MOX-Report No. 29/2016

GPU parallelization of brownout simulations with a non-interacting particles dynamic model

Miglio, E.; Parolini, N.; Penati, M.; Porcù, R.

MOX, Dipartimento di Matematica
Politecnico di Milano, Via Bonardi 9 - 20133 Milano (Italy)

mox-dmat@polimi.it http://mox.polimi.it

GPU parallelization of brownout simulations with a non-interacting particles dynamic model

Edie Miglio[a], Nicola Parolini[a], Mattia Penati[a], Roberto Porcù[a]

[a]*MOX–Department of Mathematics, Politecnico di Milano, P.zza L. da Vinci 32, 20133 Milano, Italy*

Abstract

The term brownout refers to the uplift of sand particles in the air and is generated when a helicopter is close to a dusty soil. When a brownout occurs the visibility area is remarkably restricted, thus the pilot may be disoriented and the helicopter may dangerously collide with the ground. Simulations of a brownout require tens of millions of particles in order to be significative, so that the execution of a serial program takes a very long time. In order to speedup the computation, the GPU-parallelization of a brownout simulation program is performed in order to obtain a notable speedup. The dynamics of the particles are considered in a Lagrangian way, under the effect of the gravity force and of a precomputed aerodynamic field. The particles are independent from each other since collisions between them are not taken into account. Thus trajectories are independent and the parallelization is very effective. In this paper we discuss in detail the impact of the techniques used for the GPU implementation of the parallel code on the performance.

Keywords: brownout, GPU, CUDA, parallelization, Fortran, PGI Compiler

Introduction

In aviation the term *brownout* refers to the phenomenon generated when a helicopter is close to a dusty or sandy soil and consists in a visibility area restriction due to the uplifting of sand particles in the air by vortices of the aerodynamic field, as it is shown in Figure 1.

Email addresses: edie.miglio@polimi.it (Edie Miglio), nicola.parolini@polimi.it (Nicola Parolini), mattia.penati@polimi.it (Mattia

(a) Eurocopter EC135 D-HZSG [1]

(b) Bell-Boeing V-22 Osprey [2]

Figure 1: Helicopters experiencing brownout.

When a brownout occurs, the pilot may experience disorientation and loss of control, so the helicopter may bump the ground. Other consequences of a brownout are rotor blades abrasion, damages to the components of the rotor and loss of power due to air filters occlusion. Single-rotor helicopters are generally severely affected by this problem and for dual-rotor helicopters (like *tandems* and *tiltrotors*) the situation sometimes is even worse.

Different factors may influence the occurrence and the intensity of a brownout, including rotation speed of rotor blades, rotor configuration, soil composition, weather conditions and landing angle. To prevent or limit the occurrence of this phenomenon, some remedies do exist such as landing site preparation, specific piloting techniques and helicopter aerodynamic-design solutions. Landing site preparation is possible only in specific sites, like the base camp. In cases of landing in unpredicted sites then particular piloting techniques and specific aerodynamic-design are essentials. During the past few years industries and researchers have been spending a lot of effort to further improve the knowledge about the brownout phenomenon. Numerical simulations of brownout can be found in D'Andrea [3, 4, 5], Wachspress et al. [6], Wadcock et al. [7], Gerlach [8]. In the works of Phillips et al. [9, 10, 11, 12] the brownout is simulated taking into account the *ground effects* and it is demonstrated that the shape of the sandy cloud is mainly affected by the whole helicopter system and not only by the rotor configuration. In the treatise of Tritschler et al. [13] it is described how flight path optimization can mitigate the rotorcraft brownout. Experimental

Penati), `roberto.porcu@polimi.it` (Roberto Porcù)

analysis of brownout are presented in Nathan and Green [14], Doehler and Peinecke [15]. HPC techniques and numerical methods for the acceleration of brownout simulations can be found in Hu et al. [16], Lohry et al. [17], Thomas [18] but they lack in the details of the implementation and in an accurated analysis of the performances and it is not clear if their parallelization is optimal or not. These latter topics are instead discussed in this paper and in Porcù [19] where it is also demonstrated that a more appropriate numerical method than the one used in this work can provide even better performances.

The paper is organized as follows: in the first section we describe the mathematical and numerical model implemented in the code and used for the simulations. In the second section we provide a brief digression on GPGPU and then a detailed description of the parallelization features used to overcome the most important bottlenecks of the serial program. Finally in the third section we present the numerical results obtained using firstly an analytical wind field and then a precomputed aerodynamical field; in the latter case we computed a high order solution by means of the Richardson extrapolation and we tested the convergence order of the numerical method implemented.

1. Mathematical and numerical model of the brownout

In this work the brownout is simulated in a Lagrangian way, following the motion of the sand particles. This section presents the semi-implicit Euler scheme that is the numerical model adopted for the discretization of the problem. For the aerodynamic definitions contained in the following equations we refer to Hoerner [20].

The dynamics of the particles are governed by the well known second Newton's equation $\mathbf{F}_p = m_p \mathbf{a}_p$, where $\mathbf{a}_p$ is the acceleration and $\mathbf{F}_p$ is the forcing term additively decomposed into the gravity force and the aerodynamic field:

$$\mathbf{F}_p = -\frac{1}{2}\rho_{air}|\mathbf{v}_{\mathrm{rel},p}|\mathbf{v}_{\mathrm{rel},p}\frac{\pi d_p^2}{4}C_{\mathrm{d},p} - m_p\mathbf{g}. \tag{1}$$

In relation (1) for each particle the quantity $\mathbf{v}_{\mathrm{rel},p} = \mathbf{v}_p - \mathbf{v}_{wind}(\mathbf{x}_p)$ represents the relative velocity with respect to the wind velocity at the position of the particle; $\mathbf{g}$ is the gravitational acceleration; m_p and d_p denote respectively the mass and the diameter of the particle and ρ_{air} is the density of the

air assumed uniform. The drag coefficient $C_{\mathrm{d},p}$ is inversely proportional to the Reynolds number (Re_p), as stated by the following definitions:

$$C_{\mathrm{d},p} = \frac{24}{Re_p}, \qquad Re_p = \frac{\rho_{air}|\mathbf{v}_{\mathrm{rel},p}|d_p}{\mu_{air}}, \tag{2}$$

where μ_{air} denotes the dynamic viscosity of the air, assumed uniform. Thanks to identities (2.a) and (2.b) the forcing term (1) can be simplified as:

$$\mathbf{F}_p = -3\pi d_p \mu_{air} \mathbf{v}_{\mathrm{rel},p} - m_p \mathbf{g}. \tag{3}$$

Thus at each time step the new velocity $\mathbf{v}_p^{k+1}$ and the new position $\mathbf{x}_p^{k+1}$ are computed on the basis of the following numerical scheme:

$$\begin{cases} \mathbf{x}_p^{k+1} = \mathbf{x}_p^k + \Delta t \cdot \mathbf{v}_p^{k+1}, \\ \mathbf{v}_p^{k+1} = \mathbf{v}_p^k + \Delta t \cdot \mathbf{a}_p^k, \end{cases} \tag{4}$$

where, according to (2)-(3), it is assumed:

$$\mathbf{a}_p^k = \frac{\mathbf{F}_p^k}{m_p} = -18\frac{\mu_{air}}{\rho_p d_p^2}\Big(\mathbf{v}_p^k - \mathbf{v}_{wind}(\mathbf{x}_p^k)\Big) - \mathbf{g}, \tag{5}$$

and Δt is the constant timestep. Since particles are assumed to have a spheric shape, in relation (5) formula $m_p = {}^{\pi d_p^3 \rho_p}/_6$ has been used. The numerical scheme (4) is semi-implicit and so the fulfillment of stability condition is required.

2. Parallelization on GPU architecture

CUDA extensions allow the programmer to define a particular type of functions, called *kernels*, which are invoked by the *host* (that can be the main thread on the CPU) and are run on the *device* (that is the GPU). After a kernel has been invoked, the CUDA runtime system creates a *grid* of many parallel threads residing on the device and each one executes the entire kernel. Usually a grid can contain hundreds of thousands of lightweight threads. Inside this grid, threads are organized into a two-levels hyerarchy. At the higher level, each grid is defined as a three-dimensional array of *blocks* whose dimensions can be retrieved by the CUDA keywords `gridDim.x`, `gridDim.y` and `gridDim.z`. Inside the 3D array each block is indexed by the CUDA keywords `blockIdx.x`, `blockIdx.y` and `blockIdx.z`. At the lower level each

block is defined as a three-dimensional array of threads whose dimensions are limited by the technical specifics of the GPU and are defined by the CUDA keywords `blockDim.x`, `blockDim.y` and `blockDim.z`. Inside each block a particular thread can be locally indexed by the CUDA keywords `threadIdx.x`, `threadIdx.y` and `threadIdx.z`.

Execution resources are organized in Streaming Multiprocessors (SMs) which are subdivided in Streaming Processors (SPs); to each SP can be assigned one block of threads at a time. Thus a SM can execute simultaneously a number of blocks at most equal to the number of its SPs. Actually this number depends on the quantity of resources requested by the kernel: if the necessary resources exceed the limits on the local shared memory space the CUDA runtime system automatically reduces the number of blocks assigned to each SM.

A block of threads is further subdivided into thread-units called *warps*. A warp is the execution unit of the threads inside the SMs, that is the set of threads which are really executed at the same time. Warp dimension is proper of the GPU hardware architecture. A wise implementation of warps execution is one of the aspects that can help to optimize the performances of CUDA programs.

For further details on GPUs and CUDA programming we refer to Mei Hwu and Kirk [21], Kandrot and Sanders [22], Fatica and Ruetsch [23] and to manuals [24], [25].

The parallelization of the code has been implemented using the CUDA extensions for Fortran 90 provided by the PGI Portland compiler v16.3 and all the serial and parallel simulations were run on the same computer whose specifics are synthetized in section §3.

2.1. Single-GPU implementation

The code is mainly structured as illustrated in Figure 2. The computational domain is generated into the subroutine that reads the aerodynamic grid and its dimensions are parallelly determined as described in subsection §2.1.1. The initial positions of the particles can be uniformly or randomly distributed inside the domain; this task is contained into the subroutine that sets up the initial conditions and it is described in subsection §2.1.2. The subroutine that updates all the kinematical and physical quantities is the nucleus of the simulation and its parallelization, which is fundamental for the overall speedup of the program, is described in subsection §2.1.3. The

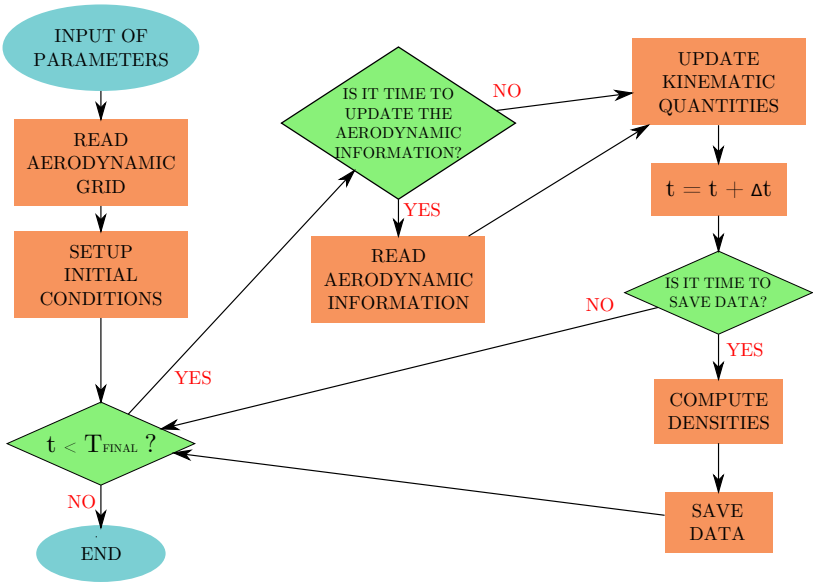


Figure 2: Flow chart of the serial program.

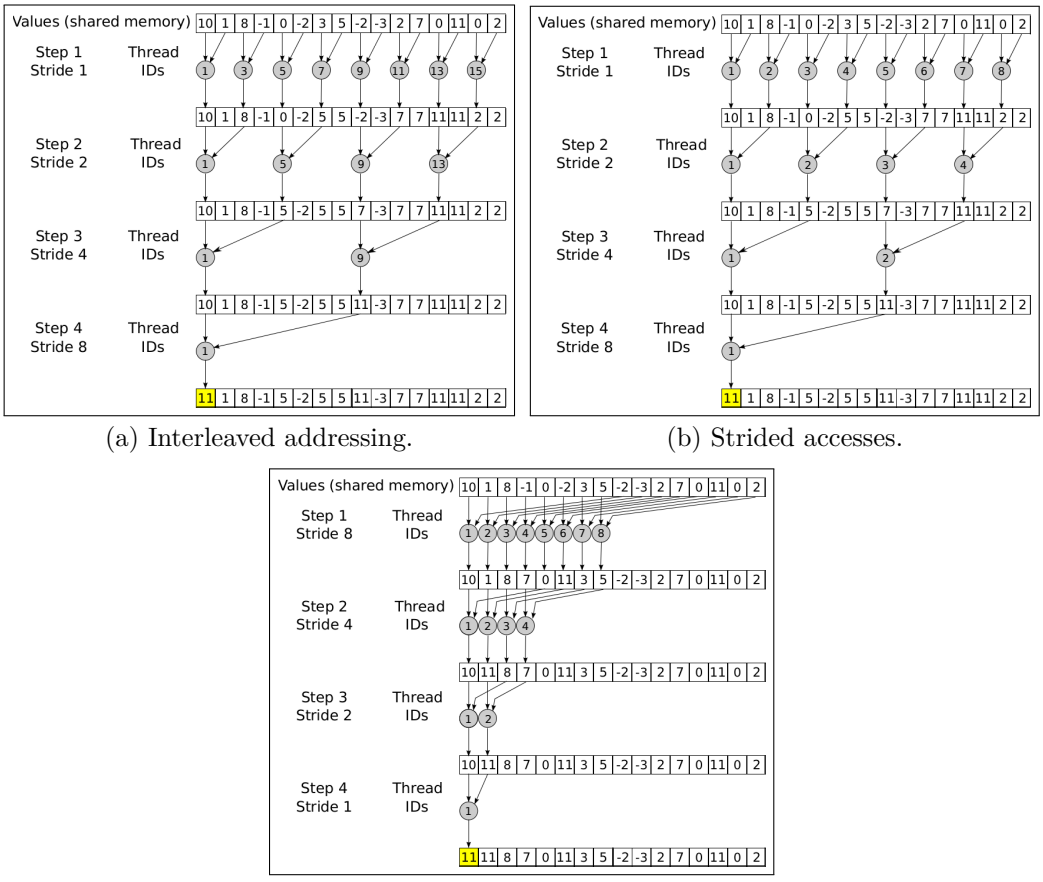
parallel determination of the densities of the sand for each cell of the domain represents the last relevant task and is described in subsection §2.1.4.

2.1.1. Parallel reduction

The wind velocity field is a given data known at scattered points in space. Since it is required for the computational domain to contain the whole aerodynamic field, it is necessary to determine the maximum and minimum values for the coordinates of the wind data. This operation can be done with a parallel reduction on each singular coordinate, thus for the sake of simplicity the following digression is restricted to the one-dimensional case.

The parallel reduction of a 1D array with CUDA requires taking into account the development of bank conflicts and execution divergence problems. For this topic and for what will be discussed in the following lines we refer to Harris [26].

Figure 3(a) describes the computation of the maximum element among an array of data by means of a parallel reduction characterized by interleaved addressing. Recalling that Fortran adopts a 1-based indexing, at the step number k the active threads are those one whose index i is equal to 1 plus a multiple of 2^k and each of them computes the maximum between the elements at positions i and $i + 2^{k-1}$ in the array. This access scheme leads to a high



(a) Interleaved addressing.

(b) Strided accesses.

(c) Sequential addressing.

Figure 3: Different implementations of the parallel reduction of a 1D array.

divergent execution and to high overhead due to the selection of active or inactive threads.

Figure 3(b) shows the same parallel reduction but with a strided addressing. In this access scheme, at the step number k the active threads are those one whose index i is less than or equal to a quantity s which is initially set equal to the dimension of the block of threads (that is the number of threads of each block) and that halves at each iteration. These threads compute the maximum between the elements at positions $(i-1) \cdot 2^k + 1$ and $(i-1) \cdot 2^k + 2^{k-1} + 1$ in the array. This method is affected by many bank conflicts (between threads of the same warp).

Finally Figure 3(c) shows the same parallel reduction with sequential

addressing. This access scheme needs a parameter s initially set equal to half the dimension of the block: at each iteration, only those threads whose index i is less than or equal to s are active and s halves. Each active thread computes the maximum between the elements at positions i and $i + s$ in the array. The cycle goes on until s is strictly greater than 0. This access-scheme is conflict free and has very low execution divergence.

All the above parallel reductions require a call to the `syncthreads` function inside each step because each active thread needs to access areas of memory that have been updated by other threads in the previous step. This operation is characterized by a high overhead since it acts like a barrier at which all threads belonging to the same block stop their execution waGiting for each other. For this reason in the code it has been implemented the parallel reduction with sequential addressing with the following further improvement: reduction cycling stops when $s \leq 32$ (32 being the warp-size) since threads of the same warp are naturally synchronous and so from now on threads synchronization is no longer needed. In this way, the latest iterations can be explicitly written (unrolled).

Table 1 reports respectively the times of execution of the serial, the parallel without unroll and the parallel with unroll reductions of a test made on a three-dimensional array of double precision values with dimensions $151 \times 151 \times 81$. It shows that, avoiding last calls to threads synchronization (thanks to unrolling), the latency of execution can be considerably reduced.

Table 1: Reduction times comparison.

	Time (ms)	Speedup	
Serial reduction	11.2058		
Parallel reduction without unroll	1.1623	9.64X	
Parallel reduction with unroll	0.5954	1.95X	18.82X

2.1.2. Random numbers generation

At initial time, the particles are arranged inside a horizontal narrow box which has the same length and width of the domain but is 15 mm tall starting from the height of 10 mm. The positions of the particles can be distributed uniformly or randomly along the three directions. In case of random placing the indices $i_{p,x}$, $i_{p,y}$ and $i_{p,z}$ of each particle are multiplied by a correspondent random number r_p in the interval taken from uniform distributions in [0,1] generated by the GPUs thanks to the CURAND library functions. In the

other case these random quantities are all set equal to 1. The following equation describes the chosen initial setting:

$$\begin{cases} x_{p,0} = x_{min,box} + \Delta x \cdot (i_{p,x} \cdot r_{p,x}), \\ y_{p,0} = y_{min,box} + \Delta y \cdot (i_{p,y} \cdot r_{p,y}), \\ z_{p,0} = z_{min,box} + \Delta z \cdot (i_{p,z} \cdot r_{p,z}), \end{cases} \qquad \begin{cases} \Delta x = \dfrac{x_{max,box} - x_{min,box}}{(\#\text{particles})_x - 1}, \\ \Delta y = \dfrac{y_{max,box} - y_{min,box}}{(\#\text{particles})_y - 1}, \\ \Delta z = \dfrac{z_{max,box} - z_{min,box}}{(\#\text{particles})_z - 1}, \end{cases}$$

where $x_{min,box}$, $y_{min,box}$ and $z_{min,box}$ are the lower boundaries of the narrow box, $i_{p,x}$, $i_{p,y}$ and $i_{p,z}$ are the 1-based numbering indexes of one particle and Δx, Δy and Δz are the spatial steps in case of a uniform spatial distribution.

With PGI Fortran compiler v16.3, libraries like CUBLAS, CUFFT and CURAND can be called by means of Fortran interfaces and of the ISO C binding as discussed in [23]. In this way random numbers generators can be called with the same signature of CUDA C, as illustrated in Listing 1.

Listing 1: Random numbers generation on GPU.

```
module FortranRand
  integer, public :: CURAND_RNG_PSEUDO_DEFAULT=100

  interface CreateGenerator
    subroutine CreateGenerator(gen, gen_type) &
    bind (C, name =' curandCreateGenerator ')
      use iso_c_binding
      integer(c_size_t) :: gen
      integer(c_int), value :: gen_type
    end subroutine CreateGenerator
  end interface

  interface GenerateUniformDouble
    subroutine GenerateUniformDouble(gen, outdata, size) &
    bind (C, name =' curandGenerateUniformDouble ')
      use iso_c_binding
      integer(c_size_t),value :: gen
      real(c_float), device :: outdata(*)
      integer(c_int), value :: size
    end subroutine GenerateUniformDouble
  end interface

end module FortranRand
```

2.1.3. Motion of the particles

The operative core of the program is the subroutine which computes the displacements of the particles on the basis of the numerical method described in (4). This process can be subdivided into four main parts: the first updates the accelerations, the velocities and the positions; the second checks if the new computed positions exceed the domain boundaries and in this case it resets the outlier particles to their respective initial condition (that is their state at $t = 0$); the third part updates the relative velocities and finally the fourth updates the drag coefficients and the Reynolds numbers.

These four tasks can be implemented into four different kernels in a biunique correspondence. The positive aspect of this pattern is that each kernel doesn't require too much space on shared memory and registers and so the optimization of the occupancy of the SMs (namely the choice of the number of threads per block) is quite easy. A higher percentage of occupancy does not necessarily mean better performances but the higher the occupancy, the more the runtime system can hide latencies. On the other hand, the negative aspect is that each kernel invocation implies an overhead that, if repeated many times, can impact the performances of the parallel application.

Thus, in order to gain an higher speedup, a two-kernels version has been implemented, grouping the first with the second and the third with the fourth task. In this case the optimization of the occupancy of the SMs is a bit harder since each kernel now requires more resources and so the number of threads per block should be chosen more accurately (the *CUDA Occupancy Calculator* [27] tool can be useful in this sense). On the other hand, this pattern halves the number of kernels invocations and their respectives overheads. A comparison between the four-kernel and the two-kernel implementations is synthetized in Table 2.

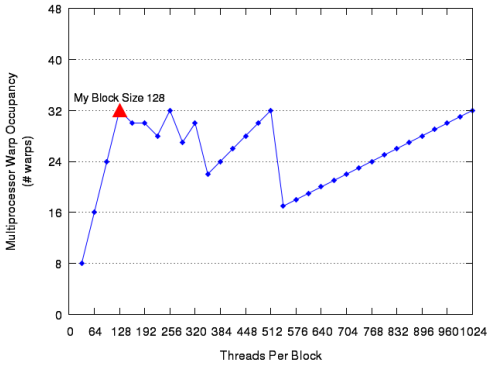
Table 2: Timings of four-kernel and two-kernel implementations.

<table>
<tr><th colspan="4">four-kernel implementation</th><th colspan="4">two-kernel implementation</th></tr>
<tr><td>1st task</td><td>1st kernel</td><td>3.71 ms</td><td rowspan="2">5.04 ms</td><td>1st task</td><td rowspan="2">1st kernel</td><td rowspan="2">4.31 ms</td></tr>
<tr><td>2nd task</td><td>2nd kernel</td><td>1.33 ms</td><td>2nd task</td></tr>
<tr><td>3rd task</td><td>3rd kernel</td><td>3.53 ms</td><td rowspan="2">4.55 ms</td><td>3rd task</td><td rowspan="2">2nd kernel</td><td rowspan="2">3.82 ms</td></tr>
<tr><td>4th task</td><td>4th kernel</td><td>1.02 ms</td><td>4th task</td></tr>
<tr><td colspan="3">Aggregate</td><td>9.59 ms</td><td colspan="2">Aggregate</td><td>8.13 ms</td></tr>
</table>

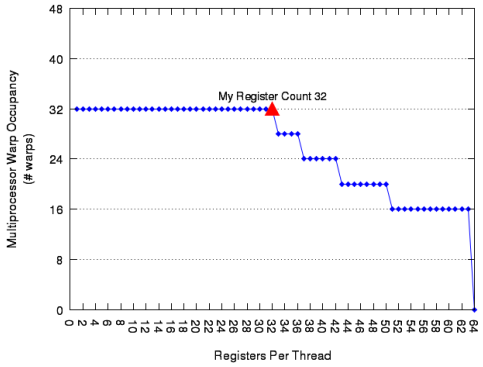
Since the above kernels are repeated for each timestep, the 1.18X speedup

guaranteed by the two-kernel design pattern becomes significant for the overall simulation time.

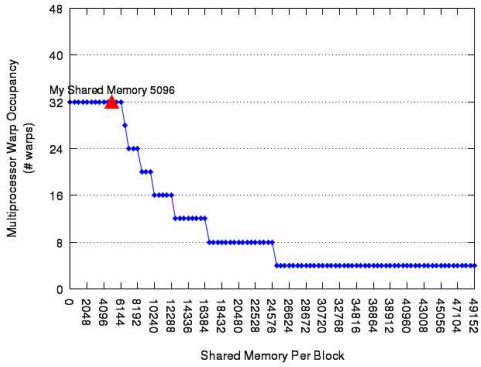
As previously said, the *CUDA Occupancy Calculator* can analyze the occupancy of the SMs for a given kernel. This utility consists of a spreadsheet that takes as input the amount of resources occupied by the kernel: these data can be retrieved by means of a profiler, such as the "NVIDIA Visual Profiler" [28]). The output consists of a table of data to whom three graphics with lines are associated. Analyzing these results it is possible to identify the factors that are limiting the occupancy.



(a) Impact of threads-number.



(b) Impact of requested registers.



(c) Impact of requested shared memory.

Figure 4: Impact of parameters variations on the occupancy of the SMs.

Figure 4 shows the graphics with lines obtained in the case of the second kernel in the two-kernels design pattern. It is possible to see that the occupancy obtained is of the 66.7%. For the first kernel it was obtained an

occupancy of 58%. These were the best possible results. The number of threads per block should always be chosen as a multiple of the warp-size. Then one strategy to identify the best configuration is to try different numbers of threads per block and analyze each case.

2.1.4. Atomic functions

A further bottleneck for the serial implementation is represented by the subroutine where the density of the sand in each cell of the domain is computed. In this task the program cycles over all the particles firstly determining the spatial indices of the cell in which a fixed particle is contained and then updating the density relative to that cell. This latter operation consists just in the addition of a quantity equal to the mass of the particle divided by the volume of the cell to the local temporary density.

The use of the atomic function `atomicAdd` is fundamental in the implementation of the kernel associated to this task. Indeed the memory space, where the temporary density is stored, might be updated by multiple threads at the same time. When a thread is asked to update a variable, it firstly makes a private copy of the value, then it updates its copy and finally it stores its updated copy into the original memory space. If two or more threads perform this operation simultaneously on the same memory address, the final result can obviously be wrong. This inconvenience is known as *race condition*.

Atomic functions perform operations using techniques that achieve mutual exclusion mechanisms in order to prevent race conditions. These functions work both on global and shared memory (in the latter case performances should be higher since accesses must be mutually exclusive only for threads of the same block).

2.2. Multi-GPU implementation

A multi-GPU version of this parallel code has been implemented in order to investigate the additional benefit of performances that can be obtained using both the GPUs Tesla K80 available on the HPC node. From version CUDA 4.0 onwards, programming multi-GPU systems has become easier, thanks to the introduction of new functions such as the `cudaSetDevice` function which allows the user to dinamically select the operative GPU.

The first relevant change with respect to the single-GPU version is the allocation of the memory on the RAM of each GPU: supposing that N GPUs are available for the computation and that a 1D array of L elements has to be allocated among the N GPUs, then the array must be divided into N

smaller arrays each one of d elements, where d is equal to the integer division of L by N, except for the last one which contains $d + r$ elements, being r the remainder after the division of L by N, as illustrated in Figure 5.

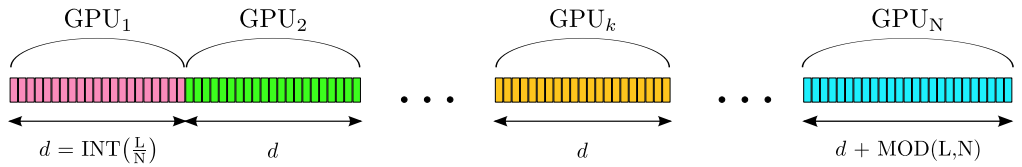


Figure 5: Memory allocation among N different GPUs.

A synthetic and exemplifying implementation of the described memory allocation is presented in Listing 2.

Listing 2: Multi-GPU memory allocation.

```
type my_type_name
  real*8,allocatable,device :: GPU_variable_1_name(:)
  real*8,allocatable,device :: GPU_variable_2_name(:)
end type my_type_name
type(my_type_name),pointer,dimension(:) :: N_GPUs_variable_name

allocate( N_GPUs_variable_name(N) )
do GPU_id = 1, N
  ierr = cudaSetDevice(GPU_id-1)
  if(GPU_id .eq. N) then
    allocate(N_GPUs_variable_name(k)%GPU_variable_1_name(d+r))
    allocate(N_GPUs_variable_name(k)%GPU_variable_2_name(d+r))
  else
    allocate(N_GPUs_variable_name(k)%GPU_variable_1_name(d))
    allocate(N_GPUs_variable_name(k)%GPU_variable_2_name(d))
  endif
enddo
```

The second important difference with respect to the single-GPU implementation is that kernels and memory operations involving more than one GPU must be invoked individually for each GPU, as illustrated in the examples of Listings 2 and 3. Since the particles do not interact it is implicitly assumed that there is no need for synchronization, unless in the very few and rare cases when the sand densities are updated.

Listing 3: Multi-GPU kernel call.

```
! highly simplified model of the subroutine in subsection §3.1.3
do time_iteration = first_iteration, last_iteration
```

```
  do GPU_id = 1, N
    ierr = cudaSetDevice(GPU_id-1)
    call kernel_1<<<grid_dimensions,blocks_dimensions>>> &
      (N_GPUs_variable_name(k)%GPU_variable_1_name)
  enddo
  do GPU_id = 1, N
    ierr = cudaSetDevice(GPU_id-1)
    call kernel_2<<<grid_dimensions,blocks_dimensions>>> &
      (N_GPUs_variable_name(k)%GPU_variable_2_name)
  enddo
enddo
```

Non-blocking CUDA functions such as kernels do not prevent the CPU thread from changing the GPU to which assign instructions before they have completed their execution. This is an important aspect since it allows the different GPUs to work as much synchronously as possible, thereby obtaining a sort of additional parallelism, as illustrated in Figure 6.

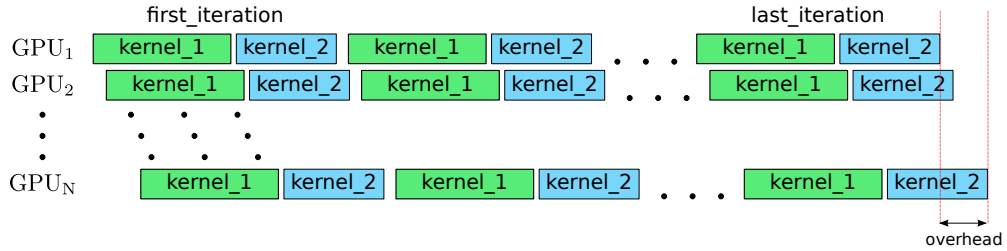


Figure 6: Kernel execution is a non-blocking CUDA function.

At the end of the last iteration the overhead due to the loops over the GPUs is a negligible percentage of the total execution time, expecially if the total number of time iterations is high. Applying these considerations to the core of the program, that is the subroutine described in subsection §2.1.3 which takes up more than the 99% of the total execution time of the simulation, it is clear that the single-CPU thread configuration that has been implemented is satisfactory and that there would be no significative speedup with a shared-memory multi-threaded CPU environment.

3. Numerical results

In this section the results obtained with two different types of aerodynamic fields (one analytical and another more realistic) are described and analyzed. The simulations were run on a Linux-based operating system

equipped with 2 octa-core Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40 GHz with 128 GB RAM, and 2 GPUs NVIDIA Tesla K80.

In the first case an analytical solenoidal flow field is considered, which does not give significative results from the point of view of the brownout phenomenon but is useful to investigate the performances of the parallelization without overheads due to file reads or writes. In the second case a realistic flow field, obtained from a previous fluidodynamic simulation consistent with the brownout problem, is considered. In the latter case the interest falls not only into the performance gain obtained by the parallelization but also into the accuracy of the numerical solution.

3.1. Analytical aerodynamic field

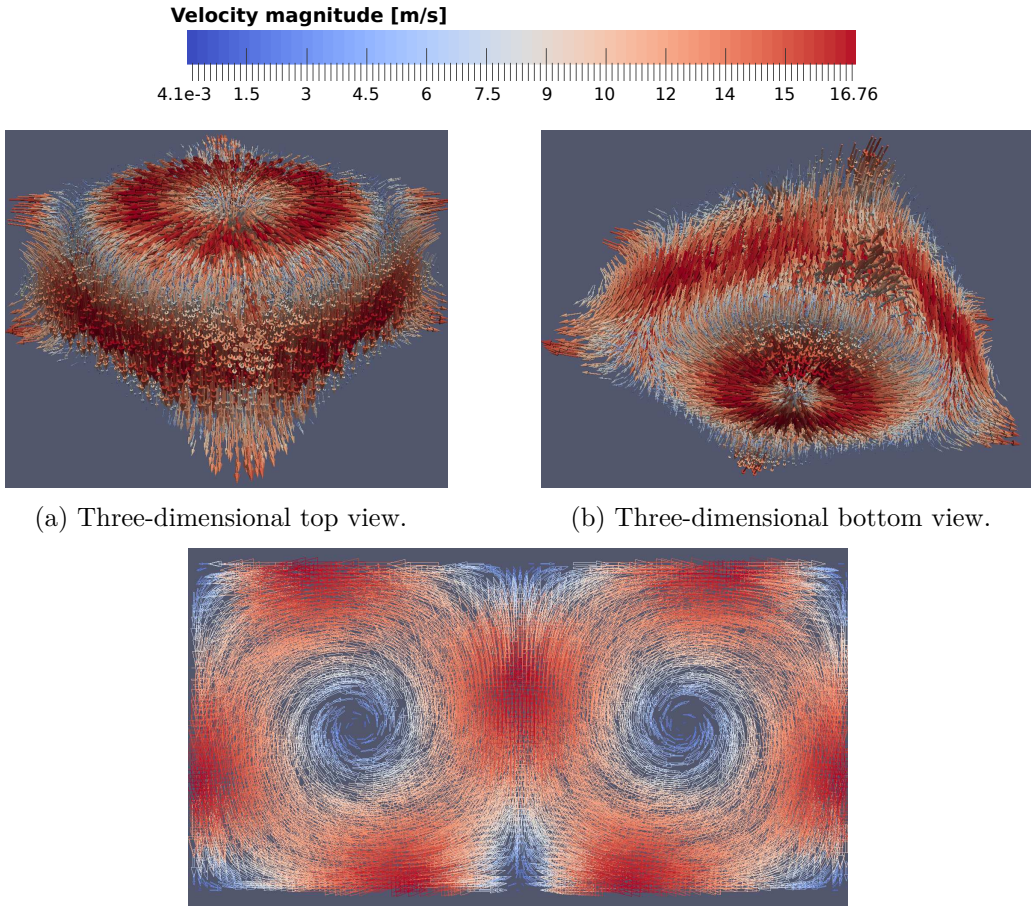
Let us consider an analytical field defined in cylindric coordinates as follows:

$$\begin{cases} v_\rho = -\dfrac{\partial \psi}{\partial z} = -A\dfrac{\pi}{L}\sin\left(\pi\dfrac{\rho}{L}\right)\cos\left(\pi\dfrac{z}{L}\right), \\ v_\theta = 0, \\ v_z = \dfrac{\partial \psi}{\partial \rho} = A\dfrac{\pi}{L}\cos\left(\pi\dfrac{\rho}{L}\right)\sin\left(\pi\dfrac{z}{L}\right), \end{cases}$$

where $\psi(\rho, z) = A\sin(\pi\rho/L)\sin(\pi z/L)$ is a potential field with the multiplicative factor A equal to $40\,\mathrm{m^2\,s^{-1}}$ and L (set equal to 7.5 m) represents the size of the domain along the radial and the vertical directions. Since v_θ is always zero and the other two components of the velocity don't depend on the angular coordinate θ, the above aerodynamic field has a cylindrical symmetry along the z direction and can be naturally assumed as a two-dimensional field. It is represented in Figure 7.

In order to discuss the performances of the serial, the single-GPU and the double-GPU simulations, two different configurations has been taken into account: one with 3.072×10^6 particles and the other with 6.144×10^6 particles. For each configuration the sand particles are uniformly subdivided into three groups of different diameters, in particular a "Small" particle has $d_p = 5\,\mu\mathrm{m}$, a "Medium" particle has $d_p = 10\,\mu\mathrm{m}$ and a "Large" particle has $d_p = 15\,\mu\mathrm{m}$.

The pictures in Figure 8 are representative of this simulation and they show respectively the positions and the modules of the velocities of the particles at the simulation time of 12.5 s. It can be observed that the disposition



(a) Three-dimensional top view.

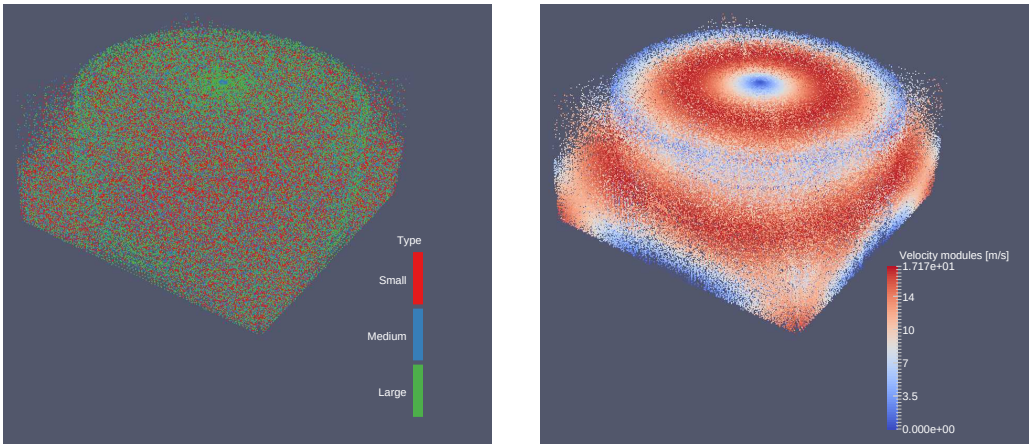
(b) Three-dimensional bottom view.

(c) Slice along a vertical plane passing through the origin.

Figure 7: Analytical aerodynamic field.

of the particles and their velocities agree with the cylindrical symmetry of the analytical field and with its distribution but there are no remarkable preferences on the type of the particle. Indeed it is not possible to distinguish a particular pattern.

The computational times needed for these formulations are synthetized in Table 3. Using one GPU the parallel program is 129.7 times faster than the serial program. Using two GPUs a further 1.3X speedup is gained. On the other hand, in the configuration with more particles, the single-GPU program is 133.4 times faster than the serial one and the double-GPU program is again 1.3 times faster than the single-GPU one.



(a) Particles positions.

(b) Particles velocity modules.

Figure 8: Analytical field simulation at time $t = 12.5\,\mathrm{s}$.

Comparing these results it is clear that the speedups between the parallel and the serial implementations are a little bit higher in the second configuration. This behavior can be explained by noticing that the bigger the number of particles is, the bigger the percentage of the parallel portion of the code is.

Furthermore observe that the performance gain provided by the double-GPU program is not much higher than the single-GPU one: this should be related to the fact that in the double-GPU simulation there isn't a good parallelism between the two devices because there is only a single thread on the host. This thread can control only one GPU at a time and so it manages the two GPUs cyclically, preventing the devices to work synchronously.

Table 3: Computational times for the analytical field.

	3.072×10^6 particles			6.144×10^6 particles		
	computational time	speedup		computational time	speedup	
Serial	7 d, 13 h, 36 min			14 d, 23 h, 47 min		
1 GPU	1 h, 24 min	129.7X		2 h, 42 min	133.4X	
2 GPUs	1 h, 24 min	167.6X	1.3X	1 h, 24 min	173.4X	1.3X

3.2. Realistic aerodynamic field

A realistic aerodynamic field has been computed in order to investigate the accuracy of the implemented numerical method. These discrete data

represent the wind velocities evaluated at fixed points in the space and at given time instants and are stored into files loaded at runtime. Successively a spatial B-spline interpolation and a time linear interpolation are performed in order to recover global $\mathcal{C}^0$ regularity of the aerodynamic field.

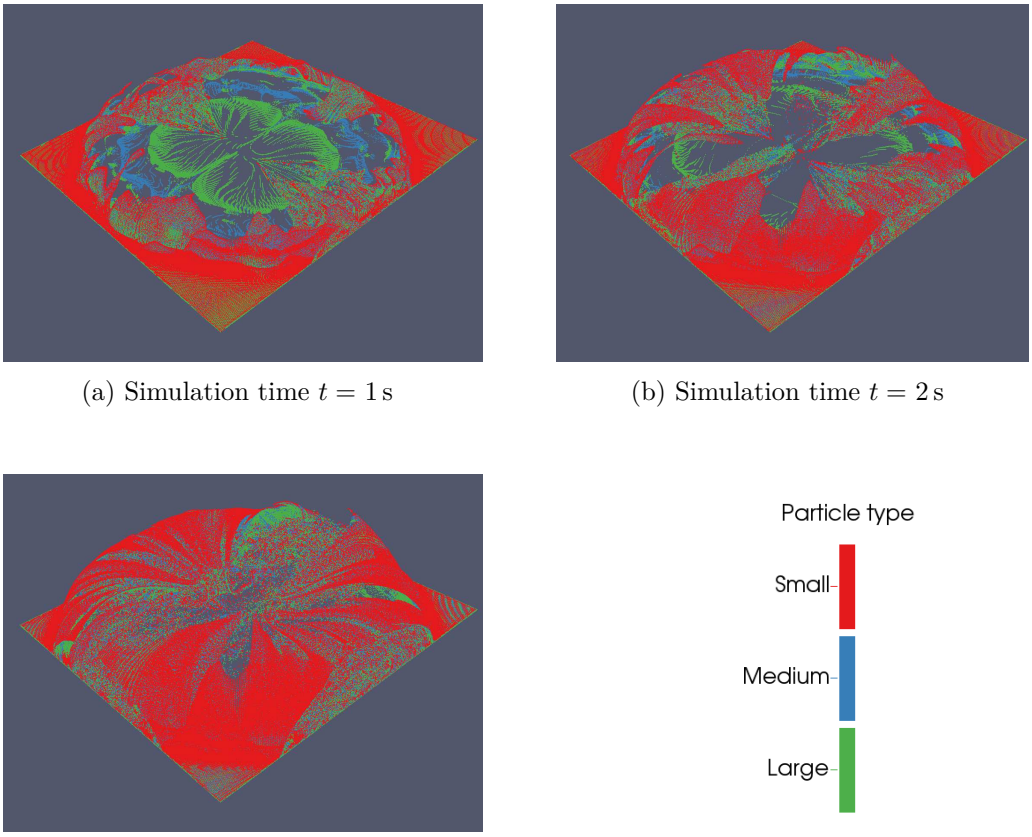
It is important to notice that the performances of the overall simulation are affected by high uncoalesced global memory accesses occurring in one of the kernels, discussed in subsection §2.1.3, which updates the particles relative velocities. Indeed, since particles positions are distributed in an unpredictable way inside the domain, it is not possible to determine a priori which subset of the wind velocities array will be needed by a particular GPU thread. For this reason each block of threads has to have access to the whole array which is too big to be stored on the shared memory and so remains on the global memory.

For this simulation it has been considered a configuration with 6.144×10^6 sand particles equally subdivided into three groups of different diameters: a "Small" particle has $d_p = 5\,\mu\text{m}$, a "Medium" particle has $d_p = 10\,\mu\text{m}$, the "Large" particle has $d_p = 15\,\mu\text{m}$). The final instant of the simulation is set at 50 s. Assuming the air dynamic viscosity equal to $\mu_{air} = 1.78 \times 10^{-5}\,\text{Pa}\,\text{s}$, then it is possible to observe that the timestep Δt must be $\leq 2 \times 10^{-4}\,\text{s}$ in order to assure numerical stability.

Figure 9 shows the displacements of the particles respectively after 1 s, 2 s and 5 s.

Let us discuss the computational gain obtained by the parallelization. The serial program takes about 16 days and 15 hours to end the execution. The single-GPU parallelization drops the computational time down to a little more than 3 hours and a half. The double-GPU implementation provides an additional speedup taking a little less than 3 hours to complete the simulation. These results are synthetized in Table 4. Comparing these speedups with those one observed in the analytical field test it can be noticed that now the performance gain provided by the parallelization is smaller. This decrease can be explained by the uncoalesced global memory accesses discussed above and by the file reads computed every N timesteps, where N is an integer fixed by the input parameters, in order to load the wind velocities data. Since the size of each file is about 12 MB, these reading operations are quite expensive and so the portion of code that is not parallelizable increases, resulting in a loss of parallelization performance (Amdahl's law).

Another noticeable aspect is that now the speedup ratio between the double-GPU and the single-GPU is a little bit smaller if compared to the



(a) Simulation time $t = 1\,\mathrm{s}$

(b) Simulation time $t = 2\,\mathrm{s}$

(c) Simulation time $t = 5\,\mathrm{s}$

Figure 9: Simulations with the real aerodynamic field at different instants.

correspondent quantity of the anaylitical field simulation. This result can be explained by observing that in the realistic field simulation the aerodynamic field information read from file is needed by both the GPUs, so that these data are transferred from host memory to the global memory of each GPU, which means two transfers for the double-GPU program in place of one single transfer in the single-GPU program.

A numerical assessment of the convergence order for the implemented method has been carried out. Different simulations with timesteps $\Delta t_i = 2^{1-i} \times 10^{-4}$ s with $i = \{0, \ldots, 5\}$ have been considered and the errors of the coordinates and velocities with respect to the values provided by Richardson extrapolation method have been computed at simulation time $t = 0.45\,\mathrm{s}$.

Table 4: Computational times for the realistic field.

	computational time	speedup	
Serial	16 d, 14 h, 52 min		
Single-GPU	3 h, 32 min	112.6X	
Double-GPU	2 h, 57 min	135.1X	1.2X

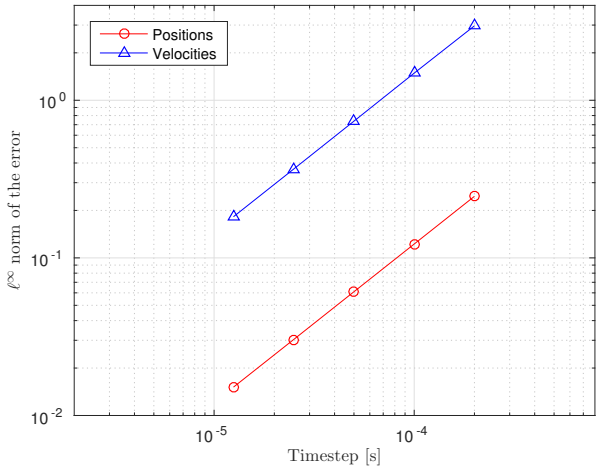


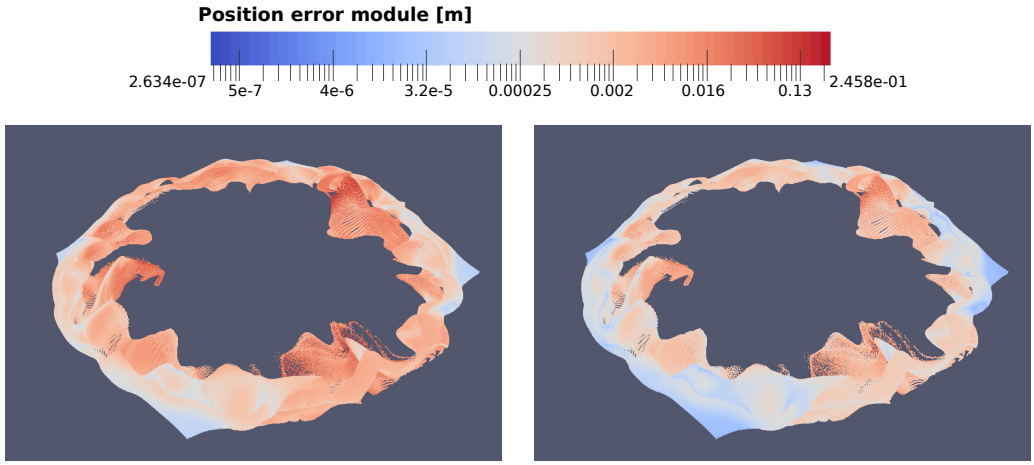
Figure 10: ℓ^∞ norm of the error over the timestep

A *loglog* plot of the ℓ^∞-norm of the error over the timestep is shown in Figure 10. Since the numerical scheme used, that is the semi-implicit Euler method, is a first-order integrator, it is clear that the performed analysis is in accord to the expected order.

Figure 11 describes the spatial distribution of the error. In this figure the particles are located at their respective computed positions. As expected the error is greater for those particles which have travelled longer, that are those one at the advancing front.

The particles that, during the computation, leave the domain at least one time are not taken into account in the shown results. Indeed when a particle goes out of the domain, its physical quantities are resetted at its corresponding initial values, thus resulting in a discontinuity on its trajectory.

Analyzing individually each type of particle it is possible to observe that the error is still decreasing with the timestep, as represented in Figure 12, and the convergence orders are still equal to 1. The grater particles are



(a) Timestep $\Delta t = 2 \times 10^{-4}$ s.

(b) Timestep $\Delta t = 5 \times 10^{-5}$ s.

(c) Timestep $\Delta t = 1.25 \times 10^{-5}$ s.

Figure 11: ℓ^∞ norm of the error of the positions in a *log* scale.

affected by a smaller error, either for their coordinates and velocities. This can be explained considering that the greater particles have more inertia so are less affected by perturbations and it should not be related to the travelled distance since the advancing front is composed by each type of particle, as illustrated in Figure 13.

Conclusions

The parallelization of the code showed the potentialities provided by CUDA programming on graphic hardwares. The problem was embarassingly parallel since we considered the trajectories of the particles independent of each other. We obtained a high speedup, indeed the serial simulation takes

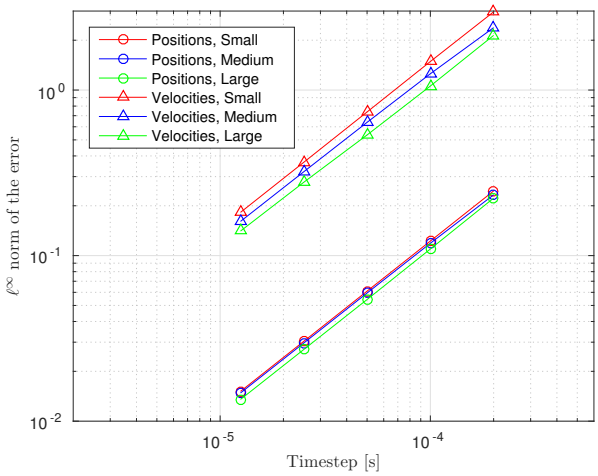


Figure 12: ℓ^∞ norm of the error for each type of particle.

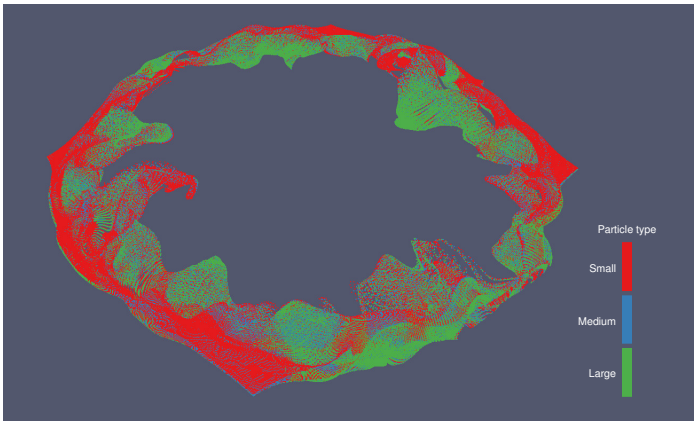


Figure 13: Distribution of the different types of particle at time $t = 0.45\,\mathrm{s}$.

about 16 days and 12 hours for its whole execution; the parallel simulation on a single GPU takes a little more than 3 hours and a half to end its work, 112.6 times faster than the serial program. The parallel implementation with two GPUs is even faster and completes its execution after about 3 hours, 1.2 times faster than the single GPU simulation.

Future developments of this work will focus on some different aspects. From the numerical point of view the most important thing to change is the implemented numerical scheme, which is a semi-implicit Euler integra-

tor. Indeed it requires a very small timestep in order to guarantee numerical stability. It has been proved, see Porcù [19], that high order methods, such as *spectral variational integrators*, even if more expensive from the computational point of view within the single timestep, can provide a relevant speedup expecially for accurate computations, thanks to a noticeable increase in the timestep size.

The second fundamental improvement that we have in mind is an extension of the mathematical model that has been actually implemented. It would be really important to take into account particle-particle interactions and also particle-ground interactions. With the latter aspect we mean not only the collisions of the particles with the ground but also the physics governing the motion of the sand particles when they're uplifted from the soil, which is not trivial but can have a remarkable impact on the overall phenomenon.

The last development that we account as interesting and fruitful concerns a further improvement of the parallelization of the code, in the sense of a hybrid multi-GPU + distributed-memory parallelization, dividing the computation over different HPC nodes by means of the MPI paradigm. We believe that such an implementation of the code would lead to the possibility of a significative increase in the number of particles used and so to more realistic simulations.

[1] J. Brüggemann, Eurocopter ec135 d-hzsg brownout, http://commons.wikimedia.org/wiki/File:Eurocopter_EC135_D-HZSG_Brownout_I_Br%C3%BCggemann.jpg.

[2] G. S. S. Williams, Osprey takes on brown-out in afghanistan, http://commons.wikimedia.org/wiki/File:Flickr_-_DVIDSHUB_-_Osprey_Takes_on_'Brown-Out'_in_Afghanistan.jpg.

[3] A. D'Andrea, Numerical analysis of unsteady vortical flows generated by a rotorcraft operating on ground: A first assessment of helicopter brownout, in: 65th Annual Forum of the American Helicopter Society, Grapevine, TX, 2009, pp. 27–29.

[4] A. D'Andrea, Unsteady numerical simulations of helicopters and tiltrotors operating in sandy-desert environment, in: Proceedings of the American Helicopter Society Specialist's Conference on Aeromechanics, 2010.

[5] A. D'Andrea, Development and application of a physics-based computational tool to simulate helicopter brownout, in: 37th European Rotorcraft Forum Proceedings, Gallarate (VA), Italy, 2011.

[6] D. A. Wachspress, G. R. Whitehouse, J. D. Keller, K. McClure, P. Gilmore, M. Dorsett, Physics based modeling of helicopter brownout for piloted simulation applications, Tech. rep., DTIC Document (2008).

[7] A. J. Wadcock, L. A. Ewing, E. Solis, M. Potsdam, G. Rajagopalan, Rotorcraft downwash flow field study to understand the aerodynamics of helicopter brownout, Tech. rep., DTIC Document (2008).

[8] T. Gerlach, Visualisation of the brownout phenomenon, integration and test on a helicopter flight simulator, The Aeronautical Journal 115 (1163) (2011) 57–63.

[9] C. Phillips, R. E. Brown, The effect of helicopter configuration on the fluid dynamics of brownout, in: 34th European Rotorcraft Forum, 2008.

[10] C. Phillips, R. E. Brown, Eulerian simulation of the fluid dynamics of helicopter brownout, Journal of Aircraft 46 (4) (2009) 1416–1429.

[11] C. Phillips, H. W. Kim, R. E. Brown, The flow physics of helicopter brownout, in: 66th American Helicopter Society Forum: Rising to New Heights in Vertical Lift Technology, 2010.

[12] C. Phillips, R. Brown, H. W. Kim, Helicopter brownout-can it be modelled?, Aeronautical Journal 115 (1164) (2011) 123–133.

[13] J. K. Tritschler, R. Celi, J. G. Leishman, Methodology for rotorcraft brownout mitigation through flight path optimization, Journal of Guidance, Control, and Dynamics 37 (5) (2014) 1524–1538.

[14] N. Nathan, R. Green, Flow visualisation of the helicopter brown-out phenomenon, Aeronautical Journal 113 (1145) (2009) 467–478.

[15] H.-U. Doehler, N. Peinecke, Image-based drift and height estimation for helicopter landings in brownout, in: International Conference Image Analysis and Recognition, Springer, 2010, pp. 366–377.

[16] Q. Hu, M. Syal, N. Gumerov, R. Duraiswami, J. G. Leishman, Toward improved aeromechanics simulations using recent advancements in scientific computing, in: Proceedings 67th Annual Forum of the American Helicopter Society, 2011, pp. 3–5.

[17] M. W. Lohry, S. Ghosh, R. G. Rajagopalan, Graphics hardware acceleration for rotorcraft brownout simulation, in: 20th AIAA Computational Fluid Dynamics Conference, 2011, p. 3224.

[18] S. Thomas, A gpu-accelerated, hybrid fvm-rans methodology for modeling rotorcraft brownout, Ph.D. thesis, University of Maryland, `http://drum.lib.umd.edu/handle/1903/14832` (2013).

[19] R. Porcù, Metodi numerici e tecniche di programmazione per l'accelerazione di un modello di dinamica di particelle non interagenti, Master's thesis, Politecnico di Milano, `https://www.politesi.polimi.it/browse?type=author&order=ASC&rpp=20&value=PORC%C3%99%2C+ROBERTO&locale=en` (4 2013).

[20] S. F. Hoerner, Fluid-dynamic drag: practical information on aerodynamic drag and hydrodynamic resistance, Hoerner Fluid Dynamics, 1965.

[21] D. B. Kirk, W. H. Wen-mei, Programming massively parallel processors: a hands-on approach, Newnes, 2012.

[22] J. Sanders, E. Kandrot, CUDA by example: an introduction to general-purpose GPU programming, Addison-Wesley Professional, 2010.

[23] G. Ruetsch, M. Fatica, Cuda fortran for scientists and engineers, NVIDIA Corporation 2701.

[24] NVIDIA, CUDA C Programming Guide, `http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`.

[25] The Portland Group, PGI CUDA Fortran Programming Guide and Reference, `http://www.pgroup.com/resources/docs.htm`.

[26] M. Harris, et al., Optimizing parallel reduction in cuda, NVIDIA Developer Technology 2 (4).

[27] NVIDIA, Cuda occupancy calculator, `http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls`.

[28] NVIDIA, Nvidia visual profiler, `https://developer.nvidia.com/nvidia-visual-profiler`.

MOX Technical Reports, last issues

Dipartimento di Matematica
Politecnico di Milano, Via Bonardi 9 - 20133 Milano (Italy)

28/2016 Antonietti, P.F.; Dal Santo, N.; Mazzieri, I.; Quarteroni, A.
A high-order discontinuous Galerkin approximation to ordinary differential equations with applications to elastodynamics

26/2016 Brunetto, D.; Calderoni, F.; Piccardi, C.
Communities in criminal networks: A case study

27/2016 Repossi, E.; Rosso, R.; Verani, M.
A phase-field model for liquid-gas mixtures: mathematical modelling and Discontinuous Galerkin discretization

25/2016 Baroli, D.; Cova, C.M.; Perotto, S.; Sala, L.; Veneziani, A.
Hi-POD solution of parametrized fluid dynamics problems: preliminary results

24/2016 Pagani, S.; Manzoni, A.; Quarteroni, A.
A Reduced Basis Ensemble Kalman Filter for State/parameter Identification in Large-scale Nonlinear Dynamical Systems

23/2016 Fedele, M.; Faggiano, E.; Dedè, L.; Quarteroni, A.
A Patient-Specific Aortic Valve Model based on Moving Resistive Immersed Implicit Surfaces

22/2016 Antonietti, P.F.; Facciola', C.; Russo, A.;Verani, M.
Discontinuous Galerkin approximation of flows in fractured porous media

21/2016 Ambrosi, D.; Zanzottera, A.
Mechanics and polarity in cell motility

19/2016 Guerciotti, B.; Vergara, C.
Computational comparison between Newtonian and non-Newtonian blood rheologies in stenotic vessels

20/2016 Wilhelm, M.; Sangalli, L.M.
Generalized Spatial Regression with Differential Regularization