GPGPU PROGRAMMING AND CELLULAR AUTOMATA: IMPLEMENTATION OF THE SCIARA LAVA FLOW SIMULATION CODE

Giuseppe Filippone[(a)] William Spataro[(b)], Giuseppe Spingola[(c)], Donato D'Ambrosio[(d)], Rocco Rongo[(e)], Giovanni Perna[(f)] , Salvatore Di Gregorio[(g)]

[(a) (b) (c) (d) (f) (g)]Department of Mathematics and HPCC, University of Calabria, Italy
[(e)] Department of Earth Sciences and HPCC, University of Calabria, Italy

[(c)]filippone@mat.unical.it, [(a)]spataro@unical.it, [(b)]g.spingola@gmail.com,
[(d)]d.dambrosio@unical.it, [(e)]rongo@unical.it, [(f)]gioper86@gmail.com, [(g)]dig@unical.it,

ABSTRACT
This paper presents an efficient implementation of a well-known computational model for simulating lava flows on Graphical Processing Units (GPU) using the Compute Unified Device Architecture (CUDA) interface developed by NVIDIA. GPUs are specifically designated for efficiently processing graphic datasets. However, recently, they are also being exploited for achieving exceptional computational results even for applications not directly connected with the Computer Graphics field. We here show an implementation of the SCIARA Cellular Automata model for simulating lava flows on graphic processors using CUDA. Carried out experiments show that significant performance improvements are achieved, over a factor of 100, depending on the problem size, adopted device and type of performed memory optimization, confirming how graphics hardware can represent a valid solution for the implementation for Cellular Automata models.

Keywords: Cellular Automata, Lava flows simulation, GPGPU programming, CUDA.

1. INTRODUCTION

High Performance Computing (HPC) (Grama et al. 2003) adopts numerical simulations as an instrument 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, Parallel Computing has undergone a significant revolution with the introduction of GPGPU technology (General-Purpose computing on Graphics Processing Units), a technique that uses the graphics card processor (the GPU – Graphics Processing Unit) for purposes other than graphics. Currently, GPUs outperform CPUs on floating point performance and memory bandwidth, both by a factor of roughly 100. As a confirmation of the increasing trend in the power of GPUs, leading companies such as Intel have already integrated GPUs into their latest products to better exploit the capabilities of their devices, such as in some releases of the Core i5 and Core i7 processing units. Although the extreme processing power of graphic processors may be used for general purpose computations, a GPU may not be suitable for every computational problem: only a parallel program that results suitable and optimized for GPU architectures can fully take advantage of the power of these devices. In fact, the performance of a GPGPU program that does not sufficiently exploit a GPU's capabilities can often be worse than that of a simple sequential one running on a CPU, such as when data transfer from main memory to video memory results crucial. 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., Tolke and Krafczyka 2008, Zuo and Chen 2010).

Among the different methodologies used for modelling geological processes, such as numerical analysis, high order difference approximations and finite differences, Cellular Automata (CA) (von Neumann 1966) 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. Well known examples are the Lattice Gas Automata and Lattice Boltzmann models (Succi 2004), which are particularly suitable for modelling fluid dynamics at a microscopic level of description. However, many complex phenomena (e.g. landslides or lava flows) are difficult to be modeled at such scale, as they generally evolve on large areas, thus needing a macroscopic level of description. Moreover, since they may also be difficult to be modelled through standard approaches, such as differential equations Macroscopic Cellular Automata (MCA) (Di Gregorio and Serra 1999) can represent a valid alternative. Several successful attempts have been carried out regarding solutions for parallelizing MCA simulation models (e.g., D'Ambrosio and Spataro 2007). In this research context, the CAMELot virtual laboratory and the libAuToti scientific library represent valid solutions for implementing and automatically parallelizing MCA models on distributed memory machines while, for shared memory architectures, some effective OpenMP parallelizations have been implemented for CA-like models, such as for fire spread simulations, Lattice Boltzmann models or lava flow modeling (Oliverio et al. 2011). However, few examples of GPGPU applications for CA-like models do exist (Tolke 2008) and to our knowledge, none regarding the MCA approach. This paper presents a implementation of a

well-known, reliable and efficient MCA model widely adopted for lava flow risk assessment, namely the SCIARA model (Rongo et al. 2008), in GPGPU environments. Tests performed on two types of GPU hardware, a Geforce GT 330M graphic card and a Tesla C1060 computing processor, have shown the validity of this kind of approach.

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

2. CELLULAR AUTOMATA AND THE SCIARA MODEL FOR LAVA FLOW SIMULATION

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, MCA can represent a valid choice especially if their dynamics can be described in terms of local interaction at macroscopic level. Well known examples of successful applications of MCA include the simulation of lava (Crisci et al. 2004) and debris flows (Di Gregorio et al. 1999), forest fires (Trunfio 2004), agent based social processes (Di Gregorio et al. 2001) and highway traffic (Di Gregorio et al. 2008), besides many others.

By extending the classic definition of Homogeneous CA, MCA facilitate the definition of several aspects considered relevant for the correct simulation of the complex systems to be modelled. In particular, MCA provide the possibility to "decompose" the CA cell state in "substates" and to allow the definition of "global parameters". Moreover, the dynamics of MCA models (especially those developed for the simulation of complex macroscopic physical systems such as debris or lava flows) is often "guided" by the "Minimisation Algorithm of the Differences" (cf. Di Gregorio and Serra 1999), which translates in algorithmic terms the general principle for which natural systems leads towards a situation of equilibrium. Refer to Di Gregorio and Serra (1999) for a complete description of the algorithm, besides theorems and applications.

2.1. The MCA lava flow model SCIARA

SCIARA is a family of bi-dimensional MCA lava flow models, successfully applied to the simulation of many real cases, such as the 2001 Mt. Etna (Italy) Nicolosi lava flow (Crisci et al. 2004), the 1991 Valle del Bove (Italy) lava event (Barca et al. 1994) which occurred on the same volcano and employed for risk mitigation (D'Ambrosio et al. 2006). In this work, the basic version of SCIARA (Barca et al. 1993) was considered and its application to the 2001 Nicolosi event and to benchmark grids shown.

SCIARA considers the surface over which the phenomenon evolves as subdivided in square cells of uniform size. Each cell changes its state by means of the transition function, which takes as input the state of the cells belonging to the von Neumann neighbourhood. It is formally defined as

$$SCIARA = < R, X, Q , P, \sigma >$$

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 $0 \leq j < j_{max}$.
- X = {(0,0), (0, -1), (1, 0), (-1, 0), (0, 1)} is the so called von Neumann neighbourhood relation, a geometrical pattern which identifies the cells influencing the state transition of the central cell.
- 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 (m);
  - Q_h is the set of values representing the lava thickness (m);
  - Q_T is the set of values representing the lava temperature ($K°$);
  - Q_o^5 are the sets of values representing the lava outflows from the central cell to the neighbouring ones (m).

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

  $$Q = Q_z \times Q_h \times Q_T \times Q_o^5$$

- P is set of global parameters ruling the CA dynamics:
  - P_T={T_{vent}, T_{sol}, T_{int}}, the subset of parameters ruling lava viscosity, which specify the temperature of lava at the vents, at solidification and the "intermediate" temperature (needed for computing lava adherence), respectively;

- $Pa=\{a_{vent}, a_{sol}, a_{int}\}$, the subset of parameters which specify the values of adherence of lava at the vents, at solidification and at the "intermediate" temperature, respectively;
- p_c, the cooling parameter, ruling the temperature drop due to irradiation;
- p_r, the relaxation rate parameter, which affects the size of outflows.

- $\sigma : Q^5 \rightarrow Q$ is the deterministic cell transition function. It is composed by four "elementary processes", briefly described in the following:
  - *Outflows computation* (σ_1). It determines the outflows from the central cell to the neighbouring ones by applying the minimisation algorithm of the differences; note that the amount of lava which cannot leave the cell, due to the effect of viscosity, is previously computed in terms of adherence. Parameters involved in this elementary process are: P_T and P_a.
  - *Lava thickness computation* (σ_2). It determines the value of lava thickness by considering the mass exchange among the cells. No parameters are involved in this elementary process.
  - *Temperature computation* (σ_3). It determines the lava temperature by considering the temperatures of incoming flows and the effect of thermal energy loss due to surface irradiation. The only parameter involved in this elementary process is p_c.
  - *Solidification* (σ_4). It determines the lava solidification when temperature drops below a given threshold, defined by the parameter T_{sol}.

3. GPU AND GPGPU PROGRAMMING

As alternative to standard parallel architecture, the term GPGPU (General-Purpose computing on Graphics Processing Units) refers to the use of the card processor (the GPU) as a parallel device for purposes other than graphic elaboration. 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 with a limited number of floating-point processors that access a common shared-memory within the multiprocessor. To better understand the enormous potential of GPUs, some comparisons with the CPU are noticeable: a medium-performance GPU (e.g. the NVIDIA Geforce GT200 family) is able to perform nearly 1000 GFLOPS (Giga Floating Point Operations per Second), while an Intel Core i7 has barely 52 GFLOPS. In addition, the most interesting aspect still is the elevated parallelism that a GPU permits. For instance, the NVIDIA GeForce 8800 GTX has 16 multiprocessors each with 8 processors for a total of 128 basic cores, while a standard multi-core CPU has few, though highly-functional, cores. Another motivation of GPUs increasing utilization as parallel architecture regards costs. Until a few years ago, in order to have the corresponding computing power of a medium range GPU of today (which costs approximately a few hundred Euros), it was necessary to spend tens of thousands of Euros. Thus, GPGPU has not only led to a drastic reduction of computation time, but also to significant cost savings. Summarizing, it is not misleading to affirm that the computational power of GPUs has exceeded that of PC-based CPUs by more than one order of magnitude while being available for a comparable price. In the last years, NVIDIA has launched a new product line called Tesla, which is specifically designed for High Performance Computing.

Supported on Windows and Linux Operating systems, NVIDIA CUDA technology (NVIDIA CUDA 2011a) permits software development of applications by adopting the standard C language, libraries and drivers. In CUDA, threads can access different memory locations during execution. Each thread has its own private memory, each block has a (limited) shared memory that is visible to all threads in the same block and finally all threads have access to global memory. The CUDA programming model provides three key abstractions: the hierarchy with which the threads are organized, the memory organization and the functions that are executed in parallel, called kernels. These abstractions allow the programmer to partition the problem into many sub-problems that can be handled and resolved individually.

3.1. CUDA Threads and Kernels

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

CUDA threads can cooperate together by sharing a common fast shared-memory (usually 16KB), eventually synchronizing in some points of the kernel, within a so-called thread-block, where each thread is identified by its thread ID. In order to better exploit the GPU, a thread block usually contains from 64 up to 512 threads, defined as three-dimensional array of type `dim3` (containing three integers defining each dimension). A thread can be referred within a block by means of the built-in global variable `threadIdx`.

While the number of threads within a block is limited, it is possible to launch kernels with a larger total number of threads by batching together blocks of threads, by means of a grid of blocks, usually defined as a two-dimensional array, also of type `dim3` (with the third component set to 1). In this case, however, thread cooperation is reduced since threads that belong to different blocks do not share the same memory and thus cannot synchronize and communicate with each other. As for threads, a built-in global variable, `blockIdx`, can be used for accessing the block index within the grid. Currently, the maximum number of blocks is 65535 in each dimension. Threads in a block are synchronized by calling the `syncthreads()` function: once all threads have reached this point, execution resumes normally. As previously reported, one of the fundamental concepts in CUDA is the *kernel*. This is nothing but a C function, which once invoked is performed in parallel by all threads that the programmer has defined. To define a kernel, the programmer uses the `__global__` qualifier before the definition of the function. This function can be executed only by the device and can be only called by the host. To define the dimension of the grid and blocks on which the kernel will be launched on, the user must specify an expression of the form `<<< Grid_Size, Block_Size >>>`, placed between the kernel name and argument list.

What follows is a classic pattern of a CUDA application:

- Allocation and initialization of data structures in RAM memory;
- Allocation of data structures in the device and transfer of data from RAM to the memory of the device;
- Definition of the block and thread grids;
- Performing one or more kernel;
- Transferring of data from the device memory to Host memory.

Eventually, it must be pointed out that a typical CUDA application has parts that are normally performed in a serial fashion, and other parts that are performed in parallel.

3.2. Memory hierarchy

In CUDA, threads can access different memory locations during execution. Each thread has its own *private memory*, each block has a (limited) *shared memory* that is visible to all threads in the same block, and finally all threads have access to *global memory*. In addition to these memory types, two other read-only, fast on-chip memory types can be defined: *texture memory* and *constant memory*.

As expected, memory usage is crucial for the performance. For example, the shared memory is much faster than the global memory and the use of one rather than the other can dramatically increase or decrease performance. By adopting variable type qualifiers, the programmer can define variables that reside in the global memory space of the device (with `__device__`) or variables that reside in the shared memory space (with `__shared__`) that are thus accessible only from threads within a block. Typical latency for accessing global memory variables is 200-300 clock cycles, compared with only 2-3 clock cycles for shared memory locations. For this reason, to improve performances variable accesses should be carried out in the shared memory rather than global memory, wherever possible. However, each variable or data structure allocated in shared memory must first be initialized in the global memory, and afterwards transferred in the shared one (NVIDIA CUDA 2011b). This means that to copy data in the shared memory, global memory access must be first performed. So, the more his type of data is accessed, the more convenient is to use this type of memory: so, for few accesses it is evident that shared memory is not convenient to use. As a consequence, a preliminary analysis of data access of the considered algorithm should be performed in order to evaluate the tradeoff, and thus, convenience of using shared memory.

4. IMPLEMENTATION OF THE SCIARA MODEL AND EXPERIMENT RESULTS

As previously stated, Cellular Automata models, such as SCIARA, can be straightforwardly implemented on parallel computers due to their underlying parallel nature. In fact, since Cellular Automata methods require only next neighbor interaction, they are very suitable and can be efficiently implemented even on GPUs. In literature, to our knowledge, no examples of Macroscopic Cellular Automata modeling with GPUs are found, while some interesting CA-like implementations, such as Lattice Boltzmann kernels, are more frequent (e.g., Tolke 2008; Kuznik et al. 2010).

In this work, two different implementations of the SCIARA lava flow computational model were carried out, a first straightforward version which uses only global memory for the entire CA space partitioning and a second, but more performing one, which adopts (also) shared memory for CA space substate allocation. What follows is an excerpt of the core of the general CUDA algorithm (cf. Section 2.1):

```
// CA loop
for(int step=0; step< Nstep; step++) {

  // add lava at craters
  crater <<<1, num_craters>>>(Aread,Awrite);
  // s1
  calc_flows<<<dimGrid,dimBlock>>>(Aread,Awrite);
  // s2
  calc_width <<<dimGrid, dimBlock>>>(Aread,Awrite);
  // s3
  calc_temp<<<dimGrid, dimBlock>>>(Aread,Awrite);
  // s4
  calc_quote <<<dimGrid, dimBlock>>>(Aread,Awrite);


// swap matrixes
copy<<<dimGrid,dimBlock>>>(Awrite,N,Aread,Substat_N);
}
cudaMemcpy(A, Aread, size, cudaMemcpyDeviceToHost );
// copy data to Host
}
```

In the time loop four basic kernels, `calc_flows`, `calc_width`, `calc_temp` and `calc_quote` are launched corresponding to the four elementary processes of SCIARA, σ_1, σ_2, σ_3 and σ_4, respectively, as described in Spingola et al. (2008). The `crater()` kernel refers to the crater cell(s), which is obviously invoked on a smaller grid that the previous ones. The model was implemented by adopting a system of double matrixes for the CA space representation, one (`Aread`) for reading cell neighbor substates and a second (`Awrite`) for writing the new substate value. This choice has proven to be efficient, since it allows to separate the substates reading phase from the update phase, after the application of the transition function, thus ensuring data integrity and consistency in a given step of the simulation. After applying the transition function to all the cell space, the main matrix must be updated, replacing values with the corresponding support matrix ones (swap matrixes phase). In this implementation, a CA step is simulated by more logical substeps where, after crater cells are updated (by means of the `crater`), lava outflows are calculated according to the σ_1 elementary process. When all outflows are computed, and therefore all outflow substates are consistent, the actual distribution takes place, producing the new value of the quantity of lava in each cell of the CA. Subsequently, each cell reads from a neighbour cell the associated outflow substate corresponding to the quantity of inflowing lava σ_2 elementary process). In this phase, the σ_3 and σ_4 elementary processes are applied to the new quantity of lava of the cell. At the end of the CA loop, data is copied back to the Host memory by the `cudaMemcpy` function.

Regarding the specific implementation, the first thing to decide on is what thread mapping should be adopted to better exploit the fine-grain parallelism of the CUDA architecture. For example, one might consider using a thread for each row or each column, as occurs in a typical data-parallel implementation (e.g., Oliverio et al. 2011). However, when working in CUDA with arrays, the most widely adopted technique is to match each cell of the array with a thread (e.g., Tolke 2008). The number of threads per block should be a multiple of 32 threads, because this provides optimal computing efficiency (NVIDIA CUDA 2011b) and thus we have chosen to build blocks of size 32 × 16, corresponding to the maximum value (512) of number of threads permitted for each block. What follows is an excerpt for defining the grid of blocks that was considered for SCIARA:

```
#define BLOCK_SIZE_X 32
#define BLOCK_SIZE_Y 16
...
int dimX; // CA x dimension
int dimY; // CA y dimension
dim3 dimBlock(BLOCK_SIZE_X, BLOCK_SIZE_Y);

int n_blocks_x = dimX/dimBlock.x;
int n_blocks_y = dimY/dimBlock.y;

dim3 dimGrid(n_blocks_x, n_blocks_y);
...
// invoke kernel functions

kernel<<<dimGrid, dimBlock>>>(...);
…
```

Once that the grid of blocks (and threads) were defined in this simple manner, kernels are managed so that each cell (*i*, *j*) of SCIARA is associated to each thread (*i*, *j*). This is simply done, for each invoked kernel (i.e., `calc_flows`, `calc_width`, `calc_temp` and `calc_quote`), by associating each row and column of the CA with the corresponding thread as in this simple scheme:

```
__global__ void kernel(...) {

  int col = blockIdx.x * blockDim.x + threadIdx.x;
  int row = blockIdx.y * blockDim.y + threadIdx.y;

// memory allocation (shared, global, etc)
    ...
/** transition function for cell[row][col] **
    ...
}
```

5. TESTS AND PERFORMANCE RESULTS

Two CUDA graphic devices were adopted for experiments: a NVIDIA high-end Tesla C1060 and a Geforce GT 330M graphic card. In particular, the Tesla computing processor has 240 processor cores, 4 GB global memory and high-bandwidth communication between CPU and GPU, whereas the less performing graphic card has 48 cores and 512 MB global memory. The sequential SCIARA reference version was implemented on a 2.66 GHz Intel Core i7 based desktop computer. The sequential CPU version is identical to the version that was developed for the GPUs, that is, no optimizations were adopted in the former version. In practice, at every step, the CA space array is scrolled and the transition function applied to each cell of the CA where lava is present.

Many tests have been performed regarding both performance and verification of the accuracy of the results. Regarding performance tests, the best implementation has regarded a version which adopts a hybrid (shared/global) memory allocation.

As known, access to a location in shared memory of each multiprocessor has a much lower latency than that carried out on the global device memory. On the other hand, an access to a shared-memory location necessary needs a first access to global memory (cf. Section 3.2). For this reason, an accurate analysis was carried out in determining how much memory access each thread does for each CA substate matrix. This investigation gave rise to a “hybrid’” memory access pattern, where shared memory allocation was adopted for those kernels accessing CA matrixes more than two times. For illustrative purposes, Figure 1 shows how

shared memory is used in the context of our GPU implementation.

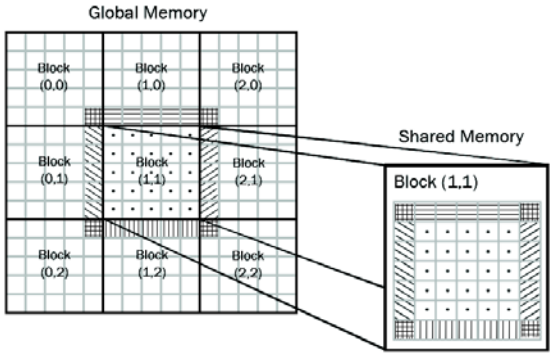

Figure 1: Memory mapping of the CA space allocated in global memory with a portion of shared memory. Shaded areas represent portions of neighbouring block areas which need to be swapped at each CA step to ensure data consistency.

A first test regarded the simulation of well-known and documented real lava flow event, the Mt. Etna Nicolosi event (Crisci et al. 2004) occurred in July, 2001. Table 1 (first row) reports the first results of tests carried out for this experiment, where the CA space is a 819 × 382 two-dimensional grid. The simulation was carried out for 15000 steps, considering one crater for lava flow emission. In order to further stress the efficiency of the GPU version, further benchmarks experiments were performed by considering four different hypothetical CA spaces, namely 512^2, 1024^2, 2048^2 and 4096^2 grids, with cells representing inclined planes, with many craters located over the grid (cf. Table 1 - from second row).

Table 1: Execution times of experiments (in seconds) carried out for evaluating the performance the GPU version of the SCIARA MCA lava-flow model on the considered hardware. The 819 × 382 matrix refers to the 2001 Mt. Etna event. Other grid dimensions refer to inclined planes. N/A (Not Available) data are due to device lack of memory capacity.

Performance results (in seconds)			
CA dim / Device	Intel i7 (sequential)	Geforce	Tesla
819×382	741	46	11.8
512^2	677	31.4	5.6
1024^2	2716	99.1	21.4
2048^2	11480	344.5	81.1
4096^2	47410	N/A	307

Timings reported for the considered GPU devices indicate their full suitability for parallelizing CA models. Even if applied to a simple MCA model, performance results show the incredible computational power of the considered GPUs in terms of execution time reduction, significantly outperforming the CPU implementation up to 150× for large grid sizes. Other tests were also performed on a completely global memory version. In this case results, here omitted for brevity, have shown how the use of shared memory can improve performances up to 50%, with respect to the total global memory version.

Eventually, to test if single-precision data can be considered sufficient for SCIARA simulations, tests were carried out on the 2001 lava flow event (15000 CA steps) and compared results produced by the GPU version with those produced by the CPU (sequential) version with single precision data (i.e., `float` type variables), and those produced still by the same GPU version against a double precision CPU implementation (i.e., `double` type variables). In each case, 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 Macroscopic Cellular Automata model using GPU architectures. As shown, the CUDA technology, in combination with the an efficient memory management, can produce a very efficient version of the SCIARA lava flow simulator. Although results are indeed already satisfactory, future developments can regard further improvements for both increasing performances and implementing more advanced MCA models.

The results obtained in this work are to be considered positive and extremely encouraging. As confirmed by the increasing number of applications in the field of scientific computing in general, GPGPU programming represents a valid alternative to traditional microprocessors in high-performance computer systems of the future.

ACKNOWLEDGMENTS

This work was partially funded by the European Com-mission - European Social Fund (ESF) and by the Regione Calabria.

REFERENCES

Barca, D., Crisci, G.M., Di Gregorio, S., Nicoletta, F., 1993. Cellular automata methods for modelling lava flow: simulation of the 1986-1987 eruption, Mount Etna, Sicily. In: Kilburn, C.R.J., Luongo, G. (Eds.), *Active lavas: monitoring and modelling*. UCL Press, London, 12, 291-309.

Barca, D., G.M. Crisci, Di Gregorio, S., Nicoletta, F. 1994. Cellular Automata for simulating lava

Flows: A method and examples of the Etnean eruptions. *Transport Theory and Statistical Physics*, 23, 195-232.

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

D'Ambrosio, D., Rongo, R., Spataro, W., Avolio, M.V., Lupiano, V. , 2006. Lava Invasion Susceptibility Hazard Mapping Through Cellular Automata. In: S. El Yacoubi, B. Chopard, and S. Bandini (Eds.), *ACRI 2006*, *Lecture Notes in Computer Science*, 4173, Springer-Verlag, Berlin Heidelberg, 452-461.

D'Ambrosio, D., Spataro, W., 2007. Parallel evolutionary modelling of geological processes. *Parallel Computing* 33 (3), 186–212.

Di Gregorio, S., Mele, F., Minei, G., 2001. Automi Cellulari Cognitivi. Simulazione di evacuazione, Proceedings "Input 2001", Seconda Conferenza Nazionale Informatica Pianificazione Urbana e Territoriale, (in Italian) Democrazia e Tecnologia.

Di Gregorio, S., Rongo, R., Siciliano, C., Sorriso-Valvo, M., Spataro, W., 1999. Mount Ontake landslide simulation by the cellular automata model SCIDDICA-3. *Physics and Chemistry of the Earth*, Part A, 24, 97-100.

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

Di Gregorio, S., Umeton, R., Bicocchi, A., Evangelisti, A., Gonzalez, M., 2008. Highway Traffic Model Based on Cellular Automata: Preliminary Simulation Results with Congestion Pricing Considerations. *Proceedings of 20th European Modeling & Simulation Symposium (EMSS)*, pp. 665-674. September 17-19, Campora S.G., CS, Italy.

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

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

NVIDIA CUDA C Programming Guide, 2011a. Available from: http://developer.download.nvidia.com/compute/cuda [accessed 27 June 2011]

NVIDIA CUDA C Best Practices Guide, 2011b. Available from: http://developer.download.nvidia.com/compute/cuda [accessed 27 June 2011]

Oliverio, M., Spataro, W. , D'Ambrosio, D. , Rongo, R., Spingola, G., Trunfio, G.A., 2011. OpenMP parallelization of the SCIARA Cellular Automata lava flow model: performance analysis on shared-memory computers. *Proceedings of International Conference on Computational Science, ICCS 2011*, Procedia Computer Science, 4, pp. 271-280.

Rongo, R., Spataro, W., D'Ambrosio, D., Avolio, M.V., Trunfio, G.A., Di Gregorio, S., 2008. Lava flow hazard evaluation through cellular automata and genetic algorithms: an application to Mt Etna volcano. *Fundamenta Informaticae*, 87, 247–268.

Spingola, G. , D'Ambrosio, D., Spataro, W., Rongo, R. Zito, G., 2008. Modeling Complex Natural Phenomena with the libAuToti Cellular Automata Library: An example of application to Lava Flows Simulation. *Proceedings of International Conference on Parallel and Distributed Processing Tecniques and Applications*, pp. 44-50. July 14-17, 2008, Las Vegas, Nevada, USA

Succi, S, 2004. *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond*, UK: Oxford University Press.

Tolke, J., Krafczyka, M., 2008. TeraFLOP computing on a desktop PC with GPUs for 3D CFD. *Int. Journ. of Comput. Fluid Dynamics*, 22 (7), 443–456.

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

Trunfio, G.A., 2004. Predicting Wildfire Spreading Through a Hexagonal Cellular Automata Model. In: P.M.A. Sloot, B. Chopard and A.G.Hoekstra (Eds.), *ACRI 2004*, *LNCS 3305*, Springer, Berlin, 2004, 725-734.

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

Zuo, W., Chen, Q., 2010. Fast and informative flow simulations in a building by using fast fluid dynamics model on graphics processing unit. *Build. Envir.*, 45, 3, 747–757.