Hindawi Publishing Corporation
Journal of Applied Mathematics
Volume 2013, Article ID 437357, 15 pages
http://dx.doi.org/10.1155/2013/437357

Research Article

TESLA GPUs versus MPI with OpenMP for the Forward Modeling of Gravity and Gravity Gradient of Large Prisms Ensemble

Carlos Couder-Castañeda,[1] Carlos Ortiz-Alemán,[1] Mauricio Gabriel Orozco-del-Castillo,[1] and Mauricio Nava-Flores[2]

[1] *Mexican Petroleum Institute, Eje Central Lázaro Cárdenas 152, Colonia San Bartolo Atepehuacan, 07730 México, DF, Mexico*
[2] *División de Ingeniería en Ciencias de la Tierra, Facultad de Ingeniería, Universidad Nacional Autónoma de México, Circuito Interior S/N, Colonia Ciudad Universitaria, 04510 México, DF, Mexico*

Correspondence should be addressed to Carlos Couder-Castañeda; ccouder@imp.mx

Received 28 May 2013; Revised 16 September 2013; Accepted 17 September 2013

Academic Editor: Luca Formaggia

An implementation with the CUDA technology in a single and in several graphics processing units (GPUs) is presented for the calculation of the forward modeling of gravitational fields from a tridimensional volumetric ensemble composed by unitary prisms of constant density. We compared the performance results obtained with the GPUs against a previous version coded in OpenMP with MPI, and we analyzed the results on both platforms. Today, the use of GPUs represents a breakthrough in parallel computing, which has led to the development of several applications with various applications. Nevertheless, in some applications the decomposition of the tasks is not trivial, as can be appreciated in this paper. Unlike a trivial decomposition of the domain, we proposed to decompose the problem by sets of prisms and use different memory spaces per processing CUDA core, avoiding the performance decay as a result of the constant calls to kernels functions which would be needed in a parallelization by observations points. The design and implementation created are the main contributions of this work, because the parallelization scheme implemented is not trivial. The performance results obtained are comparable to those of a small processing cluster.

1. Introduction

In recent years the number of publications about parallel computing applications using the GPUs architecture has remarkably increased. These applications represent an economic and powerful way to access high-performance computing [1, 2]. However, since the architecture of the GPU is different to that of a conventional CPU, the programming paradigm should be changed. This had led to the development of a new research field within scientific computing which explores the performance of the GPU to general purpose applications, such as acoustic simulation [3], propagation of seismic waves [4], seismic migration [5], molecular engineering [6], fluid dynamics [7], even for astrophysical simulations [8] and many other implementations. In a few words, the objective of the general purpose computing in GPU (GPGPU) is to develop new applications for those who pretend to solve problems of numerical simulation requiring as less computing time as possible.

Even though the GPUs have become an accessible platform for general purpose programming, they still have some limitations and its programming entails some difficulties [2]. Compute unified device architecture (CUDA) is a set of tools that includes mainly a compiler for an extension of the C language, a set of libraries, and drivers for the specific programming of NVIDIA cards. Despite that these tools have eased the programming, it is still needed to know with precision the architecture of the card with its several memory levels to obtain the maximum performance. One of the greatest drawbacks that can occur in CUDA is the

handling of critical sections or shared memory, since there are no proper instructions of exclusions, as is the case in OpenMP [13]. Therefore, when these difficulties are present in the application design, it is necessary to modify the strategy to achieve a good performance. Implementing this application in CUDA for the calculation of the direct model through a prisms ensemble represents a big challenge; this is because the memory region where the calculations are made (observation grid) is shared and therefore requires different memory allocations to avoid the data coherency problems produced when two or more processing cores access the same memory location at the same time. This drawback is not present, for example, when a model is solved through finite differences since the domain is divided between the cores and there is no overlap in the data handled by each core [9].

One of the simplest parallelization options for this problem using CUDA consists of partitioning by the number of prisms and to consequently divide the domain of the observation grid between the device cores; this approach avoids the handling of shared regions. However, this design is extremely inefficient since for each prism it would be necessary to make a call to the GPU. A typical problem of fourteen million prisms would imply the execution of a kernel by the same number, and each kernel call is computationally expensive [11]. Therefore, we propose an efficient design based on the partition by groups of prisms.

Additionally, we make experiments with double and single precision to calculate the errors that can introduce the single precision, and even when its use reduces the computing time from 30% to 50%, it is necessary to evaluate the effect of the introduced error by using only seven significant digits in the floating numbers and investigate if this error affects considerably the result of the modeling. This analysis is necessary since recently NVIDIA introduced TESLA K10 cards which handle single precision and are more economical than the TESLA K20 cards for double precision.

1.1. Related Work. Some related research works which implement an approach to calculate scalar and tensor gravity utilizing the massively parallel architecture of GPU can be found in [12], in which a parametrization based on rectilinear blocks with constant density within each block is used; however, the results show that our design yields a better performance using different memory allocations. Also a parallel program was developed to estimate the correlation imaging for gravity and gravity gradiometry data to provide a rapid approach to equivalent estimation of objective bodies with different density contrasts in the subsurface [17]; however, neither is multi-GPU implementation.

1.2. Paper Organization. This paper is organized as follows: in Section 2 we present the characteristics of the CUDA platform and the tools we used, in Section 3 the application design is explained, in Section 4 we present some numerical experiments that were made, in Section 5 we detail the validation of the code for double as well as single precision, and finally in Section 6 we compare against a 29-nodes cluster and finally we present our conclusions.

2. Architecture of the Platform

As a general-purpose architecture, CUDA includes the hardware that can have dedicated processing cards or cards which also control the visualization of the monitor and the software that includes the compiler, the card drivers, and the libraries. The programming model in CUDA consists of functions called kernels which are executed concurrently by several light threads (CUDA threads). These threads are grouped into blocks which can be of one, two, or even three dimensions. Each block can contain a maximum number of threads, defined by the architecture of the card which is being used. The blocks are executed concurrently by the stream multiprocessors (SMs) and the execution order is nondeterministic. Each SM contains a set of microprocessors which can be thought of as arithmetic logic units (ALUs) and are known as CUDA cores. The threads within a block are divided into groups of 32, called warps. A warp is executed concurrently by the CUDA cores, and the number of cores can be less than the size of the warp, as happens in the TESLA C1060 card, which has eight cores per SM but supports one warp. This configuration implies that 8 threads are executed in parallel, but 32 are processed concurrently, and this means that each core will process 4 threads previously assigned.

There is an implicit synchronization between kernel calls; that is, the next kernel cannot be executed until the previous one has finalized. There are some cards of a more advanced architecture which allow the concurrent execution of kernels, but this must be specified by the programmer. The threads within the same block can be synchronized, but synchronization between blocks cannot be achieved.

Understanding the different types and hierarchies of memory of the NVIDIA cards is essential to be able to take advantage of them. There are four types of memory: global, constant, texture, and shared. The global memory is analogous to the RAM memory used in a CPU. A CUDA application requires several data transfers from the global memory of the GPU to the CPU memory. The constant and the texture memories are cache memories and read-only by the SM. The content of the texture memory can be updated through special functions. The shared memory is included in each block of threads and is shared only by the threads in the block and is extremely fast in comparison with the global memory, but its deficiency lies in the fact that it is very limited and its size is defined by the architecture.

The key, in general, to achieve an efficient code for GPU is to correctly handle the access times (latency) to the memory, that is, to carry out the least possible data transfers between the global memory of the GPU and the principal memory of the CPU [15], followed by few calls to the kernel functions. This implies having a great amount of blocks to process or having blocks with a great amount of threads. It is also necessary to avoid an excessive read-write access to the global memory, and preferable to use the shared memory, even though a lot of times this is not possible since a great amount of data is being handled and the shared memory becomes insufficient.

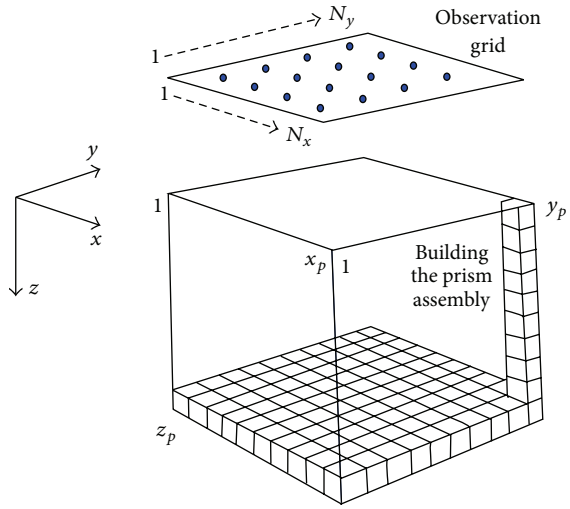

Figure 1: Construction of a prism of densities and its calculation with respect to an observation grid.

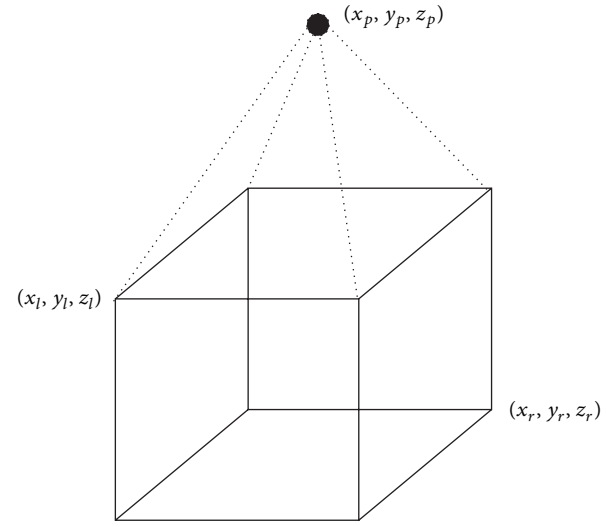

Figure 2: Illustration of the calculation of the anomaly produced by a prism with respect to an observation point.

3. Design of the Application for GPU

The application consists of calculating the gravimetric or gradiometric response produced by a rectangular prismatic body with constant density, in reference to a set of points called observation points (see Figure 1) [16]. The set of prisms is known as prisms ensemble and is not necessarily regular. An ensemble of nonregular prisms can be configured (Figure 3), with the only requisite being that they are not superimposed. Since the gravitational field complies with the superposition principle with respect to the observation, if f is the calculated response at a point (x, y), then the observed response at the point $f(x, y)$ is given by

$$f(x, y) = \sum_{k=1}^{M} G(\rho_k, x, y), \tag{1}$$

where M is the total number of prisms and ρ the density of the prism.

It is well known that the function which can calculate the gravimetric or gradiometric contribution for a given prism and a set point can be rewritten as follows:

$$g = f(x_1, y_1, z_1, x_2, y_2, z_2, x, y, z, \rho), \tag{2}$$

where (x_1, y_1, z_1) is the upper left vertex and (x_2, y_2, z_2) the lower right vertex of the prism; (x, y, z) is the observation point and ρ the density, as shown in Figure 2.

To be able to discretise the cube, we define x_p, y_p, and z_p as the face numbers in the directions x, y, and z, respectively. If the cube is discretized in an homogeneous way, then we can define M, the number of prisms, as $M = x_p \times y_p \times z_p$, and the consecutive numbering of the prisms would be first in x, then y and finally in z. In case the ensemble is irregular, x_1, y_1, z_1, x_2, y_2, z_2, and ρ should be provided through a file for each prism. We define O as the number of observation points, which is determined by $O = N_x \times N_y$, where N_x and N_y are the number of observation points in the x and y directions,

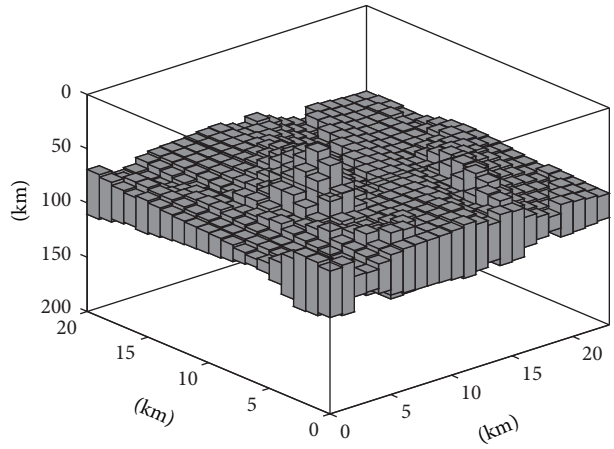

Figure 3: Illustration of an ensemble of irregular prisms.

respectively. Therefore, the number of times which is required to call the function defined in (2) to calculate the anomaly produced by a component is $M \times O$.

The first step to develop a parallel program is to search the finest granularity. This is important since CUDA handles a fine granularity paradigm. In this case it can be thought of parallelizing by prisms or by observation points, (see Figure 4), that is, by the number of elements in M or by the number of elements in O. One of the requisites which must be taken into consideration in the design is that it must be scalable, and hybrid systems must be considered since they are the most commonly used nowadays. However, many times design and type of architecture cannot be separated, especially when it is as specific as the CUDA architecture. Following the methodology proposed by Foster [10], it is necessary to analyze both parallelization schemes and examine which yields the best performance. Nevertheless, since $M \gg O$, in principle the best option of partitioning is by M, as

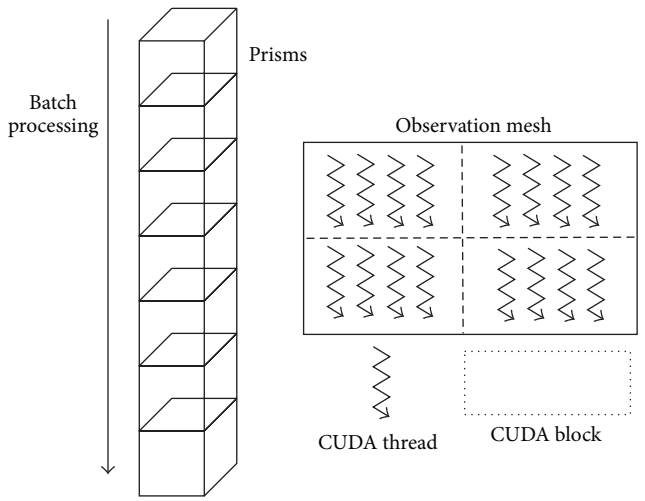

Figure 4: Partitioning by observation points.

detailed in the next section, even though this design requires a greater effort for its implementation.

3.1. Implementation in a Single GPU. The design for a GPU will consist of generating an observation grid (memory space) for each execution thread created in the TESLA card; in other words, if we create 14 blocks, each one containing 32 execution threads, we would have generated 448 observation grids. This design obeys the fact that we will select a partition by prisms, which implies less calls to kernel functions and is therefore more efficient in terms of execution time.

To give the correct dimension to our design, we analyze the simplest option of parallelization, which consists of partitioning the observation grid in the memory card for each prism.

This method of parallelization is the most trivial since it is enough to simply parallelize the cycle of the observations, which can even be done by using OpenACC, simplifying even more the work and avoiding the creation of the kernel by the programmer and leaving it to the compiler. However, a big drawback of this method is the excessive number of calls to the kernel function, which decreases performance since the parallel region is created and closed on each call. Additionally, it does not represent a big challenge design-wise to the point that the scheme can be solved with a compiler which could automatically generate parallel code.

On the other hand, the other parallelization option is to do it by prisms; in other words, make the threads divide the work by prisms. However, to avoid the coherence problems it is necessary to create a different space of memory for each execution thread, since it is not feasible to create just one memory space for a single observation grid, shared by all of the threads, since one of the principal problems which is not easy to handle in CUDA is the mutual exclusion of the threads in shared zones.

As can be seen in Figure 5, it is required to create an observation grid for each execution thread to avoid numerical consistency problems; if only one grid is occupied for all

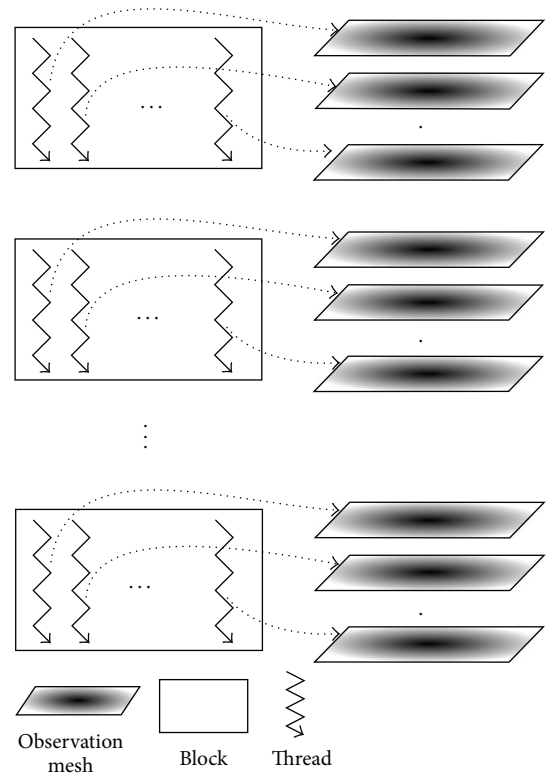

Figure 5: Partitioning by prisms using different memory spaces.

the threads, access conflicts will appear since several threads would write in the same memory location.

The number of created grids is equal to the number of created execution threads, and therefore the number of grids would be equal to the number of blocks times the number of threads contained in each block. This design allows the process at the same time as many prisms as threads are created, and therefore, for example, if 448 threads are created, then the same number of prisms will be processed in parallel in one kernel function execution and in the following call to the kernel function another 448 prisms will be processed and so successively until finalizing the process. In this way, the thread 1 will process the prisms set $\{1, 449, 897, 1345, \ldots\}$; in fact, the number of times that the kernel function is called is determined by

$$p = \left\lceil \frac{M}{T} \right\rceil, \tag{3}$$

where M is the number of prisms and T is the number of created threads, and consequently $p \ll M$. To exemplify that the prisms partitioning is better, let us suppose that 14 blocks are created, each one containing 512 execution threads, and then a total of 7,168 observation grids are generated. If we have a problem of 200,000 prisms, the number of calls to the kernel will be $\lceil 200,000/7,168 \rceil = 30$, and 30 is much smaller than 200,000. Therefore, we reduced the number of calls to the kernel in a $6,666X$ factor with respect to the partition by observation points.

```
If (K<=M) then
 For each j from 1 to Ny
   For each i from 1 to Nx
     G(Thread,i,j) = Gz(parameters) + G(Thread,i,j);
   End For
 End For
Else
 For each j from 1 to Ny
   For each i from 1 to Nx
     G(Thread, i, j) = G(Thread,i,j)+0.0
   End For
 End For
End if
```

Pseudocode 1

```
DO k = 1,T
!$acc parallel loop present (Gd,Gshared_d)
collapse (2)
DO i = 1,Nx
    DO j = 1,Ny
    Gshared_D(i,j) = Gd(k,i,j) + Gshared_D(i,j);
    END DO
END DO
!$acc end parallel loop
END DO
```

Pseudocode 2

Each thread will follow a scheme of processing over the prisms as follows: a thread t will process the sequence of prisms

$$T \times (n-1) + t, \tag{4}$$

where $n = 1, 2, 3, \ldots, p$.

The implementation will initially consist of coding in device mode the functions which calculate the vectorial components (g_x, g_y, g_z) and the tensorial components $(g_{xx}, g_{yy}, g_{zz}, g_{xy}, g_{xz}, g_{yz})$, following the definition of (2). The functions of type device can only be called by kernels and are executed by a single CUDA thread.

To calculate any component it is necessary to allocate the memory for a tridimensional array `G` of size $T \times N_x \times N_y$. We define ID $= t \cdot x + (b \cdot x - 1) \times s \cdot x$, where $t \cdot x$ is the thread identifier, $b \cdot x$ the block identifier, and $s \cdot x$ the block size. Notice that we write $(b \cdot x - 1)$ because the thread and block identifiers in FORTRAN-CUDA are numbered starting from 1. We also define K as $K = (T) * (I - 1) + \mathrm{ID}$, where T is the number of created threads (equal to the number of observation grids), while I is the number of partition over the set of prims M in which it is working upon. The I partition of the set M is a division of M into nonoverlapping and nonempty subsets of size T that cover all of M. The subsets are collectively exhaustive and mutually exclusive with respect to the set M. The cardinality of the partition is determined by T (4) and the number of partitions is defined by p (3) and I can take the values $1, 2, 3, \ldots, p$. The general scheme of the computing kernel would be defined in Pseudocode 1, where `Gz` receives the parameters defined in (2), which are:

(i) `Xa(K)`, `Ya(K)`, and `Za(K)`, the position in x, y, and z, respectively, of the upper left vertex of the prism,

(ii) `Xb(K)`, `Yb(K)`, and `Zb(K)`, the position in x, y, and z, respectively, of the lower right vertex of the prism,

(iii) `Xm(i)`, `Ym(j)`, and `Elev(i,j)`, the location in x, y, and z, respectively, of the observation point,

(iv) `Rho(K)`, the density of the prism.

To calculate the final result of the anomaly it is necessary to add all of the obtained results from the threads, this is, the final anomaly G_f at a point (i, j) is approximated as:

$$G_f(i,j) = \sum_{k=1}^{T} G(k,i,j). \tag{5}$$

To generate the reduction (the sum), we can make use of the OpenACC which automatically generates the kernel, and the structure in FORTRAN is as in Pseudocode 2.

In Pseudocode 2, `Gshared` is the bidimensional array where the reduction is made, and `Gd` is the tridimensional which contains the data of the anomaly generated by the different threads.

3.2. Multi-GPU Implementation. There are diverse possibilities to do the implementation using several GPUs [14]; one of

them is to use the same libraries provided by CUDA 4.0 to the development of peer-to-peer applications to communicate the GPUs within the same workstation; another possibility would be to use OpenMP. Nevertheless, we consider that the best option in this case is to use MPI since the number of required messages is not intensive and, unlike OpenMP or the peer-to-peer connection, it allows distributing the work to several GPUs which are not even connected in the same machine. In fact, the number of messages required to be sent for a problem with q MPI processes, each one controlling a GPU, is q itself; this is because each process sends its observation grid. The estimated communication time t_{comm} to the complete application is

$$t_{\text{comm}} = q\left(t_{\text{startup}} + t_{\text{data}}\right), \tag{6}$$

where t_{startup} is the necessary time to initialize the sending of the data and $t_{\text{data}} = N_x \times N_y$ is the size of the message, which in this case is the number of observation points.

Due to the fact that the parallelization in MPI is explicit, we need to manually distribute the number of prisms through a modular expression. Let us suppose that M is the number of prisms to calculate and that q is the number of cards that we are going to use. If every card is controlled by a process, we define the start and end of prisms to process by q as q_{start} and q_{end}, respectively. We then calculate the integer s as the quotient of M and the total number of processes q_n, and we determine the remainder r, both as

$$\begin{aligned} s &= \frac{M}{q_n}, \\ r &= \operatorname{mod}\left(\frac{M}{q_n}\right). \end{aligned} \tag{7}$$

Therefore

$$\begin{aligned} q_{\text{start}} &= q \times s + 1, \\ q_{\text{end}} &= (q+1) \times s. \end{aligned} \tag{8}$$

If $r \neq 0$ and $q < r$, then we adjust as

$$\begin{aligned} q_{\text{start}} &= q_{\text{start}} + q, \\ q_{\text{end}} &= q_{\text{end}} + (q+1). \end{aligned} \tag{9}$$

If $r \neq 0$ and $q \geq r$, then

$$\begin{aligned} q_{\text{start}} &= q_{\text{start}} + r, \\ q_{\text{end}} &= q_{\text{end}} + r. \end{aligned} \tag{10}$$

In this way we can distribute the number of prisms M over q_n GPUs, in a balanced way.

Once the precedent distribution is done, we can occupy the implementation of the previous section to process the subset of local prisms for each GPU. Let us suppose that we have two workstations containing 4 GPUs each one; then in each station the cards will be numbered as 0, 1, 2, 3. To correctly select a device for each process q, numbered from 0 to 7, we use the function

$$f(q) = \operatorname{mod}(q, n_d), \tag{11}$$

where n_d (constant) is the number of devices per machine (in this case 4). If $q = 4$ then $f(4) = 0$, and this means that the fifth process, numbered as 4, will be responsible of controlling the device 0 of the second team. It is necessary to note that this procedure works if the workload of processes is orderly distributed; in other words, in the first machine the processes 0, 1, 2, 3 are addressed and in the second 4, 5, 6, 7. If this distribution is not ordered, then the algorithm does not work properly, which is why for some MPI implementation the ordered flag can be specified as execution parameter, which orderly distributes the workload.

After selecting a device per process, we proceed to distribute the prisms between the 8 processes using (7). In this way the balance for a problem of 251,946 prisms results as follows:

$$\begin{aligned} q_0 &= \{0, \ldots, 31494\}, \\ q_1 &= \{31495, \ldots, 62988\}, \\ q_2 &= \{62989, \ldots, 94481\}, \\ q_3 &= \{94482, \ldots, 125974\}, \\ q_4 &= \{125975, \ldots, 157467\}, \\ q_5 &= \{157468, \ldots, 188960\}, \\ q_6 &= \{188961, \ldots, 220453\}, \\ q_7 &= \{220454, \ldots, 251946\}. \end{aligned}$$

After the subsets of prisms in which every card will work are defined, we apply the scheme used in Section 3.1.

4. Performance Experiments

As experiment we use a synthetic case composed by an ensemble of $700 \times 700 \times 50$ prisms with 7 spheres of contrast of variable density (Figure 6). The spheres are conformed by 251,946 prisms and an observation grid of $150 \times 100 = 15,000$ points at an elevation of 100 m. Therefore, the number of calls to a function to calculate a component of the tensor or vector is 3,779,190,000, which represents a high-performance computing problem.

We performed experiments to calculate the vectorial components G_x, G_y, and G_z, and the tensorial components G_{xx}, G_{yy}, G_{zz}, G_{xy}, G_{xz}, and G_{yz} in a single GPU, in a workstation with 4 GPUs and with two workstations containing 4 GPUs each one.

The characteristics of the workstations where the experiments took place are as follows:

(i) 2 Intel(R) Xeon(R) CPU X5690 @ 3.47 GHz,

(ii) 6 real cores per processor,

(iii) hyperthreading technology disabled,

(iv) 12 GB of RAM memory,

(v) 4 TESLA Cards model C2070,

(vi) operating system Red Hat 6.3.

The principal characteristics that we can highlight of the TESLA card C2070 are

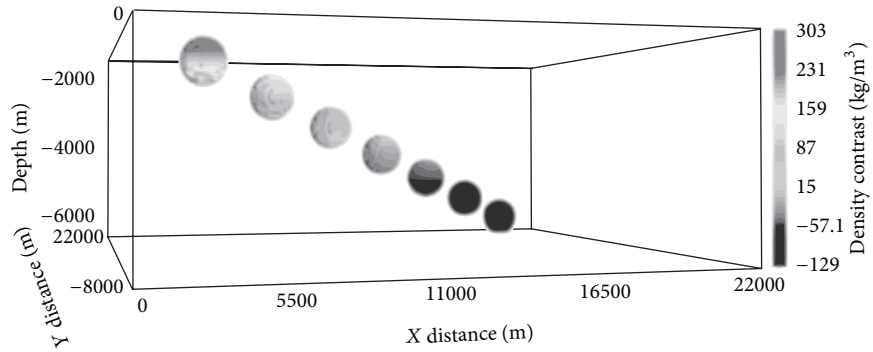

Figure 6: Synthetic problem setup with 7 spheres of variable density contrast (not scaled). Ensemble size of 22 km × 22 km × 8 km, 251,946 conform the spheres.

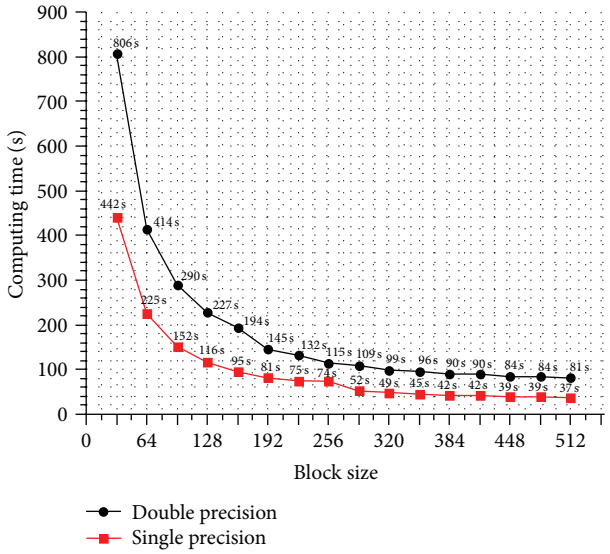

Figure 7: Comparison of the execution time using a variable block size in multiples of 32, in double and single precision.

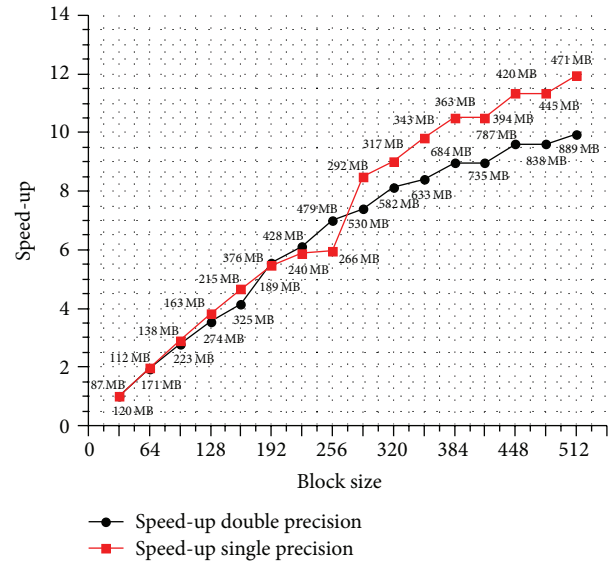

Figure 8: Speed-up of the behavior by increasing the number of threads per block in double and single precision, with its respective quantity of global memory required.

(i) 14 multiprocessors (SM),

(ii) 32 cores per multiprocessor (448 total CUDA cores),

(iii) 6 GB of global memory DDR5,

(iv) 515 theoretical GLOPS in double precision,

(v) 1.03 theoretical TFLOPS in single precision,

(vi) Frequency of the CUDA cores 1.15 GHz.

4.1. Performance of One Single Card. The first experiment was made in a C2070 card and consists of testing with different block sizes, keeping the number of blocks fixed at 14. The size of the selected block obeys the number of SMs available in the card and varies in multiples of 32 (warp size) until 512 threads per block, which means 16 experiments.

In Figure 7, the obtained computing times to solve the problem of the 7 spheres in double and single precision are shown. Notice how by increasing the size of the block, the execution time decreases exponentially to a limit or an asymptotic time which in double precision is of 81 s and in single precision of 37 s. However, even when the best computing time is obtained with a block of 512 threads, this is the most memory-consuming setup. To improve the understanding of the behavior we used the speed-up as a metric, considering a block of 32 threads as the processing unit, and we label the quantity of required memory for each case.

By increasing the block size in multiples of the warp, we increase what is known as multiprocessor *occupancy*. In general, by increasing the occupancy we improve the use of the SM, and in this application this phenomenon is observed.

The speed-up graph depicted in Figure 8 shows that the computing time reduction is practically linear by increasing the occupancy for both cases, single and double precision, and later on it starts to stabilize, which means that in fact we do obtain an increase in the performance if we increase the block size in multiples of 32 threads. In the C2070 card,

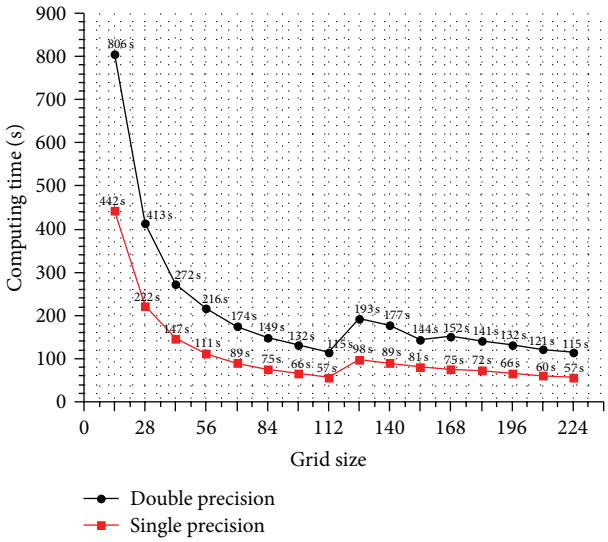

Figure 9: Comparison of the execution time using a variable grid size in multiples of 14, in double and single precision.

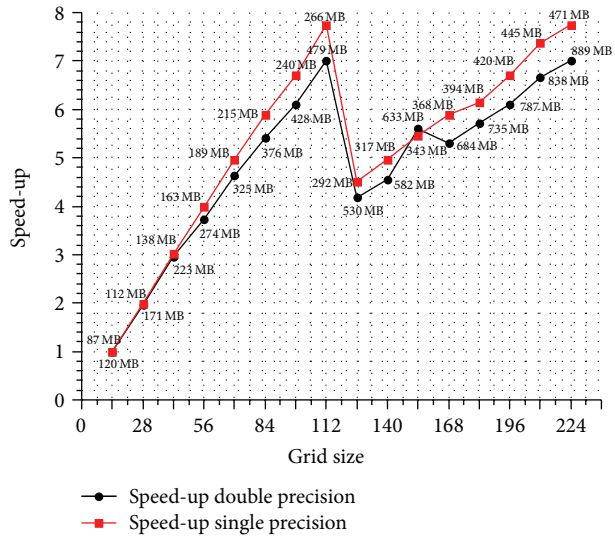

Figure 10: Speed-up of the behavior by increasing the quantity of blocks in double and single precision for a constant block size of 32 threads, with the respective quantity of required memory.

the maximum number of threads with their own memory space which could be created was 7,168 (14 blocks × 512 threads), with a size of 150 × 100 = 15,000 elements, which along with the number of prisms which must be previously stored in the memory of the card add up 889 MB in double precision and 471 MB in single precision. We observe that the amount of used memory in GPU increases, but the performance is improved. The number of blocks and their size—the number of created threads—will depend on the type of the card which is being used. There are several cards of medium range which do not exceed 500 MB of memory; thus so many grids cannot be created.

We now examine the behavior of the performance if we set the number of threads fixed at 32 per block and vary the number of blocks from 14 to 224 in multiples of 14. The results of the execution times are shown in Figure 9, where it can be noted that the minimum in execution time is reached, both for single and double precision, when 112 blocks of 32 threads are created. Later on the time increases and starts decreasing again when 224 blocks are created. Comparing with the graph in Figure 7, the behavior produced by increasing the number of blocks is not as stable as increasing the number of threads per block since occupancy is not increased as only one warp per block is handled.

It is necessary to note that even though 32 blocks of 512 threads and 224 blocks of 32 threads sum up 7,168 execution threads, the second option is slower because it increases the computing time in a 42% for double precision and 54% for single precision. To simplify the writing, if we create n blocks with m threads, using double precision we write $\langle n, m, d\rangle$ and $\langle n, m, s\rangle$ in single precision.

In this particular case, the creation of many blocks is not as efficient as increasing the number of threads, as shown in the speed-up graph (Figure 10), considering every 14 blocks as a processing unit. In this graph a decrease of the speed-up can be clearly seen when reaching 126 blocks, but after this point the speed-up increases again until reaching at 224 blocks the same value obtained at 112 blocks.

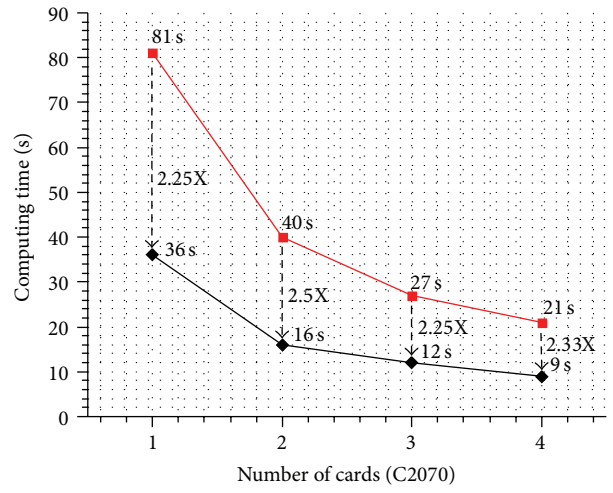

Figure 11: Execution time using up to four graphic cards in the same workstation in double and single precision.

4.2. Performance Using Multi-GPUs. In this subsection we analyze the performance using several GPUs integrated into the same workstation and distributed in two workstations, as was mentioned in Section 3.2; the choice to distribute the work between several GPUs was MPI. The best configuration found for this problem using only one C2070 card was to create 14 blocks with 512 threads, so this configuration was used. First we did experiments using four GPUs integrated into the same workstation. The results with respect to computing time are shown in Figure 11, where we considered a card as

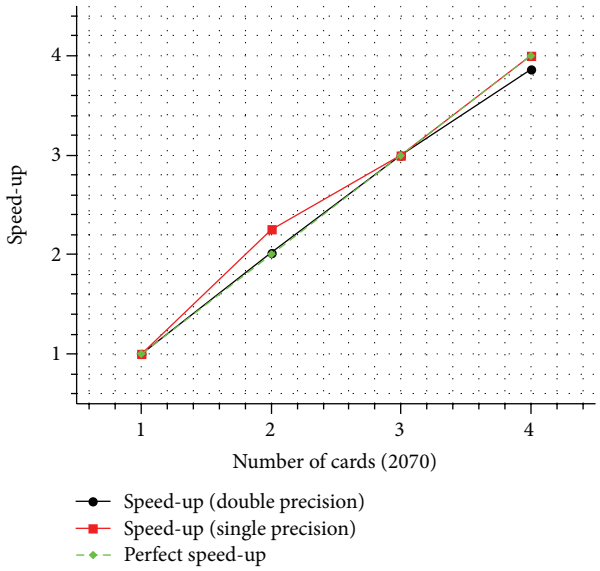

Figure 12: Speed-up of the behavior by increasing the number of processing cards in double and single precision. The number of blocks is 14 and the number of threads per block is 512 in every card, the optimal setup found for one single card.

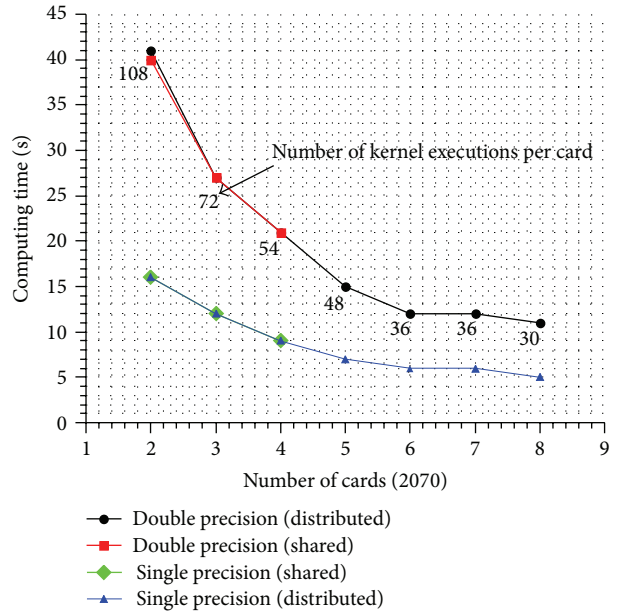

Figure 13: Execution time obtained using shared and distributed graphics cards, in double and single precision. We can see that MPI is not introducing overhead in the computing time.

a processing unit. It can be observed that the time decrease is proportional to the number of cards and that in single precision we reduced even more the time by a factor greater than $2X$. The speed-up graph shows that the decrease of time is almost linear for double and single precision (Figure 12).

We then analyzed the performance using the cards in a distributed fashion. For this we used two workstations each one with four internal cards interconnected by a network of 1000 Gbytes. The results show that the communication latency is negligible since there is no perceivable overload because of the use of MPI. This is because the used MPI functions are only required at the end of the calculation to do the reductions. The configurations which are used in a distributed fashion are (1 + 1) (A setup (1 + 1) means the use of two cards in a distributed fashion, so a configuration (m+n) implies m cards in one workstation and n in other), (2 + 1), (2 + 2), (3 + 3), (4 + 4), and the execution times are shown in Figure 13.

In the graph shown in Figure 13 it can be observed that the execution times are practically the same for cases where the application is executed both in a local or distributed fashion. Thus, there is no difference between executing with four cards in the same workstation or with two in each workstation. With respect to the execution time, this decreases proportionally to the number of cards; however, in the cases where six and seven GPUs are used in one configuration (3 + 3) and (4 + 3), respectively, the execution times are the same since the work distribution requires the same number of executions of the kernel for each card, in this case 36. This happens because the number of prisms for the case of 7 cards is not reduced for less than 7168 per card.

The behavior of the speed-up for a distributed execution is represented in Figure 14. It can be observed a practically linear speed-up and in some particular cases up to a super

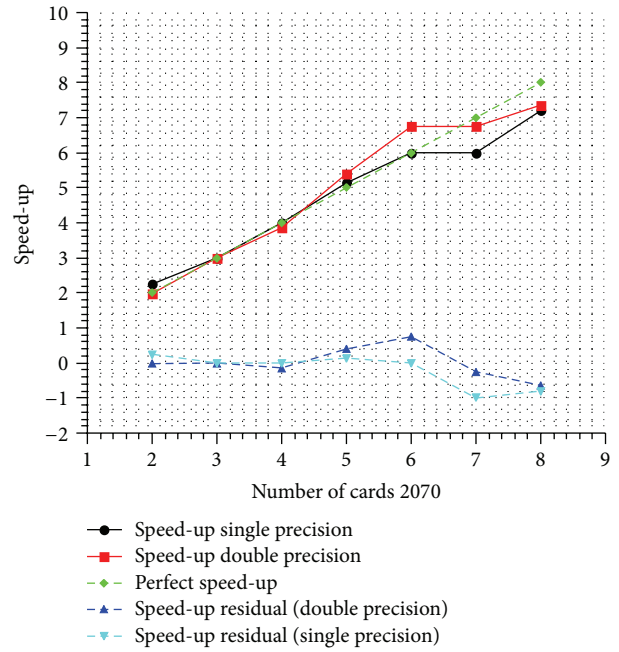

Figure 14: Speed-up obtained using shared and distributed graphics cards, in double and single precision.

speed-up, as is the case with the (3+3) configuration. Because of this we consider that the performance is excellent.

4.3. Comparison against a Cluster. To get a better perspective of the obtained performance with this CUDA implementation we compared it with the development made in OpenMP

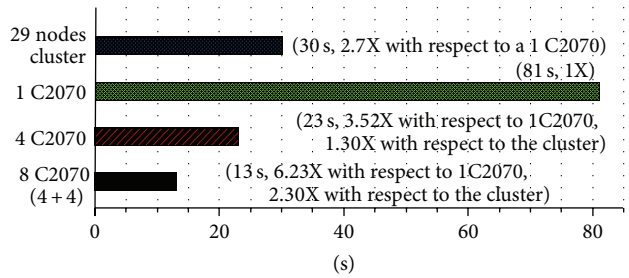

Figure 15: Comparison of the execution times for the calculation of the tensorial components in double precision between a small cluster against a single C2070 card, four C2070 cards in the same workstation, and 8 C2070 cards distributed in two workstations.

with MPI for cluster, and we also compared the results obtained in a cluster with the following characteristics:

(i) node: 1 Intel Xeon processors model X5550 with four physical cores per processor (2 threads per core),

(ii) 29 processing nodes,

(iii) hyperthreading technology enabled,

(iv) 40 GB of RAM per node,

(v) Red Hat 6.3 as operating system.

The results against which this implementation is compared were obtained from a very efficient hybrid design and its implementation using MPI and OpenMP for the calculation of the gravimetry and gradiometry. The comparison of the results shows that, using 29 nodes of the cluster, it is required an execution time of 30 s; in a C2070 card using a $\langle 14, 512, d\rangle$ configuration, the execution time is of 81 seg; four C2070 cards in the same machine communicated by MPI require 23 seg; and 8 distributed CUDA cards (4 per workstation) demand 13 s. With these times we composed the bar chart shown in Figure 15.

The comparison is made against 29 nodes because it is the optimal number of nodes for the distribution in a problem of 251,946 prisms with an observation grid of 15,000 points. It turns out that the cluster is 2.7X faster than a single CUDA 2070 card, but if we occupy 4 cards these are 1.3X times faster than the cluster, and we can say that there is an approximate equivalence, for this problem in particular, between 29 nodes and 4 C2070 cards, but if we occupy 8 cards distributively, these are 2.3X times faster than the cluster.

5. Validation of the Numerical CUDA Code

We now proceed to verify the quality of the numerical solution produced using single card C2070 and four C2070 cards. The main objective of using the CUDA programming is to reduce the computing time; however, the quality of the numerical results must be validated and verified.

Table 1: Errors of the components of the gravimetric tensor, calculated with CUDA in double precision, with respect to its sequential counterpart in double precision.

Gravity components	Error L2 1-GPU (DP)	Error L2 4-GPU (DP)
G_z	$2.0582e-09$	$2.0581e-09$
G_x	$1.7107e-09$	$1.7106e-09$
G_y	$1.1162e-09$	$1.1163e-09$

Table 2: Errors of the components of the gravimetric tensor, calculated with CUDA in single precision, with respect its sequential counterpart in double precision.

Gravity components	Error L2 1-GPU (SP)	Error L2 4-GPU (SP)
G_z	1.3080	1.3080
G_x	0.8929	0.8929
G_y	0.9275	0.9275

Table 3: Errors of the components of the gradient tensor of gravity in double precision, with respect to its sequential counterpart.

Gravity gradient tensor components	Error L2 1-GPU (DP)	Error L2 4-GPU (DP)
G_{zz}	$4.6290e-11$	$4.6373e-11$
G_{xx}	$7.7994e-11$	$7.8030e-11$
G_{yy}	$8.8905e-11$	$8.8935e-11$
G_{xy}	$5.3476e-11$	$5.3503e-11$
G_{xz}	$9.2797e-10$	$9.2792e-10$
G_{yz}	$2.7225e-10$	$2.7207e-10$

Table 4: Errors of the components of the gradient tensor of gravity in single precision, with respect to its sequential counterpart.

Gravity gradient tensor components	Error L2 1-GPU (SP)	Error L2 4-GPU (SP)
G_{zz}	0.0310	0.0310
G_{xx}	0.0385	0.0385
G_{yy}	0.0392	0.0392
G_{xy}	0.0743	0.0743
G_{xz}	0.3085	0.3085
G_{yz}	0.3148	0.3148

To measure the error we use the formula of L2 norm, or RMS, defined as [18]

$$e = \sqrt{\frac{1}{N_x N_y}\sum_{i=1}^{N_x}\sum_{j=1}^{N_y}\left|g_{i,j}^{p} - g_{i,j}^{s}\right|^2}, \tag{12}$$

where $g_{i,j}^{c}$ is the component of the tensor calculated using the GPU, and $g_{i,j}^{s}$ is the component calculated serially in the CPU.

In Table 1 the errors of the components of the gravimetric tensor parallelly calculated with a GPU are shown, using the configuration of $\langle 14, 512, d\rangle$, with respect to the serial version.

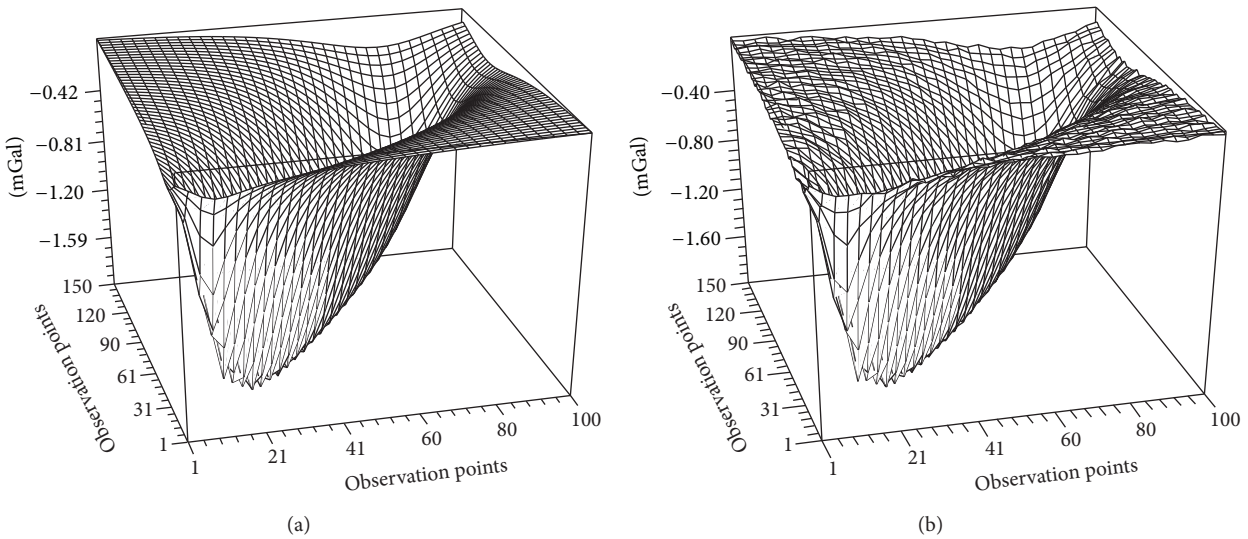

Figure 16: Component z of the vector $G(G_z)$, (a) calculated in double precision, (b) calculated in single precision. Apparently no significant differences are seen between the graphs; however there is the introduction of roughness in single precision.

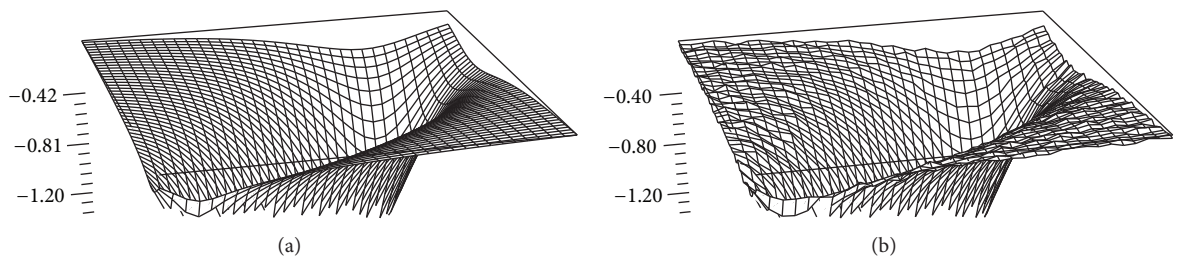

Figure 17: Zooming the component of the vector G_z, (a) calculated in double precision, (b) calculated in single precision. We can observe clearly the roughness introduced by the single precision.

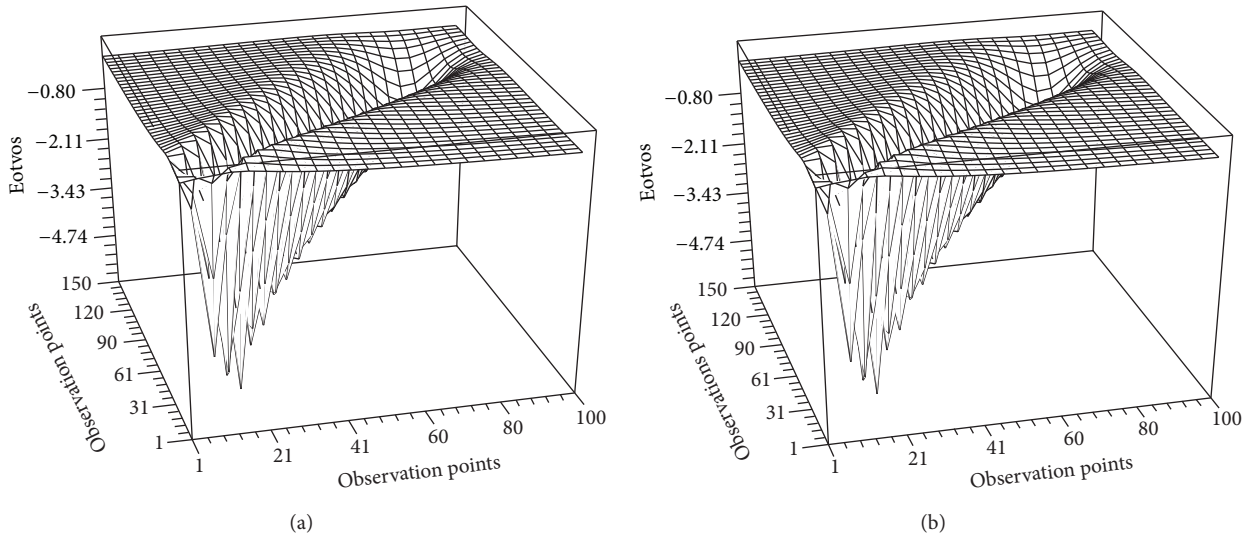

Figure 18: Component zz of the tensor $G(G_{zz})$, (a) calculated in double precision, (b) calculated in single precision. No significant differences are noticed between the plots.

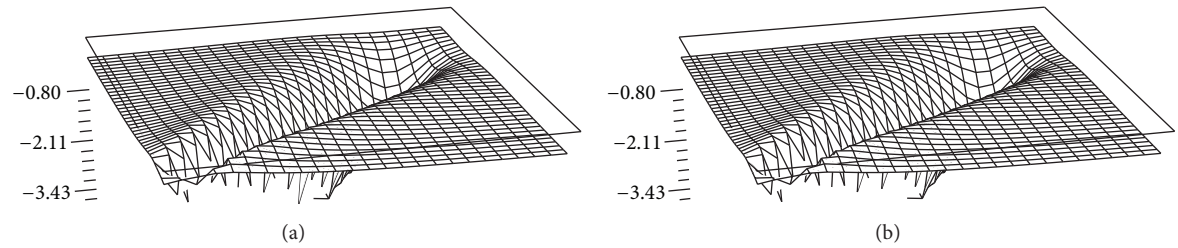

Figure 19: Zooming the component of the tensor G_{zz}, (a) calculated in double precision, (b) calculated in single precision. In this case the rugosity is not introduced by the single precision.

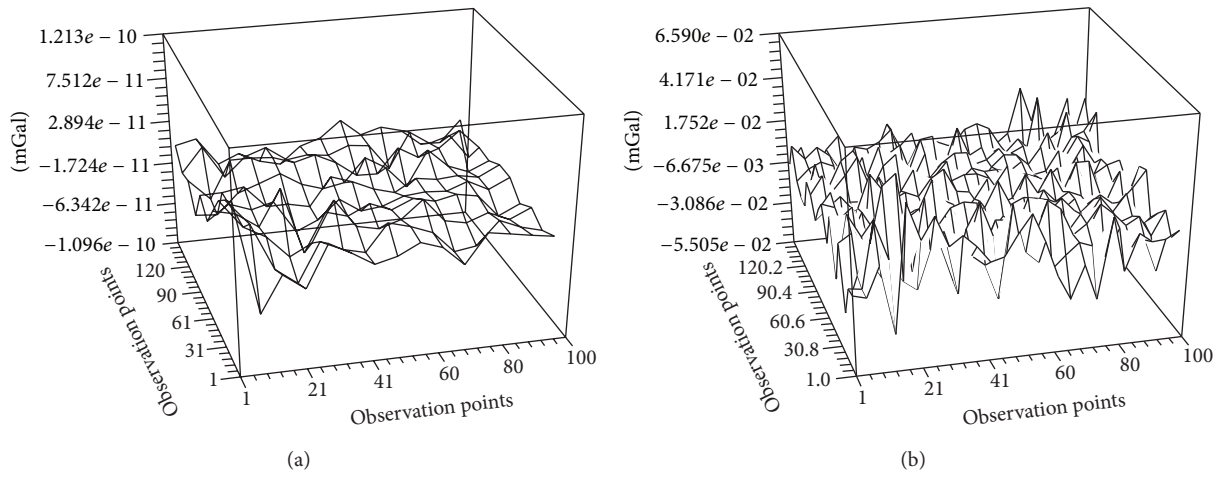

Figure 20: Absolute error of the components G_z calculated using four C2070 cards in (a) double precision and (*b*) single precision with respect to double precision CPU-cluster results.

The errors in single precision for the same configuration $\langle 14, 512, s\rangle$ are shown in Table 2.

The errors of the components of the gradient tensor of gravity calculated with CUDA in double precision, with respect to its sequential counterpart, are shown in Table 3.

And finally in Table 4 the errors of the components of the gradient tensor of gravity are shown, calculated with CUDA in single precision, with respect to its sequential counterpart.

It is necessary to mention that the sequential version is calculated in double precision and, as can be seen, there is practically no difference between the CUDA version in double precision and the reference solution. Nevertheless, in single precision the calculation of the vectors produces an error more significant than the calculation of the tensors. To observe how the error propagates in single precision, we show in Figure 16 the calculation of G_z in double and single precision, using a single GPU.

As can be noted, apparently the same anomaly result is reproduced both in double precision as in single, but in this last one the numerical roughness is pronounced. To examine this phenomenon in greater detail, we can zoom in Figure 16 to see with more detail the introduced rugosity depicted in Figure 17.

However, the numerical rugosity problem in single precision is left to the criterion of anyone interested in the result, since this precision requires practically 40% and 50% less in computing time and global memory, respectively.

To note that the single precision does not always introduce pronounced numerical rugosities, we can observe the behavior of the calculation of G_{zz}, which does not introduce rugosity problems. Figure 18 is shows the G_{zz} tensor calculated in double and single precision, and Figure 19 is zoomed to focus on the detail.

Finally we mention that single precision can be used if done properly, depending on the requirements of the particular problem. The maximum absolute error found in G_z with single precision is of 0.0566 mGal and for G_{zz} of 0.0011 Eotvos (see Figures 20 and 21). As we can see in the calculation of G_{zz} the introduced error is less than in G_z.

6. Conclusions

A parallel design for the calculation of the vectorial and tensorial components of the gravity using CUDA was implemented and validated. The numerical experiments and the obtained metrics validate that the implementation is very efficient and that it also produces good results with respect to the numerical solution.

We showed that selecting the simplest or most trivial parallelization technique does not necessarily leads to the best performance or the best use of the platform. In our particular case, even though the partitioning by prisms requires a greater inversion in the design and implementation, this

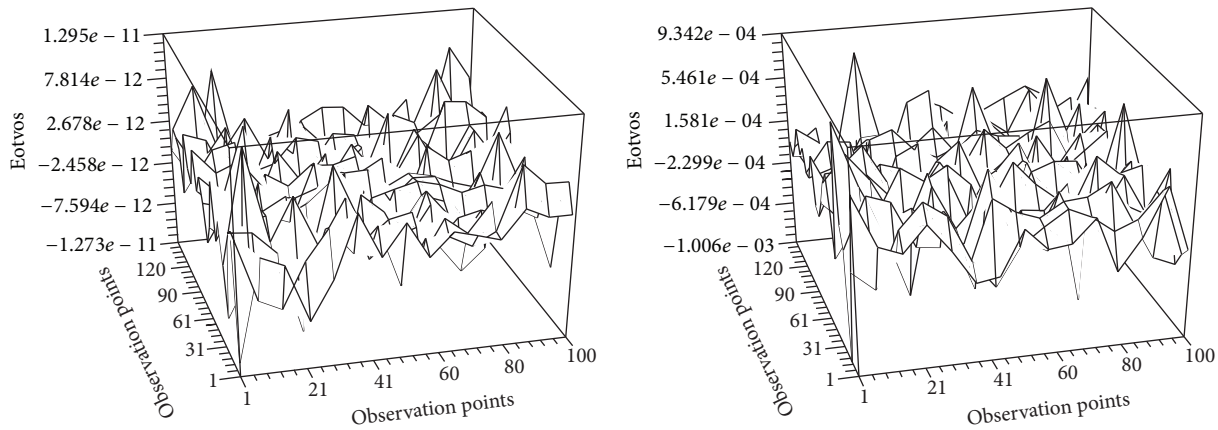

Figure 21: Absolute error of the components G_{zz} calculated using four C2070 cards in (a) double precision and (b) single precision with respect to double precision CPU-cluster results.

method is the most advantageous with respect to performance because the transfers between the CPU and the GPU are minimized, moving more executable code to the GPU and also the number of calls to the kernel functions is reduced.

The multi-GPU version using MPI as controller and balancer of the workload was correctly implemented since it produces practically the same results as those of the version for just one GPU. In double precision we can say that there is no difference between the calculations by the CPU and the GPU. The single precision can be used with confidence in the calculation of the tensorial components and, with appropriate considerations, in the calculation of the vectorial components as well.

It was shown that for our synthetic problem, approximately 29 nodes are equivalent to 4 four C2070 cards. This obviously shows the economical benefit of using CUDA, since it is cheaper to acquire 4 graphic cards than 29 nodes of processing, and clearly the maintenance and energetic consume is considerably smaller. Nevertheless, we consider that the CUDA implementation is much more costly from the point of view of the design and the required time for its programming.

We can also conclude that this implementation can serve as a design pattern to parallelize numerical schemes where the computational space cannot be disjointly divided between the processing cores, therefore minimizing the execution of the number of kernels calls.

Finally we expect that GPU computing will enable us, in a near future, to optimize the numerical burden of large scale geophysical applications such as potential field modeling of impact craters [19] and multiparameter geophysical global optimization by heuristic methods [20, 21].

Appendix

Calculation of Gravitational Quantities

The Earth's gravitational potential G is a scalar quantity and its shape can be constrained by its slope in the x, y, and z directions, called the gravitational attraction G_x, G_y, and G_z (gravity vector field). In this work, we have investigated how to parallelize the analytical calculation of the components of the gravity field vector and the gravity gradients represented by a nine-component tensor; because of the symmetrical or irrotational attribute, the gravity gradient tensor is reduced to only six independent components: G_{xx}, G_{yy}, G_{zz} (the vertical gravity gradient), G_{xy}, G_{xz}, and G_{yz}. For the right rectangular prism model, the analytical formulae for the three components vectors and the six gravity gradient components, corresponding to (2), are given by

$$G_x = \gamma\rho\sum_{i=1}^{2}\sum_{j=1}^{2}\sum_{k=1}^{2}\mu_{ijk}\left[y_j \ln\left(z_k + r_{ijk}\right) + z_k \ln\left(y_j + r_{ijk}\right) - x_i \arctan\frac{y_j z_k}{x_i r_{ijk}}\right],$$

$$G_y = -\gamma\rho\sum_{i=1}^{2}\sum_{j=1}^{2}\sum_{k=1}^{2}\mu_{ijk}\left[z_k \ln\left(x_i + r_{ijk}\right) + x_i \ln\left(z_k + r_{ijk}\right) - y_j \arctan\frac{z_k x_i}{y_j r_{ijk}}\right],$$

$$G_z = -\gamma\rho\sum_{i=1}^{2}\sum_{j=1}^{2}\sum_{k=1}^{2}\mu_{ijk}\left[x_i \ln\left(y_j + r_{ijk}\right) + x_i \ln\left(z_k + r_{ijk}\right) - y_j \arctan\frac{z_k r_{ijk}}{x_i y_j}\right],$$

$$G_{xx} = \gamma\rho \sum_{i=1}^{2}\sum_{j=1}^{2}\sum_{k=1}^{2} \mu_{ijk} \arctan \frac{y_j z_k}{x_i r_{ijk}},$$

$$G_{yy} = \gamma\rho \sum_{i=1}^{2}\sum_{j=1}^{2}\sum_{k=1}^{2} \mu_{ijk} \arctan \frac{x_i z_k}{y_j r_{ijk}},$$

$$G_{zz} = \gamma\rho \sum_{i=1}^{2}\sum_{j=1}^{2}\sum_{k=1}^{2} \mu_{ijk} \arctan \frac{x_i y_j}{z_k r_{ijk}},$$

$$G_{xy} = \gamma\rho \sum_{i=1}^{2}\sum_{j=1}^{2}\sum_{k=1}^{2} \mu_{ijk} \ln\left(z_k + r_{ijk}\right),$$

$$G_{xz} = \gamma\rho \sum_{i=1}^{2}\sum_{j=1}^{2}\sum_{k=1}^{2} \mu_{ijk} \ln\left(y_j + r_{ijk}\right),$$

$$G_{yz} = \gamma\rho \sum_{i=1}^{2}\sum_{j=1}^{2}\sum_{k=1}^{2} \mu_{ijk} \ln\left(x_i + r_{ijk}\right), \tag{A.1}$$

where $r_{ijk} = \sqrt{x_i^2 + y_j^2 + z_k^2}$, $\gamma = 6.673 \times 10^{-11}$ and $x_1 = (x_l - x_p) \times 10^3$, $x_2 = (x_r - x_p) \times 10^3$, $y_1 = (y_l - y_p) \times 10^3$, $y_2 = (y_r - y_p) \times 10^3$, $z_1 = (z_l - z_p) \times 10^3$, $z_2 = (z_r - z_p) \times 10^3$.

Glossary

CPU:	Central processing unit, of one or several cores in shared memory
Device or GPU:	Graphics processing unit. For instance, the Tesla C2070 card
Device function:	Is a function that can only be executed by a CUDA thread and is called by a kernel
Grid:	A set of blocks of threads. A kernel is executed in a grid of blocks
CUDA thread:	A CUDA thread is a light process which executes a distinct sequence of the contained code in a kernel and resides in the GPU
Block identifier:	It is analogous to the thread identifier and identifies the block within a grid. It is accessed with the variable `blocKIdx`
Thread identifier:	It is a value between 0 and n in C language or between 1 and n in FORTRAN language, which functions as thread identifier within a block. It is accessed with the variable `threadIdx`. This variable is very useful to distribute the work among different threads and can be handled up to 3 components (x, y, z), coinciding with the dimensions of blocks of threads, but not necessarily
Kernel:	A function or procedure executed parallelly in the device which is executed by the CUDA threads
Global memory:	Uncached off-chip DRAM memory
Multiprocessor:	It is a processing unit containing 8 CUDA cores
CUDA core:	It is a core of processing contained inside a multiprocessor and dispatches the threads contained in a block
Numerical rugosity:	It is defined for this work as numerical rugosity to the effect produced by truncate and round to 7 decimals the precision of the floating-point numbers, which produces values above or below the exact solution
Block size:	It indicates the number of threads contained inside the block and it is accessed through the variable `blocKDim`. It can contain the three dimensions
Multiprocessor occupancy:	It is the quotient of the number of warps executing concurrently in a multiprocessor, divided between the maximum number of warps which can be executed concurrently
Warp:	It is a group of 32 threads which execute concurrently in a GPU multiprocessor.

Acknowledgments

The authors want to acknowledge the support provided by the Mexican Institute of Petroleum to access its computing equipment, based on the financing received through project SERNER-CONACYT 128376 (IMP Y.00107) created jointly by IMP-SENER-CONACYT. They want to express their special

gratitude to Dr. Raul del Valle and Dr. Oleg Titov for all the support received in their laboratory.

References

[1] R. Vuduc and K. Czechowski, "What GPU computing means for high-end systems," *IEEE Micro*, vol. 31, no. 4, pp. 74–78, 2011.

[2] J. D. Owens, D. Luebke, N. Govindaraju et al., "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007.

[3] C. J. Webb and S. Bilbao, "Computing room acoustics with CUDA-3D FDTD schemes with boundary losses and viscosity," in *Proceedings of the 36th IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '11)*, pp. 317–320, May 2011.

[4] N. Nakata, T. Tsuji, and T. Matsuoka, "Acceleration of computation speed for elastic wave simulation using a Graphic Processing Unit," *Exploration Geophysics*, vol. 42, no. 1, pp. 98–104, 2011.

[5] R. Abdelkhalek, H. Calandra, O. Coulaud, J. Roman, and G. Latu, "Fast seismic modeling and reverse time migration on a GPU cluster," in *Proceedings of the International Conference on High Performance Computing and Simulation (HPCS '09)*, pp. 36–43, June 2009.

[6] J. Yang, Y. Wang, and Y. Chen, "GPU accelerated molecular dynamics simulation of thermal conductivities," *Journal of Computational Physics*, vol. 221, no. 2, pp. 799–804, 2007.

[7] T. Brandvik and G. Pullan, "Acceleration of a two-dimensioanl Euler flow solver using commodity graphics hardware," *Journal of Mechanical Engineering Science*, vol. 221, no. 12, pp. 1745–1748, 2007.

[8] R. Capuzzo-Dolcetta, A. Mastrobuono-Battisti, and D. Maschietti, "NBSymple, a double parallel, symplectic N-body code running on graphic processing units," *New Astronomy*, vol. 16, no. 4, pp. 284–295, 2011.

[9] L.-G. Du, K. Li, F.-M. Kong, and Y. Hu, "Parallel 3D finite-difference time-domain method on multi-GPU systems," *International Journal of Modern Physics C*, vol. 22, no. 2, pp. 107–121, 2011.

[10] I. Foster, *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*, Addison-Wesley Longman, Boston, Mass, USA, 1995.

[11] D. Michéa and D. Komatitsch, "Accelerating a three-dimensional finite-difference wave propagation code using GPU graphics cards," *Geophysical Journal International*, vol. 182, no. 1, pp. 389–402, 2010.

[12] M. Moorkamp, M. Jegen, A. Roberts, and R. Hobbs, "Massively parallel forward modeling of scalar and tensor gravimetry data," *Computers and Geosciences*, vol. 36, no. 5, pp. 680–686, 2010.

[13] L. Dagum and R. Menon, "Openmp: an industry-standard api for shared-memory programming," *IEEE Computing in Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[14] T.-Y. Liang, H.-F. Li, and J.-Y. Chiu, "Enabling mixed openmp/mpi programming on hybrid cpu/gpu computing architecture," in *Proceedings of the IEEE 26th International Parallel and Distributed Processing Symposium Workshops (IPDPSW '12)*, pp. 2369–2377, 2012.

[15] D. Komatitsch, "Fluid-solid coupling on a cluster of GPU graphics cards for seismic wave propagation," *Comptes Rendus Mécanique*, vol. 339, no. 2-3, pp. 125–135, 2011.

[16] B. Heck and K. Seitz, "A comparison of the tesseroid, prism and point-mass approaches for mass reductions in gravity field modelling," *Journal of Geodesy*, vol. 81, no. 2, pp. 121–136, 2007.

[17] Z. Chen, X. Meng, and L. Guo, "Gicuda: a parallel program for 3d correlation imaging of large scale gravity and gravity gradiometry data on graphics processing units with cuda," *Computers and Geosciences*, vol. 46, pp. 119–128, 2012.

[18] K. L. Mickus and J. H. Hinojosa, "The complete gravity gradient tensor derived from the vertical component of gravity: a Fourier transform technique," *Journal of Applied Geophysics*, vol. 46, no. 3, pp. 159–174, 2001.

[19] C. Ortiz-Alemán and J. Urrutia-Fucugauchi, "Aeromagnetic anomaly modeling of central zone structure and magnetic sources in the Chicxulub crater," *Physics of the Earth and Planetary Interiors*, vol. 179, no. 3-4, pp. 127–138, 2010.

[20] M. G. Orozco-del Castillo, C. Ortiz-Alemán, J. Urrutia-Fucugauchi, R. Martin, A. Rodriguez-Castellanos, and P. E. Villase~nor-Rojas, "A genetic algorithm for filter design to enhance features in seismic images," *Geophysical Prospecting*, 2013.

[21] J. L. Rodríguez-Zúñiga, C. Ortiz-Alemán, G. Padilla, and R. Gaulon, "Application of genetic algorithms to constrain shallow elastic parameters using in situ ground inclination measurements," *Soil Dynamics and Earthquake Engineering*, vol. 16, no. 3, pp. 223–234, 1997.