Int J Parallel Prog (2018) 46:584–606
https://doi.org/10.1007/s10766-017-0509-y

SIMD Monte-Carlo Numerical Simulations Accelerated on GPU and Xeon Phi

Bastien Plazolles[1,2] · Didier El Baz[1] · Martin Spel[2] · Vincent Rivola[2] · Pascal Gegout[3,4]

Received: 24 February 2017 / Accepted: 10 May 2017 / Published online: 17 May 2017

Abstract The efficiency of a pleasingly parallel application is studied for several computing platforms. A real world problem, i.e., Monte-Carlo numerical simulations of stratospheric balloon envelope drift descent is considered. We detail the optimization of the SIMD parallel codes on the K40 and K80 GPUs as well as on the Intel Xeon Phi. We emphasize on loop and task parallelism, multi-threading and vectorization, respectively. The experiments show that GPU and MIC permit one to decrease computing time by non negligeable factors, as compared to a parallel code implemented on a two sockets CPU (E5-2680-v2) which finally allows us to use these devices in operational conditions.

✉ Didier El Baz
elbaz@laas.fr

Bastien Plazolles
bplazoll@laas.fr

Martin Spel
martin.spel@rtech.fr

Vincent Rivola
vincent.rivola@rtech.fr

Pascal Gegout
pascal.gegout@get.omp.eu

1 LAAS-CNRS, Université de Toulouse, CNRS, Toulouse, France

2 R.Tech, Parc Technologique Delta Sud, 09340 Verniolle, France

3 Géosciences Environnement Toulouse (CNRS UMR5563), 14 Avenue Edouard Belin, 31400 Toulouse, France

4 Université de Toulouse, 31400 Toulouse, France

Keywords Parallel computing · Multi-core CPU · Xeon Phi · OpenMP · GPU · CUDA · Numerical integrator · Monte-Carlo simulations

1 Introduction

In this paper we present the benefits of using computing accelerators in order to solve a real world problem in operational conditions. We concentrate on the enveloppe drift descent analysis problem for statospheric balloon. By operational condition, we mean that our goal is to use easily displaceable computing platforms in the operational field far from any computing center and with limited energy consumption.

Currently to determine the landing point of the stratospheric balloon envelope after a drift descent, the French space agency uses a sequential code that takes around 0.1 s to compute a single drift descent trajectory, i.e. an unique landing point. In its present form this code cannot be used in operational condition to perform a fast Monte-Carlo simulation of the drift descent, since it is purely mono-core, mono threaded and would take several hours to compute 100,000 simulations.

This is why we study the benefits of computing accelerators like Graphics Processing Units (GPU) and Many Integrated Core (MIC) that can easily be transported onto operational field and wich feature low energy consumption. In the sequel, we show how this pleasingly parallel application, i.e., Monte-Carlo numerical simulations of stratospheric balloon envelope drift descent can take benefits of the characteristics of both kind of accelerators to achieve this operational challenge.

The paper is structured as follows: Section 2 summarizes the scientific background of the envelope drift descent. Section 3 introduces the concept of Monte-Carlo numerical simulation in the context of envelope drift descent analysis. Section 4 briefly presents computing accelerators. Section 5 presents the parallel implementation of our algorithm on the GPU, the Intel Xeon Phi and multi-core Intel Xeon CPU, and the different techniques that have been studied in order to optimize the parallel codes. Section 5 gives also a synthesis of these improvements and presents some general guidelines to address similar problems. Section 6 deals with the conclusion.

2 Application

In this section we present briefly the application and the physical model that governs the descent of a stratospheric balloon envelope. Then we present the method used to retrieve important parameters, i.e., atmospheric data and ballistic coefficient, in tables at each time step, and how we implement the Monte-Carlo perturbations. Finally we introduce the algorithm of the envelope drift descent.

2.1 Operation Context

Nowadays, governments impose stringent regulations to space agencies and meteorological offices concerning the recovery of their devices. In particular the French government requires the national space agency, CNES, to guarantee a safe area of

possible controlled descent during the flight of a stratospheric balloon, to prevent any human casualty or damage to property. Indeed, at the end of the flight of a stratospheric balloon, when the scientific mission is achieved, the envelope and the nacelle separate, and the nacelle, which can weight one metric tone or more, descends below a parachute while the envelope descends in the wind, both without control. To address this safety requirement, a statistical risk analysis must be performed as quickly as possible on the drift descent of the envelope of a stratospheric balloon, so as to define the recovery zone at the end of the flight. A common way is to perform a Monte-Carlo perturbation [19] of the initial conditions like the atmospheric parameters that present many uncertainties. Nevertheless, this study is time consuming. In particular, this type of study cannot be performed in operational conditions with classical CPUs. Moreover solutions like cluster computing are not well suited to operational conditions. Indeed, during balloon campaign, people usually embark all the computation means on the operational field (in Kiruna, North of Sweden or in Timmins, Canada for example), restricting the weight of the later. This particular conditions define what we are going to call our operational challenge: to be able to perform a Monte-Carlo analysis of a drift descent with only on-site computational means in operational computation time, i.e. before the flying balloon leaves the possible separation area.

2.2 Physical Model of the Envelope Drift Descent

In the model of the drift descent of the envelope, the position of the envelope is defined, in the local frame with $\mathbf{z}$ along the vertical direction, at any time step t by its altitude $z(t)$, its latitude $lat(t)$ and its longitude $lon(t)$. In this coordinate system, the motion of the envelope can be decomposed into two independents parts: the vertical motion and the horizontal motion.

2.2.1 Vertical Motion

The vertical movement of the envelope is modeled by the sum of two forces: gravitational force $\mathbf{g}$ and drag force $\mathbf{F_{Drag}}$ as shown in Fig. 1. The acceleration of the envelope is then computed according to the following Eq. (1):

$$a(t, z) = \frac{-1}{2}\rho(z)v(t)|v(t)| * B(t) - g, \tag{1}$$

where: $a(t, z)$, is the envelope vertical acceleration at time t and altitude z, in $\mathrm{m\,s^{-2}}$; $v(t)$, is the vertical speed of the envelope at time t, in $\mathrm{m\,s^{-1}}$; $\rho(z)$, is the density of the atmosphere at the altitude z, in $\mathrm{kg\,m^{-3}}$, computed from pressure and temperature via the ideal gas law; $B(t)$, is the ballistic coefficient of the envelope; g, is the Earth gravitational acceleration.

The new vertical position of the envelope, is computed at every time step by integrating (1), via Euler's method. We choose to use the Euler's method as the physical model is linear and does not present any numerical instability with the time step used for the calculation ($dt = 0.1\,\mathrm{s}$).

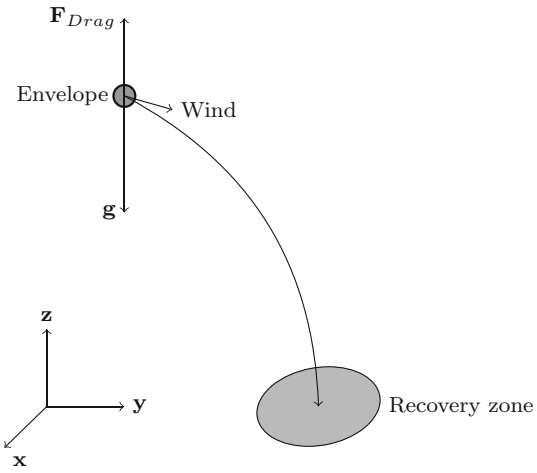

Fig. 1 Sketch of drift descent motion dynamics

2.2.2 Horizontal Motion

The horizontal movement of the envelope is only determined by the direction and the strength of the wind. Thus we define, U the speed of the wind along the zonal component (positive to the East), and V the speed of the wind along the meridional component (positive to the South), in $\mathrm{m\,s^{-1}}$. In the code, at every time step, the new position in latitude and longitude is computed , via Euler's method, according to Eqs. (2) and (3), assuming the hypothesis of a spherical Earth:

$$lat(t+dt) = lat(t) + \arcsin\left(\frac{V(z)dt}{R_T + z}\right), \tag{2}$$

$$lon(t+dt) = lon(t) + \arcsin\left(\frac{U(z)dt}{(R_T + z)\cos(lat(t))}\right), \tag{3}$$

where, R_T the Earth radius in m, z the vertical position.

2.3 Parameters Interpolation

For a given simulation we consider two tables of parameters. The first table contains the atmospheric profile, that represents the evolution of atmospheric parameters (pressure, temperature and wind) according to the altitude. These data are obtained via weather forecast. The second table contains the evolution of the envelope ballistic coefficient in function of the time. These data are obtained via reanalysis of previous flights and provide the ballistic coefficient of the envelope at given times after the begining of the drift descent.

In order to compute the Eqs. (1), (2) and (3) we need to compute the pressure, the temperature and the wind at the simulation current altitude and the ballistic coefficient at the simulation current time. This is done by interpolating those values in table: first

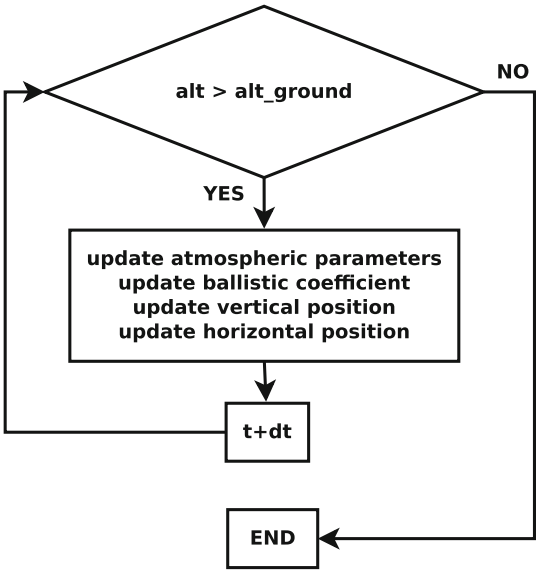

Fig. 2 Envelope drift descent calculation

we indentify the current altitude (time), then we go through the atmospheric table (ballistic coefficient table) to find out, in the table, the values that frame the current altitude (time) and we perform the interpolation between the levels, or we perform an extrapolation if the values are out of the bounds of the tables.

2.4 Envelope Drift Descent Algorithm

The numerical integration of an envelope drift descent at each time step, is composed of four parts, as shown in Fig. 2, and this loop is computed until the ground level (alt_ground) is reached:

- update the atmospheric parameters at the current altitude, via interpolated values from forecast data
- update the ballistic coefficient from the ballistic coefficient evolution table.
- update the vertical position via Eq. (1)
- update the horizontal position via Eqs. (2) and (3).

3 Monte-Carlo Simulations

3.1 Definition of the Monte-Carlo Simulations

We call Monte-Carlo simulations the numerical method composed of the following steps [19]:

- Definition of the initial conditions domain: for each parameter of the model, we define the domain that is accessible according to a certain distribution law.

– Generation of N sets of initial conditions according to informations defined previously.
– Computation of the N simulations from the sets of initial conditions defined previously: each simulation is then deterministic.
– Agregation of the results of N simulations and determination of statistical properties, like average, standard deviation,…of the results.

As we can see once the sets of initial conditions are defined, there is no communication between the simulations and even the generation of the sets of initial conditions is independant between simulations. Thus our Monte-Carlo simulations problem appears to be a pleasingly parallel problem that might benefit from the use of computing accelerators such as GPUs and coprocessors Intel Xeon Phi.

3.2 Application to the Envelope Drift Descent Analysis

In order to perform the Monte-Carlo analysis of the envelope drift descent we start, as explained in Sect. 3.1, by defining the initial condition domain. In the case of the envelope drift descent we consider perturbation on the atmospheric parameters (pressure, temperature and wind). As we wanted our method to be the less conservative on what concerns the size of the computed fallout area, a systematic perturbation of the atmospheric parameters is applied at each altitude level according to Eq. (4).

$$Parameter_{perturbed} = Parameter_{Initial} \pm \alpha_{parameter} * \sigma_{parameter}, \quad (4)$$

where: $\alpha_{parameter}$ is an uniformly distributed random value between 0 and 1. This value is generated at the beginning of the simulation and is different for each parameter of the stimulation; $\sigma_{parameter}$ is the standard deviation for the parameter computed comparing weather forecast data and true atmospheric conditions encountered by the balloon during its ascending phase. The sign of the perturbation is also randomly generated at the beginning of the simulation for each parameter of each simulation. This manner, each simulation generates a unique atmospheric profile.

4 Computing Accelerators

During the last decade a new set of computing devices, i.e., computing accelerators has emerged. These accelerators are highly parallel dedicated hardware used to perform computing functions faster than the CPU. To the best of our knowledge, computing accelerators, like GPU and coprocessor Intel Xeon Phi, have been successfully used in the solution of classic problems like linear algebra, image processing, combinatorial optimization and resolution of differential system [1–4,25]. Moreover few comparisons between the devices have been conducted, see [23,24]. The features of these new devices make them attractive for industrials to solve real world applications [16,22].

In the following sub-sections we introduce the two main families of computing accelerators: GPU and Intel Xeon Phi.

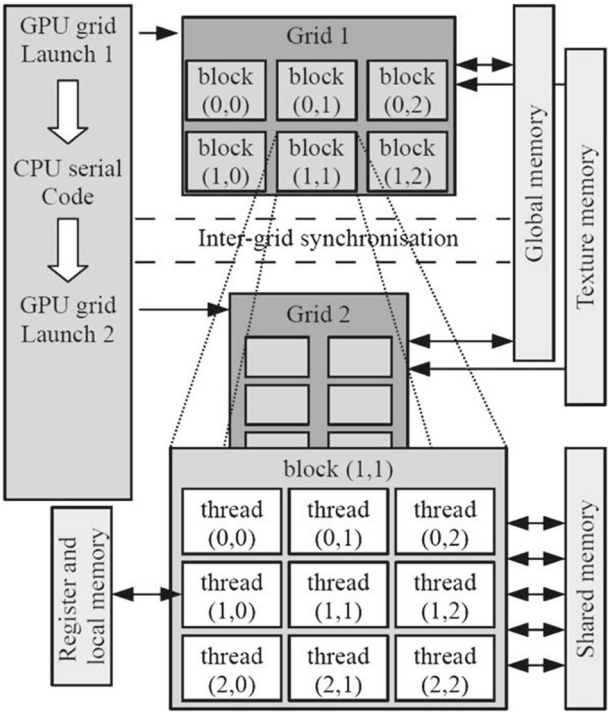

Fig. 3 Thread and memory hierarchy in GPUs

4.1 GPU

4.1.1 Architecture

Graphics Processing Units are highly parallel multithreaded, many-core architectures. They are better known for image processing. Nevertheless, NVIDIA introduced in 2006 CUDA (Compute Unified Device Architecture), a technology that enables programmers to use a GPU card to address parallel applications. Indeed, NVIDIA's GPUs are SIMT (Single Instruction, Multiple Threads) architectures, i.e. the same instruction is executed simultaneously on many data elements by the different threads. They are especially well-suited to address problems that can be expressed as data-parallel computations.

As shown in Fig. 3, a parallel code on the GPU (hereafter named the device), is interleaved with a serial code executed on the CPU (hereafter named the host). The parallel threads are grouped into blocks which are organized in a grid. The grid is launched via a single CUDA program, the so-called kernel [13].

4.1.2 Memory Hierarchy

The hierarchisation of the threads, presented in the previous subsection, is related to the memory hierarchy. There are three distinct levels of memory that are accessible for the programmer on a GPU:

– Global memory: accessible to every threads within the grid. This is the largest memory of the GPU (several GB), but exhibits the highest latency.
– Per block shared memory: accessible by each thread within a block. This memory cannot be accessed by a thread from another block.
– Per thread local memory: only accessible to one thread. This memory presents the lowest latency but is limited in size (few KB of storage).

4.1.3 GPU Synthesis

GPUs are massively parallel computing accelerators with few possibilities of vectorization. There is vectorization at Instruction Level Parallelism by placing consecutive operations in the pipeline if these operations are not data dependent [6]. The same task is executed inside the same group of 32 threads (called warp) of a given kernel but different warps of a given kernel can perform different tasks. The memory hierarchy permits one to optimize the data placement in order to reduce the memory access latencies. For this type of device the following points are particularly important.

– Provide enough threads to keep busy all the warps;
– Try to have non divergent threads in the same warp;
– Try to limit data transfer between the CPU and the GPU;
– Optimize data locality on the GPU (in order to take benefit of high memory bandwidth).

4.2 Xeon Phi

4.2.1 Architecture

In 2013, Intel released the Many Integrated Core (MIC) coprocessor: the Intel Xeon Phi also known by the code name Knights Corner during its early phase of development. The coprocessor consists of up to 61 processor cores, interconnected by a high-speed bidirectional ring (see Fig. 4) [11]. The architecture of a core is based on a modified x86 Pentium 54C cores and contains a dedicated 512-bit wide Streaming SIMD Extensions vector unit [5]. Each core can execute four hardware threads (two per clock cycle and per ring's direction). Like the GPUs, the Intel Xeon Phi is connected to the CPU via the PCI-Express connector. The memory controllers and the PCIe client logic provide a direct interface to the GDDR5 memory on the coprocessor and the PCIe bus, respectively. The design of the coprocessor permits one to run existing applications parallelised via OpenMP or MPI.

4.2.2 Execution Mode

Unlike GPUs, there are three different ways to execute an application on the Xeon Phi: the native, offload and hybrid modes (see Fig 5) [11].

The native execution mode, consists in taking advantage of the fact that the Xeon Phi has a Linux micro OS running in it, and appears as another machine connected to the host which can be reached via a ssh connection. For this mode, the application and the input files are copied into the Xeon Phi, and the application is launched from it. In the

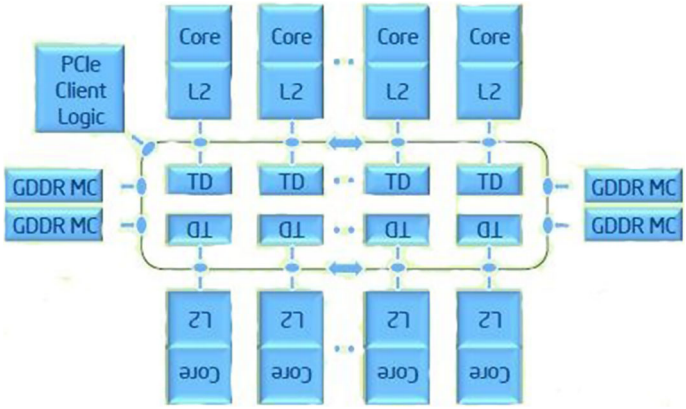

Fig. 4 Microarchitecture of the MIC coprocessor

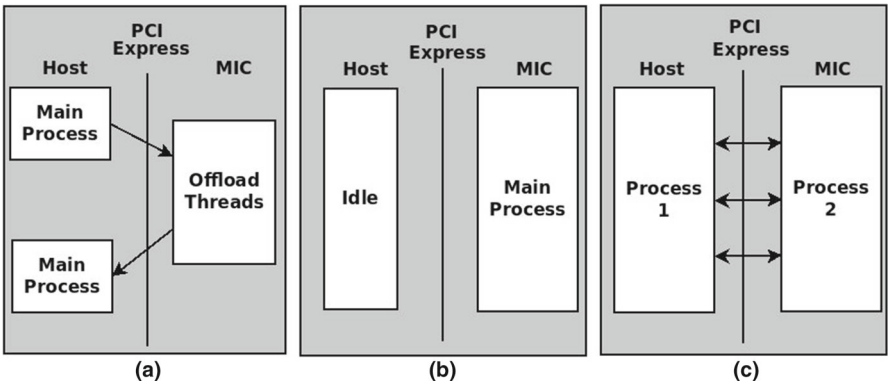

Fig. 5 Intel Xeon Phi execution modes: **a** offload mode, **b** native mode and **c** symmetric mode

beginning, the sequential part of the application runs on one core of the coprocessor, then the parallel part of the application is deployed over the different cores of the coprocessor. Existing parallel code running on CPU or cluster can be compiled with the *-mmic* option in order to run natively on the Phi without any modifications.

The offload execution mode or heterogeneous programming mode, is similar to what is done for GPU: the application starts on the host and runs a sequential part of the code. Then the CPU sends data to the Xeon Phi, and launches the parallel computation on the coprocessor.

The symmetric mode is similar to the offload mode, except that a part of the parallel computation is also performed on the host. The code needs to be compiled twice: for the host and for the coprocessor. In this mode, one of the major challenges is to properly balance work between the host and the Phi.

4.2.3 Xeon Phi Synthesis

Intel Xeon Phi are parallel computing coprocessors. They compensate their low parallelism (with regards to GPUs), by their ability to vectorize computations (SIMD

model). The same task is executed for a vector, i.e. an OpenMP process. Nevertheless different OpenMP processes can perform different tasks. For these devices, the following points are important:

– provide enough threads to keep busy all the cores;
– take advantage of the vector units;
– take care of data locality.

5 Parallel Implementations on CPU, GPU and MIC

The Monte-Carlo application studied in this paper belongs to the class of pleasingly parallel applications, since each drift descent simulation is independent. As explained in [8] this important class represents more than 20% of the total number of parallel applications and this proportion is growing.

In this section, we present the experimental setups and protocol and we describe for the different architectures, i.e., GPU, Xeon Phi and CPU, the basic parallel algorithms and the main steps of optimization of the parallel codes showing each time the gain obtained. Finally, we display and analyze the computing times of the parallel codes on the different computing systems.

5.1 Description of the Experimental Conditions

5.1.1 Hardware

We consider three computing systems (see Table 1).

– A cluster node, composed of two Intel Xeon E5-2680 v2 at 2.8 GHz with 10 physical cores each and an Intel Xeon Phi Coprocessor 7120P, with 61 cores at 1.238 GHz and 16 GB memory.
– A computing system with a NVIDIA Tesla K40, with 2880 CUDA cores at 0.745 GHz and 12 GB memory.
– A computing system with a NVIDIA Tesla K80, with 4992 CUDA cores at 0.875 GHz and 24 GB memory.

5.1.2 Software

The GPU implementation of the code is performed using CUDA 7 [14]. The Xeon Phi implementation of the code is performed using OpenMP 4.0 and compiled using the Intel Compiler version 14.0.1 (Intel Parallel Studio XE 2015).

5.2 Experimental Protocol

In this paper, we present average computing times for ten instances of each problem. The measurement starts with the loading of the first data, and stops when the

Table 1 Description of devices characteristics

Name	Arch.	Cores	Clock (GHz)	Total memory (MB)	Memory bandwdith max (GB/s)	Vector unit	Compute capability
K40	Kepler	2880	0.745	12,000	288	N.A.	3.5
K80	Kepler	4992	0.875	24,000	480 (240/GPU)	N.A.	3.7
Xeon Phi 7120P	N.A.	61	1.24	16,000	352	AVX-512 (512-bit SIMD)	N.A.
Xeon E5-2680 v2	N.A.	20	2.8	25 (cache/proc)	59.7	AVX (256-bit SIMD)	N.A.

Table 2 GPU threads repartition

Number of threads per block	1024
Number of blocks	99

Table 3 Xeon Phi threads repartition

Number of OMP threads	244
SIMD vector size	8

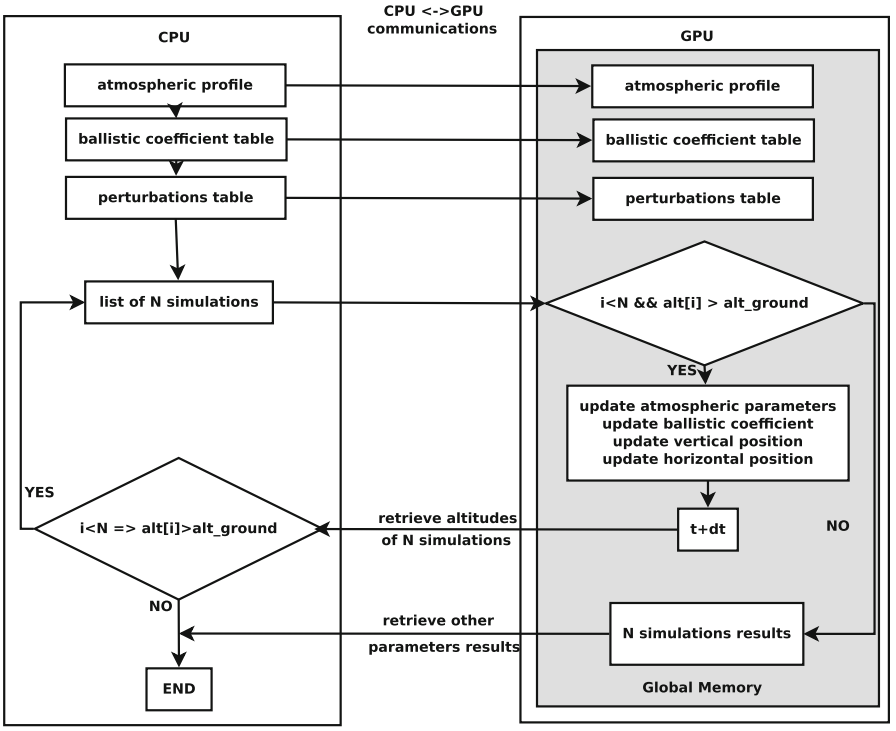

Fig. 6 GPU loop parallel algorithm

results of the last simulation are gathered. Computations are performed using double precision floating point operations. In all experiments, we use all the available cores on the CPU and Xeon Phi, and for the Xeon Phi each core runs the maximum number of threads supported (4 threads per core). As the parallel threads of our application are independent, we set the KMP_AFFINITY of the MIC and of the CPU to "*granularity=fine*" [10,18]. The threads repartition used on the GPU and Xeon Phi for these experiments is summarized in Tables 2 and 3.

Table 4 Computation time for 100,800 simulations on the K40 GPU

Parallel algorithms	Overall time (s)	Communication time (s)	Computation time (s)
Loop parallel algorithm	21.49	17.39	4.1
Task parallel algorithm	3.03	0.12	2.91
Task parallel algorithm with memory management	2.79	0.11	2.68

5.3 Parallelization on the GPU

In this subsection, we show how we have implemented the parallel algorithm on the GPU in order to take advantage of its massive parallelism capability.

5.3.1 Loop Parallel Algorithm

We begin the parallelization of our application on the GPU, using a loop parallel algorithm displayed Fig. 6, and referenced as Loop parallel algorithm in Table 4. We decide to start with this algorithm, since it is typically the kind of algorithm one gets when using GPU numerical solver of Ordinary Differential Equations library such as odeint [12], with complex models that necessitate to update data between two time steps, like in our case with atmospheric data. In this algorithm the CUDA kernel corresponds to a single iteration of the while loop shown in Fig. 2. This way, the lifetime of the CUDA kernel corresponds to one time step of the numerical integration, one thread is associated with one simulation and at each launch of the kernel we instantiate as many CUDA threads as there are simulations to perform. A new kernel is launched at each step. In order to ensure persistence of the data between two time steps, every variable is stored in the global memory. The atmospheric data, the ballistic coefficient table and the perturbations parameters that are generated by the CPU are sent to the GPU only once, before the first time step. At each time step, only the vector containing the altitude of all the simulations is copied back to the CPU in order to evaluate the advance of the simulations. The other results are copied back to the CPU when all simulations are performed. This algorithm permits one to compute 100,800 simulations (3150 warps of 32 threads) of envelope drift descent in 21.49 s.

5.3.2 Limitations of Communications

Then we start investigating the performance of our loop parallel algorithm using nvprof. It appears that most of the time is spent copying data between the device and the host, i.e., 17.93 s (80.92% of overall execution time), while the kernel execution only counts for 4.1 s (19.08% of overall execution time). In order to reduce the

communications, we decide to switch from loop parallelism to task parallelism. We redesign the kernel so that each CUDA thread performs the main while loop presented Fig. 2, instead of only one time step. This algorithm is referenced as Task parallel algorithm in Table 4. As a result, we perform only two communications between the CPU and the GPU: one at the beginning and one at the end of the application, and we only launch one kernel. Also, instead of generating the perturbation coefficients on the CPU and copying them on the GPU, we use the cuRAND library so that each CUDA thread generates it's own set of perturbation coefficients. Changing the code from the algorithm presented in the previous sub-section to this one requires little code modification as it only requires to modify three lines of code. This new algorithm performs 100,800 simulations in 3.03 s, resulting in a reduction of the computation time by a factor of 85.9%. The communications between the CPU and the GPU only count for 0.04% (0.12 s) of the execution time, while the kernel counts now for 99.86% (2.91 s). It is interesting to point that the kernel execution time is 2.91 s, while it was 4.1 s with the loop parallel algorithm. This reduction of 29.3% of the execution time of the kernel is due to the instantiation of only one kernel instead of the instantiation of 19,039 kernels as in the case of the loop parallel algorithm.

5.3.3 Memory Management

In this subsection, the locality of data is taken into account and we present a parallel algorithm for which a local copy of all the data (atmospheric and ballistic) is performed, moreover all variables are stored in the local memory. This way, the vector state of the balloon envelope (containing the position and the speed), which is updated at each computation (see Fig. 2), resides in the thread local memory reducing memory access latency. Only the final step of each simulation is stored in the global memory. This algorithm is referred to as Task parallel algorithm with memory management in Table 4. The modification of the code between the one used in the previous subsection and this one are important as they implied to redefine every variables used in the kernel, manage copy of data from global to local memory. Also, in order to ensure that each thread keeps the state vector in its local memory we use *cudaFuncCachePreferL1* in order to increase the thread L1 memory to 48 KB. The resulting gain in performance is of 5%, while the local memory overhead (i.e. the ratio of local memory traffic to total memory traffic) [15] increases from 0.06 to 92.79% as compared with the previous implementation. This algorithm is presented Fig. 7. We note that for our pleasingly parallel application we have used local and global memory and never used shared memory.

5.3.4 Multi-GPU Consideration

We have also considered the performance of the algorithm presented in the previous sub-section on a NVIDIA Tesla K80. This GPU composed of 4992 CUDA cores consists of two GK210 cards, each having 2496 CUDA cores and its own memory system that needs to be adressed separately. From a programming point of view, things are as if two GPUs were connected to the host. We note that we don't have any communication between GPUs since we are considering an embarassingly parallel

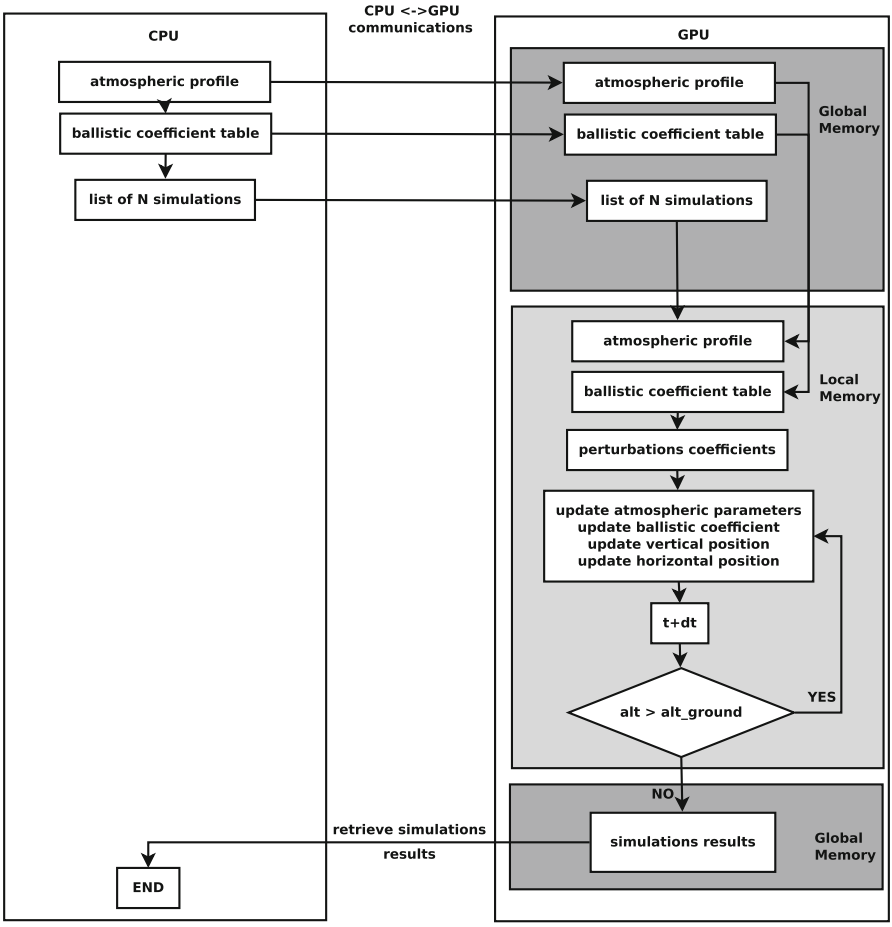

Fig. 7 GPU task parallel algorithm with memory management

Data: atmospheric profile and ballistic coefficient table
for *each GPU* **do**
    select a GPU;
    copy data on the current GPU;
    launch kernel on the current GPU;
end
synchronyze GPUs;
copy results from GPUs to CPU;

Algorithm 1: Task parallel algorithm in the multi-GPU case

problem. Thus, for the tests on the K80 card, we only modified the CPU implementation without changing the GPU implementation presented in the previous sub-section. On the GPU the only difference with what was presented previously is that a kernel will only compute N/2 simulations. We copy the atmospheric profile and the ballistic coefficient table on both GK210 cards. Then we take advantage of the asynchronism

Table 5 Computation time for 101,504 simulations on Xeon Phi

Parallel algorithms	Time (s)
Task parallel algorithm	29.98
Task parallel algorithm with vectorization	23.06
Task parallel algorithm with vectorization and memory management	8.02
Task parallel algorithm with memory management and fixed size vectorization	6.86

Data: atmospheric profile and ballistic coefficient table
#pragma omp for
for *each simulation* **do**
  compute simulation;
end

Algorithm 2: Task parallel algorithm on the Xeon Phi

between the CPU and GPU execution of the code: by default, after the launch of a kernel, the CPU code runs the next instruction without waiting the end of the kernel. So we perform a loop on the CPU on the amount of connected GPUs (here two) in order to launch the kernel on the GPUs. Then, in order to retrieve the results of both GPUs, we block the execution of the CPU code until the GPUs end their computations with *cudaDeviceSynchronize()*. These modifications are summarized in Algorithm 1. This algorithm permits one to compute 100,800 simulations of envelope drift descent in 1.4 s on the K80. We note that we can easily generalize this approach to several multi-GPU configurations.

5.4 Parallelization on the Xeon Phi

In this subsection, we show how we have implemented the parallel code on the Xeon Phi in order to take advantage of it's massive vectorization capability.

5.4.1 Task Parallel Algorithm

For the implementation of the application on the Xeon Phi, we decide to use the native programming mode, in order to avoid communications between the CPU and the Xeon Phi. Doing this, it seems natural to adopt a task parallel algorithm (referenced as Task parallel algorithm in Table 5). All the vectors are allocated using the instruction $posix\_memalign()$, and aligned on 64 bits. The parallelization across the Phi, is ensured via OpenMP as presented in Algorithm 2. We instantiate 244 OpenMP processes (i.e. 61×4 processes). Each process has to perform sequentially M times the while loop presented Fig. 2, i.e. M envelope drift descent simulations, where M is selected so that M times the number of OpenMP processes equals the total number of simulations to perform. The variables of the simulations are stored in vectors

(a)

```
Data: A[N]
Data: B[N]
Data: C[N]

for i ← 0 to N do
    C[i] = A[i] + B[i]
end
```

(b)

```
Data: A[N]
Data: B[N]
Data: C[N]

#pragma simd
for i ← 0 to N do
    C[i] = A[i] + B[i]
end
```

(c)

```
Data: A[N]
Data: B[N]
Data: C[N]
Input: NbOmpProcess
#pragma omp for
for i ← 0 to N/NbOmpProcess do
    #pragma simd
    Input: OmpProcessId
    Data: StartIter = OmpProcessId *
          N/NbOmpProcess
    for j ← StartIter to StartIter+
    N/NbOmpProcess do
        C[j] = A[j] + B[j]
    end
end
```

Algorithm 3: Illustration of the SIMD vectorization with a vectorial addition: (a) sequential for loop, (b) vectorized for loop, (c) parallelized and vectorized for loop

```
Data: atmospheric profile and ballistic coefficient table
#pragma omp for
for each simulation do
    #pragma vector aligned
    #pragma ivdep
    #pragma simd
    compute vector simulations;
end
```

Algorithm 4: Task parallel algorithm with vectorization on the Xeon Phi

shared among the OpenMP processes. The perturbation table is generated by each OpenMP process for the M simulations. The uniqueness of each perturbation is guaranteed by the management of the seed of the random number generator that depends on the rank of the OpenMP process. This algorithm computes 101,504 simulations in 29.98 s.

5.4.2 Vectorization

The above implementation does not fully use the capacity of the Xeon Phi as there is no vectorization. In this subsection we present how we have vectorized the code using the OpenMP 4.0 SIMD vector model. This model permits one to vectorize the computation of a 'for' loop. For the simplicity of presentation, we illustrate SIMD vectorization with a vectorial addition: let's consider three vectors A, B and C with N elements each and the vectorial addition C = A + B. In the sequential case, one writes a loop on the number N of elements of A, B and C as presented in the Algorithm 3.(a). This way, the addition is performed one vector element 'i' at a time. In order to vectorize this addition one can simply add OpenMP 4.0 SIMD directive as shown in Algorithm 3.(b). This way, instead of being performed sequentially, the addition will be performed via the vector unit for all the elements of the vectors A and B if the vector size is lower or equal to the dimension of the vector unit (512 bits in the case of the

```
Data: atmospheric profile and ballistic coefficient table
#pragma omp for
for each simulation do
    copy data;
    #pragma vector aligned
    #pragma ivdep
    #pragma simd
    compute vector simulations;
end
```

Algorithm 5: Task parallel algorithm with vectorization and memory management on the Xeon Phi

AVX-512); otherwise, the computation is sequentiallized with sub-vectors of 512 bits in the case of the AVX-512. The later situation (see Algorithm 3.(c)) corresponds to the case where we combine parallization and vectorization. This way, each OpenMP process, identified by its OpenMP Id: 'OmpProcessId', works on sub-vectors with size N / NbOmpProcess, where NbOmpProcess is the number of OpenMP processes. The parallelization and vectorization strategy carried out in this paper corresponds to Algorithm 3.(c). We also use the compiler directive *#pragma vector aligned*, in order to guarantee to the compiler that all memory accesses are aligned. This new algorithm is referenced as Task parallel algorithm with vectorization in Table 5 and is presented Algorithm 4. Vectorization of the code implies a heavy modification of the code as each line of computation needs to be rewritten in a vector way. The size of the vectors is defined at the application execution and is set to 8, because the size of the vector unit of the Xeon Phi is 512 bits (AVX-512) and we work with double precision arithmetics (64 bits). Thus, to compute 101,504 simulations, the application launches 244 OpenMP processes that compute 416 simulations each, launching 52 blocks of 8 vectorized simulations; this is performed in 23.06 s.

5.4.3 Memory Management

Analyzing the performance of the previous algorithm, it appears that the vectorization was not efficient due to the time spent accessing data in the global memory of the Xeon Phi. In oder to address this issue, we store state vectors (containing the position and speed of the envelope), atmospheric profile and ballistic coefficient table in the local memory of each OpenMP process creating private variables as presented in Algorithm 5. This type of parallel algorithm (referenced as Task parallel algorithm with vectorization and memory management in Table 5) computes 101,504 simulations in 8.02 s. We note that rewriting the code from the task parallel algorithm to the task parallel method with vectorization and memory management necessitates the modification of 99 lines of codes (the non vectorized code has around 500 lines).

5.4.4 Fixed Size Vectorization

During the process of optimization of the application, we fix vector size to 8 beforehand, i.e. before compilation. This corresponds to the Algorithm with memory

management and fixed size vectorization in Table 5. Thanks to this modification, the application computes 101,504 simulations in 6.86 s. It appears that this gain is due to the fact that if the size of the vector is not set at the compile time, then the Intel compiler adds extra control features that slowdown the application; thus, fixing vector size prevents this behavior.

5.5 Parallelization and Vectorization on the CPU

We note that the parallel algorithm implemented on the CPU is the same as the one implemented on the coprocessor Xeon Phi. De facto, the CPU implementation is multi-threaded, vectorized and addresses memory locality issues. The code is compiled without specific -mmic compilation option and with AVX compilation options (-mtune=core-avx-i -axCORE-AVX-I -mavx). Computational experiments show that 8 is not the vector size that gives the best performance since the Xeon E5-2680-v2 L1 cache is wider and more efficient than the one of the Xeon Phi. We obtain the best performance, i.e., the best compromise between the cache efficiency and the dimension of the vector unit, using vectors of size 128. This algorithm computes 102,400 simulations (20 OpenMP processes that compute 5120 simulations each, launching 40 blocks of 128 vectorized simulations) in 13.31 s.

5.6 Operational Challenge

For stratospheric balloon studies the operational challenge consists in computing very fast, many envelope drift descent simulations (enough to obtain a statistically unbiased result) before balloon launch or during the balloon flight. In particular during flight this analysis appears as a decision support and must be performed as fast as possible. Moreover, such computation must be performed via computing systems that can be easily embedded on the operation site, e.g. laptop or even central unit, in order to prevent connection issues.

Figure 8 displays the computing time on the different platforms for several numbers of simulations. As we can see, the K80, the K40 and the Xeon Phi 7120P, address the operational challenge, since 1,000,000 simulations require respectively 11.84, 32.4 and 64.2 s, while they require 130.5 s on a computing node with two Intel Xeon. Thus, these devices appear to be more efficient than a standard cluster node. Moreover, those devices can be plugged on a computer, and their electric consumption of around 300 W (PDT of 300 W for the Phi 7120p and the K80, and 280 W for the K40), make them suitable too operational conditions.

For the operational challenge the rapidity of the computation is not the only criterion since the accuracy of the provided solution is also very important. In order to control the accuracy of our technique of determining the probable zones of fallout of stratospheric balloon, we analyze data of the 6 flights of the 2014 CNES stratospheric balloon campaign at Timmins, Canada. For each flight of this campaign we consider the end of flight conditions (atmospheric profile and the uncertainties on this profile, the separation position in altitude, latitude and longitude) and perform 1,000,000 simulations. Then we use the method presented in references [7,20] in order to determine

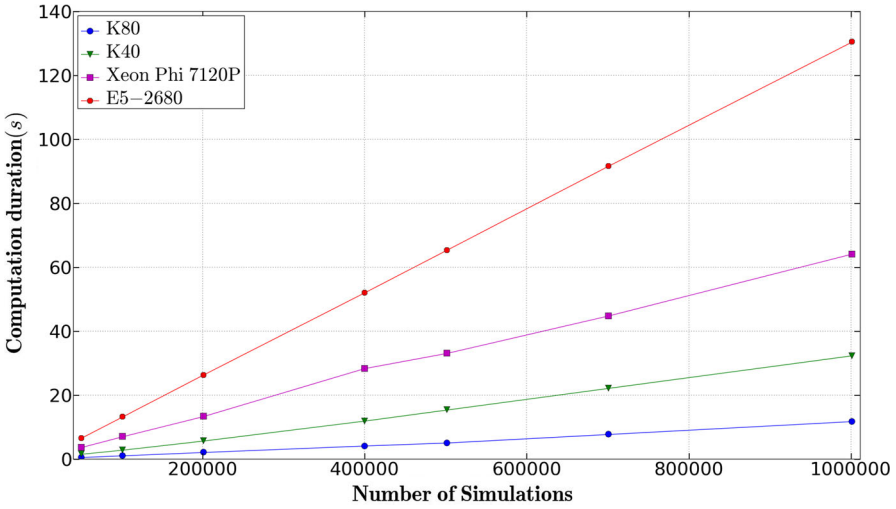

Fig. 8 Computation duration versus number of simulations

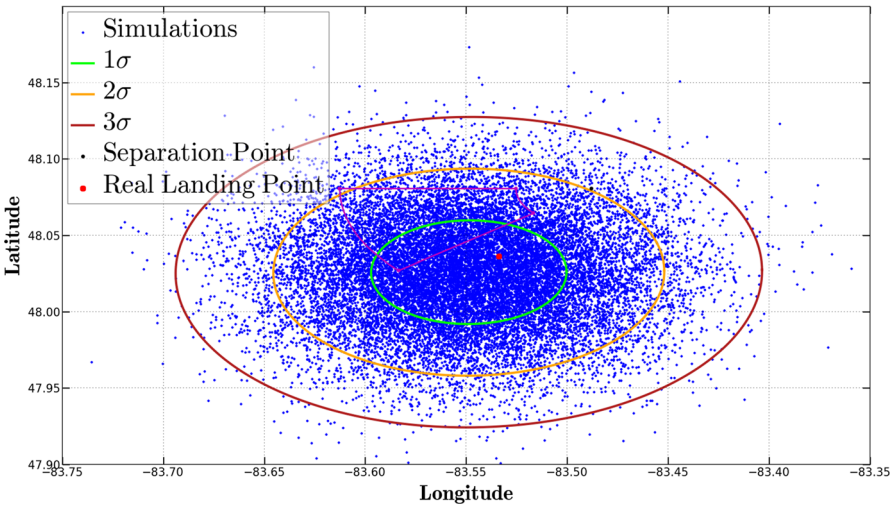

Fig. 9 Fallout area determined for the first flight of 2014 Timmins stratospheric balloon flight campaign

the 1, 2 and 3 σ confidence ellipses on the fallout area. Figure 9 represents results for the first flight of this campaign. As we can see the real landing point (red) is inside the 1 σ confidence ellipse. For the others flights, for three of them landing points were also in the 1 σ confidence ellipse while for the other two landing points were in the 2 σ confidence ellipse. These first results are very promising in terms of accuracy of the results provided.

5.7 Synthesis and Guidelines

We see that the parallel implementation on GPU and Xeon Phi of the Monte-Carlo simulation of envelope drift descent, permits one to reduce the computing time and to fulfill operational conditions. Indeed, the computing time on the K40 and K80 GPUs are respectively about four and ten times faster than the one obtained with a parallel code on the two E5-2680 CPU; similarly, the computing time on the Xeon Phi 7120P is twice as fast.

On what concerns specifically code optimization, we note that reductions in computing time have been obtained thanks to the following improvements:

- **GPU implementation** the reduction of communications between the host and device along with the reduction of number of kernels launched per simulation, changing our algorithm from a loop parallel to a task parallel approach, was very efficient. This has led to a 86% reduction in the computing time. In particular, such modifications permit one to dramatically reduce the synchronizations between threads since the application is pleasantly parallel.
- **Xeon Phi implementation** the combination of memory management and vectorization is the key issue in the Xeon Phi case. Indeed, this leads to a 73% reduction in the computing time. However we note that without memory management such an important reduction in computing time could not be obtained. Also imposing the size of vectors before compilation appears to reduce the computing time by 14%.

We showed also the necessity to reconceptualize in a vectorial way scalar algorithm on Xeon Phi which requires a lot of programing effort as it implies to fully rewrite all the code and manage specific functions that do not support vectorization like *rand()* function. On the other hand, we showed that implementation on the GPU requires less reconceptualization and thus less programming effort than on the Xeon Phi.

We believe that these guidelines are also valid for other applications related to numerical integration and more generally to numerical simulation.

6 Conclusion

The implementations of our parallel algorithm on the GPUs K40 and K80 are respectively 4 and 10 times faster than a parallel and vectorized implementation on the two sockets CPU E5-2680-v2 while the Xeon Phi 7120P is twice as fast as a vectorized and parallel CPU implementation. In order to obtain these results it is crucial to properly consider the architectures of both accelerators and to design the codes accordingly, i.e., take advantage of the massive parallelism ability of the GPUs and use the parallelism combined with massive vectorization abilities of the Xeon Phi. Also, a good understanding of the physical problem permits one to optimize the data locality and hence to improve the performance of the parallel application.

Computing accelerators appear to be very serious alternatives to clusters in order to solve real world problems in operational conditions. The application presented in the present paper is an illustration of a numerical integrator with Monte-Carlo perturbation of initial conditions. This is a general class of problems which has many

fields of applications such as atmospheric reentry of satellites [17], ballistic trajectory prediction [9], ray tracing of GNSS [6], parafoil automatic guidance [21], and we believe that thanks to the guidelines, presented in this paper, many of these applications could strongly benefit of computing accelerators.

As the methodology presented in this paper can be applied to other applications, by replacing the core of the algorithm, we are now working on generalizing our approach and designing a library that will be made available to the community.

Acknowledgements Dr. Didier El Baz and Dr. Bastien Plazolles gratefully acknowledge the support of NVIDIA Corporation with the donation of the Tesla K40 GPU used for this research work. The authors wish also to thank Dr. D. Gazen and Dr. J. Escobar of Observatoire Midi-Pyrénées for their advices and the access to the cluster in Toulouse. The authors thank the DEDALE work group coordinated by CNES, France. Finally, the authors thank the reviewers for their useful suggestions in order to improve the manuscript.

References

1. Aldinucci, M., Pezzi, G.P., Drocco, M., Spampinato, C., Torquati, M.: Parallel visual data restoration on multi-gpgpus using stencil-reduce pattern. Int. J. High Perform. Comput. Appl. **29**(4), 461–472 (2015)
2. Boyer, V., El Baz, D., Elkihel, M.: Solving knapsack problems on GPU. Comput. Oper. Res. **39**(1), 42–47 (2012). doi:10.1016/j.cor.2011.03.014. http://www.sciencedirect.com/science/article/pii/S0305054811000876. Special Issue on knapsack problems and applications
3. Boyer, V., El Baz, D.: Recent advances on GPU computing in operations research. In: Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International, pp. 1778–1787 (2013). doi:10.1109/IPDPSW.2013.45
4. Cuomo, S., Michele, P.D., Galletti, A., Marcellino, L.: A parallel pde-based numerical algorithm for computing the optical flow in hybrid systems. J. Comput. Sci. (2017). doi:10.1016/j.jocs.2017.03.011. http://www.sciencedirect.com/science/article/pii/S1877750317303010
5. Farber, R.: Programming Intel's Xeon Phi: a jumpstart introduction. http://www.drdobbs.com/parallel/programming-intels-xeon-phi-a-jumpstart/240144160
6. Gegout, P., Oberle, P., Desjardins, C., Moyard, J., Brunet, P.M.: Ray-tracing of GNSS signal through the atmosphere powered by CUDA, HMPP and GPUs technologies. IEEE J. Sel. Top. Appl. Earth Obs. Remote Sens. **7**(5), 1592–1602 (2014). doi:10.1109/JSTARS.2013.2272600
7. Hoover, W.E., States., U.: Algorithms for confidence circles and ellipses [microform]. U.S. Dept. of Commerce, National Oceanic and Atmospheric Administration, National Ocean Service Rockville, MD (1984)
8. Hwang, K., Fox, G.C., Dongarra, J.: Distributed and Cloud Computing: From Parallel Processing to the Internet of Things, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco (2011)
9. Ilg, M., Rogers, J., Costello, M.: Projectile Monte-Carlo trajectory analysis using a graphics processing unit. AIAA Atmos. Flight Mech. Conf. (2011). doi:10.2514/6.2011-6266
10. Intel: Thread affinity interface. https://software.intel.com/en-us/node/522691#KMP_AFFINITY_ENVIRONMENT_VARIABLE
11. Jeffers, J., Reinders, J.: Intel Xeon Phi Coprocessor High-Performance Programming. Morgan Kaufmann, Burlington (2013)
12. Karsten, A., Mario, M.: Odeint. http://headmyshoulder.github.io/odeint-v2/
13. NVIDIA: Nvidia. CUDA 7.0 programming guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
14. NVIDIA: Nvidia. CUDA 7.0. https://developer.nvidia.com/cuda-toolkit
15. NVIDIA: Profiler user's guide. http://docs.nvidia.com/cuda/profiler-users-guide/#nvprof-overview
16. Pennycook, S.J., Hughes, C.J., Smelyanskiy, M., Jarvis, S.A.: Exploring SIMD for molecular dynamics, using Intel Xeon processors and Intel Xeon Phi coprocessors. In: Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, IPDPS '13. pp. 1085–1097. IEEE Computer Society, Washington, DC, USA (2013). doi:10.1109/IPDPS.2013.44

17. Plazolles, B., Spel, M., Rivola, V., El Baz, D.: Monte-Carlo analysis of object reentry in earth s atmosphere based on taguchi method. In: Proceedings of the 8th European Symposium on Aerothermodynamics for Space Vehicle, Lisbon (2015)
18. Rahman, R.: Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers, 1st edn. Apress, Berkely (2013)
19. Robert, C.P., Casella, G.: Monte-Carlo Statistical Methods. Springer, New York (2004)
20. Rocchi, M.B.L., Sisti, D., Ditroilo, M., A. Calavalle, R.P.: The misuse of the confidence ellipse in evaluating statokinesigram. Ital. J. Sport Sci. **12**(2), 169–171 (2005). http://hdl.handle.net/11576/2504321
21. Rogers, J., Slegers, N.: Robust parafoil terminal guidance using massively parallel processing. AIAA Atmos. Flight Mech. Conf. (2013). doi:10.2514/6.2012-4736
22. Saini, S., Jin, H., Jesperson, D., Cheung, S., Djomehri, J., Chang, J., Hood, R.: Early multi-node performance evaluation of a knights corner (KNC) based NASA supercomputer. In: IEEE 24th International Heterogeneity Computing Whorkshop (2015)
23. Saule, E., Kaya, K., Çatalyürek, Ü.V.: Performance evaluation of sparse matrix multiplication kernels on Intel Xeon Phi. CoRR abs/1302.1078 (2013). arxiv:1302.1078
24. Teodoro, G., Kurc, T., Kong, J., Cooper, L., Saltz, J.: Comparative performance analysis of Intel (R) Xeon Phi (TM), GPU, and CPU: a case study from microscopy image analysis. In: Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium, IPDPS '14, pp. 1063-1072. IEEE Computer Society, Washington, DC, USA (2014). doi:10.1109/IPDPS.2014.111
25. ul Hasan Khan, A., Al-Mouhamed, M., Firdaus, L.: Evaluation of Global Synchronization for Iterative Algebra Algorithms on Many-Core. In: 2015 16th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD). pp. 1–6 (2015). doi:10.1109/SNPD.2015.7176173