION "flowsComputation"
30 #define KERNEL_ELEM_PROC_WIDTH_UPDATE "widthUpdate"
31 #define KERNEL_STEERING  "steering"
32 #ifdef ACTIVE_CELLS
33 #define KERNEL_ELEM_PROC_RM_ACT_CELLS "removeInactiveCells"
34 #endif
35
36 // The set of CA substates
37 struct sciddicaTSubstates {
38   struct CALSubstate2Dr *z;
39   struct CALSubstate2Dr *h;
40   struct CALSubstate2Dr *f[NUMBER_OF_OUTFLOWS];
41 };
42
43 // The set of CA parameters
44 struct sciddicaTParameters {
45   CALParameterr epsilon;
46   CALParameterr r;
```

```
47 };
48
49 // Objects declaration
50 struct CALCLDeviceManager * calcl_device_manager; //the device manager object
51 struct CALModel2D* host_CA;                       //the host-side CA
52 struct sciddicaTSubstates Q;                      //the CA substates object
53 struct sciddicaTParameters P;                     //the CA parameters object
54 struct CALCLModel2D * device_CA;                  //the device-side CA
55
56
57 // SciddicaT exit function
58 void exitFunction(void)
59 {
60   // saving configuration
61   calSaveSubstate2Dr (host_CA, Q.h, OUTPUT_PATH);
62
63   // finalizations
64   //calRunFinalize2D (sciddicaTsimulation);
65   calclFinalizeManager(calcl_device_manager);
66   calclFinalize2D(device_CA);
67   calFinalize2D (host_CA);
68 }
69
70 // SciddicaT init function
71 void sciddicaTSimulationInit(struct CALModel2D* host_CA) {
72   CALreal z, h;
73   CALint i, j;
74
75   //initializing substates to 0
76   calInitSubstate2Dr(host_CA, Q.f[0], 0);
77   calInitSubstate2Dr(host_CA, Q.f[1], 0);
78   calInitSubstate2Dr(host_CA, Q.f[2], 0);
79   calInitSubstate2Dr(host_CA, Q.f[3], 0);
80
81   //sciddicaT parameters setting
82   P.r = P_R;
83   P.epsilon = P_EPSILON;
84
85   //sciddicaT source initialization
86   for (i = 0; i < host_CA->rows; i++)
87     for (j = 0; j < host_CA->columns; j++) {
88       h = calGet2Dr(host_CA, Q.h, i, j);
89
90       if (h > 0.0) {
91         z = calGet2Dr(host_CA, Q.z, i, j);
92         calSet2Dr(host_CA, Q.z, i, j, z - h);
93
94 #ifdef ACTIVE_CELLS
95         //adds the cell (i, j) to the set of active ones
96         calAddActiveCell2D(host_CA, i, j);
97 #endif
98       }
99     }
100 }
101
102 int main(int argc, char** argv)
103 {
104   // OpenCL device, context and program declaration
105   CALCLdevice device;
106   CALCLcontext context;
107   CALCLprogram program;
108
109   // kernels paths, names and buffers (for kernel parameters)
110 #ifdef ACTIVE_CELLS
111   char * kernelSrc = KERNEL_SRC_AC;
112   char * kernelInc = KERNEL_INC_AC;
113 #else
114   char * kernelSrc = KERNEL_SRC;
```

```
115   char * kernelInc = KERNEL_INC;
116 #endif
117   CALCLkernel kernel_elem_proc_flow_computation;
118   CALCLkernel kernel_elem_proc_width_update;
119   CALCLkernel kernel_elem_proc_rm_act_cells;
120   CALCLkernel kernel_steering;
121   CALCLmem bufferEpsilonParameter;
122   CALCLmem bufferRParameter;
123
124   //OpenCL device selection from stdin and context definition
125   calcl_device_manager = calclCreateManager();
126   calclGetPlatformAndDeviceFromStdIn(calcl_device_manager, &device);
127   context = calclCreateContext(&device);
128
129   // Load kernels and return a compiled program
130   program = calclLoadProgram2D(context, device, kernelSrc, kernelInc);
131
132   // host-side CA definition
133 #ifdef ACTIVE_CELLS
134     host_CA = calCADef2D(ROWS, COLUMNS, CAL_VON_NEUMANN_NEIGHBORHOOD_2D,
          CAL_SPACE_TOROIDAL, CAL_OPT_ACTIVE_CELLS_NAIVE);
135 #else
136   host_CA = calCADef2D(ROWS, COLUMNS, CAL_VON_NEUMANN_NEIGHBORHOOD_2D,
        CAL_SPACE_TOROIDAL, CAL_NO_OPT);
137 #endif
138
139   // Add substates to the host-side CA
140   Q.f[0] = calAddSubstate2Dr(host_CA);
141   Q.f[1] = calAddSubstate2Dr(host_CA);
142   Q.f[2] = calAddSubstate2Dr(host_CA);
143   Q.f[3] = calAddSubstate2Dr(host_CA);
144   Q.z = calAddSubstate2Dr(host_CA);
145   Q.h = calAddSubstate2Dr(host_CA);
146
147   // Load data from file
148   calLoadSubstate2Dr(host_CA, Q.z, DEM_PATH);
149   calLoadSubstate2Dr(host_CA, Q.h, SOURCE_PATH);
150   // Host-side CA initialization
151   sciddicaTSimulationInit(host_CA);
152   calUpdate2D(host_CA);
153
154   //device-side CA definition
155   device_CA = calclCADef2D(host_CA, context, program, device);
156
157   // Extract kernels from program
158   kernel_elem_proc_flow_computation = calclGetKernelFromProgram(&program,
        KERNEL_ELEM_PROC_FLOW_COMPUTATION);
159   kernel_elem_proc_width_update = calclGetKernelFromProgram(&program,
        KERNEL_ELEM_PROC_WIDTH_UPDATE);
160 #ifdef ACTIVE_CELLS
161   kernel_elem_proc_rm_act_cells = calclGetKernelFromProgram(&program,
        KERNEL_ELEM_PROC_RM_ACT_CELLS);
162 #endif
163   kernel_steering = calclGetKernelFromProgram(&program, KERNEL_STEERING);
164
165   // Setting kernel parameters
166   bufferEpsilonParameter = calclCreateBuffer(context, &P.epsilon, sizeof(
        CALParameterr));
167   bufferRParameter = calclCreateBuffer(context, &P.r, sizeof(CALParameterr));
168   calclSetKernelArg2D(&kernel_elem_proc_flow_computation, 0, sizeof(CALCLmem),
         &bufferEpsilonParameter);
169   calclSetKernelArg2D(&kernel_elem_proc_flow_computation, 1, sizeof(CALCLmem),
         &bufferRParameter);
170
171   // Register transition function s elementary processes to the device-side
        CA
172   calclAddElementaryProcess2D(device_CA, &kernel_elem_proc_flow_computation);
173   calclAddElementaryProcess2D(device_CA, &kernel_elem_proc_width_update);
```

```
174   #ifdef ACTIVE_CELLS
175     calclSetKernelArg2D(&kernel_elem_proc_rm_act_cells, 0, sizeof(CALCLmem), &
            bufferEpsilonParameter);
176     calclAddElementaryProcess2D(device_CA, &kernel_elem_proc_rm_act_cells);
177   #endif
178   // Register a steering function to the device-side CA
179   calclAddSteeringFunc2D(device_CA, &kernel_steering);
180
181   // Register a function to be executed before program termination
182   atexit(exitFunction);
183
184
185   // Graphic viewer initialization
186   calglInitViewer("SciddicaT OpenCAL-GL visualizer", 5, 800, 600, 10, 10,
          CAL_TRUE, 0);
187   //calglSetLayoutOrientation2D(CALGL_LAYOUT_ORIENTATION_VERTICAL);
188
189   // Rendering objects declaration
190   struct CALGLDrawModel2D* render_3D;
191   struct CALGLDrawModel2D* render_2D;
192
193   // render_3D definition
194   struct CALGLRun2D * calgl_run= calglRunCLDef2D(device_CA,
          GRAPHIC_UPDATE_INTERVAL, 1, 4000);
195   calglSetDisplayStep(GRAPHIC_UPDATE_INTERVAL);
196
197   // 3D view rendering object
198   render_3D = calglDefDrawModelCL2D(CALGL_DRAW_MODE_SURFACE, "SciddicaT 3D
          view", host_CA, calgl_run);
199   // Add nodes
200   calglAdd2Dr(render_3D, NULL, &Q.z, CALGL_TYPE_INFO_VERTEX_DATA,
          CALGL_TYPE_INFO_USE_NO_COLOR, CALGL_DATA_TYPE_STATIC);
201   calglColor2D(render_3D, 0.5, 0.5, 0.5, 1.0);
202   calglAdd2Dr(render_3D, Q.z, &Q.z, CALGL_TYPE_INFO_COLOR_DATA,
          CALGL_TYPE_INFO_USE_CURRENT_COLOR, CALGL_DATA_TYPE_DYNAMIC);
203   calglAdd2Dr(render_3D, Q.z, &Q.z, CALGL_TYPE_INFO_NORMAL_DATA,
          CALGL_TYPE_INFO_USE_NO_COLOR, CALGL_DATA_TYPE_DYNAMIC);
204   calglAdd2Dr(render_3D, Q.z, &Q.h, CALGL_TYPE_INFO_VERTEX_DATA,
          CALGL_TYPE_INFO_USE_NO_COLOR, CALGL_DATA_TYPE_DYNAMIC);
205   calglAdd2Dr(render_3D, Q.h, &Q.h, CALGL_TYPE_INFO_COLOR_DATA,
          CALGL_TYPE_INFO_USE_RED_YELLOW_SCALE, CALGL_DATA_TYPE_DYNAMIC);
206   calglAdd2Dr(render_3D, Q.h, &Q.h, CALGL_TYPE_INFO_NORMAL_DATA,
          CALGL_TYPE_INFO_USE_NO_COLOR, CALGL_DATA_TYPE_DYNAMIC);
207   calglSetHeightOffset2D(render_3D,10);
208
209   // Scalar bar
210   calglInfoBar2Dr(render_3D, Q.h, "Debris thickness",
          CALGL_TYPE_INFO_USE_RED_SCALE, 20, 120, 300, 40);
211
212   // 2D view rendering object
213   render_2D = calglDefDrawModelCL2D(CALGL_DRAW_MODE_FLAT, "SciddicaT 2D view",
           host_CA, calgl_run);
214   render_2D->realModel = render_3D->realModel;
215   calglInfoBar2Dr(render_2D, Q.h, "Debris thickness",
          CALGL_TYPE_INFO_USE_RED_SCALE, 20, 200, 50, 150);
216
217   // calgl main loop
218   calglMainLoop2D(argc, argv);
219
220   return 0;
221 }
```

Listing 6.20: An OpenCAL-CL host-side implementation of the SciddicaT Cellular Automata model for landslide simulation.

The host-side implementation of $SciddicaT_{naive}$ is shown in Listing 6.20. Lines 4-9 include some headers. In particular, the `cal2DIO.h` OpenCAL header is included for

I/O purposes, while the `calcl2D.h` and `calgl2DRunCL.h` headers provide CA definition and graphic rendering support for the OpenCAL-CL library. Eventually, the `calgl2D.h` and `calgl2DWindow.h` OpenCAL-GL headers are included, to complete the application support for 2D and 3D rendering.

Lines 12-21 provide definitions and are almost the same of the serial implementation shown in Listing 4.14. In addition, `GRAPHIC_UPDATE_INTERVAL` defines the interval (in terms of computational steps) to be used for both data retrieval from the OpenCL compliant device and visualization purposes.

Lines 24-34 provide other definitions, specifying kernels paths and names. Note that, different paths are here defined, depending whether the active cells optimization is considered or not (cf. `ACTIVE_CELLS`, line 24). In the first case, an additional kernel is defined at line 33.

Lines 37-47 define two data types for grouping CA substates and parameters, while lines 50-54 declare host- and device-side CA objects, substates, and parameters, besides the `calcl_device_manager` object, needed to select the OpenCL compliant device.

In the main function, after OpenCL device, context and program declarations, kernel variables are declared at lines 117-120, while lines 121-122 declare buffers to be used for kernel parameter settings. The device manager is then defined at line 125, a device selected by means of the `calclGetPlatformAndDeviceFromStdIn()` function at line 126. This latter prints all the available platforms and devices to standard output and allows for their interactive selection.

OpenCL kernels, implementing transition function elementary processes and, optionally, other functions like global init or steering, are read from files, compiled and grouped together into an OpenCL container called program at line 130, with a single call to `calclLoadProgram2D()`.

The host-side CA is therefore defined at lines 133-137, and substates registered at lines 140-145. Note that, the order in which substates are registered, implicitly define the values to be assigned to substate handles to refer related data device-side.

The initial configuration is defined at lines 148-152. The `calLoadSubstate2Dr()` function is here used to initialize two substates from file, while the others are initialized through the `sciddicaTSimulationInit()` function at line 151. Note that, in order to apply the initialization performed before to the substates *next working planes*, it is necessary to call `calUpdate2D()`[3].

The device-side CA is defined at line 155. As for the case of Conway's Game of Life, the `calclCADef2D()` function initializes the object, by performing data transfer from the host to the device memory, in a transparent way to the user.

In order to be able to execute a simulation, transition function elementary processes and global functions must be registered to the device-side CA object. This is a two-step process: at first, kernels are extracted from the OpenCL program by means of the `calclGetKernelFromProgram()` function (cf. e.g. line 158) and therefore they are registered to the device-side CA object through the `calclAddElementaryProcess2D()` function (cf. e.g. line 172). In the specific case of *SciddicaT*$_{naive}$, some elementary processes require parameters, that are defined and set at lines 166-177 through the `calclCreateBuffer()` and `calclSetKernelArg2D()` functions, respectively (as described in Section 6.1). Eventually, the steering kernel is registered at line 179 through the `calclAddSteeringFunc2D()` function.

[3] Note that, in the serial implementation of SciddicaT, this operation was performed automatically by the simulation object.

Starting from line 182, an integration with the OpenCAL-GL library is implemented in order to provide a minimal GLUT Graphical User Interface and a simplified OpenGL 2D/3D visualization system to the application. Firstly, an exit function is registered to the application for memory release purposes at line 182 through the `atexit()` C function[4]. The viewer is initialized at line 186 by means of the `calglInitViewer()` function, while the next call is used to define a vertical layout for the two 2D and 3D rendering objects, defined later at lines 190-191.

The rendering tree-based structures for the 3D and 2D rendering objects are defined at lines 198-210 and 213-215, respectively. A simulation object is then defined at line 194 by means of the `CALGLRun2D()` function and run on the OpenCL compliant device at line 218 by means of the `calglMainLoop2D()` function. This latter executes kernels in parallel on the device by considering the one thread-one cell policy, manages data retrieval from compliant device to host at prefixed intervals (in terms of computational steps - cf. the `GRAPHIC_UPDATE_INTERVAL` definition at line 21), and performs the graphic rending.

For further details about OpenCAL-GL, please refer Section 4.8.1.

```
1  #ifndef kernel_h
2  #define kernel_h
3
4  #include <OpenCAL-CL/calcl2D.h>
5
6  #define NUMBER_OF_OUTFLOWS 4
7  #define F0 0
8  #define F1 1
9  #define F2 2
10 #define F3 3
11 #define Z 4
12 #define H 5
13
14 #endif
```

Listing 6.21: The SciddicaT kernels' header file, as implemented in OpenCAL-CL.

```
1  #include <kernel.h>
2
3  //first elementary process
4  __kernel void flowsComputation(__CALCL_MODEL_2D, __global CALParameterr *
       Pepsilon, __global CALParameterr * Pr)
5  {
6    calclActiveThreadCheck2D();
7
8    int threadID = calclGlobalRow();
9    int i = calclActiveCellRow(threadID);
10   int j = calclActiveCellColumn(threadID);
11
12   CALbyte eliminated_cells[5] = { CAL_FALSE, CAL_FALSE, CAL_FALSE, CAL_FALSE,
         CAL_FALSE };
13   CALbyte again;
14   CALint cells_count;
15   CALreal average;
16   CALreal m;
17   CALreal u[5];
18   CALint n;
19   CALreal z, h;
20   CALint sizeOfX_ = calclGetNeighborhoodSize();
21   CALParameterr eps = *Pepsilon;
22
23   if (calclGet2Dr(MODEL_2D,H, i, j) <= eps)
```

[4]This is necessary because, once entered the GLUT main loop, the GL Utility Toolkit does not return the control to the main application.

```
24      return;
25
26    m = calclGet2Dr(MODEL_2D,H, i, j) - eps;
27    u[0] = calclGet2Dr(MODEL_2D, Z, i, j) + eps;
28    for (n = 1; n < sizeOfX_; n++) {
29      z = calclGetX2Dr(MODEL_2D, Z, i, j, n);
30      h = calclGetX2Dr(MODEL_2D, H, i, j, n);
31      u[n] = z + h;
32    }
33
34    do {
35      again = CAL_FALSE;
36      average = m;
37      cells_count = 0;
38
39      for (n = 0; n < sizeOfX_; n++)
40        if (!eliminated_cells[n]) {
41          average += u[n];
42          cells_count++;
43        }
44
45      if (cells_count != 0)
46        average /= cells_count;
47
48      for (n = 0; n < sizeOfX_; n++)
49        if ((average <= u[n]) && (!eliminated_cells[n])) {
50          eliminated_cells[n] = CAL_TRUE;
51          again = CAL_TRUE;
52        }
53
54    } while (again);
55
56    //__global CALreal * fsubstate;
57
58    for (n = 1; n < sizeOfX_; n++) {
59      if (eliminated_cells[n])
60        calclSet2Dr(MODEL_2D, n-1, i, j, 0.0);
61      else {
62        calclSet2Dr(MODEL_2D, n-1, i, j, (average - u[n]) * (*Pr));
63        calclAddActiveCellX2D(MODEL_2D, i, j, n);
64      }
65    }
66  }
67
68  __kernel void widthUpdate(__CALCL_MODEL_2D)
69  {
70    calclActiveThreadCheck2D();
71
72    CALint neighborhoodSize = calclGetNeighborhoodSize();
73
74    int threadID = calclGlobalRow();
75    int i = calclActiveCellRow(threadID);
76    int j = calclActiveCellColumn(threadID);
77
78    CALreal h_next;
79    CALint n;
80
81    h_next = calclGet2Dr(MODEL_2D,H, i, j);
82
83
84    for (n = 1; n < neighborhoodSize; n++)
85      h_next += ( calclGetX2Dr(MODEL_2D, NUMBER_OF_OUTFLOWS-n, i, j, n) -
           calclGet2Dr(MODEL_2D, n-1, i, j) );
86
87    calclSet2Dr(MODEL_2D, H, i, j, h_next);
88
89  }
90
```

```
91  __kernel void removeInactiveCells(__CALCL_MODEL_2D, __global CALParameterr *
        Pepsilon)
92  {
93    calclActiveThreadCheck2D();
94
95    int threadID = calclGlobalRow();
96    int i = calclActiveCellRow(threadID);
97    int j = calclActiveCellColumn(threadID);
98
99    if (calclGet2Dr(MODEL_2D, H, i, j) <= *Pepsilon)
100     calclRemoveActiveCell2D(MODEL_2D,i,j);
101 }
102
103 __kernel void steering(__CALCL_MODEL_2D)
104 {
105   calclActiveThreadCheck2D();
106
107   int threadID = calclGlobalRow();
108
109   int dim = calclGetColumns() * calclGetRows();
110   int i;
111   for (i = 0; i < NUMBER_OF_OUTFLOWS; ++i)
112     calclInitSubstateActiveCell2Dr(MODEL_2D, i, threadID, 0);
113 }
```

Listing 6.22: The OpenCAL-CL kernels implementing the SciddicaT elementary processes and steering.

The device-side implementation of SciddicaT which take into account the active cells optimization is shown in Listings 6.21 and 6.22 for kernels sources and header, respectively. Non-optimized versions are here omitted for the sake of shortness.

Line 3 includes the `calcl2D.h` header, while line 6 define the number of outflows considered in SciddicaT. Eventually, lines 7-12 define the substates handles. Here, handles are assigned by taking into account the order in which they have been registered to the host-side CA (cf. Listing 6.20, lines 140-145).

As regards the kernels shown in Listings 6.22, they do not differ from that of the Game of Life (cf. 6.18). In fact, each kernel is identified by the `__kernel` keyword and, with the exception of the first one, do not take any parameter, besides the mandatory implicit macro-like `__CALCL_MODEL_2D` one. Concerning the `flowsComputation()` kernel, it takes two additional parameters, namely `Pepsilon` and `Pr`, both of type `CALParameterr`. They are declared by means of the `__global` OpenCL keyword, meaning that they will be stored in the device global memory. Setting the values for these parameters is a host-side application responsibility. Note that in the specific case of the OpenCAL-CL implementation of SciddicaT, this operation is done at lines 166-169 of Listing 6.20. Differently from the case of Life, the `calclActiveThreadCheck2D();` function is used in each kernel to prevent the computation for threads that do not correspond to active cells. For the remaining code, the OpenCAL-CL functions already discussed in Section 6.2 are employed, while the algorithms implemented into the kernels are exactly the same of those shown in Section 4.6.

A screenshot of *SciddicaT−naive* with the embedded OpenGL/GLUT visualization system is shown in Figure 6.2.

6.4 A three-dimensional example

This section describes the implementation of a simple 3D model, namely the *mod2* 3D CA, already presented and implemented by means of the serial version of OpenCAL

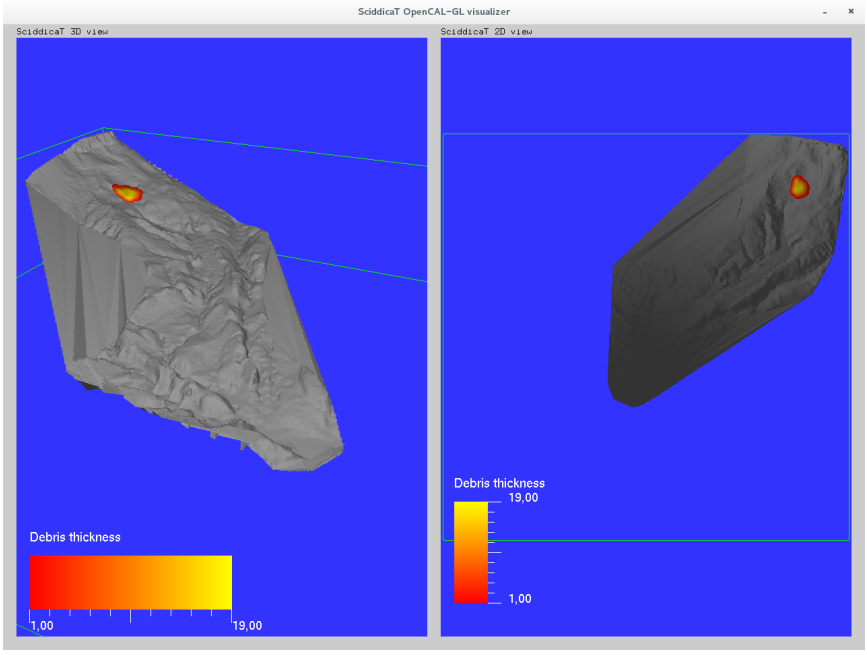


Figure 6.2: Screenshot of the SciddicaT debris flow model with a multi-view 2D and 3D visualization system based on OpenCAL-GL.

in Section 4.7. As for the previous example about SciddicaT, the *mod2* application comes together with an OpenCAL-GL-based GUI and 3D rendering system. The complete source codes for host- and device-side applications are shown in Listings 6.23 and 6.24, respectively .

```
1  // mod2 3D Cellular Automaton
2
3  #include <OpenCAL-CL/calcl3D.h>
4  #include <OpenCAL-CL/calgl3DRunCL.h>
5  #include <OpenCAL-GL/calgl3D.h>
6  #include <OpenCAL-GL/calgl3DWindow.h>
7  #include <OpenCAL/cal3DIO.h>
8
9  // Some definitions...
10 #define ROWS 25
11 #define COLS 25
12 #define LAYERS 25
13 #define KERNEL_SRC "./kernel/source/"
14 #define KERNEL_LIFE_TRANSITION_FUNCTION "mod2TransitionFunction"
15 #define PLATFORM_NUM 0
16 #define DEVICE_NUM 0
17
18 struct CALModel3D* host_CA;         //the cellular automaton
19 struct CALSubstate3Db *Q;           //the substate Q
20 struct CALCLModel3D* device_CA;     //the simulartion run
21 struct CALCLDeviceManager * calcl_device_manager;
22
23 // Callback unction called just before program termination
24 void exitFunction(void)
25 {
26   // Finalizations
27   calclFinalizeManager(calcl_device_manager);
28   calclFinalize3D(device_CA);
29   calFinalize3D(host_CA);
30 }
31
32 // Simulation init callback function used to set a seed at position (24, 0, 0)
33 void mod2SimulationInit(struct CALModel3D* ca)
34 {
35   //initializing substate to 0
36   calInitSubstate3Db(ca, Q, 0);
37   //setting a specific cell
38   calSet3Db(ca, Q, 24, 0, 0, 1);
39 }
40
41
42 int main(int argc, char** argv)
43 {
44   // Declare a viewer object
45   struct CALGLDrawModel3D* drawModel;
46
47   atexit(exitFunction);
48
49   // Select a compliant device
50   calcl_device_manager = calclCreateManager();
51   calclPrintPlatformsAndDevices(calcl_device_manager);
52   CALCLdevice device = calclGetDevice(calcl_device_manager, PLATFORM_NUM,
         DEVICE_NUM);
53   CALCLcontext context = calclCreateContext(&device);
54
55   // Load kernels and return a compiled program
56   CALCLprogram program = calclLoadProgram3D(context, device, KERNEL_SRC, NULL)
         ;
57
58   // Define of the mod2 CA object and declare a substate
59   host_CA = calCADef3D(ROWS, COLS, LAYERS, CAL_MOORE_NEIGHBORHOOD_3D,
         CAL_SPACE_TOROIDAL, CAL_NO_OPT);
```

```
60
61    // Add the Q substate to the host_CA CA
62    Q = calAddSubstate3Db(host_CA);
63
64    // Set the whole substate to 0
65    calInitSubstate3Db(host_CA, Q, 0);
66
67    //setting a specific cell
68    calInit3Db(host_CA, Q, 24, 0, 0, 1);
69
70    // Save the Q substate to file
71    calSaveSubstate3Db(host_CA, Q, "./mod2_0000.txt");
72
73    // Define a device-side CA
74    device_CA = calclCADef3D(host_CA, context, program, device);
75
76      // Register a transition function's elementary process kernel
77    CALCLkernel kernel_transition_function = calclGetKernelFromProgram(&program,
          KERNEL_LIFE_TRANSITION_FUNCTION);
78
79    // Add transition function's elementary process
80    calclAddElementaryProcess3D(device_CA, &kernel_transition_function);
81
82    // Initialize the viewer
83    calglInitViewer("mod2 3D CA viewer", 1.0f, 400, 400, 40, 40, CAL_TRUE, 100);
84
85    //drawModel definition
86    struct CALGLRun3D * calUpdater = calglRunCLDef3D(device_CA,100,1,0);
87    drawModel = calglDefDrawModelCL3D(CALGL_DRAW_MODE_FLAT, "3D view", host_CA,
          calUpdater);
88    calglAdd3Db(drawModel, NULL, &Q, CALGL_TYPE_INFO_VERTEX_DATA,
          CALGL_TYPE_INFO_USE_NO_COLOR, CALGL_DATA_TYPE_DYNAMIC);
89    calglColor3D(drawModel, 0.5f, 0.5f, 0.5f, 1.0f);
90    calglAdd3Db(drawModel, Q, &Q, CALGL_TYPE_INFO_COLOR_DATA,
          CALGL_TYPE_INFO_USE_CURRENT_COLOR, CALGL_DATA_TYPE_DYNAMIC);
91    calglAdd3Db(drawModel, Q, &Q, CALGL_TYPE_INFO_NORMAL_DATA,
          CALGL_TYPE_INFO_USE_NO_COLOR, CALGL_DATA_TYPE_DYNAMIC);
92
93    calglMainLoop3D(argc, argv);
94
95    return 0;
96  }
```

Listing 6.23: An OpenCAL-CL implementation of the mod2 3D CA with openCAL-GL graphic output.

```
1   #include <OpenCAL-CL/calcl3D.h>
2
3   #define Q 0
4
5   __kernel void mod2TransitionFunction(__CALCL_MODEL_3D)
6   {
7     calclThreadCheck3D();
8
9     int i = calclGlobalRow();
10    int j = calclGlobalColumn();
11    int k = calclGlobalSlice();
12
13    int sum = 0, n;
14    CALint sizeOf_X = calclGetNeighborhoodSize();
15
16    for (n=0; n<sizeOf_X; n++)
17      sum += calclGetX3Db(MODEL_3D, Q, i, j, k, n);
18
19    calclSet3Db(MODEL_3D, Q, i, j, k, sum%2);
20  }
```

Listing 6.24: The OpenCAL-CL kernel implementing the mod2 3D CA elementary

process.

As noted, the host code is quite similar to that of the Game of Life (cf. Section 6.2), with the difference that here some OpenCAL-GL code is added for GUI and 3D graphic rendering, as already done for the $SciddicaT_{naive}$ example (cf. Section 6.3).

The main difference with respect the previous examples is that *mod2* is a 3D CA model. In fact, the 3D versions of the previously considered 2D headers are included at lines 3-7 and, accordingly, the host and device CA, as well as the single model substate, are declared as 3D objects at lines 18-21. Eventually, note that also library functions are in their 3D versions. Their meaning is self-explaining, since they correspond to their 2D counterparts already seen in previous examples.

Similarly, device-side code is equivalent to that of the Game of Life. However, since *mod2* is a 3D CA, the kernel in Listing 6.24 also gets the third integer coordinate for the cell being processed at line 11, by means of the `calclGlobalSlice()` function. Eventually, as for the host-side code, even the device-side library functions come in their 3D version.

`OpenCAL-GL/calgl3D.h` and `OpenCAL-GL/calgl3DWindow.h` headers are included. Moreover, the OpenCAL-GL functions, already seen in the 2D version in previous Section, are here in their 3D form (see, e.g., the `calglAdd3Db()` function at line 80). They are completely equivalent to the 2D versions and therefore will not be commented. Eventually, note that statements at lines 86-91 are commented. If comments are removed, you will end with some parts of the cellular space cut down from the visualization.

Figure 6.3 shows a screenshot of the *mod2* CA.

6.5 Reduction operations

OpenCAL-CL comes with some predefined global reduction operations. In order to use them, each desired reduction must be firstly registered to the device-side CA object, by specifying the substate on which the reduction has to be performed. Registered reductions are performed by OpenCAL-CL device-side in a transparent way to the user at each computational step, just after the application of elementary processes and before steering. Hence, reduction operation results are retrieved device-side within kernels by means of other predefined OpenCAL-CL functions. Note that, since reductions are evaluated after elementary processes, the values retrieved during the first computational step are equal to zero, which is the default value. Host- and device-side reduction registration and retrieving functions are listed in Tables 6.3 and 6.4, respectively.

Note that, host-side reduction functions return void and accept a pointer to a device-side CA object and a 0-based numerical handle corresponding to the device-side substate over which the reduction must be performed. Instead, device-side reduction retrieving functions return a CALreal value and only take the 0-based numerical handle corresponding to the device-side substate over which the reduction has been performed. Listings 6.25 and 6.26 show the prototypes of the reduction function for evaluating the sum of a double-precision 2D and the corresponding retriving function, respectively. Eventually, Listings 6.27 and 6.28 show host- and device-side example codes, respectively, in which a reduction operation is considered for evaluating the sum of all the elements within a given substate.

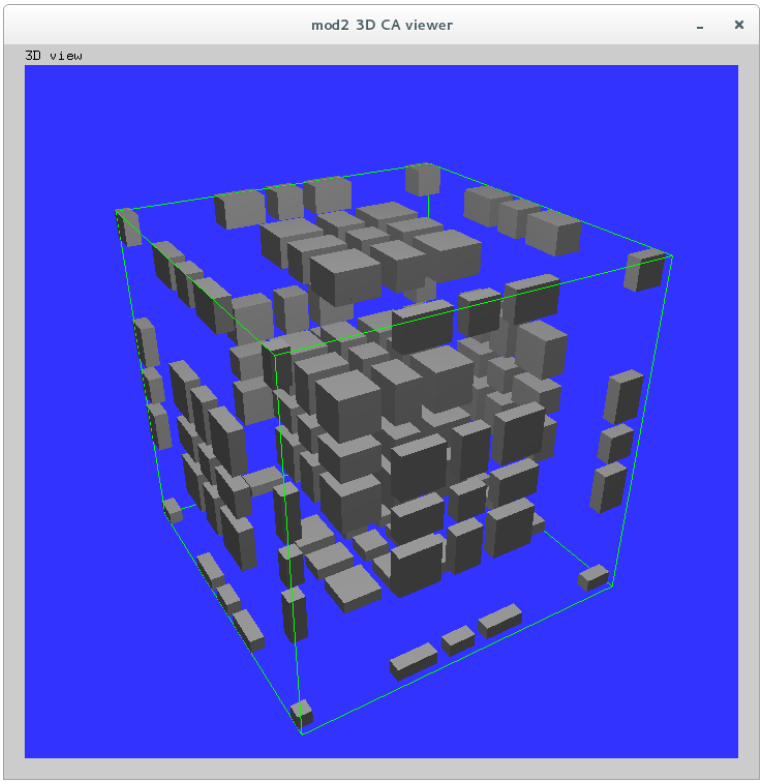


Figure 6.3: Screenshot of the *mod2* 3D CA viewer based on OpenCAL-GL.

Host-side reduction functions	Register a reduction to compute the ...
`calclAddReductionMax{2D|3D}{b|i|r}()`	maximum of a substate elements
`calclAddReductionMin{2D|3D}{b|i|r}()`	minimum of a substate elements
`calclAddReductionSum{2D|3D}{b|i|r}()`	sum of a substate elements
`calclAddReductionProd{2D|3D}{b|i|r}()`	product of substate elements
`calclAddReductionLogicalAnd{2D|3D}{b|i|r}()`	logical AND of substate elements
`calclAddReductionBinaryAnd{2D|3D}{b|i|r}()`	binary AND of substate elements
`calclAddReductionLogicalOr{2D|3D}{b|i|r}()`	logical OR of substate elements
`calclAddReductionBinaryOr{2D|3D}{b|i|r}()`	binary OR of substate elements
`calclAddReductionLogicalXOr{2D|3D}{b|i|r}()`	logical AND of substate elements
`calclAddReductionBinaryXor{2D|3D}{b|i|r}()`	binary AND of substate elements

Table 6.3: OpenCAL-CL host-side reduction registration functions.

```
void calclAddReductionSum2Dr(
     struct CALCLModel2D * device_CA,
     int devide_side_substate_handle
);
```

Listing 6.25: The calclAddReductionSum2Dr() host-side reduction registration function.

Device-side reduction functions	Get the value of a reduction operation for the ...
calclGetMax{2D|3D}{b|i|r}()	maximum of a substate elements
calclGetMin{2D|3D}{b|i|r}()	minimum of a substate elements
calclGetSum{2D|3D}{b|i|r}()	sum of a substate elements
calclGetProd{2D|3D}{b|i|r}()	product of substate elements
calclGetLogicalAnd{2D|3D}{b|i|r}()	logical AND of substate elements
calclGetBinaryAnd{2D|3D}{b|i|r}()	binary AND of substate elements
calclGetLogicalOr{2D|3D}{b|i|r}()	logical OR of substate elements
calclGetBinaryOr{2D|3D}{b|i|r}()	binary OR of substate elements
calclGetLogicalXor{2D|3D}{b|i|r}()	logical AND of substate elements
calclGetBinaryXor{2D|3D}{b|i|r}()	binary AND of substate elements

Table 6.4: OpenCAL-CL device-side reduction retrieving functions.

```
CALreal calclGetSum2Dr(
    int device_side_substate_handle
);
```

Listing 6.26: The calclGetSum2Dr() device-side reduction retrieving function.

```
1  #include <OpenCAL-CL/calcl2D.h>
2  #include <OpenCAL-CL/calcl2DReduction.h>
3  //...
4
5  // 0-based numerical handle to refer the Q substate device-side
6  #define DEVICE_Q 0
7
8
9  int main()
10 {
11   // ...
12
13   // Define a device-side CA
14   struct CALCLModel2D * device_CA = calclCADef2D(host_CA, context, program,
         device);
15
16   // Extract a kernel from program
17   CALCLkernel kernel_life_transition_function = calclGetKernelFromProgram(&
         program, KERNEL_LIFE_TRANSITION_FUNCTION);
18
19   // Register a transition function's elementary process kernel
20   calclAddElementaryProcess2D(device_CA, &kernel_life_transition_function);
21
22   // Register reduction operation to be computed after elementary processes
         and after steering
23   calclAddReductionSum2Di(device_CA, DEVICE_Q);
24
25   //...
26 }
```

Listing 6.27: An example of global reduction operation in a host-side OpenCAL-CL application.

```
#include <OpenCAL-CL/calcl2D.h>
#include <OpenCAL-CL/calcl2DReduction.h>


// 0-based numerical handle to refer the Q substate device-side
#define DEVICE_Q 0

__kernel void lifeTransitionFunction(__CALCL_MODEL_2D)
{
  calclThreadCheck2D();

  int i = calclGlobalRow();
  int j = calclGlobalColumn();

  // retrieve and print the result of the sum reduction
  // operation registered host-side
  int sum = calclGetSum2Di(DEVICE_Q);
  if(i==0 && j==0)
    printf("sum = %f \n", sum);

  //...
}
```

Listing 6.28: An example of global reduction operation in a device-side OpenCAL-CL application.