ANNALS OF GEOPHYSICS, 54, 5, 2011; doi: 10.4401/ag-5341

Special Issue: V3-LAVA PROJECT

Porting and optimizing MAGFLOW on CUDA

Giuseppe Bilotta[1,2,*], Eugenio Rustico[1], Alexis Hérault[2,3], Annamaria Vicari[2], Giovanni Russo[1], 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:
GPGPU, Modeling, High-performance computing, Parallel computation, Hazard, Lava.

ABSTRACT
The MAGFLOW lava simulation model is a 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 simulation. The accuracy and adherence to reality achieved by the physics-based cell evolution of MAGFLOW comes at the cost of significant computational times for long-running simulations. The present study describes the efforts and results obtained by porting the original serial code to the parallel computational platforms offered by modern video cards, and in particular to the NVIDIA Compute Unified Device Architecture (CUDA). A number of optimization strategies that have been used to achieve optimal performance on a graphic processing units (GPU) are also discussed. The actual benefits of running on the GPU rather than the central processing unit depends on the extent and duration of the simulated event; for large, long-running simulations, the GPU can be 70-to-80-times faster, while for short-lived eruptions with a small extents the speed improvements obtained are 40-to-50 times.

1. Introduction

Modeling and simulation of lava flows is of extreme importance for short-term and long-term hazard assessment in volcanic areas. Physical models of lava flows must take into consideration the non-linear, temperature-dependent rheology, the variation in space and time of the rheological parameters, and the irregularity of natural topography. The more physically correct a model, the more computationally intensive it is, a condition that can hinder the applicability of the model to short-term forecasting of lava emplacement during an eruption, as it is essential for scenario simulations to be completed in very short times compared to the actual evolution of the phenomenon.

One of the most successful approaches to lava-flow modeling is cellular automaton [Crisci et al. 1986, Ishihara et al. 1990, Miyamoto and Sasaki 1997, Avolio et al. 2006], in which the computational domain is represented by a (usually regular) grid of two-dimensional or three dimensional cells, each of which is characterized by some properties, such as lava height and temperature, and where the modeling of the phenomenon is described through an evolution function for the properties of the cells.

The MAGFLOW model [Vicari et al. 2007] was developed at the Sezione di Catania of the Istituto Nazionale di Geofisica e Vulcanologia (INGV-Catania), and it is one of the few cellular automata that are based on physical modeling of lava flows, including thermal effects, to describe the system evolution. The MAGFLOW model has been used successfully both to reproduce past events with well-determined characteristics (for validation) and to predict Mount Etna lava flows in real time during the eruptions of 2004 [Del Negro et al. 2008], 2006 [Hérault et al. 2009, Vicari et al. 2009] and 2008 [Bonaccorso et al. 2011]. The use of this model in scenario forecasting has been possible because of its good performance, which allows simulation of several days of eruption in a few hours.

Although the original MAGFLOW implementation is written for serial execution on standard computer processors, the cellular automaton paradigm has a very high degree of parallelism that makes it particular suitable for implementation on parallel computing hardware. In particular, our choice has been geared towards the use of graphic processing units (GPUs), as these offer very high performance in parallel computing with a total cost of ownership that is significantly lower than that of traditional computing clusters of equal performance: a workstation with a modern GPU offers a computing capability in the order of teraflops (10^{12} floating-point operations per second), while it is priced at about €1,000 and consumes about 600 W.

We chose to implement the MAGFLOW model using Compute Unified Device Architecture (CUDA), an architecture that is provided by NVIDIA for the deployment of their latest generations of GPUs as high-performance computing hardware [NVIDIA 2008]. As CUDA provides a programming environment that is closely related to the C family of programming languages [NVIDIA 2010], this has allowed us to preserve much of the code and structure from the original serial MAGFLOW implementation, while providing a significant benefit in terms of computational speed, due to the parallel nature of the hardware.

As a preliminary to a better understanding of the details of the porting of MAGFLOW from C to CUDA (Section 5) and our discussion of the optimization strategies considered (Section 6), we will briefly present the fundamental concepts of GPU programming (Section 2) and a summary of the automaton structure and evolution functions (Section 3). A summary of the overall benefits are presented in Section 4, and the conclusions are presented last.

2. Hardware platform

The videogaming request for ever more realistic gameplay has driven the development of extremely high-performance, dedicated hardware for the real-time rendering of high-quality, realistic, animated, interactive three-dimensional scenes.

The computing power of GPUs has been initially exploited for non-graphic computing tasks (GPGPU: General-purpose Programming on GPU) by going through convoluted transformations that can express the numerical problems in graphical terms. The growing interest for the use of the GPUs as general purpose high-performance computing processors has led the two main manufacturers, ATI and NVIDIA, to develop interfaces that allow programming of their GPUs with a more conventional approach. For NVIDIA cards, this has resulted in CUDA, an extension of the C programming language, to support the GPU programming model. This has led to a significant breakthrough in performance in their use as high-performance computing devices [Berczik 2008].

GPU programming follows the stream processing concept, where a set of instructions (called a *kernel*) is executed in parallel over all of the items of a dataset. Each instance of a kernel is called a *thread*; threads are grouped in one-, two- or three-dimensional blocks that are themselves arranged in one- or two-dimensional grids.

The number of threads in a kernel launch is thus given by the number of threads per block times the number of blocks in the grid. As this is in general much larger than the number of computing cores available on the card, the GPU takes care of the dispatching of the threads to the multiprocessors as they become available. The threads from the same block are always assigned to the same multiprocessor, which will automatically distribute their execution over its cores. The smallest number of threads that can run physically in lockstep on a single multiprocessor is called the *warp size*, which is 32 for CUDA cards.

As the thread dispatching and concurrency is handled by the hardware, the programmer can focus on the implementation of the actual algorithm. This has allowed GPU usage for high-performance computing to flourish in a wide range of fields, from medical imaging [Roberts et al. 2010], to cryptography [Szerwinski and Gneysu 2008], and from statistical physics [Preis 2011], to computational fluid dynamics [Hérault et al. 2010, Kuznik et al. 2010].

The details of the thread distribution only become relevant during the optimization phase and in cases where threads have to communicate with each other to solve concurrency issues (separate threads trying to update the same datum). In these cases, the developer can query the driver at runtime for the hardware details of a specific card and tune the kernel parameters accordingly.

Different hardware generations (marked as the *compute capabilities* of the device) usually have a different number of processors, as well as a different number of cores per processor and different amounts of resources, such as registers and shared memory per processor. The number of concurrent threads that are executing at any given time depends on the number of computing cores that are available, and also on the complexity of the kernels and on their use of the resources available on the multiprocessors: the overall resource use by a batch of concurrent kernels is called the *occupancy*, which is an index of the overall efficiency with which the hardware is being used.

The occupancy, however, does not tell the whole story about the performance of an implementation, as there are a number of other factors that must be taken into consideration. These include algorithm features such as the presence of conditionals, or the nature and structure of the memory access patterns. Some of these issues will be discussed when presenting the GPU implementation of MAGFLOW. More information about the CUDA platform can be found in the CUDA programming guide of NVIDIA [2010].

3. Automaton structure

In this section, we present the key aspects of the automaton underlying the MAGFLOW model, while recalling only the aspects that are more relevant to the GPU implementation. A more detailed description of the model can be found in Vicari et al. [2007].

The MAGFLOW cellular automaton has a two-dimensional structure with cells described by five scalar quantities: ground elevation, lava thickness, heat quantity, temperature, and amount of solidified lava. The system evolution is purely local, in the sense that each cell evolves according to its present status and the status of its eight immediate neighbors (the Moore neighborhood).

The domain (automaton size) is chosen to be large enough to include the prospected maximum extent of the lava flow emplacement, and it is decomposed into square cells such that their width matches the resolution of the digital elevation model (DEM) that is available for the area, which is used to set the ground elevation property of each of the automaton cells.

The lava thickness varies according to the lava influx from the vent (for cells corresponding to a vent location), plus any lava flux between neighboring cells. Cross-cell lava flux is determined according to the height difference, using a steady-state solution for the one-dimensional Navier-Stokes equations for a fluid with Bingham rheology.

The rheological parameters are the yield strength S_y and the plastic viscosity η. The Bingham flow condition, which requires that the shear stress is greater than the yield strength, is modeled by introducing a critical height h_{cr} and having flux between two adjacent cells only when $h > h_{cr}$, with h being the lava height in the cell with the higher total height. When this condition is verified, the volumetric flux is given by:

$$q = \frac{S_y h_{cr}^2 \Delta x}{3\eta}\left(a^3 - (3/2)a^2 + 1/2\right) \qquad (1)$$

where Δx is the distance between adjacent cells, and $a = h/h_{cr}$. The Bingham flow condition ensures that $a > 1$ always when the formula is applied.

The critical thickness h_{cr} is computed from the yield strength and the slope angle, to account for both pressure-driven and gravity-driven flows, according to the formula:

$$h_{cr} = \frac{S_y}{\rho g(\sin\alpha - \partial h/\partial x \cos\alpha)} \simeq S_y \frac{\sqrt{\Delta z^2 + \Delta x^2}}{\rho g(\Delta z - \Delta h)} \qquad (2)$$

where ρ is the lava density, α is the slope angle of the inclined plane, g is the gravity acceleration, Δz is the overall height difference (considering ground elevation and lava height), and Δh is the difference in lava thickness.

Following Ishihara et al. [1990], the yield strength is computer according to the formula:

$$\log_{10} S_y = 13.00997 - 0.0089T \qquad (3)$$

and the viscosity follows Giordano and Dingwell [2003]:

$$\log_{10}\eta = -4.643 + \frac{5812.44 - 427.04 \times H_2O}{T - 499.31 + 28.74\ln(H_2O)} \qquad (4)$$

where T is the temperature in Kelvin, and H_2O is the water content in weight percent (wt%). MAGFLOW assumes a constant (average) water content across the flow, and its value is a user-controlled parameter that is typically in the range 0.02-0.2.

The actual amount of lava gained or lost by a cell at each iteration is given by the total flux Q of the cell multiplied by the timestep Δt for that iteration. To prevent non-physical solutions, the timestep is controlled by ensuring that for each cell $Q\Delta t < chA$, where h is the lava height in the cell and A is the cell area. The constant $0 < c < 1$ ensures that only a fraction of the total fluid lava volume is lost at each iteration, and this should be selected o be small enough to ensure that the stationary solution of the Navier-Stokes equation used to compute the flux remains approximately valid during the next step. The timestep used by the cellular automaton is then the minimum of the Δt computed by each cell.

3.1. Computational pipeline

An iteration of the cellular automaton can be divided into the following steps, each of which can be executed independently by each cell: (i) compute the eruption flux if the cell is a vent cell; (ii) compute the fluxes with the neighboring cells; (iii) compute the maximum allowed timestep; and (iv) update the cell status.

The cell-status update itself consists of three steps: (a) compute the new lava thickness; (b) compute the heat radiation loss; and (c) transfer an appropriate amount of lava thickness to the solid lava thickness if there is solidification.

There are a few important differences between serial and parallel implementation of such an algorithm. When computing the fluxes between neighboring cells, serial execution can compute the flux between each pair of cells a single time, and then add the result (with opposite signs) to the total flux of each cell involved. On the other hand, a parallel implementation benefits more from totally independent cell evolutions: it is therefore more efficient to let each cell compute all of its fluxes, even though this means that fluxes will be computed twice.

Similarly, during the serial execution, the maximum allowed time-step is updated by each cell at the end of the flux computation loop, while a parallel implementation would store the maximum time-step of each cell in an array, and then use parallel reduction to determine its minimum.

In both cases, the simulation time is updated at the end of the iteration, and a new iteration begins if the simulation time has not passed the given end time.

4. Preliminary results

4.1. A test case: the 2001 Mount Etna eruption

To test the performance of our CUDA implementation, we compare here the timing for three different NVIDIA GPUs *versus* the CPU timing. All of these simulations were run on the same machine, which has an Intel Core2 Quad clocked at 2.83 GHz as well as all three of the GPUs specified in Table 1. The operating system used for the tests is Ubuntu 10.04, with the CUDA tool-kit version 3.2.

To compare the computational performance of the CPU and GPU implementations, we look at the average number of evolutions per second simulated by each hardware choice, and then compute the speed improvements obtained by the CPU over the GPU.

The average number of evolutions per second (obtained by dividing the total number of evolutions at the end of the

	CPU	9500 GT	GTX 280	GTX 480
CUDA compute capability	n.a.	1.1	1.3	2.0
Clock rate (GHz)	2.83	1.35	1.30	1.40
CUDA cores	n.a.	32 (4 × 8)	240 (30 × 8)	480 (15 × 32)

Table 1. Specifications of the hardware used to run the test simulations.

simulation by the seconds spent in the simulation functions) gives us better grounds for comparison than the simple runtime, because due to hardware differences, the actual number of evolutions needed to complete the simulation differ on the different hardware platforms, as further discussed in Section 5.3 (Accuracy).

In applications such as scenario forecasting, the total runtime of a simulation, i.e. the time from program launch to production of the results, is an important factor to consider, and possibly even more so than the bare computational speed improvements that we have considered. The total runtime accounts for janitorial and maintenance tasks, such as loading the initial data and saving the final results, in addition to the time spent in the actual simulation. It is therefore important to consider that the GPU code must perform some additional tasks with respect to the CPU implementation; namely, the uploading of the initial data to the GPU, the retrieval of the maximum allowed Δt at each time step, and finally the download of the final results from the GPU. However, the total cost of these additional operations is barely significant, and they only take a fraction of the overall simulation time: the overall performance of the GPU implementations closely follows the computational performance in terms of the overall speed improvements, as shown by the results presented in the next section.

We will present and discuss the timing results from two simulations, both based on the data available for the 2001 Mount Etna eruption, and which differ only in the size of the cells and the cell numbers, due to their different grid resolutions (DEMs with 10-m and 5-m cell resolutions, respectively).

4.2. Performance comparison

By considering only the raw computational power of the CPU and the GPUs, and therefore factors such as the clock rates, the number of cores, and finally that on the GPU each flux is computer twice (once per cell, instead of once per pair of cells), we can expect a maximum speed improvement of about 8 times when using the 9500 GT. Similarly, the GTX 280 can be expected to be 6-to-8-times faster than the 9500 GT (and thus about 50 times faster than the CPU), and the GTX 480 more than twice as fast as the GTX 280 (and therefore about two orders of magnitude faster than the CPU).

The actual performance numbers are presented in Tables 2 and 3. We also show the average runtimes over three simulations for each hardware choice, with the offsets for the slowest and fastest execution times (Table 4). The results show both the accuracy of the timing throughout the study, and how the runtimes essentially follow the qualitative benefits of the computational speed improvements.

The highlights of Tables 2 and 3 are the low performance of the oldest hardware generation, which comes out as 3-to-4-times slower than the forecast according to its raw computing power; only minor differences in the speed improvements are seen between the 10-m and 5-m cells. The more modern hardware, on the other hand, comes closer to the expected results for the 5-m cells, but falls short by about 50% with the 10-m cells.

These phenomena can be explained by looking at some of the factors that can have a significant (negative) impacts on the performance of a GPU. The first, and probably the most important factor, is given by the very high latency of the GPU global memory: 800 computing cycles are needed to access a single value in the global memory, with the threads stalling in a wait queue until the datum becomes available. The GPU can compensate for these long wait times by coalescing memory accesses and by running some threads while others are stalled.

Coalescing happens when a single memory access is carried out to fetch or write multiple values. This typically happens when threads with consecutive global indices in the launch grid access values that are contiguous in correctly aligned memory; the actual details about when memory accesses are coalesced depend on the hardware, with that of the more recent generations having improved coalescing capabilities than the older hardware.

The thread dispatcher built into the GPU will also take care of the execution of threads that are ready to run while the others are stalled waiting for data: this will allow the

	CPU	9500 GT	GTX 280	GTX 480
10 m DEM cell	86.2	127.8	2501.6	4415
5 m DEM cell	17.1	31	833.5	1685

Table 2. Average number of automaton evolutions per second.

	CPU	9500 GT	GTX 280	GTX 480
10 m DEM cell	1	1.5	29	51.2
5 m DEM cell	1	1.8	48.7	98.5

Table 3. Speed improvement factors over the CPU.

	CPU	9500 GT	GTX 280	GTX 480
10 m DEM cell	6:07:21 ± 4s	4:05:26 ± 3s	13:58 ± 2s	8:29 ± 2s
5 m DEM cell	89:35:59 ± 35s	43:04:26 ± 21s	1:39:31 ± 2s	52:46 ± 1s

Table 4. Total runtime (h:m:s).

latency of subsequent accesses to be covered by the execution of the other threads, as long as there are enough threads ready to be executed.

When all of the GPU multicores are busy running threads, the GPU is said to be saturated. Saturation typically needs tens or hundreds of thousands of threads, depending on the complexity of the kernels, and it is also a very important factor in achieving optimal GPU performance. A rule of thumb to detect saturation is a linear increase in the execution time with the size of a problem. Indeed, Table 2 shows such a trend for the 9500 GT, but not for the GTX cards.

A third aspect that can have an impact on the performance is given by the number of divergences in the code. Divergences happen when threads in the same block run different branches in conditional code paths. When such a situation occurs, thread execution is serialized by the GPU, losing the benefit of the parallel execution, which is a crucial aspect of the GPU performance.

In the MAGFLOW case, the first phase of an eruption has conditions that are typically not favorable for the GPU, as there are only a very few active cells, and a number of these are either neighbors to the vents or are boundary cells for the lava flow, both of which represent situations that increase the number of divergences. When the number of such cells is dominant, the GPU will suffer significant performance loss, resulting in little benefit over the CPU execution.

The GPU works closer to peak performance when the number of cells covered by the main body flow grows to be significant. This usually happens towards the end of the simulation, if the lava flow grows to cover a large area, or if the resolution of the DEM is not too coarse.

5. Porting details

5.1. Memory structures

Most of the automaton data is stored in the GPU global memory. To allow easier coalescing, and thus to speed up memory access, we follow the best practice recommended by NVIDIA: the use of "structures of arrays" rather than "arrays of structures". While the original serial implementation of MAGFLOW defined a cell structure holding the data for that cell, and the automaton was modeled as a list of cells, the CUDA version of MAGFLOW defines six arrays with the CPU and 10 arrays with the GPU. These six CPU arrays are ground elevation, vent index (which is 0 if the cell has no vent, or the vent number otherwise), lava height, solid lava height, heat quantity, and temperature.

Each of these arrays has the number of elements equal to the number of cells in the automaton. With the GPU, there are six arrays that mirror those of the CPU, plus four additional arrays that hold lava fluxes, heat fluxes, the maximum time-step, and a marker to denote whether the cell has an erupting vent or not. Again, these arrays have as many elements as there are cells in the automaton.

The only CPU arrays that get initialized and uploaded to the GPU are the ground elevation array, where the data are read from a specified DEM file in ASCII grid format, and the vent index array. The other arrays are only used during checkpoints, to download the data from the GPU and to save the results on file. They are also used when loading a previous checkpoint to continue a simulation that was interrupted abruptly.

5.2. CUDA kernels

In MAGFLOW, the CUDA computational kernels can be distributed in a natural way by having one thread per cell in the automaton. The iteration structure also suggests a natural way to split the algorithm into computational kernels:

Erupt, which is executed only on vent cells and which computes the effusion rate at the given time;

CalcFlux, which is executed on each active cell and which computes the ingoing and outgoing lava fluxes with respect to all 8 neighbors, and the maximum timestep for that cell;

Reduction, from the CUDPP library [Sengupta et al. 2008], which operates a parallel reduction to determine the maximum allowed timestep;

UpdateCells, which is executed on each active cell to update its status, and thus to complete the iteration.

The instant timings for each kernel, i.e. the actual kernel runtime at each iteration, are plotted in Figure 1.

The instant timings show that the runtime for CalcFlux, the timestep reduction and UpdateCells grows with the number of active blocks. The Erupt kernel, on the other hand, has an overall constant runtime, which is one-to-three orders of magnitude lower than the runtime of each of the other kernels. A small decrease in the execution time is seen in the Erupt kernel due to the actual eruption ending a few days before the end of the simulation.

On the GTX 280 and GTX 480, the runtime of CalcFlux has a slightly negative slope after the maximum number of blocks is reached. This is due to the increasing homogeneity of the status of the active cells as the eruption progresses, and the consequent decrease in the number of divergences for the conditional kernel code.

The differences in the runtimes between Erupt and the other kernels is particularly visible for the oldest hardware, while with the newer hardware, CalcFlux and UpdateCells gain up to an order of magnitude in execution speed. The same is not observed for Erupt, which is easily explained by noting that CalcFlux and UpdateCells are highly parallel, as they are distributed across all of the active cells, while Erupt is limited to the very small number of vent cells. Therefore, the Erupt kernel does not scale as well as the others with the improvements offered by the newer hardware. On the GTX 480, the Erupt runtime is of the same order of magnitude

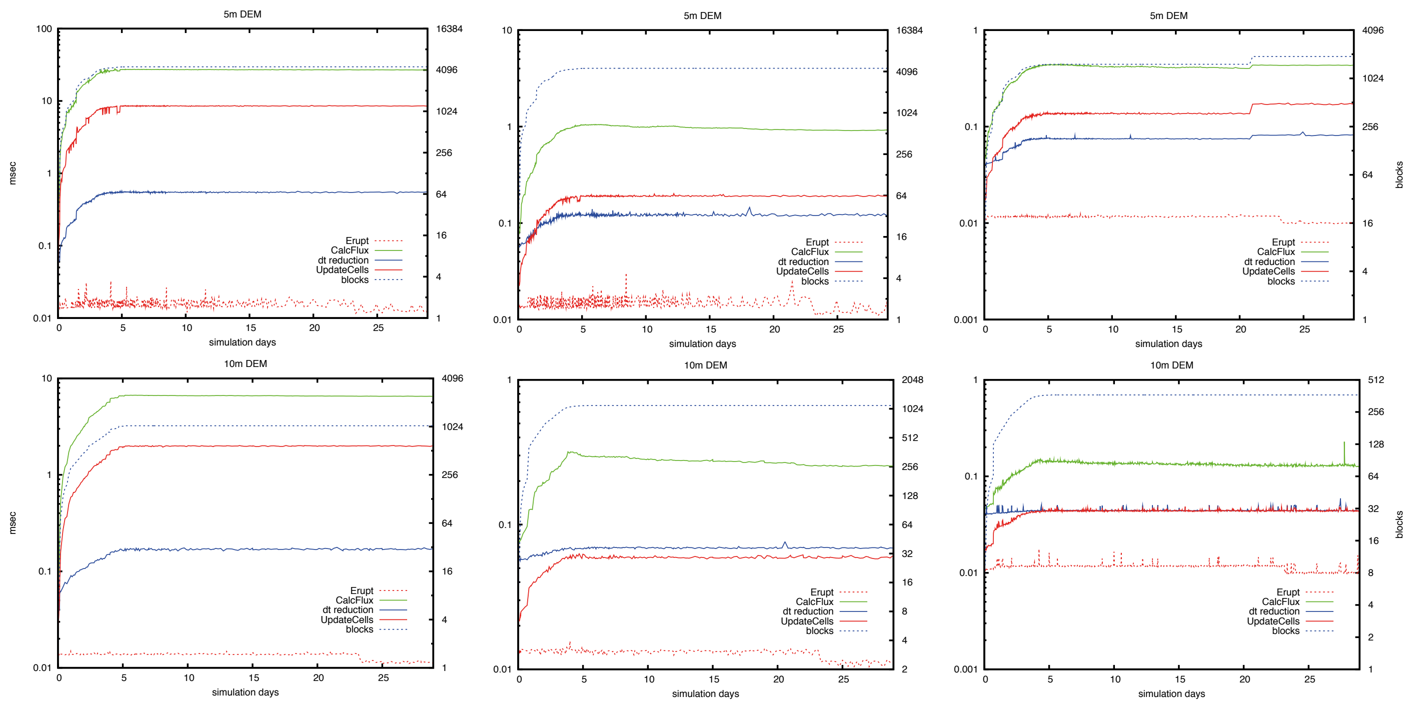

Figure 1. Instant timings for each kernel on the 9500 GT (left), GTX 280 (center) and GTX 480 (right). The kernel timing is on a base-10 log axis, and the block sizes are on a base-2 log axis.

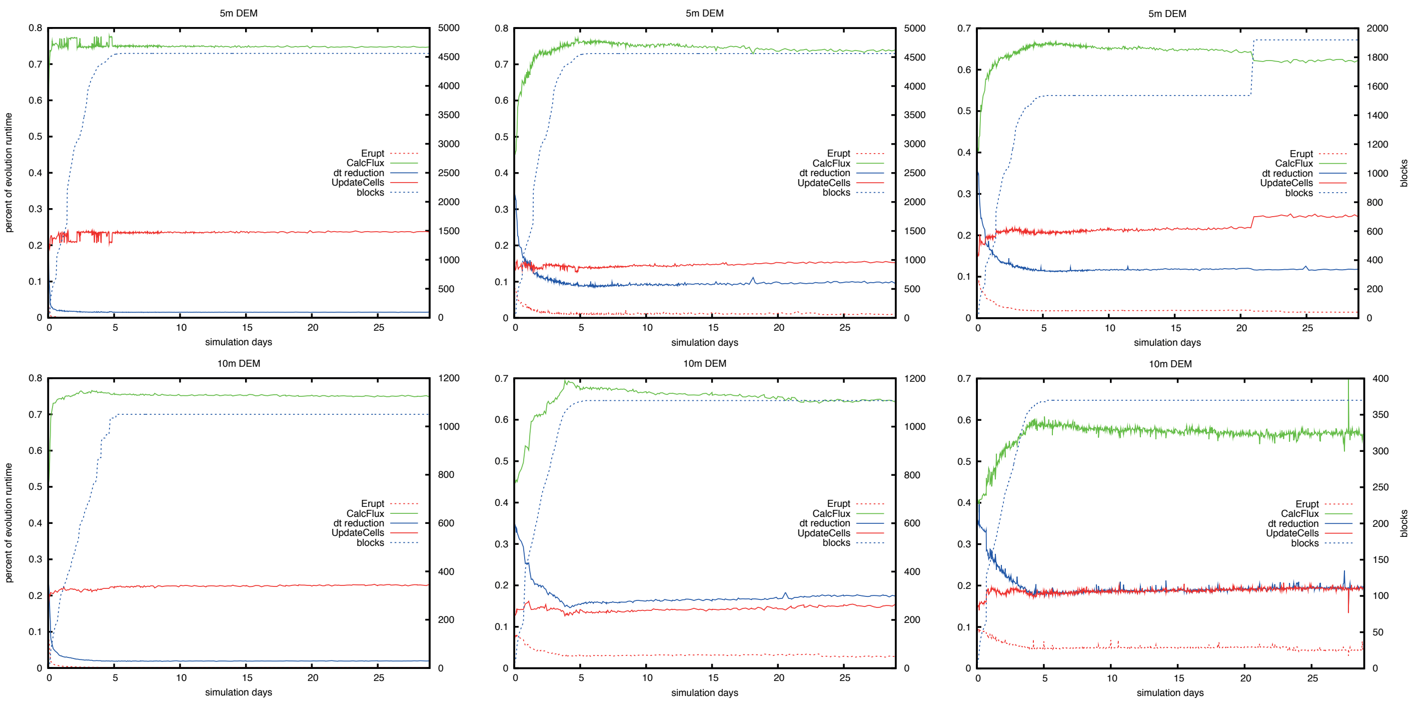

Figure 2. Percent timing for each kernel on the 9500 GT (left), GTX 280 (center) and GTX 480 (right).

(10^{-2} ms) as the hard lower boundary imposed by the kernel launch time.

The same factors also influence the distribution of the runtime percentages at each iteration, as plotted in Figure 2. The percentages are obtained by dividing the runtime of the single kernel by the sum of all of the kernel runtimes at the given iteration. These clearly show that CalcFlux takes more than half of the total iteration runtime, as expected by its high computational density, but with a decreasing weight (from over 75% on the 9500 GT down to 65% or less on the GTX 480) as the hardware improves and the kernel runtimes grow smaller.

5.3. Accuracy

For historical reasons, the first generation of NVIDIA GPUs was strongly optimized for floating-point operations in single precision; the later hardware revisions introduced support for double precision, which was about eight-times slower than single precision. Although the support for double precision has greatly improved with the latest generation of NVIDIA GPUs (known as Fermi), our CUDA implementation of MAGFLOW uses single precision to support a wider range of hardware.

The existing CPU implementation of MAGFLOW can be used with either single or double precision, which is important both for performance comparison with the GPU implementation (in our benchmarks here, we used single precision on both the CPU and the GPU), and to help determine the numerical robustness of the algorithm. Indeed, the lack of associativity of the floating-point operations (on both CPUs and GPUs) can produce slight differences in the computed fluxes, which can depend, for example, on the order in which the neighborhood is transversed (clockwise vs counterclockwise, which neighbor the cycle is started from, etc.)

The differences in the flux computation can accumulate, although even on long-running simulations (a month of simulated time), the number of cells invaded by lava and the cell status components (height, heat, solid height, temperature) remain well within 1% discrepancy. The algorithm itself is therefore particularly stable numerically. Further details, with a more complete analysis of the sensitivity of the model can be found in Bilotta et al. [submitted 2011].

We elected to further improve the accuracy of the GPU code by using Kahan summation [Kahan 1965] during the flux computation. This incurs a performance penalty of less than 5%, due to the handful of extra summations that need to be computed, but it stabilizes the results with respect to the neighbor access patterns.

Kahan summation was also considered for the evolution of each of the cell variables, but the idea was discarded due to its inefficiency. Using Kahan summation for the cell evolution requires one additional array for each cell status variable (height, heat, solid height, temperature), to store the Kahan reminders, and this results in the need for twice the access to the global memory during the UpdateCells kernel. The overall effect on the performance is between 25% and 30%, which is not justified for differences in the results of less than 1%.

6. Optimization

6.1. Register use

The effects of latency hiding that are provided by having more blocks per multiprocessor can be seen by studying the effects of register use.

For example, the CalcFlux kernel with Kahan summation compiles by default for the use of 33 registers, which results in a limit of six blocks per multiprocessor on the GTX 280; forcing the compiler to use no more than 32 registers, thus allowing eight concurrent blocks, results in a 5% speed improvement for the kernel. On the 9500 GT, however, a reduction in the number of registers (thus bringing the concurrent blocks from three to four) provides no significant benefits.

6.2. Active cells and block structure

At each iteration, only the cells that either have lava or have a neighbor with lava can have non-zero thermal or mass flux, and therefore can evolve into a different status at the end of an iteration. In the CPU implementation of MAGFLOW, the number of cells for which fluxes were computed was therefore minimized by the tracking of a list of active cells, which was updated at the end of each iteration when a previously empty cell had a positive lava flux.

A similar approach with the GPU would result in scattered memory reads and very low coalescing, with a drastic reduction in performance. To favor the regular access patterns that increase coalescing and improve the memory-access performance, our choice was to compute at each iteration the smallest rectangular bounding box that covered any cells with fluid or solid lava, plus one additional row/ column on each of the four sides of the rectangle.

With the appropriate memory and block layout (see also Sections 6.3 and 6.6) the two-dimensional structure of the blocks and the grid allows fast memory access, which on the Fermi architecture provides good exploitation of the cache. The bounding box also ensures that all of the active cells (in the sense used for the CPU implementation) are included in the computation, although it also includes a number of additional cells for which fluxes do not actually need to be computed. Figure 3 illustrate a fictitious case that represents

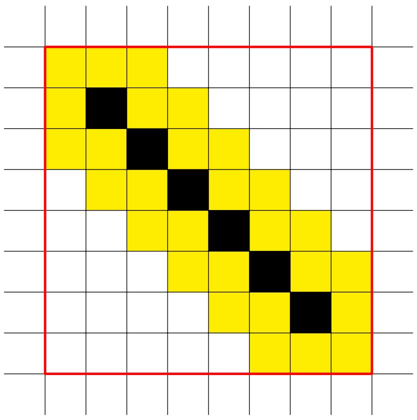
Figure 3. Bounding box inefficiency. Black squares mark cells with lava, yellow squares mark active cells without lava, white squares (inside the red frame) indicate cells for which computations are run but which are known to have zero flux.

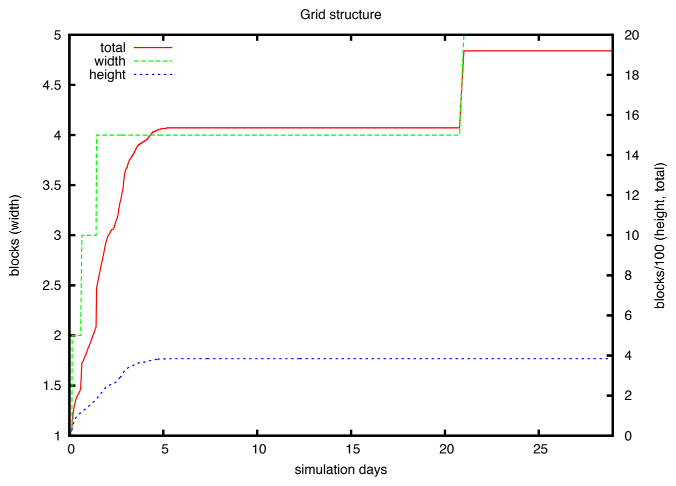

Figure 4. Width and height of the computational grid, and resulting number of blocks, during the 5-m simulation with the GTX 480. The left scale tracks the width of the grid in blocks, the right scale tracks the height of the grid and the total number of blocks, in hundreds of blocks. The large step around day 20 is caused by the width of the grid increasing from 4 to 5, when the height of the grid is already at 384 blocks.

the worst possible situation, in which the lava flow has a linear emplacement which is at 45° with respect to the automaton orientation.

An interesting effect of the bounding box covering is that especially during the latter phases of the simulation, a single extra active cell in one direction can cause the number of cells for which computations are run to jump suddenly by a significant factor.

This phenomenon can be seen in Figure 1, when comparing the behavior of the GTX 480 (right) with that of the GTX 280 (center) for the 5-m DEM resolution, around about day 20. The detailed plot of the grid structure with the distinct width and height increases (Figure 4) clearly shows the anomalous step caused by the 384 × 4 block grid expanding to a 384 × 5 grid.

6.3. Optimal thread and memory layout

The optimal block structure for kernel execution depends on the compute capability of the hardware, due to the following differences between the hardware generations: the number of cores and registers and the amount of shared memory available for each multiprocessor, the improved memory coalescing available in devices with a compute capability 1.2 or higher, and the availability of L1 and L2 cache memory in devices with a compute capability of 2.0 or higher.

The thread layout can be tuned both in terms of the number of threads per block and with respect to the two-dimensional structure of the block. For example, blocks with 64 threads can be distributed as 64 threads in a row (1 × 64), 64 threads in a column (64 × 1), a square block with 8 × 8 threads, or any other combination. The choice can have a drastic impact on performance, as it contributes to the improvement of coalescing of memory access.

In particular, it is important that the thread block structure closely follows the memory structure. The linear layout of the two-dimensional array storing cell data in global memory can be either row-major, with the data in the same row that occupies consecutive column-ordered memory locations, or column-major, with the data in the same column that occupies consecutive row-ordered memory locations.

With row-major global arrays, a block structure of 1 × 64 is optimal because threads in the same block will access consecutive memory locations, which improves coalescing. Conversely, with column-major data disposition, a block structure of 64 × 1 will provide the best performance. The results for row-major data ordering are shown in Table 5, with the average number of evolutions per second correlated to the block structure, assuming a constant block size of 64 threads.

We can see that in the case of the 9500 GT, the oldest device, the block structure does not have any significant influence (10% variation). In the case of the more recent hardware, however, the correct block structure can provide up to a five-times boost in performance. This can be explained by the very small number of concurrent blocks that can execute on a multiprocessor in the 9500 GT and to its reduced coalescing capabilities, which means that the global memory access latency is the dominant slowing down factor. In the case of the GTX 280, instead, the improved coalescing and the higher number of blocks that can be dispatched to a single multiprocessor result in much more reduced memory latencies in the case of the optimal block structure.

These effects are not as obvious in the case of the GTX 480 due to the L1 cache that is provided by its Fermi architecture, which helps further to reduce the memory latency. Indeed, if we let the block size grow with the GTX 480, we can see how the performance improves until the number of blocks per multiprocessor drops below the level at which the cache system can compensate for this (Table 6). In this case, a more two-dimensional block structure helps to improve the performance, as the L1 cache becomes more effective.

Block size (rows × cols)	9500 GT	GTX 280	GTX 480
1 × 64	114	2501	3449
8 × 8	127	1347	3345
64 × 1	120	553	869

Table 5. Influence of block structure on the average number of evolutions per second, with 64 threads per block.

	1× 64	1 × 128	1 × 192	2 × 64	3 × 64	4 × 64
GTX 480	3449	3634	3151	4285	4326	4210
GTX 280	2501	2150	n.a.	2299	n.a.	n.a.

Table 6. Influence of increased block size on the GTX 480 and GTX 280.

Table 6 shows that the GTX 480 reaches its peak performance with 192 threads per blocks, with a 3 × 64 structure, while blocks with more than 64 threads are counter productive on the older hardware generations.

6.4. *Texture* versus *global memory*

Graphic cards provide a special memory area, called the texture memory, that is optimized for two-dimensional access and that provides caching even on the older hardware generations. Although the two-dimensional structure of the cellular automaton suggests that this feature can provide some benefit, experimentation shows that the reads and writes of the coalesced global memory obtained with the highly regular access patterns of appropriate thread/ block structuring leads to a higher data throughput than for the noncoalesced but cached texture memory. Indeed, the loss of coalescing due to the use of the texture memory can reduce the implementation performance by two or three times.

6.5. *Shared memory* versus *multiple data reads*

With a fully parallelized implementation, each cell computes the interaction with all of its neighboring cells, with the result that each interaction is computed twice. As flux computations depend on costly operations, such as the exponential, it is possible to halve the number of operations by computing each interaction once and then updating both of the cells involved.

To achieve this, each block must load into the shared memory the height, total solid height, and temperature of the whole block, plus a ring of neighboring cells; this is achieved according to the loading pattern illustrated in Figure 5. The shared memory also holds the lava and heat flux for each cell in the block, before it is stored back in the global memory.

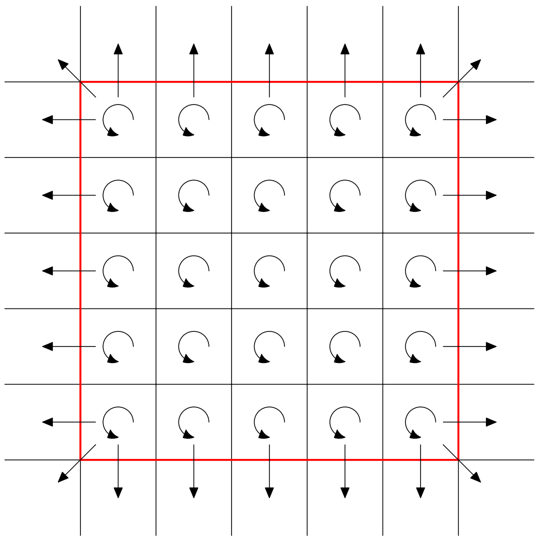
Figure 5. Load pattern for shared memory use. Each block cell (inside the red frame) loads its own data, and the boundary cells also load data from the outer ring of their neighboring cells.

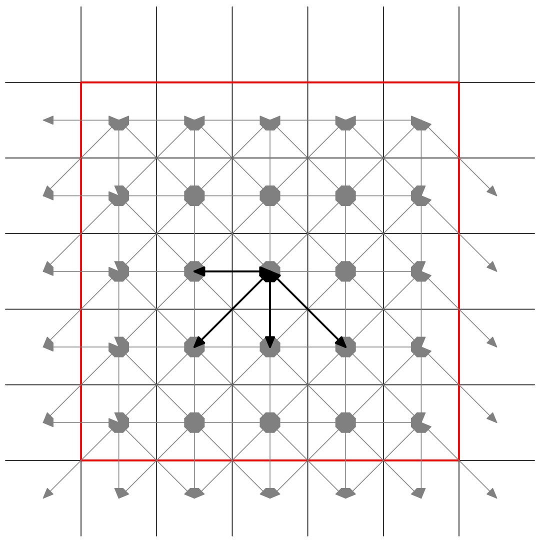
Figure 6. Flux computation pattern. When all of the cells use the same pattern (highlighted in the central cell), each pair of cells interacts only once, although the cells in the outer rim are missing some neighbors, which requires additional interactions to be computed.

Each cell then computes the interaction with its left, bottom left, bottom and bottom right neighbors, while updating both its own flux and that of its neighbor. Thread synchronization is used between the updates, to ensure that all of the writes occur correctly. The leftmost, rightmost and topmost cells also compute the interactions with the outer ring that are not computed in the common loop (Figure 6).

In this case, a block structure with a square shape provides the best benefit, as the amount of shared information is higher. However, the benefits of this approach depend on the compute capability of the device: the enormous increase in shared memory use results in a proportional reduction in the concurrent block execution. Additionally, the different behavior of the extremal cells in each block causes divergences at every execution, both during the initial phase of the kernel execution (data loading), and during the computational phase.

While a net benefit of 35% has been measured for the older, memory-bound hardware, such as the 9500 GT, due to the reduced computation, the much smaller number of concurrent blocks results in an equal loss in kernel performance for the GTX 280, as the reduced number of computations are not sufficient to offset the reduced number of concurrent threads. Given the obsolescence of the older hardware, the structure without shared memory is preferred. The L1 caching introduced in the Fermi architecture (GTX 480) provides an automatic way to reduce memory access, which is more efficient than manual shared memory use.

6.6. The Fermi L1 cache

With the new Fermi architecture, NVIDIA introduced two caching levels for global memory access (L1 and L2). The appropriate use of this capability can bring a substantial benefit to the execution of kernels that require repeated access to a small number of global memory values from multiple threads in the same block. This is indeed the case for our flux computation kernels, where all of the neighboring cells have to access each other's height, lava height, solid lava height and temperature.

As much of the hardware for the L1 cache is common with the shared memory hardware, CUDA provides an interface that allows the programmer to choose whether shared memory is preferred (in which case 48 kb will be used for the shared memory, and 16 kb will be used for the L1 cache), or whether the L1 cache is preferred (in which case 16 kb will be used for shared memory, and 48 kb will be used for the L1 cache). A third option (which is the default) informs CUDA that the kernel should use whatever setting is currently active, which is typically to prefer shared memory, unless the programmer has set the preference as the L1 cache for a previous kernel.

As in our case shared memory does not bring any particular benefit for the more recent GPUs, we can set CUDA to prefer L1 caching to shared memory without affecting the number of kernels being executed.

The performance comparison (Table 7) shows a consistent gain of about 2% that derives from the preference for L1. We can get deeper insight into how the cache affects the performance by making use of the visual profiler provided in the latest versions of CUDA, and by analyzing the impact that the cache settings have on the cache hit rate (Figure 7) and on the average kernel runtime (Figure 8).

	1 × 64	2 × 64	3 × 64	4× 64
Prefer shared	3449	4285	4326	4210
Prefer L1	3543	4337	4415	4328

Table 7. Average number of evolutions per second on the GTX 480 that depend on the shared *versus* L1 preference setting for kernel launch.

We observe that the effects on the two kernels of the cache preference and the block structure are quite different, as expected. As CalcFlux needs to access the data for each cell multiple times, the cache hit rate without any specific optimization is already relatively high (>50%), and is limited by the actual cache size, as expressed by the increase in the cache hit rate with the preference for L1 caching.

In contrast, the UpdateCells kernel only accesses data for a single cell in each thread, and therefore it can exploit the cache at a much lower level; this is indicated by the lack of significant increase in the cache hits when L1 is preferred, for the default 1 × 64 block structure. As the block structure becomes more rectangular with the increase in the first dimension, the Fermi hardware can make better use of the cache lines, and the cache size becomes an increasingly limiting factor.

The average kernel runtime provides a comparison with the older CUDA architecture, thereby indirectly showing the influence the new L1 cache can have on performance. In Figure 8 there is a growth in the average of the GTX 280 kernel runtimes, as expected by the increased block size and the diminishing number of blocks per multiprocessor.

In contrast, for the GTX 480, the CalcFlux kernel shows a constant decline in the average runtime with the increase in the blocksize, which can be explained by the cache efficiency and the high number of memory accesses required by the kernel. In the UpdateCells case, on the other hand, we again see a slight increase in the average runtime, due to the very small number of memory accesses and the consequent minor

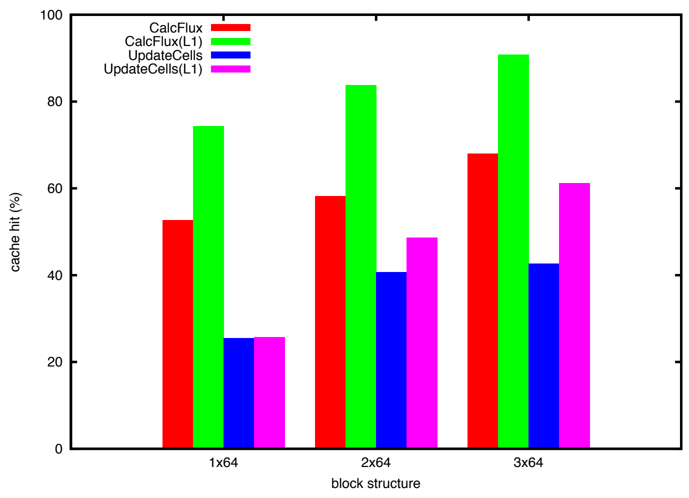

Figure 7. Cache hit rate for the different block structures and the L1 cache preferences.

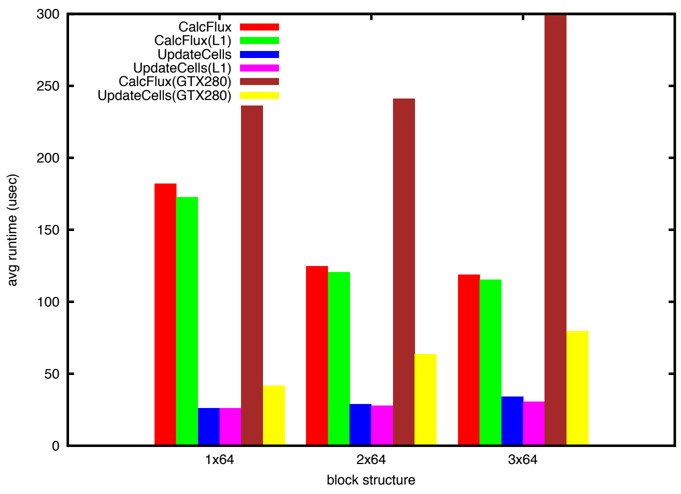

Figure 8. Average running time for each kernel for the different block structures and L1 cache preferences.

impact of the caching. The L1 influence can, however, still be seen by considering that the GTX 280 suffered almost a 100% increase in average runtime with a triple block size, while for the GTX 480 this effect was limited to a 30% increase.

6.7. *CPU* versus *GPU vents*

The eruption phase cannot really exploit the GPU parallel computation capabilities. However, it is more efficient to run the kernel on the GPU directly rather than running it on the CPU and then uploading the new data to the GPU at every iteration, because of the high host-to-device latency.

6.8. *Vector types* versus *separate memory array*

The cellular automaton structure needs four floats to describe each cell status, plus two floats to describe mass and heat flux from cell to cell. This suggests that using the native float4 and float2 vector types might result in more efficient code and better memory accesses.

As with the shared memory, however, the actual benefit is only present for the lower-end hardware (9500 GT), where the bottleneck is caused by the memory access, and the lower number of blocks limits the number of blocks running concurrently: switching from arrays of floats to arrays of vector types in this case results in a gain in performance of up to two times.

For the GTX 280, however, the same change results in a 10% loss in performance, due to the higher register use of the resulting kernel. This again can be mostly recovered by forcing the compiler to use no more than 32 registers for the kernel, which limits the performance loss to 1%; this can be attributed to the slightly inferior bandwidth that vector types have over scalars.

7. Conclusions

The MAGFLOW cellular automaton has an intrinsically highly parallel structure that can substantially exploit the high-performance parallel hardware of modern GPUs. A straightforward conversion of the serial CPU code leads to only minor improvements in the execution speed on the older GPUs, while a speed improvement of almost 50 times can be obtained with the last generation of CUDA cards during the simulation of lava flows from typical Mount Etna eruptions. Further benefits are harder to achieve for typical eruptions due to the inability for these events to saturate the hardware, although long-running eruptions or lava flows spanning a wider area can provide speed improvements of almost 100 times.

We believe that the optimization strategies presented in this study can also be useful for other CUDA applications that are based on similar mathematical structures and algorithms. In particular, the same computational approach can be extended to include other hazardous natural phenomena where the physical modeling is close to that of lava flow, with the primary examples being land slides and debris flow.

An optimization that has not been implemented yet is the possibility to use different block sizes for the CalcFlux and UpdateCells kernels. The profiling conducted on the GTX 480 card appears to indicated that a block size of 1×64 is optimal for UpdateCells, while 3×64 is optimal for CalcFlux; however, UpdateCells only takes 10% of the total evolution time, and the consequent speed improvement of 30% on its runtime would result in a net impact of less than 1% over the whole simulation runtime; this is actually within the standard runtime fluctuations encountered during our tests.

Other optimizations might also be possible, in particular concerning the bounding box strategy used to determine the kernel launch grid: the current strategy can launch a significant number of kernels on cell which are far from the lava flow and would therefore not need flux calculations nor updating. A more sophisticated strategy can be used to map execution blocks only on the minimal number of cells surrounding the lava flow that can be covered, while still retaining the efficient memory access patterns that ensure coalescence (and high L1 cache use on the GTX 480). Experimentation is needed to determine whether the complexity of such an irregular grid mapping would be balanced out by the fewer blocks needed to cover the active area of the automaton.

Even without this additional optimization, the increase in simulation speed is already sufficient to provide a 7-day forecast in a couple of minutes, and a month forecast in 10 to 15 min. This thus matches, or indeed surpasses, the speed of more simplistic lava-flow simulation models, while still retaining the reliability of the physics-based evolution.

Acknowledgements. This study was carried out with financial support from the V3-LAVA project (INGV-DPC 2007-2009 contract).

References

Avolio, M.V., G.M. Crisci, S.D. Gregorio, R. Rongo, W. Spataro and G.A. Trunfio (2006). SCIARA g2: An improved cellular automata model for lava flows and applications to the 2002 Etnean crisis, Comput. Geosci.-UK, 32 (7), 876-889.

Berczik, P. (2008). SPH Astrophysics – "State-of-the-art", In: SPHERIC 2008 Keynotes; URL: http://cfd.mace.manchester.ac.uk/sph/meetings/3rdSPHERIC_workshop/Berczik_keynote_SPHERIC_2008.pdf (last visited: November 09, 2011).

Bilotta, G., A. Cappello, A. Hérault, A. Vicari, G. Russo and C. Del Negro. Sensitivity analysis of the MAGFLOW Cellular Automaton model, (submitted 2011, *sub judice*).

Bonaccorso, A., A. Bonforte, S. Calvari, C. Del Negro, G. Di Grazia, G. Ganci, M. Neri, A. Vicari and E. Boschi

(2011). The initial phases of the 2008-2009 Mount Etna eruption: A multidisciplinary approach for hazard assessment, J. Geophys. Res., 116, 1-19.

Crisci, G.M., S. Di Gregorio, O. Pindaro and G.A. Ranieri (1986). Lava flow simulation by a discrete cellular model: first implementation, International Journal of Modelling and Simulation, 6 (4), 137-140.

Del Negro, C., L. Fortuna, A. Hérault and A. Vicari (2008). Simulations of the 2004 lava flow at Etna volcano using the magflow cellular automata model, B. Volcanol., 70 (7), 805-812.

Giordano, D. and D.B. Dingwell (2003). Viscosity of hydrous Etna basalt: implications for Plinian-style basaltic eruptions, B. Volcanol., 65 (1), 8-14.

Hérault, A., A. Vicari, A. Ciraudo and C. Del Negro (2009). Forecasting lava flow hazards during the 2006 Etna eruption: Using the MAGFLOW cellular automata model, Comput. Geosci.-UK, 35 (5), 1050-1060; doi: 10.1016/j.cageo. 2007.10.008.

Hérault, A., G. Bilotta and R.A. Dalrymple (2010). SPH on GPU with CUDA, J. Hydraul. Res., 48 (extra issue), 74-79.

Ishihara, K., M. Iguchi and K. Kamo (1990). Numerical simulation of lava flows on some volcanoes in Japan, In: Lava Flows and Domes: Emplacement Mechanisms and Hazard Implications (IAVCEI Proceedings in Volcanology), edited by J.H. Fink, Springer-Verlag, 174-207.

Kahan, W. (1965). Pracniques: further remarks on reducing truncation errors, Communications of the ACM, 8 (1), 40-41.

Kuznik, F., C. Obrecht, G. Rusaouen and J.J. Roux (2010). Lbm based flow simulation using gpu computing processor, Comput. Math. Appl., 59, 2380-2392.

Miyamoto, H. and S. Sasaki (1997). Simulating lava flows by an improved cellular automata method, Comput. Geosci.-UK, 23 (3), 283-292.

NVIDIA (2008). CUDA Zone. URL: http://www.nvidia.com/object/cuda_home.html (last visited: November 09, 2011).

NVIDIA (2010). NVIDIA CUDA C Programming Guide, version 3.2.

Preis, T. (2011). Gpu-computing in econophysics and statistical physics, Eur. Phys. J.-Spec. Top., 194, 87-119.

Roberts, M., J. Packer, M.C. Sousa and J.R. Mitchell (2010). A work-efficient gpu algorithm for level set segmentation, In: Proceedings of the Conference on High Performance Graphics, HPG '10, Eurographics Association, Aire-la-Ville, Switzerland, 123-132.

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

Szerwinski, R. and T. Gneysu (2008). Exploiting the power of gpus for asymmetric cryptography, in Cryptographic Hardware and Embedded Systems CHES 2008, edited by E. Oswald and P. Rohatgi, vol. 5154 of Lecture Notes in Computer Science, Springer, Berlin/Heidelberg, 79-99.

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 (10), 1465-1471.

Vicari, A., A. Ciraudo, C. Del Negro, A. Hérault and L. Fortuna (2009). Lava flow simulations using discharge rates from thermal infrared satellite imagery during the 2006 Etna eruption, Nat. Hazards, 50 (3), 539-550.

*Corresponding author: Giuseppe Bilotta,
Università di Catania, Dipartimento di Matematica e Informatica,
Catania, Italy; email: bilotta@dmi.unict.it.