Special Issue: V3-LAVA PROJECT

Scalable multi-GPU implementation of the MAGFLOW simulator

Eugenio Rustico[1,*], Giuseppe Bilotta[1,2], Alexis Hérault[2,3], Ciro Del Negro[2], Giovanni Gallo[1]

[1] *Università di Catania, Dipartimento di Matematica e Informatica, Catania, Italy*

[2] *Istituto Nazionale di Geofisica e Vulcanologia, Sezione di Catania, Osservatorio Etneo, Catania, Italy*

[3] *Conservatoire des Arts et Métiers, Département Ingénierie Mathématique, Paris, France*

Article history
Received November 23, 2010; accepted July 15, 2011.
Subject classification:
Computational geophysics, General purpose GPU modeling, HPC, Parallel programming, GPU, Multi-GPU, Hazard, Lava.

ABSTRACT

We have developed a robust and scalable multi-GPU (Graphics Processing Unit) version of the cellular-automaton-based MAGFLOW lava simulator. The cellular automaton is partitioned into strips that are assigned to different GPUs, with minimal overlapping. For each GPU, a host thread is launched to manage allocation, deallocation, data transfer and kernel launches; the main host thread coordinates all of the GPUs, to ensure temporal coherence and data integrity. The overlapping borders and maximum temporal step need to be exchanged among the GPUs at the beginning of every evolution of the cellular automaton; data transfers are asynchronous with respect to the computations, to cover the introduced overhead. It is not required to have GPUs of the same speed or capacity; the system runs flawlessly on homogeneous and heterogeneous hardware. The speed-up factor differs from that which is ideal (#GPUs×) only for a constant overhead loss of about $4E^{-2} \cdot T \cdot \#GPUs$, with T as the total simulation time.

1. Introduction

Numerical problems that expose a high degree of intrinsic parallelism can often be mapped to more than one level of parallelism. If it is possible to model a problem as a set of pseudo-independent subproblems, then it is possible to split the set of computations needed to solve it into two or more partitions that can be executed separately. Very few numerical problems have completely inter-dependent subproblems, such as astrophysical n-body simulations; sometimes it is nevertheless possible to parallelize such problems by means of numerical approximations or by adding specific constraints to the model. Cellular automaton methods are by definition made up of quasi-independent subproblems: the state of every cell of the automaton depends only on the previous state of the same cell and on the state of the neighbor cells. Given a cell and its neighborhood, we can compute its next state in parallel and with almost no knowledge about the other cells. The same reasoning applies to sets of cells that are in turn independent from other sets, except for their neighborhood. In the general case, it is possible to exploit a two-level parallelism and to split an already parallel cellular automaton simulation into parts to be executed on different *devices*, which can include a graphics processing unit (GPU) card, a central processing unit (CPU) core, the node of a cluster, or even a computer in a network. The real feasibility and the benefit of this theoretical possibility are, however, to be evaluated case by case, as there may be model peculiarities that require specific adaptations or that conflict with the technical limitations of the underlying architecture. The most limiting factors are typically the latency and/or the bandwidth of the inter-device communication channel, which is usually orders of magnitude slower than any intra-device communication.

The MAGFLOW lava simulation model [Vicari et al. 2007] is the cellular automaton developed by the Sezione di Catania of the Istituto Nazionale di Geofisica e Vulcanologia (INGV), and it represents the peak of the evolution of cell-based models for lava-flow simulations. The conversion of the MAGFLOW cellular automaton from serial single-CPU code to a parallel, GPU-based implementation is discussed in Bilotta et al. [2011] in this volume; here, we show how we exploited a second level of parallelism by running a simulation on two or more GPUs simultaneously.

There are at least two ways to exploit a two-levels parallelism in the evolution of a cellular automaton. The simplest and most intuitive is to spatially partition the automaton into disjointed areas and to assign each area to a separate device. As we need to operate locally and read information about the neighboring cells, these areas can never be truly disjointed; it is therefore more correct to refer to them as *subsets* rather than *partitions*. We could instead operate the division in the domain of computations, thus realizing a *pipeline*: each device is assigned to all of the cells in the domain, but elaborates only for a specific phase of the evolution. Unfortunately, this approach presents three major drawbacks:

1. It is not possible to scale to an arbitrary number of devices.

2. It is not trivial to balance the computational load among the devices.

3. It requires the complete domain to be constantly transferred from one device to another, to keep every device of the system updated about the current automaton status.

For these reasons, we chose the former division and we focus on how the automaton can be split in the spatial domain.

2. The Compute Unified Device Architecture architecture

A GPU is a specialized microprocessor that is traditionally used to offload and accelerate graphic rendering pipelines from a CPU. While the market of videogames has been more and more demanding of computational power, GPUs have evolved to include on-board microchips with direct hardware support for common two-dimensional (2D) and 3D graphics primitives. They have become incredibly faster than CPUs in parallel, arithmetic-intensive tasks, like matrix manipulations and coordinate projections. In 2001, NVIDIA released the first chip capable of programmable shading, i.e. a chip that it was possible to program with a customized arithmetic pipeline for very sophisticated surface renderers. As GPUs became programmable and really fast in matrix and vector operations, engineers and scientists started to use them for non-graphical calculations, by mapping a non-graphic problem to a graphic one, and letting a GPU compute it. In 2007, NVIDIA released the Compute Unified Device Architecture (CUDA), a hardware and software architecture that has explicit support for general purpose programmability. CUDA has enabled programmers to implement generic algorithms on NVIDIA GPUs using an extension of the C programming language. Soon after, the main competitor of NVIDIA, ATI, also released a general purpose GPU architecture called Stream, which has a slightly different access interface (an assembly-like language).

A third, architecture-independent, parallel computing platform, OpenCL, was proposed by Apple in 2008 and released in 2009, as the result of a collaboration with AMD, IBM, Intel and NVIDIA. OpenCL offers an abstraction layer between the application and the parallel computing hardware, which allows an application to be transparently executed on multicore CPUs as well as on GPUs; fine-tuning of the OpenCL code is, however, more complex, due to the wider variety of platforms on which it can run.

We started porting MAGFLOW to GPUs in 2007, when CUDA was the only platform that allowed the use of standard C language to write kernels. Since the OpenCL interface is similar to the low-lever CUDA C API, we are considering the possibility to port the MAGFLOW simulator also to OpenCL.

A modern GPU hosts a set of Single Instruction Multiple Data (SIMD) multicore processors, with a few hundred total cores. From the perspective of a programmer, a GPU can be seen as a coupled, external computer: it has its own RAM and processors, and the typical workflow consists of sending the data to the GPU, asynchronously requesting a computation, and finally transferring back the resulting output. A sequence of instructions compiled to be executed on a GPU is called a *kernel*. Kernels are instantiated in parallel threads, which for the convenience of the programmer, are grouped into 1D, 2D or 3D blocks; the thread blocks themselves can be organized into 1D or 2D grids.

With a slight change in the programming logic, numerical problems that expose a high level of parallelism can achieve speed-ups of two orders of magnitude with respect to the correspondent CPU implementations. The main breakthrough of this phenomenon, however, is the high cost effectiveness of GPU-based solutions. A workstation with a modern GPU easily reaches the theoretical speed of 1.4 TFlops (10^{12} floating-point operations per second) with an expense of less than €1,000 and a power consumption of about 600 W. Even dedicated hardware like the NVIDIA Tesla cards, which share the computing hardware with consumer-level GPUs, is still a very cost-effective solution. It is therefore not surprising that in May, 2010, a hybrid CUDA-enabled Chinese supercomputer, called Nebulae, reached the theoretical peak performance of 2.98 PFlops, ranking it first in the list of the most powerful commercial systems in the World [Meuer et al. 2010]. Although CUDA is still young and continuously evolving, it is nowadays a *de-facto* standard among low-end and medium-end parallel computing platforms.

3. MAGFLOW on single GPUs

In this section, we present only a concise summary of the implementation of the MAGFLOW numerical model on CUDA GPUs, as a necessary basis to understand the multi-GPU structure and the additional problems that arise in the exploitation of a second level of parallelism. For a more detailed description, please refer to Bilotta et al. [2011], in this volume.

The MAGFLOW cellular automaton consists of a two-dimensional grid of cells, each of which is described by five scalar quantities: ground elevation, lava thickness, heat quantity, temperature, and amount of solidified lava. The ground elevation is initialized once, and it remains constant throughout the simulation. We compute the *state* of a cell at time t from the state of the cell and its immediate neighbors at time $t-1$. The lava thickness varies according to the lava emission of the vents on the cell, if any, and to the amount of cross-cell lava flux, between the cell and its immediate neighbors (eight neighbors for square of cells). The flux depends on the height difference, and is computed using a steady-state solution for the 1D Navier–Stokes equations for a fluid with Bingham rheology. The actual amount of lava gained or lost by a cell at the end of each iteration of the automaton is given by the total flux of the cell (vent emissions plus cross-cell flux) multiplied by the time-step *dt* for that iteration. The higher the *dt*, the smaller the number of iterations we need to complete the simulation. However, the time-step is upper-bounded by a numerical constraint, to prevent nonphysical solutions. At each iteration, we are thus interested in computing the maximum time-step value allowed by each cell. We choose as the global time-step the minimum of these values, thus ensuring both adherence to reality and the minimal number of iterations to be computed.

Each iteration is made up of the following steps:

1. For each cell, compute the eruption flux if the cell is a vent cell.

2. For each cell, compute the flux transfer with the neighboring cells and the maximum allowed time-step.

3. Find the global minimum time-step among the computed maximums.

4. For each cell, update the lava thickness and the other fields (i.e. solidification, radiation loss), according to the chosen time-step.

It is straightforward to implement these steps in four dedicated CUDA kernels. Steps 1, 2 and 4 are written from scratch, while step 3 is provided by a well-known, highly optimized, array-scan library [Sengupta et al. 2008]. The first level of parallelism is represented by the "for each" at the beginning of the statements: one *thread* on the GPU computes the evolution of a single cell of the automaton, with a 1:1 mapping. Note that only in steps 1 and 4 are the cells fully independent of each other: step 2 requires access to the neighboring cells, and step 3 requires a global selection algorithm. This consideration is particularly important when designing a multi-device structure: additional subproblem dependencies amplify the inter-device communication latencies, and can bring the overall performance down if an appropriate synchronization strategy is not implemented.

4. MAGFLOW on multiple GPUs

We now analyze how the cross-cell dependencies in steps 2 and 3 affect the structure of the multi-GPU MAGFLOW, and why it is nontrivial to synchronize the available devices efficiently.

The 2D arrays holding the automaton-cell data are stored in row-major order, following the C convention. This structure must be taken into account for the optimization of memory transfers, as accessing a burst of consecutive addresses is faster than accessing the same amount of data sparsely distributed. As a consequence, we only split the domain horizontally, so that the synchronization involves transfers of row segments. In special cases of very narrow, rectangular domains, we can load transposed data, depending on which choice leads to shorter rows.

Let D be the number of available GPU devices on a machine. On a workstation, we usually have $1 \leq D \leq 3$, while on a dedicated rack, we migh have up to eight devices per PCI domain. Also, let R be the number of rows and C the number of columns of the automaton. We split our $R{\times}C$ grid in horizontal strips, and we assign bordering strips to consecutive GPU indices; we will refer to these strips as *subdomains*. If we split the domain into equal parts, assuming for simplicity that R is a multiple of D, the device of index d will be assigned to zero-based rows $(R/D) * d$, to $(R/D) * (d + 1) - 1$.

However, to compute cross-cell lava fluxes of row $(R/D) * d$, device d also needs to read the cells in row $(R/D) * (d + 1) - 1$; i.e. the last row of device $d - 1$. The same reasoning applies to row $(R/D) * (d+1) - 1$, which requires reading access to row $(R/D) * (d + 1)$. Devices $d - 1$ and $d + 1$ need in turn to read the first and the last rows assigned to device d, respectively; the minimum overlapping between contiguous subdomains is therefore two rows high. We call rows $(R/D) * d$ to $(R/D) * (d + 1) - 1$ *internal* rows for device d, while rows $(R/D) * d - 1$ and $(R/D) * (d + 1)$ are *external* rows for the same device.

Every device computes and writes the status of all of its internal rows; the external rows are just copies accessed for reading only. Figure 1 is a visual representation of this scheme. Steps 1 and 4 can be computed in parallel, with no communication at all among the devices; no significant changes are required with respect to the single-GPU code. Step 2 can be computed independently in different devices, up to a minimal overlapping border. Step 3 is easily extended from 1 to D devices: each device provides its own subdomain

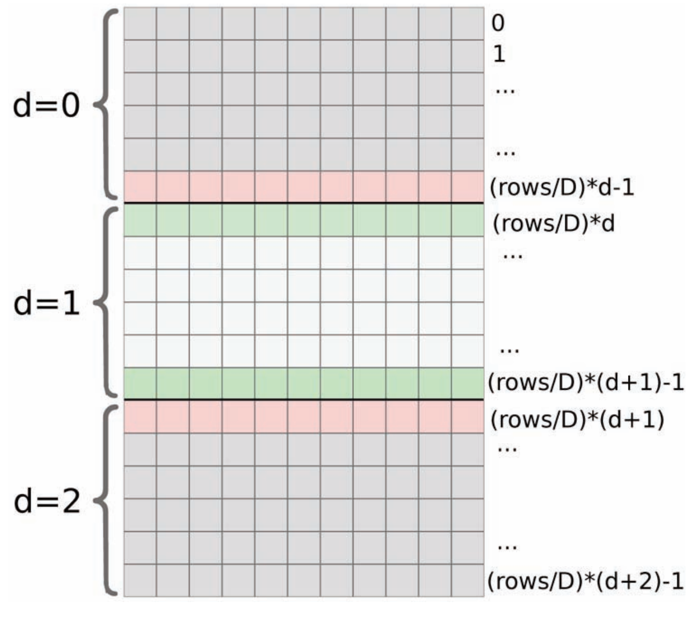


Figure 1. Representation of subdomain overlapping. The internal rows of the strip assigned to device $d = 1$ are green, the external rows (i.e. one internal to device $d = 0$ and one internal to $d = 2$) are light red.

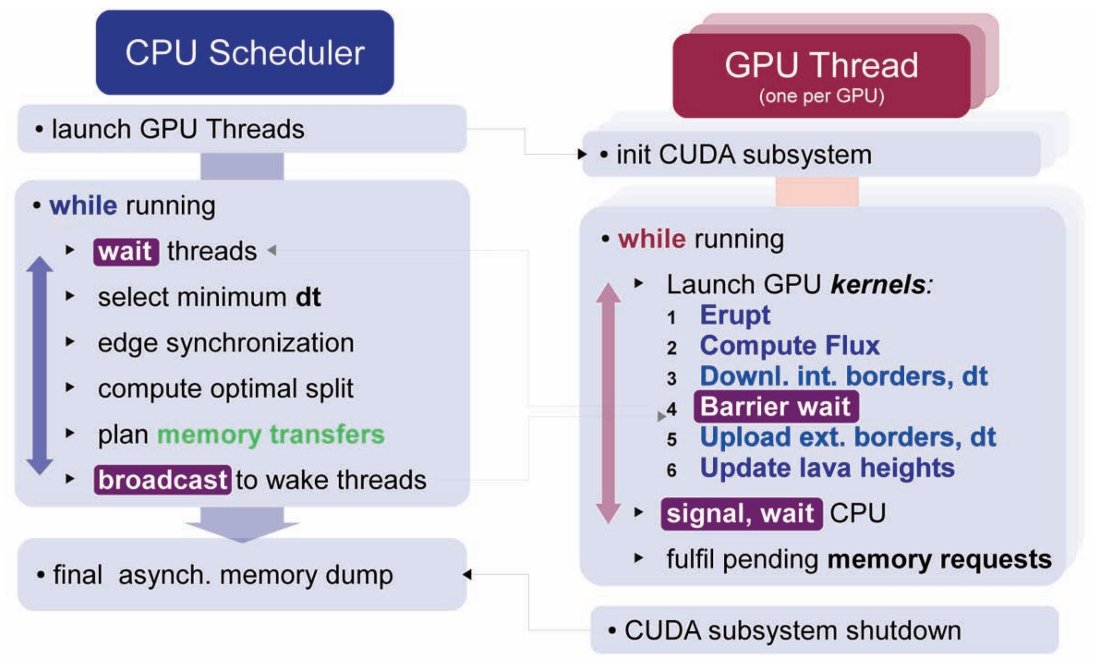


Figure 2. Overall structure of the simulator. The main thread launches and synchronizes one GPUThread per device.

minimum, the CPU quickly finds the smallest of the *D* local values and the global minimum value is passed back to the GPUs, to be used in step 4. This requires a barrier in the iteration: after having found the local minimum, all of the GPUs must communicate this value and wait for the global one. The barrier consists of a device flushing call (`cudaThreadSynchronize`) followed by a CPU thread barrier (`pthread_barrier_wait`).

Let us use a CPU-centered nomenclature and call a *download* any transfer of data from the GPU to the CPU, and an *upload* any transfer in the opposite direction. At the end of each iteration, after step 4, every device downloads its first and last internal rows to a shared CPU buffer and uploads from this the external rows, which have just been downloaded by the neighboring devices; another break is required as a synchronizing barrier between the download and the upload, to ensure that each device reads up-to-date values.

So far, every device has to stop two times for each iteration: once to obtain the global time-step and once to wait for downloads to complete. With a little design change, however, we can make every GPU stop only once, therefore speeding up the whole process. The "trick" consists of exchanging the amount of flux at the end of step 2 instead of the lava thickness at the end of step 4; this exploits the barrier needed for the time-step selection; the fluxes are already computed when the devices wait for the time-step, and they can be updated later when the time-step is available. Figure 2 represents the sequence of operations that every device executes according to the new logic. This was an important breakthrough, as stopping twice was the main source of latency in our early prototypes.

5. Implementation details

The CUDA parts of the MAGFLOW implementation have undergone almost no changes since the single-GPU version; the rest of the simulator, on the other hand, has been completely redesigned.

In the multi-GPU implementation, we introduced the class GPUThread, which encapsulates all of the pointers and primitives that are needed to handle a GPU; one GPUThread per device is allocated. Every GPUThread starts a dedicated thread in constant communication with the associated device. The wait mechanism is based on the barrier primitives of the NTPL pthread implementation [Drepper and Molnar 2005]; as an alternative, it is possible to choose a busy-wait mechanism, which is more responsive, but far more CPU-consuming.

The main thread has a flux exchange buffer to enable the GPUs to download and upload their internal and external rows; this is double buffered to keep data consistency over consecutive iterations. The main thread manages the split, balances the workload, tracks the simulation time, and periodically requests a state dump from the GPUs according to a user-defined save frequency. An optional auxiliary thread periodically checks the status of the whole system, for debugging purposes.

It is possible to specify at the command line several options, like the input files, the possibility to disable asynchronous data transfers, the list of devices to be used, and so on; the devices can be heterogeneous and can even belong to different generations (e.g. Fermi and Tesla). It is possible to specify a device more than once: this causes two or more GPUThreads to use the same physical device, thus emulating a multi-GPU behavior on a single-GPU machine.

The data is not multi-GPU aware: it is possible to save the state of a single-GPU simulation and to load it into a multi-GPU environment, and *vice-versa*. The transfer times for the exchanges of updated overlapping borders are covered by exchanging the internal borders as soon as they are ready, while the rest of the cells are still being processed. Thus, the system scales on any number of CUDA-enabled devices with compute capability 1.1 or higher, as capability 1.0 does not support concurrent memory transfers and kernel execution.

The execution time of each iteration is lower bounded by the slowest device; a dynamic load balancing is necessary to limit the relative time differences and to avoid undesired bottlenecks.

A space domain subdivision is currently performed and dynamically updated as the bounding box grows; the domain is distributed proportionally to the number of multiprocessors in each active device.

While this approach works reasonably well, there also many unpredictable, run-time factors that can influence the performance of a device (thread scheduling order, optimization of the CUDA run-time in kernel executions, PCI bus conflicts, lava flow topology, etc.). The only way to take these factors into account is to distribute the computing load according to the execution times of the previous iterations.

We are currently implementing a smart balancing policy that analyzes the fluctuations of the execution times in relation to the subdomain sizes, which takes into account all of the explicit and implicit factors in an *a-posteriori* analysis (i.e. based on the effective execution times for assigned subdomains). However, due to the scattered nature of these timings and the risk of balancing oscillations in local minima, such an analysis is far from trivial to design, and it requires the use of advanced signal-processing techniques.

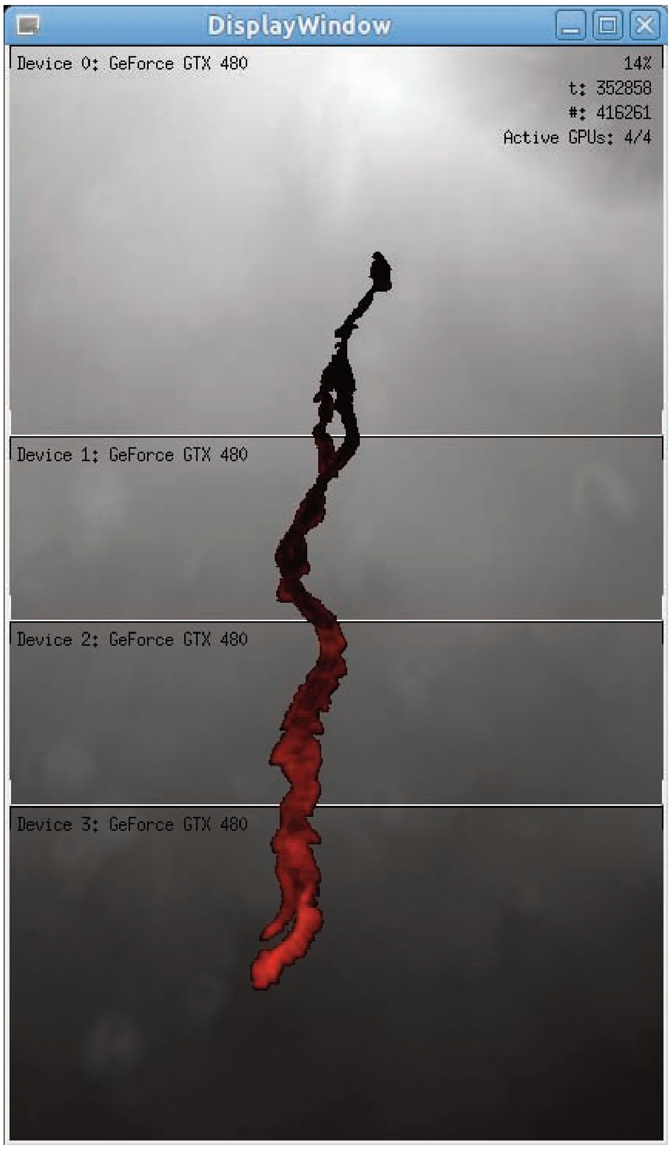
Another approach to dynamic load balancing relies on overriding the CUDA run-time scheduler policies with a constantly running custom kernel [Chen et al. 2010]. This approach is especially useful when the problem modeled has a large number of small, heterogeneous, independent tasks, at the price of additional design complexity and the reduced possibility for *a-posteriori* balancing.

6. Interface

The system can provide a graphical display of the state of the simulation (Figure 3). This display is not aimed at being a realistic representation of the phenomenon, but at showing the course of the simulation in a concise and immediate way.

Basic status information are shown, including the GPUs being used and the corresponding subdomain split. It is possible to draw the subdomain limits and/or the minimum bounding box that contains all of the cells with non-zero amounts of lava.

The window is optimally zoomed to fit onto the monitor with any automaton resolution. It is possible to draw the topography and the lava heights produced, with an adaptive palette related to the thickness of fluid or solid lava.



```
Device 0: GeForce GTX 480                14%
                                   t: 352858
                                   #: 416261
                              Active GPUs: 4/4
Device 1: GeForce GTX 480
Device 2: GeForce GTX 480
Device 3: GeForce GTX 480
```

Figure 3. Screenshot of the simulator interface while running a 4-GPUs simulation.

7. Preliminary performance analysis

The main bottleneck of multi-device set-ups is the inter-device communication latency. Although code refactoring has brought several improvements that are not specific to multi-GPUs, like the ability to save state files asynchronously with respect to computations, we needed to exploit all of the advanced techniques offered by CUDA to overcome the communication overhead that arose to maintain continuous synchronization.

Figure 4 shows the average execution times of each step of the automaton evolution during a single-GPU simulation, in milliseconds. As expected, computing the cross-cell fluxes is the most expensive step in terms of computational time, which takes 0.106 ms (68%), as well as the only step that changes significantly during the

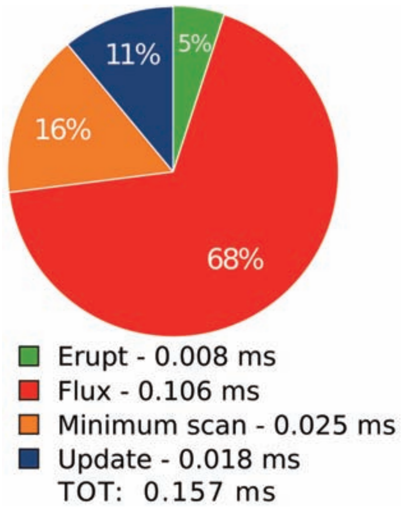


Figure 4. Average execution times of each step during a single-GPU simulation, 5 m DEM.

computation (see also Figure 5); finding the minimum time-step takes 0.025 ms; computing the lava heights from the flux takes 0.018 ms, while eruption requires only 0.008 ms. It is more interesting, however, to analyze how these times vary in relation to the number of active cells in the automaton (i.e. the number of threads running on the GPU), as is plotted in Figure 5. Two steps basically take constant times: the eruption and the minimum scan. The time for updating the cells increases linearly with the number of cells, with very slow growth (0.02 ms from 1 to 60,000 cells). We do not expect to have any performance gain with the distribution of the work of these steps to more than one device. Computing fluxes follow a more interesting trend. Their execution time is almost constant until about 6,000 cells, where there is a sudden increase, and it grows linearly afterwards. When an increase in the number of active cells does not lead to a higher execution time, we are not effectively using all of the cores of the GPU; the sudden increase corresponds to the hardware saturation point, from which point on we expect a positive linear relationship between the number of threads on the device and the execution time.

Splitting the automaton before the saturation of all of the active devices would therefore give us no speed-up. Our multi-GPU implementation enables one GPU at a time, as long as all of the already active devices are saturated. The profiling of the application with CUDA Visual Profiler showed that a device should not be considered saturated before at least one order of magnitude more blocks are assigned than the number of its multiprocessors. Every time a new device is activated, the domain is split again according to the estimated power ratios (i.e. number of multiprocessors) of active GPUs.

8. Multi-GPU execution times

When the automaton has been distributed over two or more GPUs, we observe an important change in the proportion of execution times. Figure 6 shows the average timings over 5,000 iterations of a 5-m simulation just after the automaton has been split in half.

Computing the flux and updating the cells take almost half the time, while the eruption and minimum scan are basically unchanged, as expected. However, the total time is only 10% smaller, because we have to take into account the downloading and uploading of the overlapping rows, and together, these transfers take more than 20% the new total

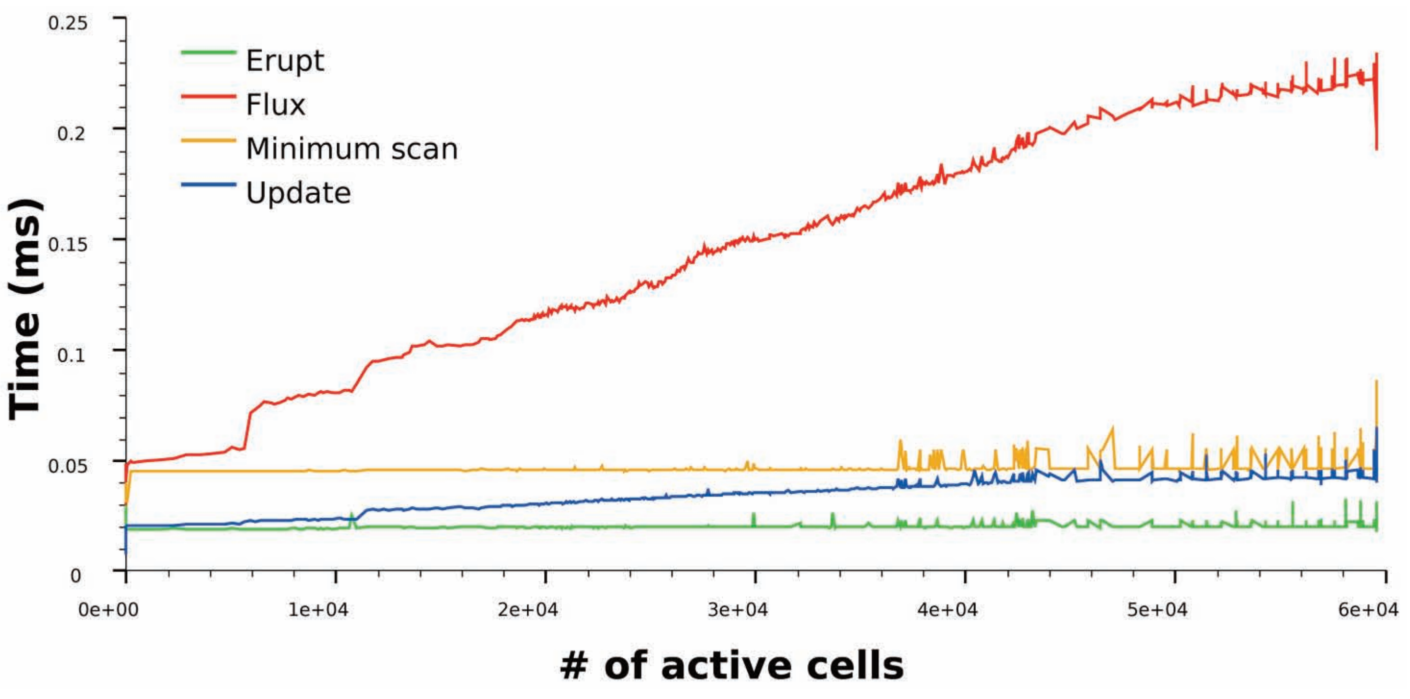


Figure 5. Execution times for each step, with respect to the number of active cells in the domain, 5 m DEM.

time. Making this overhead negligible is the main challenge of any multi-GPU system where continuous inter-device communication is needed. We have covered these latencies almost completely by the transfer of the overlapping borders simultaneously with the computation of the internal cells, with the results shown in Table 1.

Our test-bed was the simulation of lava eruptions on Etna volcano in the year 2001, on DEMs with 2 m and 5 m resolution, loading from a 20% complete state, where the active box had already reached its maximum extension. This was simulated on a TYAN-FT72 rack mounting a dual-Xeon processor, 16 Gb RAM and 6×GTX480 cards. All of the tests have been numerically validated, and the time comparison refers to the same simulation with the same options (e.g. the stated dumping frequency), as computed by the single-GPU code.

The execution times obtained differ from the ideal times only for a constant overhead loss of about $4E^{-2} \cdot T \cdot \#\text{GPUs}$, with T being the total simulation time. Figure 7 shows the real execution times, the ideal times, and the differences between the two. In this specific example, splitting the simulation across more than 5 GPUs is not convenient, as the cost becomes comparable with the simulation time.

In general, the maximum number of GPUs across which it is convenient to distribute the computation depends on the total extent of the simulated flow, i.e. on the number of active cells, as well as on the additional cost mentioned above. For example, the same eruption simulated on a 5 m DEM can benefit at most from being distribute across 3 GPUs.

The linear cost is mainly due to the lack of an *a-posteriori* balancing policy: a lava flow topology can cause two equally shaped subdomains to require slightly different amounts of computations (due, e.g., to different numbers of cells with the actual flow, different memory access patterns, etc.).

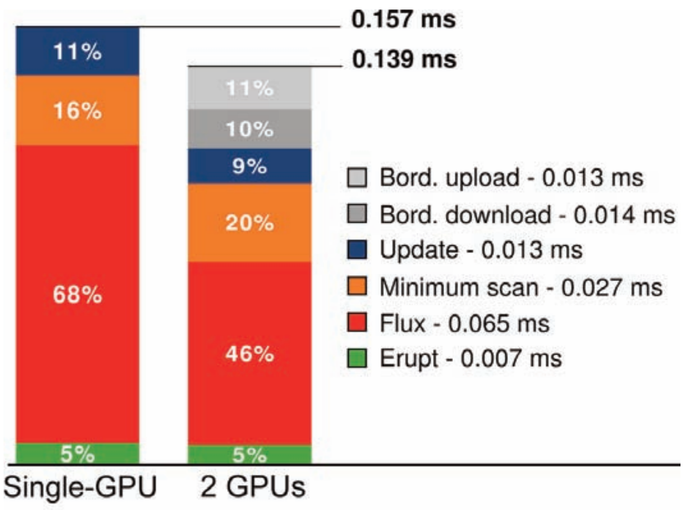


Figure 6. Average execution times of each step before and after splitting the domain, 5 m DEM resolution.

	1 GPU	2 GPU	3 GPU	4 GPU	5 GPU	6 GPU
Real time (s)	14377	7737	5813	4993	4562	4500
Ideal time (s)	14377	7189	4792	3594	2875	2396
Real cost (2)	0	549	1021	1399	1687	2104

Table 1. Execution times and cost in seconds, for 1 to 6 devices, and 2 m DEM computed by the single-GPU code.

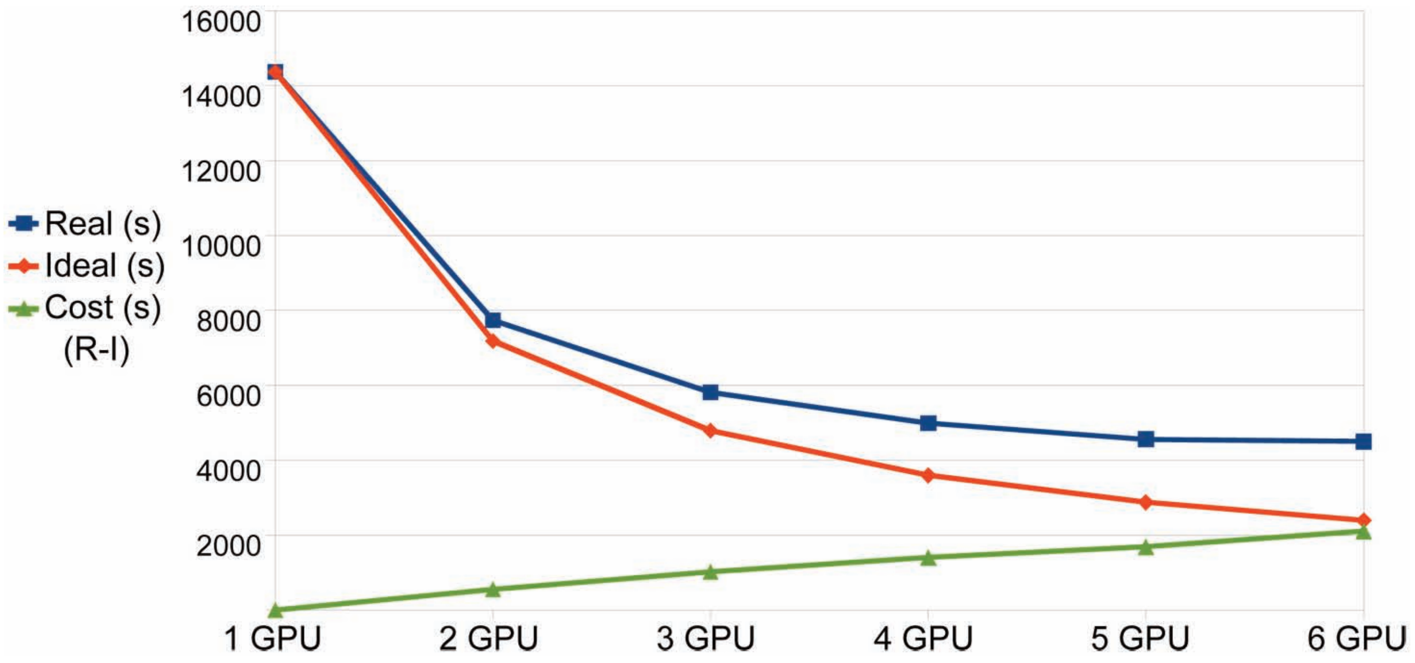


Figure 7. Plot of the execution times (~18 h of simulated time; DEM, 2 m), ideal times and linear cost, for 1 to 6 devices.

9. Conclusions and future work

We have presented here a CUDA-based multi-GPU implementation of the MAGFLOW cellular automaton that transparently scales on an arbitrary number of homogeneous or heterogeneous GPUs. Although running a physics-adherent simulation on GPUs is already faster than real-time, even faster simulations might open new use scenarios and bring higher levels of flexibility in model validation and scenario forecasting.

The speed-up obtained differs from the ideal one by a cost function that is linear with the number of GPUs, which is due to the slightly different execution times of equally shaped areas with different numbers of cells containing lava and different flow topologies.

It is still possible to improve upon these results, both for the model and for the technical implementation sides. We are currently performing further tests to shortly complete a system that has automatic fine-tuning of the execution parameters, to fully exploit the hardware computational power. By interlacing array data, memory transfers can be merged into a single transfer per iteration. Although CUDA Toolkit 3.2 is far more mature than the first release, some features still need to be completed, and some of them are not well documented yet. Above all, a more advanced profiling tool that can obtain absolute execution times of GPU operations is essential, to get accurate information about the real overlap among GPU transfers and computations, which is currently not possible with the standard profiling tools delivered with CUDA.

The recent release of CUDA 4.0 has introduced new multi-GPU-specific features, like device-to-device transfers and a unified address space for both CPU and GPU memories. While inter-device transfer costs are already almost completely covered, we plan to test these new features in the very near future.

Finally, we will use the techniques and know-how developed to date in other computationally expensive numerical problems, such as mesh-free fluidodynamics and more generic task-driven multi-GPU schedulers.

References

Bilotta, G., E. Rustico, A. Hérault, A. Vicari, G. Russo, C. Del Negro and G. Gallo (2011). Porting and optimizing MAGFLOW on CUDA, Annals of Geophysics, 54 (5), 580-591 ; doi: 10.4401/ag-5341 (this issue).

Chen, L., O. Villa, S. Krishnamoorthy and G.R. Gao (2010). Dynamic load balancing on single- and multi-GPU systems, In: Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium, 1-12; doi: 10.1109/IPDPS.2010.5470413.

Drepper, U. and I. Molnar (2005). The Native POSIX Thread Library for Linux, February 21, 2005, 17 pp.; URL: http://people.redhat.com/drepper/nptl-design.pdf.

Meuer, H., E. Strohmaier, J. Dongarra and H. Simon (2010). The TOP500 Project; URL: http://www.top500.org/lists/2010/06.

Sengupta, S., M. Harris and M. Garland (2008). Efficient parallel scan algorithms for GPUs, NVIDIA Technical Report NVR-2008-003, Dec. 2008, 17 pp.; URL: http://mgarland.org/files/papers/nvr-2008-003.pdf

Vicari, A., A. Hérault, C. Del Negro, M. Coltelli, M. Marsella and C. Proietti (2007). Modeling of the 2001 lava flow at Etna volcano by a celluar automata approach, Environ. Modell. Softw., 22, 1465-1471.

*Corresponding author: Eugenio Rustico,
Università di Catania, Dipartimento di Matematica e Informatica, Catania, Italy; email: rustico@dmi.unict.it.