Accelerating Lava Flows Simulations with GPGPU and OpenCL

A. De Rango, M. Macrì, D. D'Ambrosio, W. Spataro
Department of Mathematics and Computer Science, University of Calabria, Italy

Abstract - *The introduction of the GPU (graphics processing units) has marked a revolution in the field of Parallel Computing allowing to achieve computational performance unimaginable until a few years ago. Widely adopted in the Scientific Computing Field, this hardware has proven to be extremely reliable and suitable to simulate Cellular Automata (CA) models for modeling complex systems whose evolution can be described in terms of local interactions. This paper presents an effective implementation of a well-known numerical model for simulating lava flows on Graphical Processing Units (GPU) based on the OpenCL (Open Computing Language) standard. Carried out experiments show that significant performance improvements in terms of speedup are achieved, adopting also some original optimizations strategies, confirming the validity of OpenCL and both low-cost and high-end graphics hardware as an alternative to expensive solutions for the simulation of CA models.*

Keywords: Cellular Automata, GPGPU, OpenCL, Parallel Software Tools, Modeling and Simulation.

1 Introduction

Numerical models are adopted in High Performance Computing (HPC) ([12]) for solving complex equation systems which rule the dynamics of complex systems as, for instance, a lava flow or a forest fire. In recent years, the introduction of the GPU (graphics processing units) has marked a revolution in the field of Parallel Computing allowing to achieve computational performance unimaginable until a few years ago. Nevertheless, GPU applications to the important field of Computational Fluid Dynamics (CFD) are increasing both for quantity and quality among the Scientific Community (e.g., [23], [11]).

With GPGPU (General Purpose computing with GPU) it is possible to obtain computational performances of a theoretical order of teraflops (thousands of megaflops), still characterized by production costs that are extremely low compared to classical parallel systems. GPGPU adopts use of the GPU for operations different from graphics rendering, for which these devices were originally designed. This method has been particularly widespread in 2007 with the release of CUDA by Nvidia, who introduced software and hardware specialized for GPGPU computing. As Nvidia, other GPU manufacturers adapted their devices to this new methodology and have released software development environments for the realization of parallel programs. Hardware manufacturers have released APIs (Application Programming Interface) compatible only with their devices and this limited the development of portable software. However, in 2008 the standard OpenCL (Open Computing Language) was released for the implementation of parallel programs on heterogeneous systems. Gradually, all major manufacturers of GPU and CPU have released their implementation of OpenCL, providing developers an instrument capable of producing portable software on a large number of devices. Today we have reached the conclusion that hybrid systems based on CPU and GPU represent the future of supercomputing.

Among the different methodologies used for modelling processes, such as numerical analysis, high order difference approximations and finite differences, Cellular Automata (CA) ([26]) has proven to be particularly suitable when the behaviour of the system to be modelled can be described in terms of local interactions. Originally introduced by von Neumann in the 1950s to study self-reproduction issues, CA are discrete computational models widely utilized for modeling and simulating complex systems. Regarding the modeling of natural complex phenomena, Crisci and co-workers proposed a method based on an extended notion of homogeneous CA, firstly applied to the simulation of basaltic lava flow, which makes the modeling of spatially extended systems more straightforward and overcomes some unstated limits of the classical CA, such as having few states and look-up table transition functions. Mainly for this reason, the method is known as Complex Cellular Automata (CCA) (or Macroscopic Cellular Automata [9] or Multicomponent Cellular Automata [1]).

This paper presents an implementation of a well-known, reliable and efficient CCA model adopted for lava flow risk assessment, namely the SCIARA model [22], in GPGPU environments. Tests performed on two types of GPU hardware, a AMD Sapphire 280x graphic card and a Tesla K40c computing processor, and by adopting difference implementation strategies, have shown the validity of this kind of approach.

In the following sections, after a brief description of the basic version of the SCIARA CCA model for lava flows, a quick overview of GPGPU paradigm together OpenCL is presented. Subsequently, the specific model implementation and performance analysis referred to benchmark simulations and of a real event are shown, while conclusions and possible outlooks are reported at the end of the paper.

2 Cellular Automata and the SCIARA lava flow simulation model

As previously stated, CA are dynamical systems, discrete in space and time. They can be thought as a regular *n*-dimensional lattice of sites or, equivalently, as an *n*-dimensional space (called cellular space) partitioned in cells of uniform size (e.g. square or hexagonal for n=2), each one embedding an identical finite automaton. The cell state changes by means of the finite automaton transition function, which defines local rules of evolution for the system, and is applied to *each* cell of the CA space at discrete time steps. The states of neighbouring cells (which usually includes the central cell) constitute the cell input. The CA initial configuration is defined by the finite automata states at time t=0. The global behaviour of the system emerges, step by step, as a consequence of the simultaneous application of the transition function to each cell of the cellular space.

When dealing with the modelling of spatial extended dynamical systems, CCA can represent a valid choice especially if their dynamics can be described in terms of local interaction at macroscopic level. Examples of successful applications of CCA include the simulation of lava flows [6], debris flows [16], density currents [20], water flux in unsaturated soils [17], soil erosion by rainfall [17] as well as pyroclastic flows [5], and forest fires [25].

For the OpenCL parallelization of the CA, the release fv2 of the SCIARA numerical model for simulating lava flows was adopted. SCIARA is a family of bi-dimensional CCA lava flow models, successfully applied to the simulation of many real cases such as the 2001 Mt. Etna (Italy) Nicolosi lava flow [6] and the 1991 Valle del Bove (Italy) lava event [2], which occurred on the same volcano and was employed for risk mitigation. In formal terms, the SCIARA-fv2 model [22] is defined as:

$$SCIARA\text{-}fv2 = < R, L, X, Q, P, \tau, \gamma >$$

where:

- R is the set of square cells covering the bi-dimensional finite region where the phenomenon evolves;
- $L \in R$ specifies the lava source cells (i.e. craters);
- X = {(0, 0), (0, 1), (-1, 0), (1, 0), (0, -1), (-1, 1), (-1,-1), (1, -1), (1, 1)} identifies the pattern of cells (Moore neighbourhood) that influence the cell state change; in the following we will refer to cells by indexes 0 (for the central cell) through 8;
- $Q = Q_z \times Q_h \times Q_T \times Q_f^8$ is the finite set of states, considered as Cartesian product of "substates". Their meanings are: cell altitude a.s.l., cell lava thickness, cell lava temperature, and lava thickness outflows (from the central cell toward the eight adjacent cells), respectively;
- $P=\{w, t, T_{sol}, T_{vent}, r_{Tsol}, r_{Tvent}, hc_{Tsol}, hc_{Tvent}, \delta, \rho, \varepsilon, \sigma, c_v\}$ is the finite set of parameters (invariant in time and space) which affect the transition function (please refer to [22] for their specifications);
- $\tau : Q^9 \rightarrow Q$ is the cell deterministic transition function, divided in *elementary processes* and applied to each cell at each time step, which describes the dynamics of lava flows, such as cooling, solidification and lava outflows from the central cell towards neighbouring ones. In particular, In the fv2 version of SCIARA, the so called *elementary processes* [9] describing the cell's transition function are: (i) σ1, which determines lava outflows based on an opportune version of the Minimisation Algorithm of Differences; (ii) σ2, which determines lava thickness computation; (iii) σ3, which determines lava temperature and (iv) σ4, which determines the eventual lava solidification.
- $\gamma : Q_h \times \mathrm{N} \rightarrow Q_h$ specifies the emitted lava thickness from the source cells at each step $k \in N$ (N is the set of natural numbers).

3 OpenCL and GPGPU programming

In recent years, mainly due to the stimulus given by the increasingly demanding performance of gaming and graphics applications in general, graphic cards have undergone a huge technological evolution, giving rise to highly parallel devices, characterized by a multithreaded and multicore architecture and with very fast and large memories. A GPU can be seen as a computing device that is capable of executing an elevated number of independent threads in parallel. In general, a GPU consists in a number (e.g., 16) of SIMD (Single Instruction, Multiple Data) multiprocessors (or compute units) with a limited number of floating-point processors that access a common shared-memory within the multiprocessor.

OpenCL [21] is a framework that allows the user to perform tasks *both* on GPU than on CPU. The OpenCL routines can be performed on the GPU or CPU which are produced by major parallel computing brands, such as AMD, Nvidia, and Intel. Specifically OpenCL is nonproprietary, because it is based on a public standard, and can be freely downloaded.

The goal of OpenCL is thus to unify the programming model software to run the code on heterogeneous devices. In fact, today OpenCL supports different platforms that include CPUs (e.g. Intel, AMD, ARM, etc), GPUs (e.g., AMD, Intel, Nvidia), besides FPGA and DSP (Digital Signal Processors). As known, in Parallel Computing developers can create and manipulate concurrent task. When developers need to program a solution in OpenCL, they must decompose the problem in different tasks. Parallel programming assigns computational task to multiple processing elements that are executed at the same time. In the OpenCL language, these tasks are called *kernels*. A kernel is a special function written in C99 that is intended to be performed by one or more OpenCL devices. The kernels are sent to the devices through the host program. The host program is written in C / C ++

and runs on the user’s development system. The host application manages the connected devices using a container called *context*. To create a kernel, the host selects a function from a container called *program*. Subsequently, it associates the kernel with its data and sends it to a structure called *command queue*. The command queue is the mechanism by which the host tells devices what to do and subsequently, when a kernel is queued, the device will perform the corresponding function. An OpenCL application can configure different devices to perform different tasks, and each task can operate on different data. OpenCL provides thus a full task-parallelism. Figure 1 shows the kernel distribution among OpenCL-compliant devices.

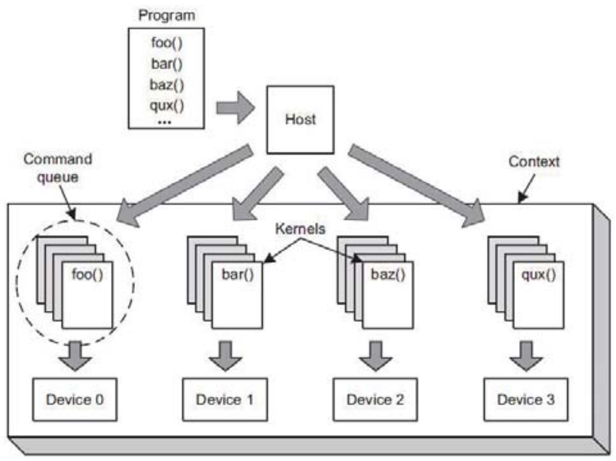

Figure 1: Assignment of the kernel to the devices contained in the *context* structure (figure taken from [21]).

Thus, in order to create an OpenCL application it is necessary to:

- Create a host program to manage the available devices and assign them the kernels to be performed;
- Create kernels, i.e. the routines to be performed on the selected devices from the host program.

3.1 Creation of the host program

The host program of an OpenCL application is written in C/C ++, but there are libraries created by third parties that allow to develop an application using the java and python languages [21]. The library defines six essential structures for the creation of the program host: *platform*, *device*, *context*, *program*, *kernel*, and *command queue*.

To access the computing devices on the system, OpenCL defines three structures: *platform*, *device* and *context*. Every manufacturer that supports OpenCL releases an SDK (software development kit) which contains an implementation of OpenCL compatible with its devices. The structure *platform* provides access to the OpenCL implementations installed on the system and to use all devices of the manufacturer. For example, installing a Nvidia SDK, all Nvidia devices on the system can be accessed via its platform. Devices are represented by the *device* structure and to be used they must be inserted into a container called *context*. In the host program, several context instances containing more devices can be defined. However, devices belonging to different contexts cannot communicate with each other and cannot be inserted in the same context devices belonging to different platform (for example, one can not create a context containing a AMD and a Nvidia device).

3.2 Kernel assignment and execution

The structure that allows communication between the host program and OpenCL devices is the *command queue*. Through the command queue, not only a kernel is assigned to a device, but can also perform data transfer operations, between two devices or between a device and the host program. Moreover, thanks to this structure one can carry out synchronizations between different kernels and profiling operations. To assign a kernel to a device one needs to decide how the data should be partitioned and assigned to compute units. Depending on the partitioning chosen by the user, kernel instantiations called *work-items* are carried out. Each work-item (i.e., thread), represents an execution of the same kernel but on different portions of data (i.e., in a SIMD fashion). For the assignment of kernels, OpenCL provides two functions:

`clEnqueueTask`. The task assigned by the host program to the device will run as a single work-item.

`clEnqueueNDRangeKernel`. The task assigned by the host program to the device will be split into multiple work items that will be executed in parallel.

The host program must therefore define the number of work items to be used and optionally may decide to divide the work items in groups called *work-groups*. The work items contained in a work-group have a shared memory block (local memory) which permits to access data much faster than the global memory shared by all the work items. Furthermore, the work items within a work-group can be synchronized. As expected, the latter of the two functions is fundamental, as it allows to perform tasks in parallel.

3.3 Data Transfer

Generally, the execution of a task includes the processing of data. From the moment the host program assigns a kernel to a device, it is necessary that the device has the data that is used to run the kernel. To send data to the device the function `clSetKernelArg` is used to allow the association of a data set to an argument of the kernel function. Basic data types that can be associated to the kernel are:

Pointers to primitive data types Associates a given primitive type.

Pointers to buffer object Associates a large set of data. A buffer object (represented by the structure `cl_mem`) can be

created using the function `clCreateBuffer`, but in order that data transfer takes place correctly, data must be stored on the host program contiguously.
After a buffer object is associated to a kernel, it is possible to reuse the same structures to transfer data both between two devices, between a device and the host program, etc.

3.4 Memory hierarchy

As reported before, a kernel function can be associated to the data required for processing. The host program is responsible for transferring the data to the device. Each device has different memory spaces (cf. Figure 2) in which to store the data received from the host program:

Global memory. Stores data accessible by all the work items for both reading and writing.

Constant memory. Similar to the global memory but data can be accessed in read-only.

Local memory. Stores data accessed by the work items contained in the same work-group.

Private memory. Stores data accessible by a single work-item.

All data from the host program is initially stored in the global/constant or private memory (the local memory can be allocated only by the host program but not initialized). The global/constant memory is larger than the others, but access to it is slower. Work items can indeed access the local memory much faster (100×) than that in the global/constant memory. Access to private memory is faster but its dimension is very small. With regards to constant memory, some devices have an apposite portion of memory, in other cases the constant memory space coincides with that of the global memory. To specify the memory space in which a given data must be stored the qualifiers `__global`, `__constant`, `__local`, `__private` are used. If omitted, data will be stored in private memory.

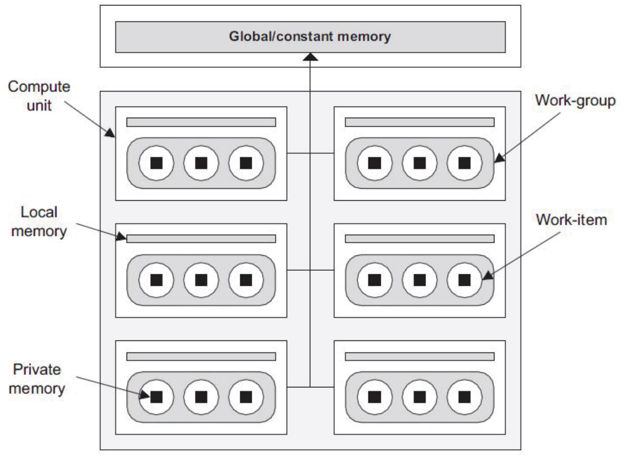

Figure 2: The OpenCL Memory model (figure taken from [21]).

3.5 Data Partitioning

The function clEnqueueNDRangeKernel described in subparagraph 3.3 allows to perform a task in parallel. To use this feature, one must:

- define on how many dimensions data is distributed (a two-dimensional matrix, etc.);
- define the number of work items for each dimension;
- define the number of work items in a work-group for each dimension.

To perform a task in parallel each work-item must be able to access the data portion that has been assigned to it. To each work-item is associated an ID that distinguishes it from all others, and generally these IDs are used to partition the data in a typical SPMD fashion. For example, suppose that the data consist of an array of `n` elements and also to have `n` work-items with their relative ID. Data can be partitioned by associating each work item to the array element with index corresponding to the ID. Moreover, for the work-items an ID that identifies them in a work-group is also associated. In this case, the purpose is to give the possibility to partition the data, even if here the partitioning occurs within a work-group. Other information that is accessible to the work items for the partitioning of data are:
• the total number of work items for each dimension;
• the total number of work-item contained in a work-group for each dimension;
• the total number of work-group
• the ID of the work-group to which the work-item belongs.

4 Implementation of the Sciara model

As previously stated, CA models, such as SCIARA, can be straightforwardly implemented on parallel computers due to their underlying parallel nature. In fact, since CA methods require only next neighbor interaction, they are very suitable and can be efficiently implemented even on GPUs. In literature, to our knowledge, few examples of Complex Cellular Automata modeling with GPUs are found, while some interesting CA-like implementations, such as Lattice Boltzmann kernels, are more frequent (e.g., [24], [15]).

The approach here adopted resembles many approaches in the field: typically, the CA parallel implementation involves two memory regions, which will be called `CAcurrent` and `CAupdated`, representing the current and next states for the cells respectively. For each CA step, the neighbouring values from `CAcurrent` are read by the local transition function, which performs its computation and writes the new state value into the appropriate element of `CAupdated`.

In accordance to the recent literature in the field (e.g., [3], [10]), in the GPGPU parallel implementation of the model, most of the automaton data (i.e. both the `CAcurrent` and `CAupdated` memory areas) was stored in the GPU global memory. In addition, the initialization of the CA (`CAupdated`) implies a copy from CPU to GPU. At the end of the computation, results from the device are copied back to the host through a GPU to CPU data transfer. In addition, at each step, in order to update the status of the previous step with the current one, a copy between the two CA data buffer memory areas on the device takes place.

A crucial step is to identify the set of instructions (i.e. the elementary processes of the transition function) that can be performed independently on the cells of the CA space. The instructions will be invoked in parallel using a OpenCL kernel for each of elementary process. Note that at each step of the simulation, only a few cells of the entire cellular space are involved in the computation. Thus, a typical problem related to GPGPU parallelization (as reported later) which can affect the speedup of the model, can lead to an overuse of computationally inactive work-items.

In particular, we will describe two different strategies that were adopted for an efficient parallelization of the SCIARA-fv2 simulation model. The first of these, defined as Whole Space Strategy (WS), is based on a naive approach to the problem consisting in the use of only global memory which is shared by the totality of work-items that make up the mapping grid. The second version, called Active Cells Strategy (AC), has a significant performance improvement of the algorithm, achieved thanks to the adoption of a data structure that manages the CA computationally active cells. In this strategy the computation takes place within a grid of work-items that adapts dynamically to the active cells.

4.1 Naïve implementation

The first strategy for the parallelization of the SCIARA-fv2 model is based on a one work-item - one cell approach, where each cell in the cellular space is computed by OpenCL work-item. The Whole Space (WS) strategy version involves the use of global only memory, where each kernel runs on a grid of work-items divided into work-groups and mapped on the entire cellular space. Work items are thus executed in parallel and synchronized each time an elementary process ends. Importantly, the elementary processes must be defined in such a way that the work items are executed independently from each other and that each work-item accesses exclusively to the portion of data that it has been assigned, since OpenCL does not provide mechanisms for synchronization of work-item belonging to different work-group [21]. The host program assigns the kernel to devices by sending them the CA model data (sub-states, type of neighborhood, size of the space cell, etc.). The execution cycle is then managed by the host program, while the transition function is performed on the devices.

The following excerpt shows execution cycle:

```
steps = 0;
wiNum = ROWS * COLS;
while (steps < maxStep) { // CA steps
    // kernel execution for each elementary process
    clEnqueueNDRangeKernel(queue,updateVentsEmissi
on, dimNum, NULL, wiNum, NULL, 0, NULL, NULL);
    // update substates
    clEnqueueCopyBuffer(queue,          CAupdated,
CAcurrent, 0, 0, bufDim, 0, NULL, NULL);

    clEnqueueNDRangeKernel(queue,   empiricalFlows,
dimNum, NULL, wiNum, NULL, 0, NULL, NULL);
    clEnqueueCopyBuffer(queue,          CAupdated,
CAcurrent, 0, 0, bufDim, 0, NULL, NULL);

    clEnqueueNDRangeKernel(queue,     width_update,
dimNum, NULL, wiNum, NULL, 0, NULL, NULL);
    clEnqueueCopyBuffer(queue,          CAupdated,
CAcurrent, 0, 0, bufDim, 0, NULL, NULL);

    clEnqueueNDRangeKernel(queue,
updateTemperature, dimNum, NULL, wiNum, NULL, 0,
NULL, NULL);
    clEnqueueCopyBuffer(queue,          CAupdated,
CAcurrent, 0, 0, bufDim, 0, NULL, NULL);

    steps++;
}
```

As an example, the following excerpt reports the kernel definition for the `empiricalFlows` lava outflow computation elementary process (i.e., σ1).

```
__kernel  void  empiricalFlows(__global  double  *
SUBSTATES, Parameters parameters) {

int i = get_global_id(0);
int j = get_global_id(1);
int SLT = 2; //lava thickness substate index
int F = 3; //outflows substates index

// check if cell contains lava
if (SUBSTATES[ROWS*COLS*SLT +(i*ROWS + j)] > 0) {
   double outflows[MOORE_NEIGHBORS];
   outflowsMin(SUBSTATES,    i,    j,    outflows,
parameters); //minimization algorithm application

    // update outflows substate
    for (int k = 1; k < MOORE_NEIGHBORS; k++)
       if (outflows[k] > 0)
         SUBSTATES[ROWS*COLS*(F+k-1) +(i*ROWS + j)]
         = outflows[k];
    }
}
```

While a similar straightforward strategy has proven to be effective in other parallelizations and applications (e.g. [14], [8], [7]), the speedups here achieved were not quite positive, probably due to the excessive use of computationally inactive threads and overuse of global memory. At the contrary, the following approach has given more positive results and can be considered as a starting point for more sophisticated applications.

4.2 Active cells optimization

The *active cells optimization strategy* (AS) allows to apply the transition function only on the *active* cells, omitting those that are in a quiescent state. The active cells are contained within a list and elementary processes can add to the list new cells, or remove them. An array designed to contain all active cells is initially allocated. Specifically, for each active cell, a work-item is instantiated and can add new cells to the list, or remove them. However, to maintain the list in a consistent state, accesses must be performed by the work-item in an exclusive manner. Unfortunately, OpenCL doesn’t support exclusive accesses to global data, so an alternative approach was here adopted. In particular, each cell is associated with a Boolean value that indicates whether the cell is active or inactive. The work-item, in order to add or remove a cell, must change this value. This information is used by a *stream compaction* algorithm to build the list of active cells.

Stream compaction algorithms ([4], [18]) are used to remove unwanted items in a set of scattered data. In this case, data is located in an array of Boolean values whose elements correspond to the cells of the cellular space. Items with a true value correspond to the active cells, while false ones correspond to the inactive cells. The implemented stream compaction algorithm, adapted from [13] takes as input this array and computes an array containing only the active cells. Suppose we have `p` work-items and a Boolean array with `n` elements, with `n> p`. The array is divided into `p` equal parts. The algorithm consists of three phases:

1. Each work-item counts the number of active cells in its part of the array.
2. Each work-item computes the array index of the output array from which it can start entering the active cells contained in the portion of its array.
3. Each work-item enters the active cells of its portion in the output array starting from the offset calculated in previous step.

Figure 3 shows the functioning of the algorithm for a matrix representing a CA space size of 4×4 with 6 active cells highlighted in red. The array used to track active cells is given as input to the stream compaction algorithm that outputs the list with only the active cells.

The second stage uses a two phase algorithm called *prefix sum*. Specifically, the algorithm takes as input the array calculated in the previous step containing the sum of the active cells calculated by each work-item. The array is seen as a balanced tree where its elements are the nodes of the tree. In the first phase, the tree is crossed from the leaves to the root calculating, for each level of the tree, the partial sums of the nodes of the previous level (by a parallel reduction pattern). In second phase, the tree is traversed from the root (containing the total number of active cells) to the leaves. At each iteration, each node sets the value of the right child to the sum of its value and the value of the left child. In addition, each node sets the value of its left child to its value. At the end of this phase, an array is created that specifies, at each location, the position from which each work-item can write its active cells in the global active cells output array.

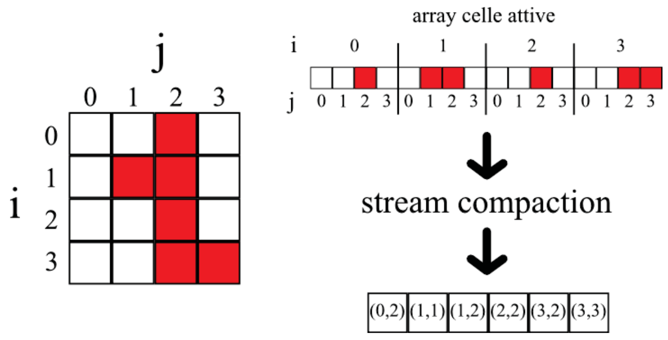

Figure 3: Stream compaction algorithm example referred to a 4 x 4 CA.

5 TESTS AND PERFORMANCE RESULTS

Two graphic devices were adopted for experiments: a NVIDIA high-end Tesla K40c and a AMD Sapphire 280x graphic card, both with a theoretical peak performance about 3,5 GFLOPS. In particular, the Tesla computing processor has 2880 stream processors (i.e., CUDA cores) and 15 compute units, 12 GB global memory and high-bandwidth communication between CPU and GPU, whereas the AMD graphic card has 2048 stream processors, 32 compute units and 3 GB global memory. The sequential SCIARA-fv2 reference version was implemented on a 2.8 GHz Intel Quad-core Xeon based desktop computer. The sequential CPU version is identical to the versions that were developed for the GPUs, that is, at every step, the CA space array is scrolled and the transition function applied to each cell of the CA where lava is present.

A first test regarded the simulation of well-known and documented real lava flow event, the Mt. Etna Nicolosi event [6] occurred in July, 2001. The simulation was carried out for 10000 steps, considering two craters for lava flow emission. In order to further stress the efficiency of the GPU version, further benchmarks experiments (stress test) were performed by considering 200 lava sources equally spaced over the area. Moreover, the two parallelizations (i.e., Whole Space and Active Cells strategies) reported in section 4 were considered. Table 1 reports the results of tests carried out for experiments, where the CA space is a 517 × 378 two-dimensional grid.

From all performed experiments, we can note that in all cases the versions using the active cells optimization are more efficient. As expected, this is due to the fact that the optimization does not apply the transition function to cells which are in a quiescent state, and thus many unnecessary calculations are not executed. Instead, in the standard implementation, there is an excessive use of computationally inactive threads and overuse of global memory.

For the first series of tests, which takes into account only two sources lava, we can see that the WS parallel version reaches a speedup of 22× (23× for the Tesla K40c) compared to its sequential version. In the case of the parallel version with active cells (AS) optimization, performance reaches a maximum speedup of only 2× (3× for the Tesla K40c), with respect to its corresponding sequential active cells version. This is due to the fact that with only two sources of lava the active cells are relatively few and therefore also the sequential implementation results extremely efficient. Moreover, performance of the AMD Sapphire 280x GPU is unexpectedly comparable to that of the Nvidia GPU K40c for the standard WS version, probably due the high boost clock of the AMD hardware with respect to the lower GPU clock of the Nvidia card (1000 MHz vs 745 MHz). However, performances related to the optimized AS version in terms of execution times are better for the Nvidia hardware, probably due to the higher number of streaming processors (i.e., 2880 vs 2048) that can be fully exploited.

As for the stress test, we can observe a speed up of 41× (40× for the Tesla K40c) for the parallel version that is not optimized, and a speedup of 39× (41× for the Tesla K40c) for the parallel version with active cells optimization compared to the corresponding sequential versions. Here, we can see that the speedup of the versions with active cells optimization is comparable with the versions without optimizations, due to the relative equal (and elevated) number of cells involved in the stress test. Even in this case, both considered hardware give the same performance, as for the previous test. A likely explanation is due to the fact that on one hand the Nvidia hardware has a higher number of streaming processors while, on the other, the AMD card has more compute units, which basically compensate both hardware computing capabilities.

Table 1: Execution times of experiments (in seconds) carried out for evaluating the performance the GPU version of the SCIARA CCA lava-flow model on the considered hardware. Experiments refer to the Whole Space (WS).

CA dim / Device	**Xeon (sequential)**	**K40c (WS)**	**Sapphire (WS)**
517 ×378 (2 craters)	1122	49	52
517 ×378 (200 craters – Stress Test)	2790	69	68

Even if timings achieved for the single case simulation cannot be considered positive, the stress test experiments have revealed the full suitability of the parallel system for intensive computations like applications, such as for the construction of hazard maps (e.g., [7], [16]). Typically, the most general approach for computing a hazard map in a extended area consists of a Monte Carlo approach in which a high number (e.g. thousands) of different simulations are carried out and on geological-geomorphological field survey and statistical analysis.

Table 2: Execution times of experiments (in seconds) carried out for evaluating the performance the GPU version of the SCIARA CCA lava-flow model on the considered hardware. Experiments refer to the Active Cells (AC) strategies (see text).

CA dim / Device	**Xeon (sequential)**	**K40c (AC)**	**Sapphire (AC)**
517 ×378 (2 craters)	62	22	30
517 ×378 (200 craters – Stress Test)	2005	49	51

Eventually, to test if single-precision data can be considered sufficient for SCIARA simulations, tests were carried out on the 2001 lava flow event (10000 CA steps) and compared results produced by the GPU version with those produced by the CPU (sequential) version with double precision CPU implementation (i.e., `double` type variables). Comparison results were satisfactory, since the areal extensions of simulations resulted the same, except for few errors of approximation in a limited number of cells. In particular, comparing the GPU version with the CPU single-precision version approximation differences at the third significant digit were only for 4% of cells, while differences were less for remaining cells. Differences were even minor compared to the previous case by considering the single precision GPU version and a CPU version which adopts double-precision variables.

6 CONCLUSIONS

This paper reports the implementation of a Complex Cellular Automata model using GPU architectures. As shown, the OpenCL technology, in combination with the an efficient memory management, can produce a very efficient version of the SCIARA-fv2 lava flow simulator. Several tests were carried out to evaluate the implemented parallelizations. In particular, tests using the GPU Sapphire 280x show that the parallel version, without optimizations, achieved a 41× speedup compared to its sequential version. The parallel version with optimization active cells also reached a speedup of 41× compared to its corresponding sequential one. As expected, in all cases, versions using the active cells optimization have resulted to be more efficient than versions without the optimization.

Future work will also regard the exploitation of graphic hardware for the construction of hazard maps, such as in [16] which are fundamental for determining locations that are subject to future events and their related risk. The positive performances obtained for the more intensive computations (stress test) will imply the extension of the AC strategy in a multi-simulation context, by using OpenCL to accelerate simultaneous concurrent SCIARA-fv2 lava flows simulations.

The results obtained on the SCIARA model are therefore to be considered positive, but further testing should be performed to assess the functionality of the adopted strategies on other models and their ability to fruitfully exploit parallel systems resources.

Acknowledgments - This research was partially funded by the UE POR FSE CALABRIA PIA 2010 “Laboratorio in Campo” Project DDG n. 17198. Authors also gratefully acknowledge the support of NVIDIA Corporation for this research.

REFERENCES

[1] M.V. Avolio, S. Di Gregorio, W. Spataro, G.A. Trunfio. “A theorem about the algorithm of minimization of differences for multicomponent cellular automata”. In: Sirakoulis GC, Bandini S, editors, ACRI. Springer, volume 7495 of Lecture Notes in Computer Science, 289-298, 2012.

[2] D. Barca, G.M. Crisci, Di Gregorio, S., Nicoletta, F. “Cellular Automata for simulating lava Flows: A method

and examples of the Etnean eruptions". Transport Theory and Statistical Physics, 23, 195-232, 1994.

[3] G. Bilotta, E. Rustico, A. Hérault, A. Vicari, G. Russo, C. Del Negro, G. Gallo. "Porting and optimizing MAGFLOW on CUDA", Annals of Geophysics 5 (54), 2011.

[4] M. Billeter, O. Olsson, U. Assarsson U. "Efficient stream compaction on wide SIMD many-core architectures". In: Proceedings of the Conference on High Performance Graphics 2009. ACM, 159–166, 2009.

[5] G.M. Crisci, S. Di Gregorio, R. Rongo, W. Spataro. "PYR: a Cellular Automata model for pyroclastic flows and application to the 1991 Mt. Pinatubo eruption". Future Generation Computer Systems 21 (7), 1019-1032, 2005.

[6] G.M. Crisci, S. Di Gregorio, R. Rongo, W. Spataro. "The simulation model SCIARA: the 1991 and 2001 at Mount Etna". Journal of Vulcanology and Geothermal Research, 132, 253-267, 2004.

[7] D. D'Ambrosio, G. Filippone, D. Marocco, R. Rongo, W. Spataro. "Efficient application of GPGPU for lava flow hazard mapping". The Journal of Supercomputing, vol. 65, no. 2, 630–644, 2013.

[8] M. De La AsunciòN, J.M. Mantas, M.J. Castro, E. D. Fernàndez-Nieto. "A MPI-CUDA implementation of an improved Roe method for two-layer shallow water systems". Journal of Parallel and Distributed Computing, 72, 9, 1065–1072, 2012.

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

[10] S. Di Gregorio, G. Filippone, W. Spataro, G.A. Trunfio. "Accelerating wildfire susceptibility mapping through GPGPU". Journal of Parallel and Distributed Computing 73: 1183 – 1194, 2013.

[11] M. Domínguez Jose, J.C. Crespo Alejandro, Gómez-Gesteira Moncho. "Optimization strategies for CPU and GPU implementations of a smoothed particle hydrodynamics method". Computer Physics Communications, 184, 3, 617-627, 2013.

[12] A. Grama, G. Karypis, V. Kumar, A Gupta. "An Introduction to Parallel Computing: Design and Analysis of Algorithms", Second Edition. USA: Addison Wesley, 2003.

[13] M. Harris, S. Sengupta, J.D Owens. "Parallel prefix sum (scan) with CUDA", GPU Gems 3 (39), 851-876, 2007.

[14] D. Jacobsen, J. C. Thibault, , I. Senocak. "An MPI-CUDA implementation for massively parallel incompressible flow computations on Multi-GPU clusters". In: American Institute of Aeronautics and Astronautics (AIAA) 48th Aerospace Science Meeting Proceedings, 2010.

[15] F. Kuznik, C. Obrecht, G. Rusaouen, J.J. Roux. "LBM based flow simulation using GPU computing processor". Computers and Mathematics with Applications, 59, 2380–2392, 2010.

[16] F. Lucà, D. DAmbrosio, G. Robustelli, R. Rongo, and W. Spataro. "Integrating geomorphology, statistic and numerical simulations for landslide invasion hazard scenarios mapping: an example in the Sorrento peninsula (italy)". Computers & Geosciences, vol. 67, 163–172, 2014.

[17] G. Mendicino, A. Senatore, G. Spezzano, S. Straface. "Three-dimensional unsaturated flow modeling using cellular automata". Water Resources Research, 42, 2006.

[18] H. Nguyen. Gpu Gems 3. First. Addison-Wesley Professional, 2007.

[19] NVIDIA CUDA C Programming Guide, 2011a. Available from: http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz3a2J6b3gl [accessed May 2015]

[20] T. Salles, S. Lopez, M. Cacas, T. Mulder. "Cellular automata model of density currents". Geomorphology, 88: 1 – 20, 2007.

[21] M. Scarpino. OpenCL in action. Manning Publications Co., 2012.

[22] W. Spataro, M.V. Avolio, V. Lupiano, G.A. Trunfio, R. Rongo, D. D'Ambrosio. "The latest release of the lava flows simulation model sciara: First application to Mt Etna (Italy) and solution of the anisotropic flow direction problem on an ideal surface". Procedia Computer Science 1: 17-26, 2010.

[23] J.C. Thibault, I. Senocak. "Accelerating incompressible flow computations with a Pthreads-CUDA implementation on small-footprint multi-GPU platforms". The Journal of Supercomputing, 59, 2, 693 – 719, 2012.

[24] J. Tolke. "Implementation of a lattice Boltzmann kernel using the compute unified device architecture developed by NVIDIA". Comput. Vis. Sci., 13 1, 29–39, 2008.

[25] G.A. Trunfio, D. D'Ambrosio, R. Rongo, W. Spataro, S. Di Gregorio. "A new algorithm for simulating wildfire spread through cellular automata". ACM Transactions on Modeling and Computer Simulation (TOMACS), 22, 6, 2011.

[26] J. von Neumann (Edited and completed by A. Burks),. Theory of self-reproducing automata. USA: University of Illinois Press, 1966.