Simulation of sonic waves along a borehole in a heterogeneous formation: Accelerating 2.5-D finite differences using [Py]OpenCL

Ursula Iturrarán-Viveros a,*, Miguel Molero b

a Facultad de Ciencias, Universidad Nacional Autónoma de México, Circuito Escolar S/N, Coyoacán C.P. 04510, México D.F., Mexico
b Centro de Acústica Aplicada y Evaluación No Destructiva, CAEND (CSIC), Arganda del Rey, 28500 Madrid, Spain

ARTICLE INFO

Article history:
Received 21 January 2013
13 March 2013
Accepted 14 March 2013
Available online 27 March 2013

Keywords:
Borehole modeling
Cylindrical 2.5-D finite differences
Sonic well-logs
Graphics cards
Accelerating GPUs
PyOpenCL

ABSTRACT

This paper presents an implementation of a 2.5-D finite-difference (FD) code to model acoustic full waveform monopole logging in cylindrical coordinates accelerated by using the new parallel computing devices (PCDs). For that purpose we use the industry open standard Open Computing Language (OpenCL) and an open-source toolkit called PyOpenCL. The advantage of OpenCL over similar languages is that it allows one to program a CPU (central processing unit), a GPU (graphics processing unit), or multiple GPUs and their interaction among them and with the CPU, or host device. We describe the code and give a performance test in terms of speed using six different computing devices under different operating systems. A maximum speedup factor over 34.2, using the GPU is attained when compared with the execution of the same program in parallel using a CPU quad-core. Furthermore, the results obtained with the finite differences are validated using the discrete wavenumber method (DWN) achieving a good agreement. To provide the Geoscience and the Petroleum Science communities with an open tool for numerical simulation of full waveform sonic logs that runs on the PCDs, the full implementation of the 2.5-D finite difference with PyOpenCL is included.

© 2013 Elsevier Ltd. All rights reserved.

1. Introduction

The problem of propagation of elastic waves in a fluid contained in a circular borehole through an elastic solid of infinite extent has been widely studied during the last decades, to mention but a few authors see for example, Biot (1952), Kurkjian et al. (1994), Bouchon (1993), Tichelaar et al. (1995), Dong and Toksöz (1995), Paillet and White (1982), Paillet and Cheng (1986), Schmitt (1989), Kostek et al. (1998), and Tang and Cheng (2004). Although the topic of modeling full wave forms for sonic logging has been covered to a great extent in the literature, the objective of this paper is to give a novel implementation to model elastic wave propagation of the acoustic monopole logging tool that accelerates the FD simulations by using either GPUs or multi-core CPUs which are cheaper than large clusters of CPUs. Parallel FD algorithms have been used either with the Message-Passing-Interface (MPI) technique or OpenMP architecture, see (Sheena et al., 2006). Despite these implementations can significantly reduce the overall computation time for sonic well-logging simulations, the cost of building these systems is very expensive. We use a FD method similar to the one of Mittet and Renile (1996), Randall et al. (1991) and Chang and Randall (1988), but we do not include viscoelasticity. We assume an isotropic medium (i.e. only the two Lamé elastic constants will be considered $\lambda$ and $\mu$).

GPUs highly parallel structure makes them more effective than general-purpose CPUs for algorithms where processing of large blocks of data is done in parallel. Therefore, GPUs offer tremendous potential for performance and efficiency to important large-scale applications of computational science and engineering. However, exploiting this potential can be challenging, as one must adapt to the specialized and rapidly evolving computing environment currently exhibited by these parallel computing devices. We choose to use OpenCL since it does not assume any particular computing device, it does not provide any concept corresponding to the wrap specific to NVIDIA's GPUs, and wavefronts specific to AMD's ones. In addition, OpenCL is suitable for parallel programming using multi-core CPU devices. By eliminating such vendor-specific definitions from the specification, OpenCL can support various kinds of parallel computing devices such as any brand of GPU or CPU, Field-Programmable Gate Array (FPGA), digital signal processors (DSP), IBM's Cell Broadband engine and more. On the other hand, a dynamic high-level scripting language with the massive performance: PyOpenCL gives easy access to OpenCL parallel computation through Python. The advantage of PyOpenCL is that it is a complete mature application programming interface (API) wrapper, it is easy to program, no need to compile and it has automatic error checking. For a comprehensive documentation on OpenCL see Munshi et al. (2012) and for PyOpenCL see Klöckner (2010) and Klöckner et al. (2012).
Several examples of seismic applications using GPUs have been recently published, but most of the literature is dedicated to implementations in CUDA (Compute Unified Device Architecture) developed for NVIDIA GPUs with their benefits and restrictions. Some of the most thriving papers are by Michéa and Komatitsch (2010), Komatitsch et al. (2010) and Komatitsch (2011). The authors based their implementations on a previous work by Micikevicius (2009). This approach utilizes thousands of threads (called work items in OpenCL), traversing the 3-D volume slice-by-slice as a 2-D front of threads to maximize data reuse from shared memory (local memory in OpenCL).

2. Mathematical formulation

Axisymmetric 2-D modeling is an economical approach to model 3-D wavefields by an approximation of the structural model to be axisymmetric along the borehole axis (see Fig. 1(b)), including the source, and then solving the elastodynamic equation in cylindrical coordinates. The computer memory requirements are only slightly larger than those for 2-D computations but reduces the computational time to nearly as short as that of the 2-D calculations. We consider the cylindrical coordinate system \((r, \theta, z)\) where \(0 \leq r \leq \infty\), \(0 \leq \theta < 2\pi\) and \(-\infty < z < \infty\), see Fig. 1(a). The wave equations for 2-D finite difference formulation are given by

\[
\rho \frac{\partial^2 v_r}{\partial t^2} = \frac{1}{r} \frac{\partial}{\partial r} \left( r \frac{\partial v_r}{\partial r} \right) + \frac{1}{r^2} \frac{\partial^2 v_r}{\partial \theta^2} - \frac{1}{r} \frac{\partial \sigma_{r\theta}}{\partial r},
\]

(1)

\[
\rho \frac{\partial^2 v_\theta}{\partial t^2} = \frac{1}{r} \frac{\partial}{\partial r} \left( r \frac{\partial v_\theta}{\partial r} \right) + \frac{1}{r^2} \frac{\partial^2 v_\theta}{\partial \theta^2} - \frac{\sigma_{r\theta}}{r},
\]

(2)

\[
\rho \frac{\partial^2 v_z}{\partial t^2} = \frac{1}{r} \frac{\partial}{\partial r} \left( r \frac{\partial v_z}{\partial r} \right) + \frac{1}{r^2} \frac{\partial^2 v_z}{\partial \theta^2} - \frac{1}{r} \frac{\partial \sigma_{rz}}{\partial r},
\]

(3)

where \(\rho\) is the density, \(v_r\), \(v_\theta\) and \(v_z\) are the velocity components, the time derivatives are denoted by a dot (i.e. if \(u_i\) is the radial displacement, then \(v_i = \partial u_i / \partial t = \dot{u}_i\), similarly with the other components) and here we are not considering body force components.

The velocity–stress relations are given by

\[
\sigma_{rr} = (\lambda + 2\mu) \frac{\partial v_r}{\partial r} + \lambda \left( \frac{v_r}{r} + \frac{1}{r} \frac{\partial v_\theta}{\partial \theta} + \frac{\partial v_z}{\partial z} \right),
\]

(4)

\[
\sigma_{r\theta} = (\lambda + 2\mu) \left( \frac{v_r}{r} + \frac{1}{r} \frac{\partial v_\theta}{\partial \theta} \right) + \lambda \left( \frac{v_\theta}{r} + \frac{1}{r} \frac{\partial v_r}{\partial \theta} + \frac{\partial v_z}{\partial z} \right),
\]

(5)

\[
\sigma_{rz} = (\lambda + 2\mu) \frac{\partial v_z}{\partial z} + \lambda \left( \frac{v_z}{r} + \frac{1}{r} \frac{\partial v_r}{\partial \theta} + \frac{1}{r} \frac{\partial v_\theta}{\partial r} \right),
\]

(6)

\[
\sigma_{\theta\theta} = \mu \left( \frac{\partial v_r}{\partial r} + \frac{\partial v_z}{\partial z} \right),
\]

(7)

\[
\sigma_{\theta z} = \mu \left( \frac{\partial v_\theta}{\partial \theta} + \frac{1}{r} \frac{\partial v_z}{\partial r} \right),
\]

(8)

\[
\sigma_{zz} = \mu \left( \frac{\partial v_z}{\partial r} + \frac{\partial v_\theta}{\partial \theta} \right).
\]

(9)

The angular dependencies are simple sinusoids with frequency \(n\). There is a phase shift between the different components, such that if the angular dependence for the \(v_\theta, \sigma_{r\theta},\) and \(\sigma_{\theta\theta}\) is \(\sin(n\theta)\), the angular dependence for the others is \(\cos(n\theta)\). In this work we only consider the case of a monopole source \((n=0)\), but multipole sources can be easily taken into account just by taking \(n > 0\).

3. Numerical implementation

In this section we give a discretization that we use to numerically approximate and solve the set of equations given in Eqs. (1)–(9). Let us consider common numerical schemes to compute partial derivatives such as forward and backward, spatial, first order approximations denoted by \(\partial_i^+ u_j(x_i)\) and \(\partial_i^- u_j(x_i)\) respectively, where \(i, j = \{ x, y, z \}\) or in cylindrical coordinates \(i, j = \{ r, \theta, z \}\). Sometimes these are also known as the Euler scheme or method. We will take the mean for density and the harmonic mean for the shear modulus \(\mu\).

We need harmonic average as this makes the shear stresses on the boundary go to zero if \(\mu = 0\) (i.e. if the medium is a fluid) on any adjoining constant property cell, whereas using a mean we could not

**Fig. 1.** (a) Configuration of the coordinate system. (b) Configuration of the fluid filled-borehole with axial symmetry (along the borehole) and the staggered grid used for the spatial discretization.
achieve this. Since we are using staggered grids we need forward and backward mean operators defined as follows:

\[ I_f^+ [u_i(x_i)] = \frac{1}{2} (u_i(x_{i+1}) + u_i(x_i)) - u_i(x_{i+1/2}), \]

(10)

\[ I_b^+ [u_i(x_i)] = \frac{1}{2} (u_i(x_i) + u_i(x_{i-1})) - u_i(x_{i-1/2}). \]

(11)

For some cases we need the derivative of a variable in the middle of two grid points, since we are using a simple forward or backward first order spatial derivative this is not a problem. However, we might need the values of the material properties at points in the middle of grid points, so we have to use the average of the neighborhood's values. Using these previous definitions and having in mind the staggered grid and the place where all variables are located, see Fig. 1(b), the spatial partial derivatives of the stresses are also computed with a forward or backward first order scheme.

Next we show the discretization of Eqs. (1)–(9). The spatial and temporal staggering of the velocity and stresses is necessary so that all FD approximations be properly centered. Velocities are updated first, followed by stresses, i.e.

\[ v^{l+1} = v^l, \]

(12)

\[ \sigma^{l+1/2} = \sigma^{l+1/2}, \]

(13)

\( I \) being the \( l \)-th time step. In a discrete form Eq. (1) is given as follows:

\[ v^l_{t+1} = v^l_t + \frac{\Delta t}{l} [I_f^+ \left( \frac{\sigma_{rz}}{r} \right) + \partial_r \sigma_{rr} + \frac{\Delta r \sigma_{zr}}{r} + \partial_z \sigma_{zz} - I_b^+ \left( \frac{\sigma_{r\theta}}{r^2} \right) + \frac{\Delta \theta \sigma_{r\theta}}{r^2}]. \]

(14)

where \( I_f^+ \) and \( I_b^+ \) denote the forward and backward averaging operators given in (10) and (11) respectively. Similarly, the discrete form of (2) is given by

\[ \sigma^l_{\theta} = \sigma^l_{\theta} + \frac{\Delta t}{\rho} \left[ I_f^+ \left( \frac{2 \rho \sigma_z}{r} \right) + \partial_r \sigma_{r\theta} - \frac{\Delta r \rho \sigma_{zr}}{r} + \frac{\Delta \theta \rho \sigma_{z\theta}}{r} \right]. \]

(15)

Note that here we did not average \( \rho \) because both variables \( v_t \) and \( \rho \) are defined on the same nodes of the grid. In the same way for (3) we have

\[ v^l_{z+1} = v^l_z + \frac{\Delta t}{l} [I_f^+ \left( \frac{\sigma_{rz}}{r} \right) + \partial_r \sigma_{zz} + \frac{\Delta r \sigma_{zr}}{r} + \partial_z \sigma_{zz}]. \]

(16)

Once we have numerically solved Eqs. (1), (2) and (3) for \( v^{l+1}, v^{l+1}_t \) and \( v^{l+1}_z \) respectively, we can feed these velocities into the discretized stresses as follows:

\[ \sigma^{l+1}_{rr} = \sigma^{l-1}_{rr} + \Delta t \left( \lambda + 2 \mu \right) \delta^l_r v_r + \lambda \left[ I_f^+ \left( \frac{v_r}{r} \right) + \frac{\Delta r \mu v_r}{r} + \partial_r \sigma_{zr} \right] + \frac{\partial_r m_{rr}}{r}. \]

(17)

\[ \sigma^{l+1}_{r\theta} = \sigma^{l-1}_{r\theta} + \Delta t \left( \lambda + 2 \mu \right) \delta^l_r v_\theta + \lambda \left[ I_f^+ \left( \frac{v_\theta}{r} \right) + \frac{\Delta r \mu v_\theta}{r} + \partial_\theta \sigma_{r\theta} \right] + \frac{\partial_\theta m_{r\theta}}{r}. \]

(18)

\[ \sigma^{l+1}_{zz} = \sigma^{l-1}_{zz} + \Delta t \left( \lambda + 2 \mu \right) \delta^l_r v_z + \lambda \left[ I_f^+ \left( \frac{v_z}{r} \right) + \frac{\Delta r \mu v_z}{r} + \partial_z \sigma_{zz} \right] + \frac{\partial_z m_{zz}}{r}. \]

(19)

\[ \sigma^{l+1}_{\theta\theta} = \sigma^{l-1}_{\theta\theta} + \Delta t H_{\theta} \left[ \mu_\theta \left( -\frac{\Delta \theta \mu v_\theta}{r^2} \right) \right], \]

(20)

\[ \sigma^{l+1}_{z\theta} = \sigma^{l-1}_{z\theta} + \Delta t H_{\theta} \left[ \delta^l r v_\theta - \frac{\Delta \theta \mu v_\theta}{r^2} \right], \]

(21)

\[ \sigma^{l+1}_{z\theta} = \sigma^{l-1}_{z\theta} + \Delta t H_{\theta} \left[ \delta^l r v_\theta - \frac{\Delta \theta \mu v_\theta}{r^2} \right], \]

(22)

where

\[ H_{\theta} [\mu_{\theta}] = \frac{2 \mu_{\theta} \mu_{\theta} + 1}{(\mu_{\theta} + \mu_{\theta} + 1)}, \]

(23)

\[ H_{\theta} [\mu_{\theta}] = \frac{2 \mu_{\theta} \mu_{\theta} + 1}{(\mu_{\theta} + \mu_{\theta} + 1)}. \]

(24)

are harmonic averaging and the derivatives of the moment tensors are given by \( m_{rr}, m_{r\theta} \) and \( m_{zz} \). In Eq. (20) we need to average \( \mu \) in the \( z \)-direction because \( \sigma_{z\theta} \) lives on nodes half size shifted than \( \mu \), in the \( z \)-direction. In contrast to the conventional FD operators, high-order FD operators require evaluation of the displacements and the stresses in the fluid, also for negative \( r \)-values in the neighborhood of \( r = 0 \). Since mathematically \( r < 0 \) is forbidden in cylindrical coordinates, this evaluation is also forbidden, so that this evaluation is a pure mathematical trick. The \( r \) dependence for negative \( r \) values can be found from analytical expressions for the field quantities, see (Kurkjian and Chang, 1986). Using these expressions it can be shown that

\[ v_\theta(t) = (-1)^n v_\theta(-t), \]

(25)

\[ \sigma_{z\theta}(t) = (-1)^n \sigma_{z\theta}(-t), \quad n = 0, 1, 2, \]

(26)

where \( n = 0 \) is a monopole and \( n = 1 \) is a dipole. We apply absorbing boundaries as in (Cerjan et al., 1985) for the top, bottom and right boundaries of the domain. The left boundary is where the axis of the borehole is located therefore we have a free-boundary condition along this boundary side.

### 3.1. Singularities (at \( r=0 \)) and source

Since mathematical singularities introduced by changing Cartesian to cylindrical coordinates, due to the existence of a singular solution in the fluid column (when \( r=0 \)) the problem is not well posed, see (Gustafsson et al., 1995). Some terms in the finite-difference scheme need special treatment at \( r=0 \) to avoid division by zero. From symmetry considerations, we find that only symmetric field quantities have non-zero values at \( r=0 \). Applying this and the fact that \( \sigma_{ij} = 0 \) for \( i\neq j \), the only non-zero quantities at \( r=0 \) (in the monopole case) are \( \sigma_{ij}, i=j=0, \theta, z \) and \( v_\theta \). We exploit the formulae derived from limiting operations with the L'Hôpital rule, further details that can be found in (Takenaka et al., 2003). The point source that we are considering is given by

\[ f(r, \phi, z) = \delta(r-r_0)\delta(z-z_0)R(t), \]

(27)

where \( \delta \) is the impulse function, \((r_0, z_0)\) is the coordinate where the source is located, \( R(t) \) is a Ricker wavelet given by \( R(t) = (b^2-1/2) \exp(-b^2), \) where \( b = t-t_0 \)/\( t_p \), the characteristic period is \( t_p \) and the delay is \( t_0 \). We apply the source term to the main stresses. We consider a point source located at the borehole axis i.e. \( r_0 = 0 \) which corresponds to a monopole source. When \( r_0 > 0 \) this represents a ring function of radius \( r_0 \) oriented in the plane \( z = z_0 \) and centered at the borehole axis, but this case will not be considered here.

### 4. Validation and numerical results

In order to validate the finite differences we compare the results obtained with a reference solution obtained by means of the discrete wavenumber method (DWM). A complete description of the DWM method can be found in (Tadeu et al., 2001), (Bouchon, 2003), and (Cheng and Blanch, 2008). In Fig. 2 we can see the traces for the reference solution and for the FD. The case when there is a prospecting tool it is not considered. The source is placed at \( r_0 = 0 \) and \( z_0 = 3 \) m, the nearest receiver is located at \( r_1 = 0, z_1 = 5.7 \) m and the separation between each of the receivers is \( 0.15 \) m. We consider eight receivers as standard logging tools (although new tools have thirteen receivers). We choose a canonical case for a monopolar source and a fast formation (see Table 1). We see that the reference solution and the numerical solution obtained using the PyOpenCl code are in good
agreement. At the top of Fig. 2 we plot the eight traces and at the bottom we have a zoom of first and the last one.

We use a Ricker wavelet with characteristic period $t_p = \frac{1.25}{C^2} \times 10^{-4}$ s and the delay is $t_s = 2t_p$, the central frequency of the pulse being $f_c = 8000$ Hz. For all numerical experiments shown here we consider the radius of the borehole equals to 0.1 m. Once the numerical method is validated we use the code to compute some canonical cases, see Tang and Cheng (2004). Table 1 shows the elastic/liquid properties used in these simulations. In order to assure numerical stability we apply the same Courant condition as in Cartesian coordinates and we take $\Delta r = \Delta z$.

We denote the velocity of the compressional wave propagating through the liquid as $V_f$ and the density of the fluid as $\rho_f$. Figs. 3 and 4 show the cases of a monopole source in a fast and slow formations, respectively. We have included in Fig. 5 a case where there is an invaded zone of width 0.5 m located next to the borehole and a slow layer intercepting the borehole, the size of this layer is 0.3 m. Table 2 summarizes the computational parameters used for these simulations. The number of grid points along the $r$-axis is $N_R$ and along the $z$-axis is $N_Z$. The total number of times steps is $N_T = 4500$. The central frequency is $f_c = 8000$ Hz.

### 5. Performance analysis

Average computation times are calculated to compare the effectiveness of the parallel code running the PyOpenCL in a GPU versus CPU. We have tested the code speed for 19 different numbers of grid sizes. The experiment platforms consist of five kinds of GPUs and one CPU multi-core listed in Table 3. In addition, we tested the ATI Radeon AMD 7970 running under two different operating systems: Windows 7 and Linux Ubuntu 12.04 see Table 3. According to the results in Table 4, we can achieve a $34.2 \times$ increase in computation speed over the CPU parallel computation for a grid with size $(1370 \times 4240)$ and over $15.7 \times$ for a larger grid $(1820 \times 5640)$. The interesting point here is that we are able to compare the performance running exactly the same code using different devices. Since this is our first attempt we have not explore yet the use of local memory but this is an issue that needs further scrutiny. The advantage of OpenCL (PyOpenCL) is that just by switching the device one wants to use, it enables the execution of the code on that specific device (provided that one has an AMD card, because with the drivers of NVIDIA cards one cannot do this). In Fig. 6 the same information given in Table 4 is displayed in a plot. Note that although we are comparing a Tesla T10 among the different platforms, a cheaper card such as the GeForce GTX 560 Ti has a better performance than the Tesla card.

### Table 1

<table>
<thead>
<tr>
<th>Formation</th>
<th>$V_p$ (m/s)</th>
<th>$V_s$ (m/s)</th>
<th>$\rho$ (kg/m$^3$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fast formation</td>
<td>4000</td>
<td>2300</td>
<td>2500</td>
</tr>
<tr>
<td>Slow formation</td>
<td>3000</td>
<td>1200</td>
<td>2500</td>
</tr>
<tr>
<td>Slow layer</td>
<td>1800</td>
<td>967</td>
<td>2200</td>
</tr>
<tr>
<td>Invaded zone</td>
<td>3000</td>
<td>1800</td>
<td>2300</td>
</tr>
<tr>
<td>Slow formation</td>
<td>3000</td>
<td>1200</td>
<td>2500</td>
</tr>
<tr>
<td>Fluid</td>
<td>1500</td>
<td>0</td>
<td>1000</td>
</tr>
</tbody>
</table>
but we have to remember that this model of the Tesla card is old in contrast to the GeForce GTX 560 Ti, not to mention the AMD 7970 that is the latest generation of AMD Radeon graphics cards.

Although a comparison with a serial code would not be fair because it is not based on the same ground basis, we ran the serial implementation in Fortran using an Intel(R) Core(TM) i5-2500 at...
3.30 GHz (4 cores) and the computation time is 723.0839 s for a grid size \((830/2 \times 2650)\). This gives us a speed up of \(26/2\) with respect to the best performance of the GPUs. Since the serial program and the parallel code are not the same this represents only a quantitative comparison. Next section describes the PyOpenCL implementation that we use for these experiments.

### 6. Description of the PyOpenCL code

The hardware implementation of a GPU is quite different from that of a CPU. A GPU has several hundred processors and several different kinds of memory, we can implement parallel computing using several hundred processors. All processors in a GPU are basically designed to execute the same code, so GPU computing accelerates simple iterative calculations efficiently. We choose to use PyOpenCL programming, where the CPU is the host that controls the GPU whilst the GPU acts as the device. In addition, PyOpenCL programming enables the use of the multi-core CPUs for both host as well as computing device. We include the full 2.5-D finite difference code with PyOpenCL as an electronic supplementary material. The description of the general algorithm to run the 2.5-D finite differences is given in Appendix A. We have implemented the parallel finite difference code by defining a Python class called FD252D that is the core of the program because it contains the functions to initialize the OpenCL environment, set the scenario, set the staggered properties, configure absorbing boundary conditions, initialize all the needed variables, set the source and receivers, run the finite differences and set the definitions of all kernels. The kernels are routines that completely run on the GPU or multi-core CPU. Kernels are executed in a parallel manner, and each singular instance of a kernel is called a work-item. These work-items are mapped, in hardware, to the stream cores of GPUs and CPUs. Due to the nature of our partial differential equation that we are solving and the fact that we need to split one time step into two halves, see Eqs. (12) and (13), our approach to parallelize this problem on a GPU consists on defining two kernels. One kernel to compute velocities and the other to

### Table 2

<table>
<thead>
<tr>
<th>Formation</th>
<th>(r) (m)</th>
<th>(z) (m)</th>
<th>(\Delta r) (m)</th>
<th>(\Delta z) (s)</th>
<th>NR</th>
<th>NZ</th>
<th>(f) (Hz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monopole fast</td>
<td>3</td>
<td>8</td>
<td>0.004687</td>
<td>2.9246876e−7</td>
<td>639</td>
<td>1706</td>
<td>8000</td>
</tr>
<tr>
<td>Monopole slow</td>
<td>3</td>
<td>8</td>
<td>0.00375</td>
<td>3.1256e−7</td>
<td>865</td>
<td>2263</td>
<td>8000</td>
</tr>
</tbody>
</table>

### Table 3

<table>
<thead>
<tr>
<th>GPU</th>
<th>(1) ATI Radeon HD 6770M</th>
<th>(2) ATI Radeon AMD 7970</th>
<th>(3) Tesla T10</th>
<th>(4) NVIDIA GeForce GTX 560 Ti</th>
<th>(5) NVIDIA GeForce GT 240</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>(6) Intel(R) Core(TM) i7-2760QM at 2.40 GHz 4 cores</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operating system</td>
<td>(1) MacOS Apple Mountain Lion 10.8</td>
<td>(2a) Windows 7 (32-bit Python)</td>
<td>(2b) Linux Ubuntu 12.04 (64-bit Python)</td>
<td>(3) Linux Red Hat Linux</td>
<td>(4) Linux Ubuntu 12.04</td>
</tr>
</tbody>
</table>

3.30 GHz (4 cores) and the computation time is 723.0839 s for a grid size \((830 \times 2650)\). This gives us a speed up of \(26 \times \) with respect to the best performance of the GPUs. Since the serial program and the parallel code are not the same this represents only a quantitative comparison. Next section describes the PyOpenCL implementation that we use for these experiments.
compute stresses imposing a barrier after the execution of each kernel to ensure that the second kernel has all the values it needs. In Fig. 7(a) we can see a simplified diagram for the sequential computation that was our starting point. We have two double-loops (one for each half time step) one that runs for all values of \( r \) and the other for all values of \( z \). In opposition to the sequential computation we have the parallel computation in which the two double loops are avoided and the computation of velocities is done at once and then when all work items finished with this computation (this is accomplished by imposing a barrier), the computation of stresses is done at once, see Fig. 7(b).

The main program starts by identifying and setting the platform or device to run the code (i.e., a GPU or a CPU), imports the class FD25D and input the parameters needed such as the position of the receivers, it defines the size of the frame for snapshots and starts the main loop which corresponds to the time iteration.

### Table 4
Comparison of average computation times for 2.5-D finite differences on five different GPUs and a CPU multi-core. The experiment of comparing two different operating systems with the same card is illustrated for the AMD7970 where we use Windows and Linux (Win/Lin) and both results are displayed in this Table. We choose 19 different grid sizes for each experiment and fixed the number of time steps to 4500 in all experiments. We write a symbol □ to denote the cases where the memory of the card was not able to support the given grid size.

<table>
<thead>
<tr>
<th>Device/grid size</th>
<th>GPU</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AMD 6770M (480 cores)</td>
<td>AMD7970 (Win/Lin) (2040 cores)</td>
</tr>
<tr>
<td>200 × 600</td>
<td>23.14 s</td>
<td>2.60 s/6.31 s</td>
</tr>
<tr>
<td>250 × 880</td>
<td>36.26 s</td>
<td>3.33 s/7.28 s</td>
</tr>
<tr>
<td>380 × 1160</td>
<td>57.18 s</td>
<td>4.64 s/7.85 s</td>
</tr>
<tr>
<td>470 × 1140</td>
<td>81.89 s</td>
<td>6.75 s/9.93 s</td>
</tr>
<tr>
<td>560 × 1170</td>
<td>137.07 s</td>
<td>9.11 s/11.65 s</td>
</tr>
<tr>
<td>650 × 2000</td>
<td>154.28 s</td>
<td>11.09 s/14.00 s</td>
</tr>
<tr>
<td>740 × 2280</td>
<td>197.03 s</td>
<td>13.71 s/16.50 s</td>
</tr>
<tr>
<td>830 × 2650</td>
<td>237.10 s</td>
<td>15.58 s/18.23 s</td>
</tr>
<tr>
<td>920 × 2840</td>
<td>332.70 s</td>
<td>18.96 s/21.42 s</td>
</tr>
<tr>
<td>1010 × 3120</td>
<td>358.09 s</td>
<td>23.51 s/26.25 s</td>
</tr>
<tr>
<td>1100 × 3400</td>
<td>447.32 s</td>
<td>27.29 s/29.25 s</td>
</tr>
<tr>
<td>1190 × 3680</td>
<td>537.31 s</td>
<td>30.86 s/33.86 s</td>
</tr>
<tr>
<td>1280 × 3960</td>
<td>717.29 s</td>
<td>35.67 s/38.21 s</td>
</tr>
<tr>
<td>1370 × 4240</td>
<td>657.71 s</td>
<td>40.62 s/43.25 s</td>
</tr>
<tr>
<td>1460 × 4520</td>
<td>793.98 s</td>
<td>45.74 s/49.03 s</td>
</tr>
<tr>
<td>1550 × 4800</td>
<td>827.32 s</td>
<td>51.95 s/54.50 s</td>
</tr>
<tr>
<td>1640 × 5080</td>
<td>968.40 s</td>
<td>57.99 s/60.93 s</td>
</tr>
<tr>
<td>1730 × 5360</td>
<td>1166.27 s</td>
<td>□/69.17 s</td>
</tr>
<tr>
<td>1820 × 5640</td>
<td>1243.58 s</td>
<td>□/78.93 s</td>
</tr>
</tbody>
</table>
7. Conclusions

We have developed a code that can be used to simulate acoustic signatures due to a borehole intercepting inhomogeneities having physical properties different from those of the host medium (with radial symmetry). This includes layers intercepting the borehole with different properties, such as invaded zones or a casing. The use of graphics cards enables us to increase the speed in 2.5-D FD simulation by more than 32\times\text{times} (using a small-scale grid) with respect to the parallel computation of the CPU quad-core. For a large-scale grid (1820\times5640) we hit a 15.7\times increase in speed. This opens the door in the very near future to study not only forward problems but more interesting full wave-form inverse problems using this technology. Further work should be addressed into the scalability of 3-D simulations and the use of multi-GPUs. Additional investigation is needed to assure that the local memory is used as much as possible to be able to take real advantage of GPU computing. The advantage of PyOpenCL over other programming languages capable to handle graphic cards is that it is not attached to any brand of card and it allows one to freely chose the device the user wants to use to run the parallel code just by switching between CPU or GPU. If a CPU is set the PyOpenCL code automatically runs the parallel code into the CPU and all its cores. Since parallel computing with graphic cards is a fairly new subject in the scientific community, the development tools equivalent to CPU tools which shield the programmer from having to deal with full complexity of hardware are unavailable or rudimentary. We aim to contribute exploring its characteristics and potentialities hoping that achieving any step in this long path, will provide some insights and help to mature this new technology. Finally, the code described here and used to obtain the results presented in this work is available as an open source tool to share with the scientific community.

Acknowledgments

We would like to thank the two anonymous reviewers for kind comments and suggestions that helped to improve this manuscript. We also like to thank Antonio J.B. Tadeu for kindly providing the code of the reference solution (using the discrete wavenumber method) that helped us to validate the finite differences. This work was partially supported by DGAPA-UNAM under project IC100511 and the Spanish Economy and Competitiveness Ministry (TEC2012-38402-C04-03).

Appendix A. Software and hardware requirements

A version of the code can be found in http://code.google.com/p/borehole-pyopencl/. In order to run the present code you need the following open-source software: Python (we use version 2.7.3), NumPy, Matplotlib, SciPy, PyOpenCL (OpenCl Library), PIL and glumpy. A manual with the description of the input parameters is included in the same web site where the code is hosted.

To run the code you just have to type: python main.py and this will invoke the FD25D class and all its necessary constructors to start with the time loop. The main program executes the following algorithm:

**Algorithm 1.** Main program to run the FD25D with its kernels to speedup the computation.
Require: Velocities for the different ring sections $V_\rho$, $V_\phi$, density $\rho$, sizes of the scenario: maximum radius $r_{\text{max}}$, minimum radius $r_{\text{min}}$, and maximum depth $z_{\text{max}}$, time iterations (Timelter), central frequency $f_0$, number of receivers, position of first and last receivers, position of the source ($b_0/s_0$) number of collars or rings of media with different properties, and device (GPU or CPU).
1: Create a scenario (size of the model, number of nodes in each direction, velocities, densities)
2: Initialize the GPU/CPU
3: Material setup (initialize the values of the material variables for each node in the grid)
4: Set the absorbing boundaries
5: Initialize the variables for OpenCL
6: Initialize the receivers (Positions)
7: while $t < $ Timelter do
8: Launch the kernel that computes velocities
9: Launch the kernel the computes stress components
10: $n = n + 1$
11: end while
12: Save simulation results into a file

Appendix B. Supplementary data

Supplementary data associated with this article can be found in the online version at http://dx.doi.org/10.1016/j.cageo.2013.03.014.

References

Micikevicius, P., 2009. 3D finite-difference computation on GPUs using CUDA. In: GPGPU-2 Proceedings of the Second Workshop on General Purpose Processing on Graphics Processing Units, Washington DC, USA, pp. 79–84.