A First Multi-GPU/Multi-Node Implementation of the Open Computing Abstraction Layer

Alessio De Rango[a], Davide Spataro[b], William Spataro[a], Donato D'Ambrosio[a,*]

[a] *Department of Mathematics and Computer Science, University of Calabria, Italy*
[b] *ASML, Eindhoven, The Nederlands*

Abstract

Here we present a first multi-node/multi-GPU implementation of OpenCAL for grid-based high-performance numerical simulation. OpenCL and MPI have been adopted as low-level APIs for maximum portability and performance evaluated with respect to three different benchmarks, namely a Sobel edge detection filter, a Julia fractal generator, and the SciddicaT Cellular Automata model for fluid-flows simulation. Different hardware configurations of a dual-node test cluster have been considered, allowing for executions up to four GPUs. Optimal performance has been achieved in consideration of the compute/memory bound nature of both benchmarks and hardware configurations.

Keywords: Complex Systems Modeling, Grid Based Numerical Modelling, OpenCL, GPGPU Computing, MPI

1. Introduction

Dedicated High-Performance Computing (HPC) machines are nowadays large ensembles of powerful computing nodes with heterogeneous hardware interconnected via network. In fact, beside traditional CPUs, GPGPU (General-Purpose Computation on Graphics Processing Unit) has become mainstream nowadays (see e.g., [1], [2], [3]). The use of this recent and advanced computing hardware is constantly being applied in many fields of science and engineering and has

*Corresponding author
Email address: donato.dambrosio@unical.it (Donato D'Ambrosio)

recently been more and more applied with success in Scientific Computing [4]. In this field, and in particular in modeling and simulation, parallel machines are used to obtain approximate numerical solutions of differential equations which describe a physical system rigorously, as for example for Maxwell's equations at the foundation of classical electromagnetism or the Navier-Stokes for fluid dynamics [5]. The classical approach, based on Calculus, often fails to solve these kinds of equations analytically, making a numerical computer-based approach absolutely necessary. An approximate numerical solution of a partial differential equation can be obtained by applying a number of methods, as Cellular Automata (CA), the Finite Element Method (FEM) or Finite Difference Method (FDM) [6], which yield approximate values at a discrete number of points over the considered domain.

Different software systems have been proposed aiming at making the programming of recent HPC systems simpler, faster and less error-prone, often providing abstract reference modeling paradigms. Among them, well-known examples are OPS [7, 8], and OP2 [9, 10], Domain Specific Languages (DSLs) for structured and unstructured grid applications, respectively, able to run on clusters of CPUs and GPUs. AQUAgpusph [11] is another well-known example of a modern software library for high-performance computing targeting many-core accelerators and providing the Smoothed-Particle Hydrodynamics (SPH) reference abstract model. Similarly, ASL [12] is an example of accelerated multi-physics simulation software based, among others, on the Lattice Boltzmann Method, while CAMELot [13, 14] and libAuToti [15] represent valid solutions for CA modeling and execution on clusters of CPUs.

OpenCAL (Open Computing Abstraction Layer) [16] was recently introduced for modeling systems based on the computational domain discretization in structured grids of cells or node points, allowing for execution on multi-core CPUs and many-core devices like GPUs. A first application of OpenCAL to the simulation of a 3D particle system was already shown in [17]. In this paper, we present an extension of the original specification of OpenCAL that expands the set of supported systems to multi-GPU workstations and clusters of intercon-

nected multi-GPU nodes. Like in its original implementation, the Application Programming Interface (API) exposes a DSL based on the Extended Cellular Automata formalism (XCA) [18]. Also known as Complex or Multi-Component Cellular Automata, XCA were introduced for the modeling of macroscopic fluid-flows (see e.g., [19, 20, 21, 22, 23, 24, 25, 26]), forest fire spreading [2, 27], eco-hydrologic dynamical systems [28, 29, 30, 31], besides others. Note that, being XCA a general structured-grid based computational paradigm, other numerical methods are supported by OpenCAL, such as Finite Differences and Cellular Automata (of which XCA can be considered an extension). Low-level parallel programming details can be ignored and different efficient serial, multi-core, single-GPU, multi-GPU and cluster of GPUs applications obtained.

In the following, Section 2 outlines the general architecture of OpenCAL and presents the specific features of the multi-GPU and multi-node implementations. Subsequently, Section 3 briefly describes three different compute/memory-bound applications considered as benchmarks. Specifically, they are a Sobel convolutional graphics filter, a Julia set fractal generator, and the SciddicaT fluid-flow simulation model. Section 4 reports and comment the achieved computational results on different hardware configurations, from single GPU to a test cluster with a total of four GPUs. Eventually, Section 5 concludes the paper with outcomes and future developments.

2. The Multi-GPU/Multi-Node implementation of OpenCAL

OpenCAL is a portable parallel computing abstraction layer for numerical simulation. It is released as a C/C++ software library under the Lesser GNU Public License (LGPL) version 3, and is freely available at GitHub. As already stated, the API provides a DSL based on the XCA formalism [18]. Informally, XCA, compared to classical CA, are different because of the following points:

- The cell's state is decomposed in *substates*, each of them belonging to the set of admissible values of a given characteristic assumed to be relevant for the modeled system and its evolution (e.g., lava temperature or lava

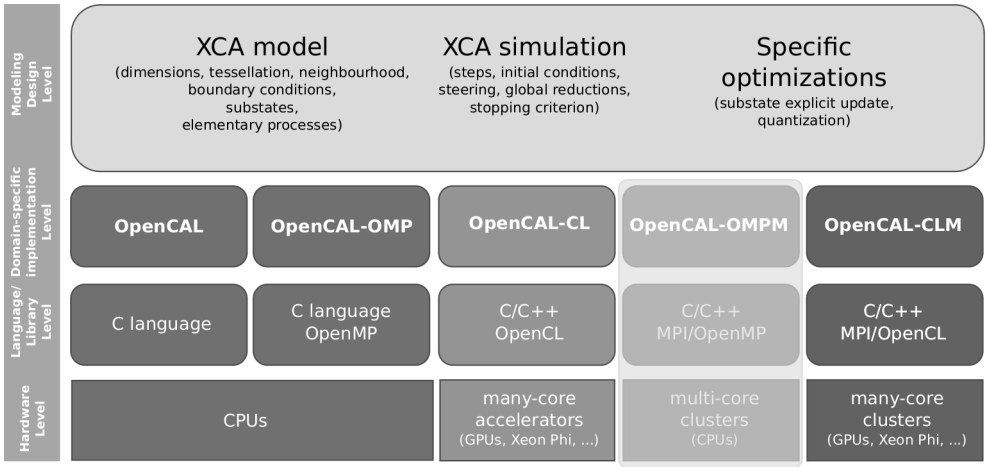


FIGURE 1: General OpenCAL architecture. The computational model, together with the simulation process and possible optimizations, is designed at the higher level of abstraction. The different OpenCAL components can be found at the implementation abstraction layer, allowing for a straightforward implementation of the designed computational model. OpenCAL-based applications can be therefore executed at the hardware level on both multi-core CPUs, many-core devices and clusters of many-core accelerators. The execution on distributed memory multi-core systems is currently still under development.

thickness in the case of a lava flow simulation model). The set of states for the cell is simply obtained as the Cartesian product of the considered sets of substates;

- A set of *parameters*, commonly used to characterize the dynamic behaviors of the considered phenomenon, can be defined;
- The cell's transition function can be split into *elementary processes*. 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*, defining substates changes of the cell as function of the substates of the cell itself (e.g. the solidification of the lava inside the cell due to the temperature drop);
- *Steering* operations are allowed, e.g., global reductions over the whole, or a subset, of the cellular space (i.e., the computational domain).

In its first release [16], OpenMP- and OpenCL-based components were released, permitting to exploit multi-core CPUs and many-core GPUs, respectively. Parallelism was almost completely made transparent and the *quantization optimization* built in to accelerate the computation in case of topologically connected phenomena (e.g., the simulation of a bounded fluid flow developing over a wide topographic surface). In fact, quantization permits to avoid to compute the next state of *stationary* cells. Based on user-defined criteria, a set of *active* cells, A, is maintained updated by the system, and the transition function application restricted to it.

Algorithm 1 describes the OpenCAL main loop in pseudo code. The `init()` function, generally used for initialization purposes, is called once at the beginning. Subsequently, if quantization is used, the set of active cells is updated, as well as substates. The `step` counter and the `halt` variable, that is used to check the simulation termination condition, are set to the initial step and to *false*, respectively, and the main simulation loop follows. At each step, elemen-

Algorithm 1: OpenCAL main implicit simulation process.

```
init()                                  // Call the init() global function
if quantization then
    update (A)                  // Update the set of non-stationary cells
forall q ∈ Q do
    update (q)                                  // Update the substate q
step ← initial_step
halt ← false
while ¬halt ∧ (step ≤ final_step ∨ final_step = CAL_RUN_LOOP) do
    forall e of σ do
        forall (A ≠ ∅ ∧ i ∈ A)∨ i ∈ R do
            e(i)  // Apply the elementary process e to the cell i
        if quantization then
            update (A)                // Update the array of active cells
        forall q ∈ Q do
            update (q)                          // Update the substate q
    steering()                  // Call the steering() global function
    if quantization then
        update (A)                      // Update the set of active cells
    forall q ∈ Q do
        update (q)                              // Update the substate q
    halt ←stopCondition()                  // Check the stop condition
    step ← step + 1
return
```

tary processes, e, defining the transition function, σ, are applied in parallel to each cell i of the cellular space. If quantization is considered, stationary cells are skipped. Depending on which OpenCAL version is considered, elementary processes are implemented as standard C callback functions or OpenCL kernels, and are executed in the same order in which they were defined by the user. In both cases, they are *registered* to the OpenCAL model object, to be transparently applied. Active cells and substates are then updated. Moreover, if defined, the `steering()` global function is executed and active cells and substates again updated. The `stopCondition()` function is eventually called and the step counter increased. The simulation loop continues while the `halt` variable, whose value is set by the `stopCondition()` function, is *false* or the final step of computation is not met.

The current OpenCAL software architecture is depicted in Figure 1. At the higher level of abstraction, the XCA model can be defined, allowing one to account for both local rules and global laws of evolution for the system. Domain topology and extent, boundary conditions, cell/node states (split in *substates*), local rules of evolution (expressed in terms of *elementary processes*, defining the automaton transition function), and neighborhood pattern (for local rules application), can be formalized. The simulation process can also be designed at this level, including initial conditions, optional global operations (e.g. steering or global reductions), and termination criteria. Four different components can be found at the implementation level, namely OpenCAL (serial reference version), OpenCAL-OMP (OpenMP-based multi-thread implementation), OpenCAL-CL (OpenCL-based component) and OpenCAL-CLM (MPI-based implementation for single and multiple OpenCL instances). With respect to the original implementation, the contribution of this work consisted in extending OpenCAL-CL to support single-node/multi-GPU workstations (for a *pure* multi-GPU execution) and to introduce the new OpenCAL-CLM component to support clusters of multi-GPU nodes. Note that, this latter component also permits to run single-GPU instances of OpenCAL-CL within a multi-GPU workstation (for a *hybrid* MPI/OpenCL multi-GPU execution). Eventually, an OpenMP/MPI

component is also planned as a future development, which will permit to exploit clusters of multi-core CPUs. The library and hardware levels complete the OpenCAL general architecture pointing out the underlying parallel APIs and computing system targets.

The first implementation of the OpenCAL-CL/CLM components was deliberately straightforward and permitted to build the new components on top of the existing OpenCAL-CL one. In particular: A classic data-parallel decomposition scheme was adopted; Halos exchange was scheduled to take place after the execution of each elementary process (i.e., after at least a substate was updated); A different halo was considered for each defined substate (even for those substates that were not affected by the elementary process).

Figure 2 shows an example of the data-parallel domain decomposition scheme adopted by OpenCAL-CL/CLM for the case of a two-dimensional domain and a dual-node system with two GPUs per node. Periodic boundary conditions are assumed for the sake of simplicity. The domain is decomposed along the rows and preliminarily assigned to the available nodes. This phase is managed by the OpenCAL-CLM component through MPI over the interconnection network. In turn, each sub-domain is further partitioned among the available GPUs by considering a similar data partitioning scheme, relying on OpenCL (for the pure multi-GPU implementation) or on MPI (for the hybrid MPI/OpenCL solution on the single node). Note that the library permits for non-uniform domain partitioning (e.g., the user can assign more rows to more performing GPU devices) in order to balance the workload among nodes/GPUs with different computational capabilities.

During computation, halos exchange is transparently managed by OpenCAL-CL/CLM, by exploiting the PCI Express bus (in case of halos residing on two or more GPUs on the same node) and/or the interconnection network. Preliminarily, substates's halos are packed in order to reduce communication latency and to maximize the bandwidth. Nevertheless, communication time can sensibly grow according to number and type of involved substates. In summary, for a halo exchange to take place, it is necessary to:

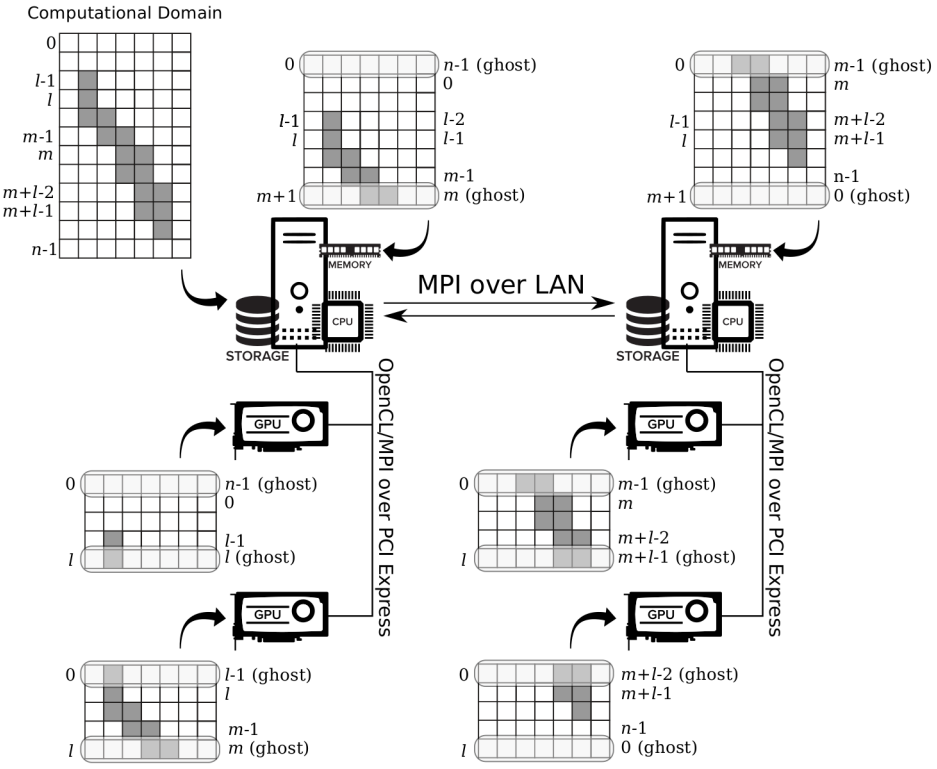


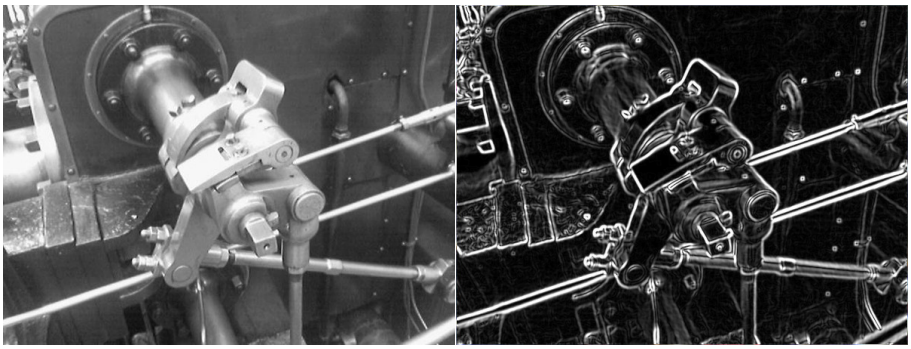
FIGURE 2: Domain decomposition adopted by the multi-node/multi-GPU release of OpenCAL. The figure shows the adopted row-major order decomposition of a two-dimensional computational domain for the case of a dual-node cluster, with two GPUs per node. MPI is adopted for halo exchange over the interconnection network. Each sub-domain can be further partitioned among the available GPUs within the node by considering the same partitioning scheme. At node level, either OpenCL (for a *pure* multi-GPU implementation) or MPI (for a *hybrid* OpenCL/MPI multi-GPU implementation) can be used for halo exchange (cmp. Section 4). In this example, a homogeneous computing system is assumed, therefore data is equally partitioned among the available GPUs. However, in real systems, non-uniform partitions can be adopted.

1. Pack and upload boundary data of all defined substates to the CPU (device to host memory transfer);
2. If the GPUs involved in the communications are controlled by different nodes, an extra communication step over the network is performed between the two nodes;
3. Unpack and upload boundary data to the receiving GPU (host to device memory transfer).

Note that, if multi-GPU execution entirely relies on OpenCAL-CL, the host application is involved in the halo exchange among the GPUs, as usually occurs in OpenCL applications. In this case, the process is serialized host-side, as for the OpenCL API specifications, even if non-blocking enqueueing read/write calls are considered. As known, the same host-side serial policy can be found in the OpenGL graphics API and represents one of the reasons that led Khronos Group (i.e. the consortium that defines the OpenCL and OpenGL API specifications, beside others) to propose the new Vulkan compute/graphics unified API, which can exploit the parallelism both host- and device-side. As a consequence, non-optimal exploitation of host-side resources could be achieved by OpenCAL-CL in multi-GPU systems. Nevertheless, the previously mentioned hybrid approach can be alternatively adopted to exploit parallelism also at CPU level, where OpenCAL-MPI can be used to run a single-GPU OpenCAL-CL process for each GPU in the node.

3. Benchmark Applications

In order to evaluate the performance of the new OpenCAL-CL/CLM components, a set of three benchmarks, namely a Sobel convolutional graphics filter, a Julia set fractal generator, and the SciddicaT fluid-flow simulation model, were developed in OpenCAL. In this Section, the three benchmarks are briefly described, while in Section 4 they are characterized as compute/memory bound applications with respect to the adopted GPUs.



(a) (b)

FIGURE 3: Example of application of the Sobel edge detection convolutional graphics filter. (a) Original bitmap (`https://en.wikipedia.org/wiki/Sobel_operator`). (b) Result after the application of the Sobel filtering process.

3.1. The Sobel Edge Detection Filter

The Sobel edge detection filter belongs to the wide family of convolutional graphics filters [32, 33], commonly used to modify the spatial frequency characteristics of an image. In general, the filtering is applied to each point of the domain (i.e., to each pixel of the image) and consists of determining the new value of the point as the weighted summation of its neighbors. For this purpose, a (usually small square) matrix K, called *kernel* of the convolution, defining the weights to be used, is considered. Formally, convolution can be expressed by the following formula:

$$f'_{ij} = \sum_{i'=0}^{n-1} \sum_{j'=0}^{m-1} f_{(i+i'-n/2)(j+j'-m/2)} k_{(i'-n/2)(j'-m/2)} \tag{1}$$

where f_{ij} and f'_{ij} are the old and new value of the point at coordinate (i, j), respectively, m and n the vertical and horizontal size of the kernel, while k_{ij} is the value of kernel at location (i, j).

In the case of the Sobel edge detection filter, a two-step process is considered,

one per each (horizontal and vertical) direction. Accordingly, the following two kernels are adopted:

$$K_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} \quad K_y = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}$$

and applied separately to produce separate measurements, $G_{x_{ij}}$ and $G_{y_{ij}}$, of the gradient component in each orientation, by applying Equation 1. These values are eventually combined to find the absolute magnitude of the gradient at each point as $f'_{ij} = \sqrt{G^2_{x_{ij}} + G^2_{y_{ij}}}$.

Figure 3 shows an example of an application of the Sobel edge detection filter to the red channel of a sample image. For the purpose of this work, however, the filter, as implemented in OpenCAL, was applied to the single channel of the gray-scale image in Figure 4 (cmp. Section 3.2).

3.2. The Julia Set Fractal Generator

Julia sets [34] are examples of fractals generated by mapping the discrete points (or pixels) of a grid $C = \{(x, y) \mid 0 \leq x < S_x, 0 \leq y < S_y\}$ to a rectangular region of the complex plane by applying the rule $z_0 = z_0(x, y) = Re(x)+Im(y)i$, where $i^2 = -1$ and $Re(x)$ and $Re(y)$ are functions of the x and y coordinates, respectively. Subsequently, z_0 is iteratively updated by considering a recurrence formula of the kind $z_{n+1} = z_n^2 + c$, where c is a complex parameter. By changing c, different Julia sets are obtained. The iterative process ends when the z module becomes greater than a given threshold $T \in \mathbb{R}$ (in this case the point (x, y) is said to be divergent and does not belong to the set), or after a predefined number of iterations N (in this case the point is said to be convergent and belongs to the set). Formally, the Julia set of the grid C can be defined as:

$$J(C) = \{z \in C \ : \ |z_n| < T, \ \forall n \leq N\}$$

Figure 4 shows the fractal considered in this work. It was obtained by considering $c = -0.391 + -0.587i$, and by mapping the discrete points of a

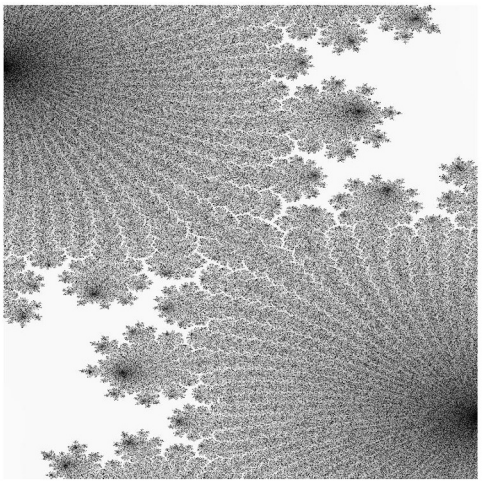


FIGURE 4: A Julia fractal set consisting of 15,000 x 15,000 pixels. Divergent pixels are in gray tones, with clearer tones identifying points diverging faster, while convergent pixels are in black.

$15,000 \times 15,000$ grid (i.e. $S_x = S_y = 15,000$) as:

$$Re(z) = \frac{3(x - \frac{S_x}{2})}{KS_x}$$
$$Im(z) = \frac{2(y - \frac{S_y}{2})}{KS_y}$$

where the *zoom factor* K was set to 3. Eventually, the theshold T was set to the value 10^3 and $N = 10^4$ iterations were considered to evaluate the process of convergence.

3.3. The SciddicaT Landslide Simulation Model

SciddicaT is a classic example of Extended Cellular Automata model for simulating fluid-flows [35]. The model is extremely simple and fast. Nevertheless, it demonstrated to be able to properly simulate non inertial real phenomena like landslides on complex topographic surfaces. In brief, six information layers account for main system's characteristics, while three elementary processes determine its dynamical evolution. In the XCA formalism, information layers

are expressed in terms of substates (i.e., as double buffered matrices, used alternatively for read and write access and swapped after the application of each elementary process). Specifically, they are: Q_z, which stores the information about the cell's altitude, Q_h representing the fluid thickness, and Q_o^4 being the outflows from the central cell to the four neighbors (belonging to the von Neumann neighborhood). The system's evolution is thus obtained by applying the following elementary processes simultaneously to each cell of the computational domain:

- $\sigma_1 : (Q_z \times Q_h)^5 \times p_\epsilon \times p_r \rightarrow Q_o^4$ computes outflows from the central cell to the four neighbors by applying the *minimization algorithm of the differences* [18]. A preliminary control avoids the computation of negligible outflows (i.e., if the fluid thickness is smaller than or equal to a predefined threshold p_ϵ). If this is not the case, the outflows are given by $q_o(0,m) = f(0,m) \cdot p_r$ $(m = 0, \ldots, 3)$, were $f(0,m)$ are the outgoing flows towards the 4 adjacent cells, as evaluated by the minimization algorithm, and $p_r \in]0,1]$ a relaxation rate factor considered to damp outflows in order to obtain a smoother convergence to the system's global equilibrium. The Q_o^4 substates are updated accordingly with the values of the computed outflows.

- $\sigma_2 : Q_h \times (Q_o^4)^5 \rightarrow Q_h$ evaluates the new value of fluid thickness inside the cell by considering mass exchange in the cell's neighborhood: $h^{t+1}(0) = h^t(0) + \sum_{m=0}^{3}(q_o(0,m) - q_o(m,0))$. Here, $h^t(0)$ and $h^{t+1}(0)$ are the mass thickness inside the cell at the t and $t+1$ computational steps, respectively, while $q_o(m,0)$ represents the inflow from the $n = (m+1)^{th}$ neighbor. The Q_h substate is updated accordingly to account for the mass balance within the cell.

- $\sigma_3 : Q_o^4 \rightarrow Q_o^4$ resets the outflow substates (i.e., sets them to zero) for the next computational step.

According to [35], the model parameters p_ϵ and p_r were set to the values

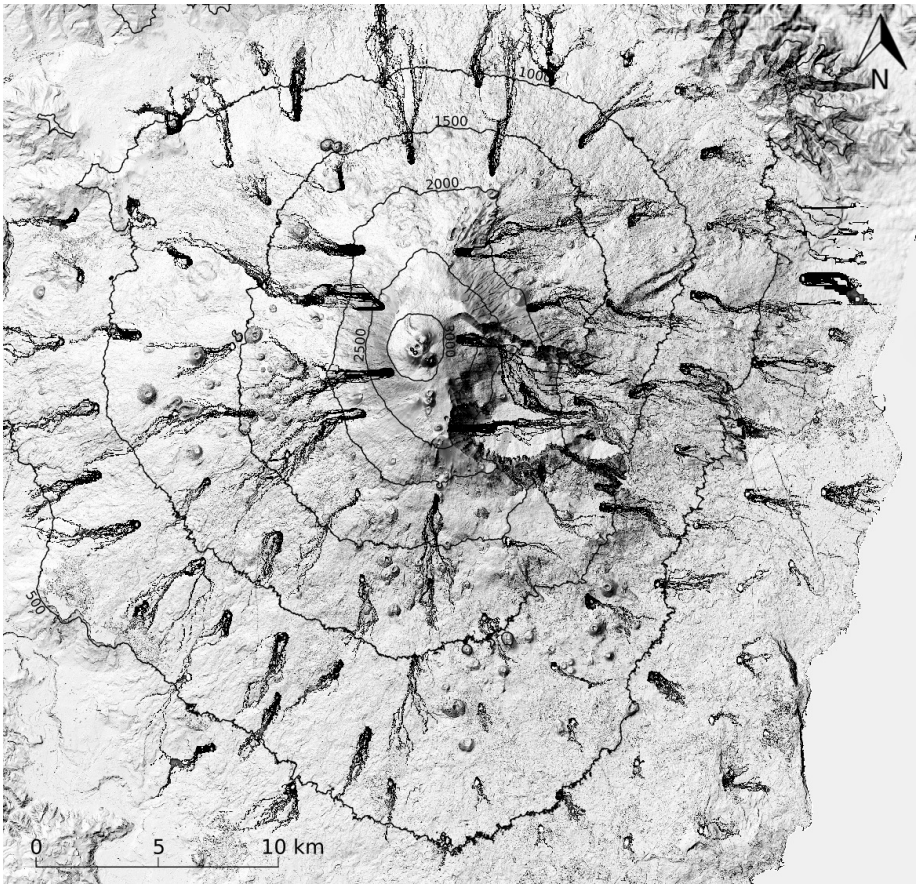


FIGURE 5: SciddicaT simulation of 100 flows over a 3,593 rows per 3,730 columns wide surface with square cells of 10 m side.

0.001 and 0.5, respectively, and 100 flows were simulated for a total of 4,000 computational steps over a wide surface represented by a $3,593 \times 3,730$ DEM (Digital Elevation Model), with square cells of 10 m side. Outcomes are shown in Figure 5.

4. Computational Results and Discussion

The JPDM-SS dual node test cluster was adopted for evaluating the performance of the OpenCAL-CL/CLM components. The cluster nodes, namely JPDM-1 and JPDM-2, were interconnected by a Cisco Cacalyst 3750 Series switch via standard Gigabit Ethernet (with a theoretical bandwidth of 820 Gbit/s). JPDM-1 was equipped with two Intel E5-2650 Xeon CPUs, two GTX 980 and one Tesla K40 Nvidia GPUs, while JPDM-2 with two E5440 Xeon processors and two Titan Xp Nvidia GPUs. In particular, the GTX 980 and the Titan Xp are game oriented graphics devices, while the Tesla K40 is a compute-dedicated solution. This is confirmed by the theoretical peak performance in single (fp32) and double (fp64) precision floating-point operations. Specifically, the GTX 980 reaches 4.98 TFLOPS in fp32, which drops to 0.156 TFLOPS in fp64. Similarly, the Titan Xp reaches the theoretical performance of 12.1 TFLOPS in single precision, value that drops to only 0.378 TFLOPS in double precision. On the contrary, theoretical better fp64 performance characterizes the Tesla K40 solution, which provides 4.3 TFLOPS in fp32 and 1.43 TFLOPS in double precision. Other specifications of the above many-core devices are: the GTX 980 (Maxwell architecture) has 2048 CUDA cores, 4 GB global memory and 112 GB/s theoretical bandwidth communication for double precision data between CPU and GPU, the Titan Xp (Pascal Architecture) device has 3840 cores, 12 GB global memory and 273.85 GB/s bandwidth, while the Tesla K40 has 2880 cores, 12 GB global memory and 144 GB/s bandwidth.

A total of ten simulations were executed for each of the considered benchmarks (cmp. Section 3) by considering different hardware configurations, ranging from single-node/single-GPU to multi-node/multi-GPU systems. Integer

values were adopted for the Sobel benchmark, while double-precision floating point values were considered for the others, and speed-up evaluated with respect to the elapsed times of the corresponding OpenCAL-based serial implementation (as executed on the - more performing - Intel E5-2650 Xeon processor), by taking into account the minimum recorded times.

The compute/memory bound nature of the benchmarks was preliminarily investigated. For this purpose, the algorithmic instruction/byte ratio r was considered and compared with the so-called device-dependent balanced instruction/byte ratio r_d, this latter typically representing the number of fp32 operations per byte issued in order to obtain peak compute and bandwidth performance [36]. More specifically, if I (expressed in GInst/s) and M (expressed in GB/s) are the peak instruction and memory throughput, respectively, the balanced instruction/byte ratio is defined as $r_d = I/M$.

Accordingly, by considering that the Tesla K40 has a total of 2880 cores, each one with a frequency of 0.745 GHz, and by assuming that a fp32 operation is completed in 3 clock cycles, the theoretical fp32 instruction throughput is $I = 0.745 * 2880/3 = 715.2$ GInstr/s. By also considering that its fp32 theoretical bandwidth is $M = 288$ GB/s, the fp32 theoretical balanced ratio for the Tesla K40 is $r_{K40}^{(fp32)} = 715.2/288 = 2.83$. This value is reduced by one third in case of fp64 operations, since the number of double precision units of the K40 is 960 (i.e., one third of the total amount of single precision units which are 2880) [37]. The balanced ratio in case of fp64 is therefore $r_{K40}^{(fp64)} = 0.94$. As for the Tesla K40, the same evaluations allowed to evaluate the theoretical balanced ratios: $r_{GTX}^{(fp32)} = 3.43$, $r_{GTX}^{(fp64)} = 0.1$ (since the fp64 units of the GTX 980 GPU are 1/32 of the fp32 ones). Moreover, regarding the Titan Xp, we obtained: $r_{Titan}^{(fp32)} = 3.7$, $r_{Titan}^{(fp64)} = 0.12$ (since fp64 units are 1/32 of the fp32 ones even for the Titan Xp). Eventually, we assume $r_*^{(int)} = r_*^{(fp32)}$, where the * character is used as wildcard to indicate any GPU here considered. The above evaluations are summarized in Table 1.

In order to evaluate the compute/memory bound nature of the developed benchmarks, we assumed that an algorithm is considered as compute bound for

r_d	fp32	fp64	int
GTX 980	3.43	0.1	3.43
Titan Xp	3.7	0.12	3.7
Tesla K40	2.83	0.94	2.83

TABLE 1: Balanced instruction/byte ratio for the GPUs adopted in this work.

a given GPU if its instruction/byte ratio r is greater than the device balanced ratio r_d, while it is memory bound in the other case.

By considering the $N = 10^4$ iterations adopted for evaluating the convergence condition within the Julia elementary process, with a total of 4 calls to math instructions per iteration and only two memory accesses, the instruction/byte ratio for the Julia benchmark was evaluated to be $r \approx 13.333 \cdot 10^3 >> r_*^{(fp64)}$. The different orders of magnitude of the instruction/byte ratio with respect to the balanced one of all the considered devices suggest that the benchmark can significantly take advantage of a compute dedicated device.

In the case of Sobel, by considering that the only defined elementary process executes 36 integer operations against a total of 21 memory accesses (substate updating included), the instruction/byte ratio was evaluated to be $r \approx 1.71 < r_*^{(int)}$, pointing out a similar memory bound nature of the benchmark with respect to the considered devices. Accordingly, computational devices with more recent architectures are expected to perform better on this benchmark.

Eventually, SciddicaT exposed both kinds of bounds within its elementary processes. In particular, σ_1 instruction/byte ratio was evaluated to be about $r = 1.4$. This value was obtained by considering that the elementary process requires in average 21 fp64 operations and 15 memory accesses. However, since a preliminary (optimization) control is applied, its application is skipped for those cells with a negligible amount of fluid (cmp. Section 3), being fully applied to about only the 3.9% of the domain cells during the whole simulation (cmp.

also [16]). On the contrary, σ_2 and σ_3 were characterized by a instruction/byte ratio of 0.8 and 0.0, respectively (i.e., no fp operations are performed by σ_3), thus resulting compute bound. By considering also the 36 memory accesses needed for updating purposes at the end of each elementary process, the overall SciddicaT instruction/memory ratio was evaluated to be $r \approx 0.17$. In this case, the algorithm resulted compute bound for the GTX 980 and the Titan Xp GPUs in a slight measure (since $r \approx r_{GTX}^{(fp64)} \approx r_{Titan}^{(fp64)}$), while more memory bound for the Tesla K40. Accordingly, devices more similar in terms of fp64 balanced ratio, like the adopted game-oriented GPUs, should run SciddicaT at almost their best, while the compute dedicated Tesla K40 device should perform sub-optimally.

In the remaining part of this Section, computational results are presented, with reference to the adopted computing systems.

4.1. Single-Node/Single-GPU and Single-Node/Multi-GPU Computational Results

The speed-up achieved by the three developed benchmarks on the three available GPUs are shown in Figure 6. The figure reports also the elapsed times in seconds on top of each speed-up bar. The Titan Xp outperformed the GTX 980 in all tests, while the compute dedicated Tesla K40 performed better on the Julia benchmark in absolute terms. Compared with its sequential implementation (as executed on JPDM-1, which is equipped with the more performing CPUs), Julia ran about 89 times faster on the Tesla K40, 11 times faster on the GTX 980 and 36 times faster on the Titan Xp. This result confirms the correctness of the previous considerations, being the K40 the most suitable GPU among those considered in this work to run compute bound double-precision algorithms (cmp. Table 1). On the contrary, for opposite reasons, the other benchmarks performed better on the game-oriented GPUs.

The second series of tests regarded the single-node/dual-GPU execution on the JPDM-1 and JPDM-2 workstations. The Tesla K40 GPU was not considered in these tests to maintain the dual-GPU systems perfectly balanced.

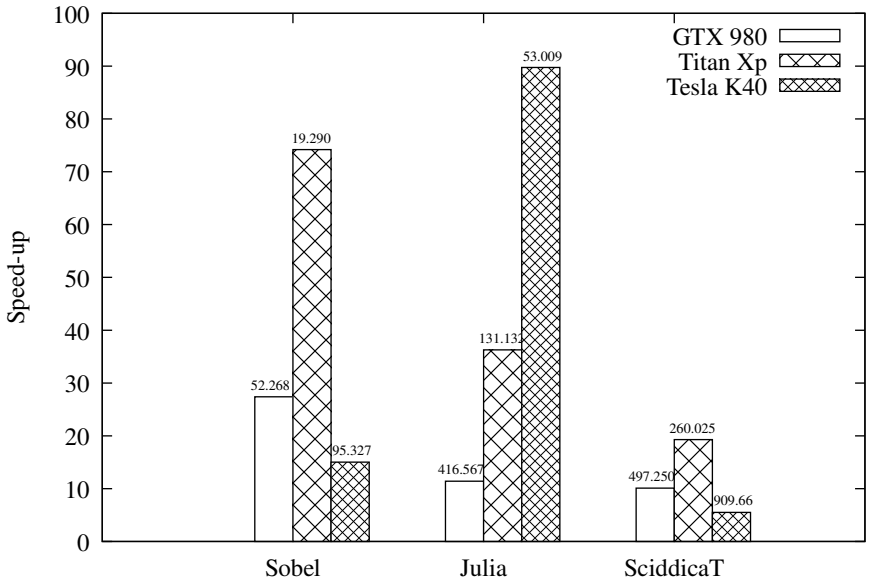
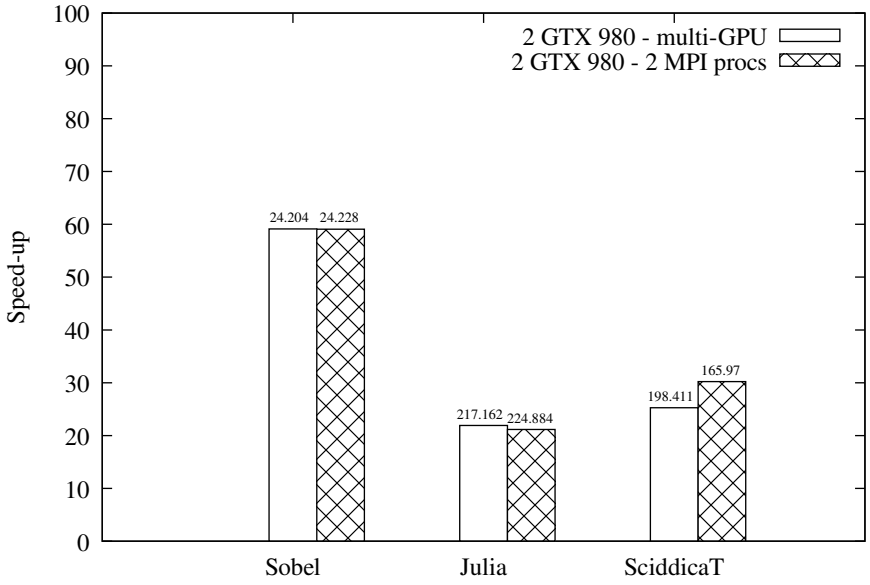
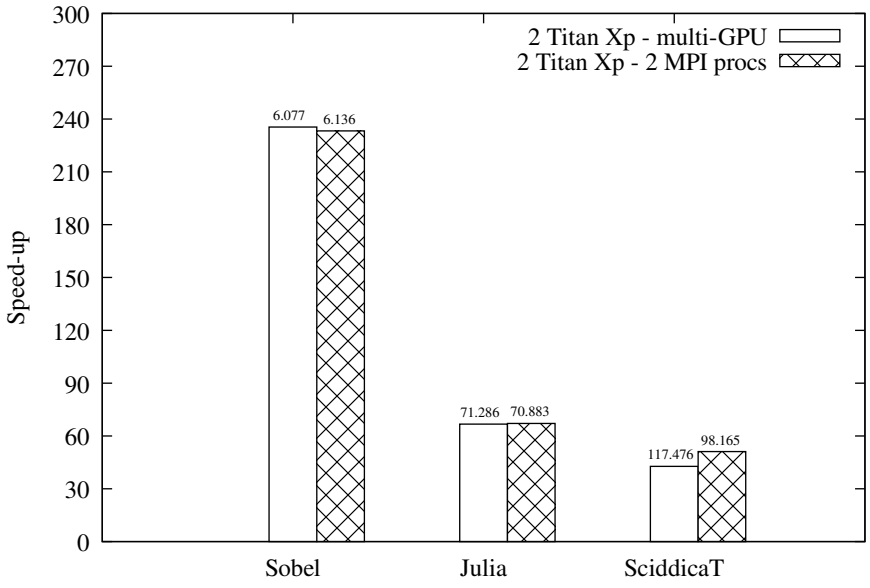


FIGURE 6: Speed-ups achieved by the different OpenCAL-CL single-GPU executions of the Sobel, Julia and SciddicaT benchmarks. Elapsed times in seconds are also shown on top of each speed-up bar. The sequential reference times were taken on the JPDM-1 workstation and are 1,431 s, 4,758 s and 5,015 s for the Sobel, Julia and SciddicaT, respectively. The adopted GPUs are an Nvidia GTX 980, Nvidia Titan Xp and Nvidia Tesla K40.



(a)



(b)

FIGURE 7: Speed-ups achieved by the different single-node/multi-GPU executions of the Sobel, Julia and SciddicaT benchmarks. Elapsed times in seconds are also shown on top of each speed-up bar. The sequential reference times were taken on the JPDM-1 workstation and are 1,431 s, 4,758 s and 5,015 s for the Sobel, Julia and SciddicaT, respectively. An equal partitioning of the domain for each benchmark was considered. (a) Results referred to the JPDM-1 node consisting of two GTX 980 GPUs. (b) Results referred to the JPDM-2 node consisting of two Titan Xp GPUs.

Accordingly, an equal partitioning of the computational domains was adopted, by assigning an equal number of rows to each GPU. Moreover, two different types of execution were considered: a first one based on a *pure* multi-GPU approach, where halos were directly exchanged by means of OpenCL read/write buffer enqueueing operations on the PCI Express bus, a second by a *hybrid* OpenCL/MPI approach, where two MPI processes were considered for running two single-GPU OpenCAL-CL instances and for halos management (which however occurred still on the PCI Express bus). It is worth to note that both the pure and hybrid execution policies were obtained by simply changing few parameters in the (so-called) OpenCAL *cluster* configuration file, with no changes required at source code level. Figures 7-a and 7-b show the results achieved on aforementioned workstations. The figures report also the elapsed times in seconds on top of each speed-up bar. Superlinear effects were here observed for the Sobel and SciddicaT benchmarks (i.e., the computation runs more than p times faster than the serial implementation, being p the number of processing units), probably due to typical cache issues effects. In fact, the algorithms require the access to neighboring cells' data, that can be prefetched in cache. This is implicitly confirmed by the fact that the Julia benchmark, which does not require the access to data besides the one referring the central cell, did not show superlinear speed-up. In absolute terms, the dual Titan Xp system (JPDM-2) performed considerably better than the dual GTX 980 one, with the (integer-based) Sobel application running about four times faster. The other (double precision) benchmarks resulted roughly two times faster on JPDM-2 with respect to JPDM-1.

It is worth to note that, independently from the workstation considered, both types of execution produced similar results for the Sobel and Julia models, while SciddicaT evidences an about 17% discrepancy in favor of the OpenCL/MPI hybrid execution policy. The reason beneath these results can be imputed to the host-side serial nature of OpenCL, as already discussed in Section 2. The problem did not emerge for the Julia benchmark since it did not require communications to take place during the computation, while it was of negligible entity

for Sobel, where a low amount of data and few messages were needed for each computational step. Eventually, the aforementioned discrepancy resulted with major evidence for the SciddicaT model since the halos have bigger size and are also exchanged more frequently (three times per step). Specifically, each Sobel communication required a total of 120 KB (only one integer substate is defined in Sobel), while 358.08 KB were necessary for SciddicaT (six double-precision subtates are defined in SciddicaT). Moreover, Sobel halo management required 8 read/write operations, versus the 48 ones required by SciddicaT. These messages were managed by the host application serially in the pure OpenCL-based execution policy (even if nonblocking OpenCL calls were used), while they were executed in parallel when the hybrid OpenCL/MPI policy was considered. On the contrary, in case of hybrid execution, the two MPI processes performed the halo management in parallel. In this manner, the single process was devoted to only 24 halo exchanges for the case of SciddicaT, by reducing the overall time spent in this process.

4.2. Multi-Node Multi-GPU Computational Results

Eventually, the dual-node JPDM-SS cluster was adopted for the final speed-up tests with a total of four GPUs, namely the two GTX 980 on JPDM-1, and the two Titan Xp on JPDM-2. Two execution policies were considered, as for the single-node case. In the first one, MPI was considered for messages that take place only between nodes, while OpenCL execution at multi-GPU level, including halo exchange between the GPUs in the node. In the second, MPI was considered for both messages between the nodes and for halo exchange at node level. Eventually, different domain partitioning where considered between nodes to account for their different computational capabilities, while the same amount of rows were assigned to the GPUs within each node.

Figure 8 shows the achieved speed-ups, together with the corresponding execution times.The best results were achieved in correspondence of the data partitioning that permitted to maximize the network bandwidth, as expected. As an example, Figure 9 shows the bandwidth measured during the SciddicaT

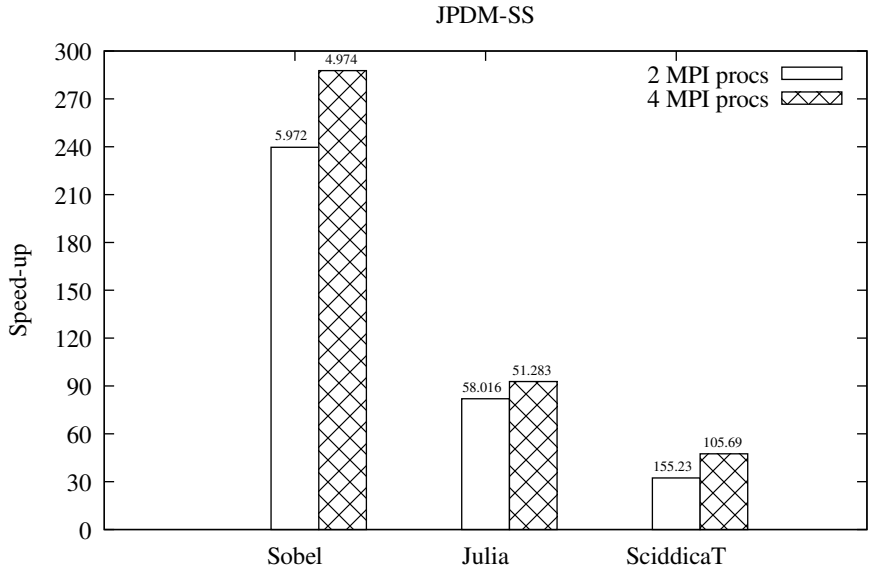


FIGURE 8: Speed-ups achieved by the different multi-node/multi-GPU executions of the Sobel, Julia and SciddicaT benchmarks. Elapsed times in seconds are also shown on top of each speed-up bar. The sequential reference times were taken on the JPDM-1 workstation and are 1,431 s, 4,758 s and 5,015 s for the Sobel, Julia and SciddicaT, respectively. Non uniform domain partitioning was adopted in order to balance the workload between the nodes. Optimal partitioning corresponded to maximun network bandwidth (cmp. Figure 9).

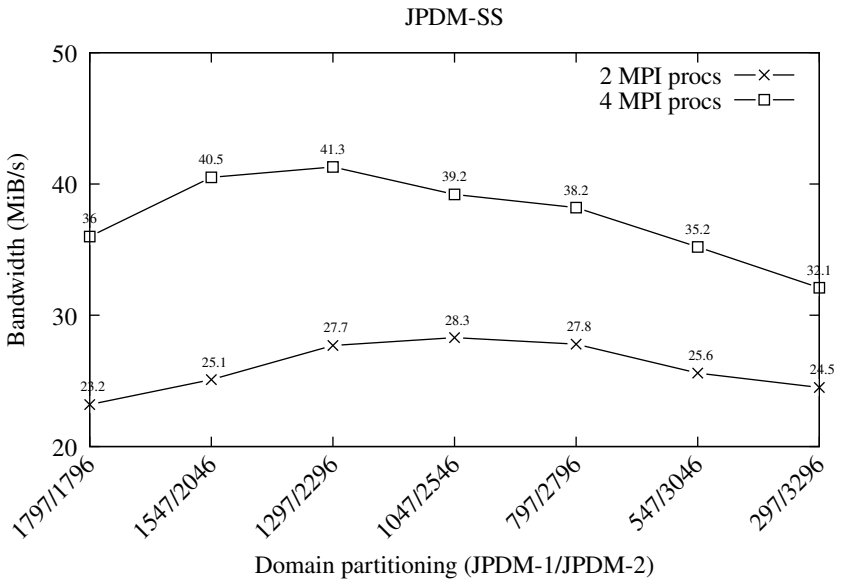


FIGURE 9: Bandwidth registered in correspondence of different domain partitioning for the execution of the SciddicaT benchmark on the JPDM-SS dual-node cluster. Labels of the form r_1/r_2 specifies that r_1 and r_2 domain rows are assigned to the JPDM-1 and JPDM-2 nodes, respectively.

execution on the different data partitions considered. In relative terms, JPDM-SS exhibited better results with respect to its single nodes for the Sobel and Julia benchmarks. The greater improvement was registered for Sobel, which evidenced a speed-up of about 1.4 with respect to JPDM-2 (i.e., the fastest node). Only SciddicaT showed a slight slow-down, nevertheless performing 1.57 times better than JPDM-1. In this case, the analysis of the obtained result is more difficult, due to the presence of the interconnection network. The registered peak bandwidth of 41.3 MiB/s (cmp. Figure 9) does not saturate the Gigabit channel and therefore did not represent a bottleneck in the SciddicaT execution. Nevertheless, as already asserted in [16], the SciddicaT behavior on JPDM-SS can be attributed to the memory-bound nature of the algorithm which requires a fast interconnection network to minimize GPUs idle time.

5. Conclusions and future outlooks

This paper presented an extension of the OpenCAL parallel software library for grid-based modeling to multi-GPU and multi-node computing systems. In particular, the OpenCL-based OpenCAL-CL module was extended to allow for multi-GPU execution on devices interconnected by PCI Express bus (i.e., inside the same node), while the new MPI-based component OpenCAL-CLM was designed and implemented to allow for concurrent execution of multiple OpenCAL-CL instances, thus also allowing to exploit distributed memory computing solutions.

Different tests were carried out on three different benchmark applications. Specifically, they were a Sobel edge detection convolutional graphics filter, a Julia set fractal generator, and the SciddicaT fluid-flow simulation model. Regarding the available parallel system, a dual node test cluster was adopted with nodes equipped with multi-GPU hardware. In particular, one node adopted 2 Nvidia GTX 980, while two Titan Xp and a Tesla K40 Nvidia devices were installed on the other.

The benchmarks were preliminarily characterized as compute/memory bound

algorithms with respect to the considered computational devices. For this purpose, the balanced instruction/byte ratio was adopted for hardware characterization. All many-core devices resulted quite similar for both single-precision floating point and integer operations, while the Tesla K40 resulted considerably more suitable for double-precision computation, as expected. Subsequently, also the three benchmarks were characterized by considering their instruction/byte ratio and compared with the same metrics of the computational devices. The Julia benchmark resulted considerably compute bound with respect all devices, thus performing better on the compute dedicated Tesla K40. On the contrary, Sobel resulted to be memory bound always, performing better according to the different GPUs architectures (the more recent performing better). Eventually, SciddicaT was evaluated to be slightly compute bound for the GTX 980 and the Titan Xp game-oriented GPUs, and more memory bound for the Tesla K40. Accordingly, even in this case, performance followed a trend according to the devices architectures (the more recent still performing better).

For the purpose of dual-GPU performance evaluation, the nodes JPDM-1 and JPDM-2 of the test cluster JPDM-SS were adopted, with two GXT 980 in the first node and two Titan Xp in the second. Moreover, two execution policies were considered, one based on an OpenCL-based multi-GPU implementation, the other on a hybrid approach where MPI was considered for running two concurrent OpenCL single-GPU instances. In all cases, superlinear effects were observed due to cache issues, except for the Julia benchmark that showed a almost linear scalability due to its pronounced compute bound nature.

Eventually, the JPDM-SS cluster was employed for multi-node/multi-GPU tests. Even in this case, the above two execution policies were considered at single-node level. Due to the non-optimal interconnection network and to the unfavorable instruction/byte ratio of the adopted algorithms, sub-linear speed-ups were registered with respect to those achieved on the (most performing) JPDM-2 node, with SciddicaT showing even a slow-down. However, it is worth to note that SciddicaT was the most penalized algorithm on the distributed memory configuration. In fact, the optimization adopted in its more compute

bound elementary process σ_1 actually reduced its application to only the 3.9% of the computational domain, undermining the scalability of the whole algorithm. Nevertheless, SciddicaT performed better on JPDM-SS than on its slower JPDM-1 node.

The OpenCAL-CL and OpenCAL-CLM here presented and tested are however preliminary. Future improvements will regard different design and implementation aspects. Among them, the next release of OpenCAL will provide different domain decomposition alternatives, currently restricted to only row-major partitioning. Another important issue that will be addressed is the improvement of the OpenCAL-CL host execution that currently serializes communications (as for OpenCL API specifications), for instance by taking advantages of possible computation-communication overlapping. Furthermore, an improved halo management will be introduced to avoid sending/receiving unchanged data and to reduce communication frequency which currently occurs at each computational step. Eventually, the system will be tested on a cluster with a high-bandwidth/low-latency compute dedicate interconnection network (e.g., Infiniband), in order to obtain more precise information on the OpenCAL performance on High-Performance Computing solutions.

Acknowledgments

Authors are grateful to Nvidia for providing the GTX980 GPU, Titan Xp and Tesla K40 coprocessor devices that were used in the benchmark tests.

References

[1] J. Was, H. Mroz, P. Topa, GPGPU Computing for Microscopic Simulations of Crowd Dynamics, Computing and Informatics 34 (6) (2015) 1418–1434.

[2] B. Arca, T. Ghisu, G. Trunfio, Gpu-accelerated multi-objective optimization of fuel treatments for mitigating wildfire hazard, Journal of Computational Science (11) (2015) 258–268.

[3] D. D'Ambrosio, G. Filippone, D. Marocco, R. Rongo, W. Spataro, Efficient application of GPGPU for lava flow hazard mapping, The Journal of Supercomputing 65 (2) (2013) 630–644.

[4] G. H. Golub, J. M. Ortega, Scientific computing: an introduction with parallel computing, Academic Press, London, UK, 2014.

[5] S. Schneiderbauer, M. Krieger, What do the Navier-Stokes equations mean?, European Journal of Physics 35 (1) (2014) 015020.

[6] S. Mazumder, Chapter 2 - The Finite Difference Method, in: S. Mazumder (Ed.), Numerical Methods for Partial Differential Equations, Academic Press, 2016, pp. 51 – 101.

[7] I. Reguly, G. Mudalige, M. Giles, D. Curran, S. McIntosh-Smith, The OPS domain specific abstraction for multi-block structured grid computations, in: Proceedings of WOLFHPC 2014: 4th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing - Held in Conjunction with SC 2014: The International Conference for High Performance Computing, Networking, Storage and Analysis, 2014, pp. 58–67.

[8] S. Jammy, G. Mudalige, I. Reguly, N. Sandham, M. Giles, Block-structured compressible Navier-Stokes solution using the OPS high-level abstraction, International Journal of Computational Fluid Dynamics 30 (6) (2016) 450–454.

[9] M. Giles, G. Mudalige, B. Spencer, C. Bertolli, I. Reguly, Designing OP2 for GPU architectures, Journal of Parallel and Distributed Computing 73 (11) (2013) 1451 – 1460.

[10] I. Reguly, G. Mudalige, C. Bertolli, M. Giles, A. Betts, P. Kelly, D. Radford, Acceleration of a Full-Scale Industrial CFD Application with OP2, IEEE Transactions on Parallel and Distributed Systems 27 (5) (2016) 1265–1278.

[11] J. Cercos-Pita, AQUAgpusph, a new free 3D SPH solver accelerated with OpenCL, Computer Physics Communications 192 (2015) 295–312.

[12] Advanced Simulation Library, http://asl.org.il/, accessed: 2017-10-16.

[13] G. Dattilo, G. Spezzano, Simulation of a cellular landslide model with CAMELOT on high performance computers, Parallel Computing 29 (10) (2003) 1403–1418.

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

[15] G. Spingola, D. D'Ambrosio, W. Spataro, R. Rongo, G. Zito, Modeling Complex Natural Phenomena with the libAuToti Cellular Automata Library: An example of Application to Lava Flows Simulation, in: PDPTA - International Conference on Parallel and Distributed Processing Techniques and Applications, 2008, pp. 277–283.

[16] D. D'Ambrosio, A. D. Rango, M. Oliverio, D. Spataro, W. Spataro, R. Rongo, G. Mendicino, A. Senatore, The open computing abstraction layer for parallel complex systems modeling on many-core systems, Journal of Parallel and Distributed Computing. doi:https://doi.org/10.1016/j.jpdc.2018.07.005.

[17] A. De Rango, P. Napoli, D. D'Ambrosio, W. Spataro, A. Di Renzo, F. Di Maio, Structured Grid-Based Parallel Simulation of a Simple DEM Model on Heterogeneous Systems, 2018, pp. 588–595. doi:10.1109/PDP2018.2018.00099.

[18] S. Di Gregorio, R. Serra, An empirical method for modelling and simulating some complex macroscopic phenomena by cellular automata, Future Generation Computer Systems 16 (1999) 259–271.

[19] D. D'Ambrosio, S. Di Gregorio, G. Iovine, Simulating debris flows through a hexagonal cellular automata model: SCIDDICA S3-hex, Natural Hazards and Earth System Science 3 (6) (2003) 545–559.

[20] M. Avolio, S. Di Gregorio, V. Lupiano, P. Mazzanti, SCIDDICA-SS3: a new version of cellular automata model for simulating fast moving landslides, The Journal of Supercomputing 65 (2) (2013) 682–696.

[21] D. D'Ambrosio, R. Rongo, W. Spataro, G. Trunfio, Meta-model assisted evolutionary optimization of cellular automata: An application to the SCIARA model, Lecture Notes in Computer Science 7204 (PART 2) (2012) 533–542.

[22] D. D'Ambrosio, R. Rongo, W. Spataro, G. Trunfio, Optimizing cellular automata through a meta-model assisted memetic algorithm, Lecture Notes in Computer Science 7492 (PART 2) (2012) 317–326.

[23] M. Oliverio, W. Spataro, D. D'Ambrosio, R. Rongo, G. Spingola, G. Trunfio, OpenMP parallelization of the SCIARA Cellular Automata lava flow model: Performance analysis on shared-memory computers, in: Procedia Computer Science, Vol. 4, 2011, pp. 271–280.

[24] D. D'Ambrosio, R. Rongo, W. Spataro, M. Avolio, V. Lupiano, Lava invasion susceptibility hazard mapping through cellular automata, Lecture Notes in Computer Science 4173 (2006) 452–461.

[25] M. Avolio, G. Crisci, S. Gregorio, R. Rongo, W. Spataro, D. D' Ambrosio, Pyroclastic flows modelling using cellular automata, Computers and Geosciences 32 (7) (2006) 897–911.

[26] G. 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) (2005) 1019–1032.

[27] M. Avolio, S. Di Gregorio, G. Trunfio, A randomized approach to improve the accuracy of wildfire simulations using cellular automata, Journal of Cellular Automata 9 (2-3) (2014) 209–223.

[28] G. Mendicino, J. Pedace, A. Senatore, Stability of an overland flow scheme in the framework of a fully coupled eco-hydrological model based on the Macroscopic Cellular Automata approach, Communications in Nonlinear Science and Numerical Simulation 21 (1-3) (2015) 128–146.

[29] G. Ravazzani, D. Rametta, M. Mancini, Macroscopic cellular automata for groundwater modelling: A first approach, Environmental Modelling and Software 26 (5) (2011) 634–643.

[30] G. Mendicino, A. Senatore, G. Spezzano, S. Straface, Three-dimensional unsaturated flow modeling using cellular automata, Water Resources Research 42 (11) (2006) W11419.

[31] G. Cervarolo, G. Mendicino, A. Senatore, A coupled ecohydrological-three-dimensional unsaturated flow model describing energy, H2O and CO2 fluxes, Ecohydrology 3 (2) (2010) 205–225.

[32] L. G. Shapiro, G. C. Stockman, Computer Vision, 2001.

[33] M. Hadwiger, J. M. Kniss, C. Rezk-salama, D. Weiskopf, K. Engel, Real-time Volume Graphics, A. K. Peters, Ltd., Natick, MA, USA, 2006.

[34] L. Carleson, T. W. Gamelin, Complex Dynamics. Springer, Springer, 1993.

[35] M. Avolio, S. Di Gregorio, F. Mantovani, A. Pasuto, R. Rongo, S. Silvano, W. Spataro, Simulation of the 1992 Tessina landslide by a cellular automata model and future hazard scenarios, International Journal of Applied Earth Observation and Geoinformation 2 (1) (2000) 41–50.

[36] V. Volkov, Understanding Latency Hiding on GPUs, Ph.D. thesis, EECS Department, University of California, Berkeley (2016).

[37] v.v., Nvidia's next generation cuda compute architecture: Kepler tm gk110 - whitepaper, Tech. rep., Nvidia Coorporation (2012).