Parallel Monte-Carlo Simulations on GPU and Xeon Phi for Stratospheric Balloon Envelope Drift Descent Analysis

Bastien Plazolles*†, Didier El Baz*, Martin Spel†, Vincent Rivola†, and Pascal Gegout‡
*LAAS-CNRS, Université de Toulouse, CNRS, Toulouse, France
Email: elbaz@laas.fr
†R.Tech - Parc Technologique Delta Sud, 09340 Verniolle, France
Email: bastien.plazolles@rtech.fr
martin.spel@rtech.fr
vincent.rivola@rtech.fr
‡Géosciences Environnement Toulouse (CNRS UMR5563), 14 avenue Edouard Belin, F-31400 Toulouse, France
Université de Toulouse, F-31400 Toulouse, France
Email: pascal.gegout@get.omp.eu

***Abstract*—A performance evaluation of parallel Monte-Carlo simulations on GPU and MIC is presented and the application to stratospheric balloon envelope drift descent is considered. The experiments show that GPU and MIC permit one to decrease computing time by a factor of 4 and 2, respectively, as compared to a parallel code implemented on a two sockets CPU (E5-2680-v2) which allows us to use these devices in operational conditions.**

***Keywords*-parallel computing; multi-core CPU; CUDA; OpenMP; GPU; Xeon Phi; numerical integrator; Monte-Carlo perturbations;**

I. Introduction

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 without control. To address this safety requirement, a statistical risk analysis must be performed 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 [1] 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.

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 0.1s to compute a single drift descent trajectory, i.e. an unique landing point. In its present form this code cannot be used to perform a Monte-Carlo simulation of the drift descent, as 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 GPU and MIC, that can easily be transported on operational field and that do not require too much energy. In the sequel, we keep the algorithm of the CNES operational code. We show how our parallelization strategy can take benefits of the characteristics of both kind of accelerators to achieve this operational challenge.

The paper is structured as follows: Section 2 deals with computing accelerators. Section 3 summarizes the scientific background of the envelope drift descent and the method used for the Monte-Carlo analysis. Section 4 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 code. Section 4 gives also a synthesis of these improvements and presents some general guidelines to address similar problems. Section 5 deals with the conclusion.

II. Computing Accelerators

During the last decade a new set of computing accelerators has emerged. These accelerators are highly parallel dedicated hardware used to perform computing functions faster than

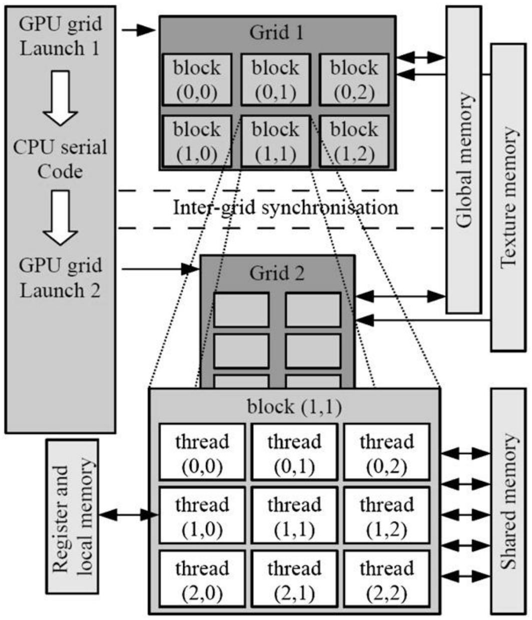

Figure 1. Thread and memory hierarchy in GPUs

the CPU. To the best of our knowledge, computing accelerators, like GPU and Xeon Phi, have been successfully used in the solution of classic problems like linear algebra or image processing [2], [3]. Moreover few comparisons between the devices have been conducted, see [4] and [5]. The features of these new devices make them attractive for industrials to solve real world applications [6], [7].

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

A. GPU

1) Architecture: Graphics Processing Units (GPUs) 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. 1, 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 [8].

2) Memory hierarchy: The hierarchisation of the threads, presented in the previous subsection, is related to the memory hierarchy. There are 3 distinct levels of memory 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).

3) Future developments: In 2016, NVIDIA plans to release the new PASCAL architecture exhibiting a higher bandwidth memory (up to 1TB/s), two times more flops than current Maxwell architecture (with an equivalent number of CUDA cores), more memory and more mixed-precision computing unit. Also, aware that one of the biggest caveat of the GPU computing is the latency induced by the transfer of data between the CPU and the GPU via the PCI-Express link, NVIDIA will add to the Pascal architecture NVLink. The new technology NVLink will let data move between GPUs and CPUs five to twelve times faster, than with current PCI-Express link (PCI-E 3.0) thanks to a higher bandwidth [9]. Also Pascal architecture will take advantage of HBM Gen2 stacked memory; denser memory chips will permit to increase HBM memory from 16 up to 32 GB, with a bandwidth of 1 TB/s.

4) GPU synthesis: GPUs are massively parallel computing accelerators with few possibilities of vectorization (vectorization at Instruction Level Parallelism by placing consecutive operations in the pipeline if these operations are not data dependent [10]). 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 optimize the memory access latencies. For these devices, it is important to provide enough threads to keep busy all the warps, to try to have non divergent threads in the same warp, to try to limit data transfer between the CPU and the GPU and to optimize data locality on the GPU.

B. Xeon Phi

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 is composed of up to 61 processor cores, interconnected by a high-speed bidirectional ring (see Fig. 2) [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 [12]. Each core can executes four hardware

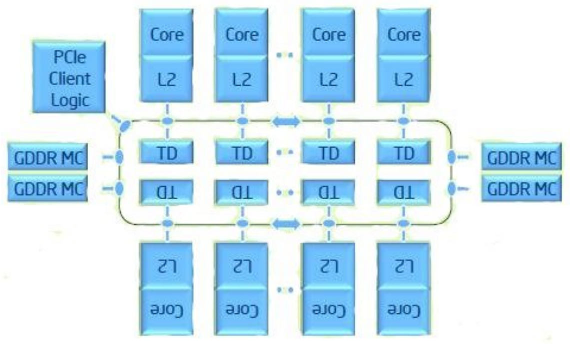

Figure 2. Microarchitecture of the MIC coprocessor

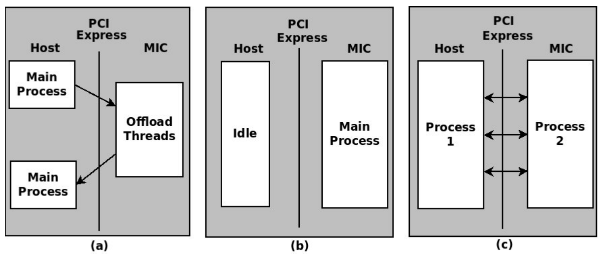

Figure 3. Intel Xeon Phi execution modes: (a) offload mode, (b) native mode and (c) Symmetric mode

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 provides 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.

2) Programming mode: Unlike GPUs, we count three different ways to execute an application on the Xeon Phi: the native, offload and hybrid modes (see Fig 3) [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 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.

3) Future developments: Intel is developing the future of the Intel Xeon Phi product family, with the release for the first semester of 2016 of the Knights Landing products. The cores of this new architecture will be based on heavily modified Silvermont Atom core, each core having two 512-bit vector units (AVX-512). The Knights Landing processor will have three times the single threaded performance as the custom Pentium 54C cores used in the Knights Corner. This new architecture is planned with 3 teraflops double precision. The new Knights Landing will be released in two versions: a coprocessor one, that could be connected to a CPU via a PCIe bus, just like the Knights Corner, and a standard CPU form factor. Also Intel has already planned Knights Hill, the 3rd generation of Intel Xeon Phi product family based on the 2nd generation of Intel Omni-Path architecture [13].

4) Xeon Phi synthesis: Intel Xeon Phi are parallel computing accelerators. They compensate their low parallelism (comparing to GPUs), by their ability to vectorize computations. The same task is executed for a vector, i.e. an OpenMP process, but different OpenMP processes can perform different tasks. For these devices, it is important to provide enough threads to keep busy all the cores, to take advantage of the vector units, and to take care of data locality.

III. Application

In this section we introduce the equations that govern the descent of a stratospheric balloon envelope. Then we present the method used to retrieve 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.

A. 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 $\vec{z}$ 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.

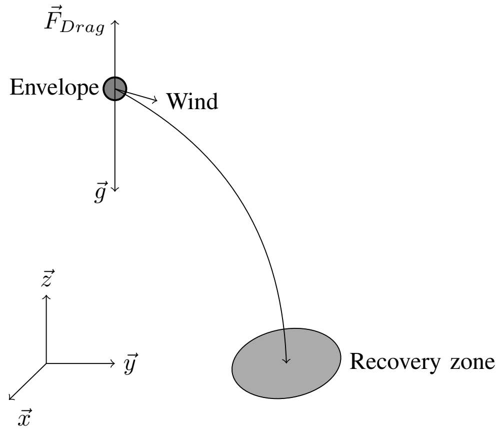

Figure 4. Sketch of drift descent motion dynamics

1) Vertical motion: The vertical movement of the envelope is modeled by the sum of two forces: gravitational force $\vec{g}$ and drag force $\vec{F_{Drag}}$ as shown in Fig. 4. The acceleration of the envelope is then computed according to the following equation (1):

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

where: $a(t,z)$, is the envelope vertical acceleration at time t and altitude z, in $m.s^{-2}$; $v(t)$, is the vertical speed of the envelope at time t, in $m.s^{-1}$; $\rho(z)$, is the density of the atmosphere at the altitude z, in $kg.m^{-3}$, 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 ($dt = 0.1s$).

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 $m.s^{-1}$. In the code, at every time step, the new position in latitude and longitude is computed , via Euler's method, according to equations (2) and (3), assuming the hypothesis of a spherical Earth:

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

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

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

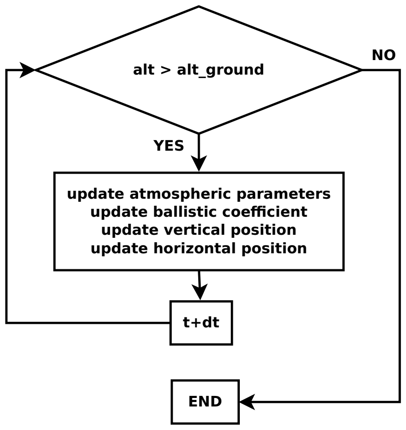
alt > alt_ground
NO
YES
update atmospheric parameters
update ballistic coefficient
update vertical position
update horizontal position
t+dt
END

Figure 5. Envelope drift descent calculation

B. Parameters interpolation and Monte-Carlo perturbation

The atmospheric parameters (pressure, temperature, wind) are retrieved at every time step of the simulation by interpolation of atmospheric data obtained via weather forecast. In the same manner the envelope ballistic coefficient is interpolated from an evolution table based on reanalysis of previous flights.

In order to perform the Monte-Carlo analysis of the envelope drift descent, the atmospheric parameters (pressure, temperature and wind) are perturbed. As we wanted our method to be the less conservative on what concern the size of the computed fallout area, a systematic perturbation of the atmospheric parameters is applied at each time step according to equation (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 parameters of each simulation. $\sigma_{parameter}$ 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. The perturbation parameters ($\alpha, \sigma, sign$) are stored in three vectors of N elements (N, the number of simulation to perform).

Thus our application perfectly fits for the usage of computing accelerators such as GPUs or Xeon Phi, as it has a lot of independent task, working on different data.

C. 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. 5:

- update the atmospheric parameters at the current altitude, via interpolated values from forecast data and perturbations derived from equation (4).

Table I
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	12000	288	N.A.	3.5
Xeon Phi 7120P	N.A.	61	1.24	16000	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.

- update the ballistic coefficient from the ballistic coefficient evolution table.
- update the vertical position via equation (1)
- update the horizontal position via equations (2) and (3).

IV. 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 [14] 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 algorithm and the main steps of optimization of the parallel code showing each time the gain obtained. Finally, we show the performances of the final codes, and we compare the parallel computing times on the different computing systems.

A. Description of the experimental conditions

1) Hardware: We consider two computing systems (see Table 1).

- A cluster node, composed of two Intel Xeon E5-2680 v2 at 2.8GHz with 10 physical cores each and an Intel Xeon Phi Coprocessor 7120P, with 61 cores at 1.238 GHz and 16GB memory .
- A computing system with a NVIDIA Tesla K40, with 2,880 CUDA cores at 0.745GHz and 12 GB memory.

2) Software: The GPU implementation of the code is performed using CUDA 7 [15]. 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).

B. Experimental protocol

In this paper, we present average computing times for ten instances of problems. The measurement starts with the loading of the first data, and stops when the 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*" [16], [17].

Table II
COMPUTATION TIME FOR 100,800 SIMULATIONS ON GPU

Parallel implementation	Overall Time (s)	Communication Time (s)	Computation Time (s)
Loop parallelism	21.49	17.39	4.1
Task parallelism	3.03	0.12	2.91
Task parallelism with memory management	2.79	0.11	2.68

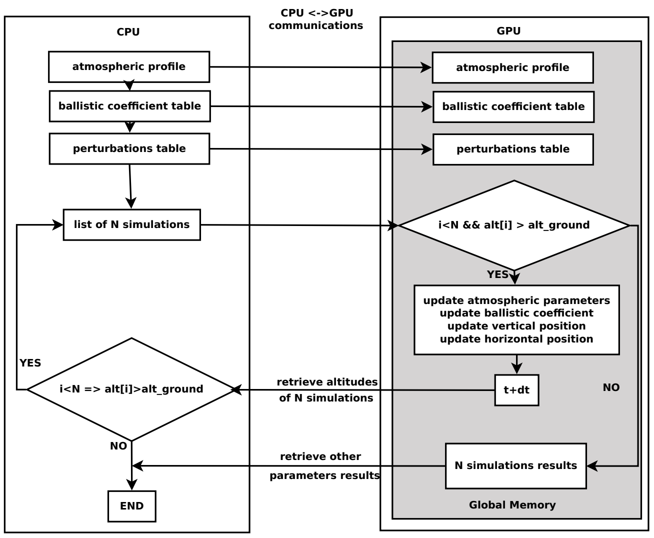
CPU
CPU <->GPU communications
GPU
atmospheric profile
ballistic coefficient table
perturbations table
list of N simulations
atmospheric profile
ballistic coefficient table
perturbations table
i<N && alt[i] > alt_ground
YES
update atmospheric parameters
update ballistic coefficient
update vertical position
update horizontal position
YES
i<N => alt[i]>alt_ground
retrieve altitudes of N simulations
t+dt
NO
NO
retrieve other parameters results
N simulations results
END
Global Memory

Figure 6. GPU loop parallel algorithm

C. 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 it's massive parallelism capability.

1) Loop parallel algorithm: We begin the parallelization of our application on the GPU, using a loop parallel algorithm represented Fig. 6, and referenced as Loop parallelism in Table II. We decide to start with this algorithm, because it is typically the kind of algorithm one gets when using GPU numerical solver of Ordinary Differential Equations library such as odeint [18], with complex models that necessitate to update data between two time step, 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. 5. 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 on 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 21.49s.

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., 17.93s (80.92% of overall execution time), while the kernel execution only counts for 4.1s (19.08% of overall execution time). In order to reduce the communications, we decide to change our parallel implementation for a task parallel algorithm. We redesign the kernel so that each CUDA thread performs the main while loop presented Fig. 5, instead of only one time step. This model is referenced as Task parallelism in Table 2. 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. This new algorithm performs 100,800 simulations in 3.03s, 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% (0.12s) of the execution time, while the kernel counts now for 99.86% (2.91s). It is interesting to point that the kernel execution time is 2.91 s, while it was 4.1s 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.

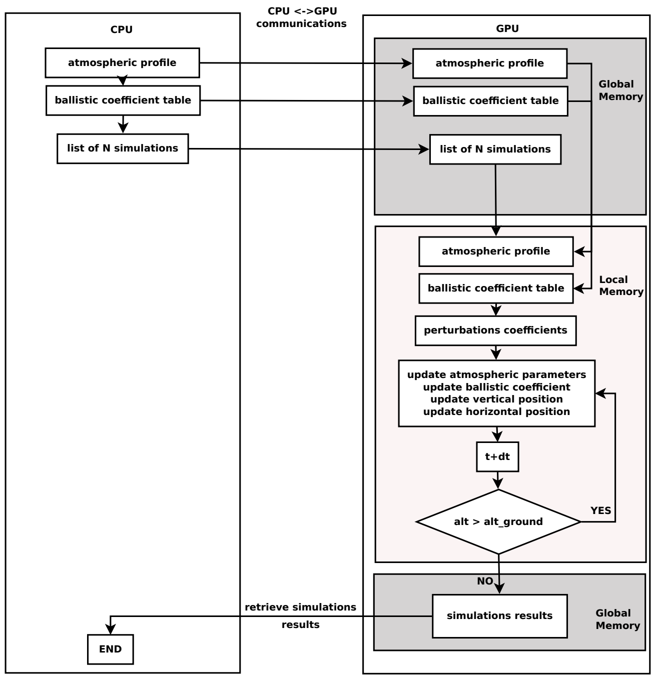
CPU
CPU <->GPU communications
GPU
atmospheric profile
ballistic coefficient table
list of N simulations
atmospheric profile
ballistic coefficient table
list of N simulations
Global Memory
atmospheric profile
ballistic coefficient table
Local Memory
perturbations coefficients
update atmospheric parameters
update ballistic coefficient
update vertical position
update horizontal position
t+dt
alt > alt_ground
YES
NO
retrieve simulations results
simulations results
Global Memory
END

Figure 7. GPU task parallel and memory management algorithm

3) Memory management: At this point we still have not considered the locality of data, storing all variables in the global memory. So we modify the kernel in order to perform a local copy of all the data (atmospheric and ballistic), and we define all the variables 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 Figure 5), 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 parallelism and memory management in Table II. 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) [19] increases from 0.06% to 92.79% as compared with the previous implementation. This algorithm is presented Fig. 7.

D. 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.

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

Table III
COMPUTATION TIME FOR 100,800 SIMULATIONS ON XEON PHI

Parallel implementation	Time (s)
Task parallelism	29.98
Task parallelism with vectorization	23.06
Task parallelism with vectorization and memory management	8.02
Task parallelism with memory management and fixed size vectorization	6.86

between the CPU and the Xeon Phi. Doing this, it seems natural to adopt a task parallel algorithm (referenced as Task parallelism in Table III). All the vectors are allocated using the instruction $posix\_memalign()$, and aligned on 64 bits. The parallelization across the Phi, is ensured via OpenMP. We instantiate 244 OpenMP processes. Each process has to perform, sequentially, M times the while loop presented Fig. 5, i.e. M envelope drift descent simulations. M is selected in order that M times the number of OpenMP process equals the total number of simulations to perform. The variables of the simulations are stored in vectors shared among the OpenMP processes. The perturbations table are generated by each OpenMP process for it's M simulations. The uniqueness of each perturbation is guaranteed by the management of the seed of the random number that depends on the rank of the OpenMP process. This algorithm computes 100,800 simulations in 29.98s.

2) Vectorization: The above implementation does not fully use the capacity of the Xeon Phi as there is no vectorization. So we modify the above algorithm to correct this. Our strategy consists in vectorizing with the special instruction *#pragma simd*, the while loop presented Fig. 5 and keeping the OpenMP parallelization introduced previously. We also add the compiler directive *#pragma vector aligned*, to guarantee to the compiler that all memory accesses are aligned. This new algorithm is referenced as Task parallelism with vectorization in Table 3. The size of the vectors is defined at the application execution and is set to 32. Thus, to compute 101,504 simulations, the application launches 244 OpenMP processes that compute 416 simulations each, launching 13 blocks of 32 vectorized simulations; this is performed in 23.06s.

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 decide to store state vectors (containing the

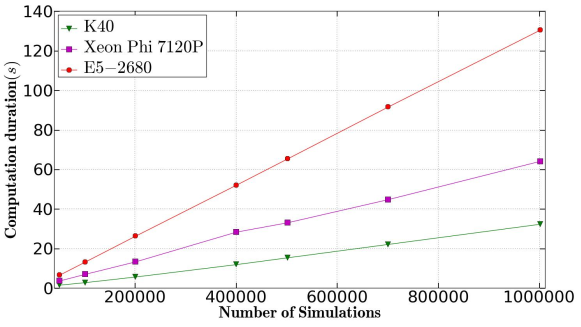

Figure 8. Computation duration versus number of simulations

position and speed of the envelope) in the local memory of each OpenMP process creating private variables. This new implementation (referenced as Task parallelism with vectorization and memory management in Table III) computes 101,504 simulations in 8.02s.

4) Fixed size vectorization: During the process of optimization of the application, we fix vector size beforehand, i.e. before compilation, to 32. This is the Algorithm with memory management and fixed size vectorization of Table III. Thanks to this modification, the application compute 101,504 simulations in 6.86s. It appears that this gain is due to the fact that if the size of the vector is not set at the compile time, the Intel compiler adds extra control features that slowdown the application. So fixing the value will prevent this behavior.

E. Parallelization on the CPU

We note that the parallel algorithm implemented on the CPU is the same as the one implemented on the 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). As the Xeon E5-2680-v2 L1 cache is wider and more efficient than the one of the Xeon Phi, experiments show that 32 is not the vector size that gives the best performance. 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 5,120 simulations each, launching 40 blocks of 128 vectorized simulations) in 13.31s.

F. 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.

Fig. 8 displays the computing time on the different platforms for several numbers of simulations. As we can see, the K40 and the Xeon Phi 7120P, address the operational challenge, since 1,000,000 simulations require respectively 32.4 and 64.2s, while they require 130.5s on a computing node with two Intel Xeon. Thus these devices appear to be more efficient than a standard cluster node. Moreover, both devices can be plugged on a computer, and their electric consumption of around 300W (PDT of 300W for the Phi 7120p, and 235W for the K40), make them suitable too operational conditions.

G. 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 GPU is about four 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.

- On what concerns GPU implementation: the reduction of communications between the host and device along with the reduction 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.
- On what concerns 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 believe these guidelines are also valid for other applications related to numerical integration and more generally to numerical simulation.

V. Conclusion

The implementations of our parallel algorithm on the GPU K40 and Xeon Phi 7120P are more than twice as fast as a parallel and vectorized implementation on the two sockets CPU E5-2680-v2. 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 [20], ballistic trajectory prediction [21], ray tracing of GNSS [10], parafoil automatic guidance [22], 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 here can be applied to more complex processes 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.

Acknowledgement

The authors gratefully acknowledge the support of NVIDIA Corporation with the donation of the Tesla K40 GPU used for this research work.

The authors wish to thank Dr. D. Gazen and Dr. J. Escobar of Observatoire Midi-Pyrénées for their advices and the access to the cluster in Toulouse.

Finally the authors thank the DEDALE work group coordinated by CNES, France.

References

[1] C. P. Robert and G. Casella, *Monte-Carlo Statistical Methods*. Springer, 2004.

[2] V. Boyer and D. El Baz, "Recent Advances on GPU Computing in Operations Research," in *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, May 2013, pp. 1778–1787.

[3] A. ul Hasan Khan, M. Al-Mouhamed, and L. Firdaus, "Evaluation of Global Synchronization for Iterative Algebra Algorithms on Many-Core," in *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2015 16th IEEE/ACIS International Conference on*, June 2015, pp. 1–6.

[4] G. Teodoro, T. Kurc, J. Kong, L. Cooper, and J. Saltz, "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*, ser. IPDPS '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 1063–1072. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2014.111

[5] E. Saule, K. Kaya, and Ü. V. Çatalyürek, "Performance Evaluation of Sparse Matrix Multiplication Kernels on Intel Xeon Phi," *CoRR*, vol. abs/1302.1078, 2013. [Online]. Available: http://arxiv.org/abs/1302.1078

[6] S. Saini, H. Jin, D. Jesperson, S. Cheung, J. Djomehri, J. Chang, and R. Hood, "Early Multi-Node Performance Evaluation of a Knights Corner (KNC) Based NASA Supercomputer," in *IEEE 24th International Heterogeneity Computing Whorkshop*, 2015.

[7] S. J. Pennycook, C. J. Hughes, M. Smelyanskiy, and S. A. Jarvis, "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*, ser. IPDPS '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 1085–1097. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2013.44

[8] NVIDIA. Nvidia. CUDA 7.0 Programming Guide. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[9] ——. www.nvidia.com.

[10] P. Gegout, P. Oberle, C. Desjardins, J. Moyard, and P.-M. Brunet, "Ray-Tracing of GNSS Signal Through the Atmosphere Powered by CUDA, HMPP and GPUs Technologies," *Selected Topics in Applied Earth Observations and Remote Sensing, IEEE Journal of*, vol. 7, no. 5, pp. 1592–1602, May 2014.

[11] J. Jeffers and J. Reinders, *Intel Xeon Phi Coprocessor High-Performance Programming*, ser. Morgan Kaufmann. Elsevier Science & Technology Books, 2013.

[12] R. Farber. Programming Intel's Xeon Phi: A Jumpstart Introduction. [Online]. Available: http://www.drdobbs.com/parallel/programming-intels-xeon-phi-a-jumpstart/240144160

[13] E. Gardner. What public disclosure has Intel made about Knights Landing? [Online]. Available: https://software.intel.com/en-us/articles/what-disclosures-has-intel-made-about-knights-landing

[14] K. Hwang, G. C. Fox, and J. Dongarra, *Distributed and Cloud Computing: From Parallel Processing to the Internet of Things*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[15] NVIDIA. Nvidia. CUDA 7.0. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[16] R. Rahman, *Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers*, 1st ed. Berkely, CA, USA: Apress, 2013.

[17] Intel. Thread Affinity Interface. [Online]. Available: https://software.intel.com/en-us/node/522691#KMP_AFFINITY_ENVIRONMENT_VARIABLE

[18] A. Karsten and M. Mario. odeint. [Online]. Available: http://headmyshoulder.github.io/odeint-v2/

[19] NVIDIA. Profiler user's guide. [Online]. Available: http://docs.nvidia.com/cuda/profiler-users-guide/#nvprof-overview

[20] B. Plazolles, M. Spel, V. Rivola, and D. El Baz, "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.

[21] M. Ilg, J. Rogers, and M. Costello, "Projectile Monte-Carlo Trajectory Analysis Using a Graphics Processing Unit," *AIAA Atmospheric Flight Mechanics Conference*, 2011. [Online]. Available: http://dx.doi.org/10.2514/6.2011-6266

[22] J. Rogers and N. Slegers, "Robust Parafoil Terminal Guidance Using Massively Parallel Processing," *AIAA Atmospheric Flight Mechanics Conference*, 2013. [Online]. Available: http://dx.doi.org/10.2514/6.2012-4736