Asociación Argentina
de Mecánica Computacional

Mecánica Computacional Vol XXIX, págs. 7147-7157 (artículo completo)
Eduardo Dvorkin, Marcela Goldschmit, Mario Storti (Eds.)
Buenos Aires, Argentina, 15-18 Noviembre 2010

SIMULATION OF WAVE PROPAGATION IN SEMI-INFINITE DOMAINS USING THE FINITE DIFFERENCE METHOD ON A GPU BASED ON CLUSTER.

Marcelo P. M. Zamith[a], Diego N. Brandão[a], Mauricio Kischinhesvky[a], Regina Célia P. Leal-Toledo[a], Otton T. Silveira Filho[a], Esteban W. G. Clua[a], Anselmo A. Montenegro[a] and André Bulcão[b]

[a]*Institute for Computing, Federal Fluminense University, Passo da Pátria Street, 156 Niterói, Rio de Janeiro, Brasil, {mzamith, dbrandao, kisch, leal, otton, esteban, anselmo}@ic.uff.br, http://www.ic.uff.br*

[b]*Centro de Pesquisas e Desenvolvimento Leopoldo Américo Miguez de Mello (Cenpes), Petrobrás, Av. Horácio Macedo, 950, Cidade Universitária. Rio de Janeiro, Brasil, bulcao@petrobras.br, http://www.petrobras.br*

Keywords: GPU Computing, Finite Difference Method, Wave Propagation, MPI.

Abstract.

The scattering of acoustic waves has been considered of pratical interest for many areas. Relevant works are reported in geophysics, medical images, structures' damage identification, etc. This work applies the finite difference method to simulate the scattering of acoustic waves in semi-infinite non-homogeneous media. Solving these problems can demand a high computational effort and, in some cases, make the proposed simulation impractical. The approach through high performance computing, using tools as MPI (Message Passing Interface) and GPUs (Graphic Processor Units), can soften this limitation. This work proposes a solution of such problem by using a heterogeneous cluster based on GPUs, taking advantage of its high level of parallelism. Computational results illustrate the viability of the adopted approach.

1 INTRODUCTION

Hyperbolic Partial Differential Equations (PDEs) describe a large variety of physical phenomena governed by wave behavior. The acoustic wave equation is the simplest of such models, but it can be applied to describe complex problems as wave propagation in geophysics (Michea and D.Komatitsch, 2010), structures' damage identification (Fernandes et al., 2010) and so on.

Numerical methods are necessary to provide approximate solutions to real wave propagation problems. Often, numerical modeling of the scattering of acoustic waves requires high-spatial solution; the discretization of the domain may contain millions of elements, which imposes a significant computation burden. Thus, the simulation of scattering of acoustic waves phenomena is a computationally demanding task (Kern and Méfire, 1996).

In solving large domains, single processor computers are limited and often incapable of managing the required memory and computational time requirements. In order to improve the performance of these finite difference methods, it is necessary to explore other computational strategies such as parallelization, using clusters or grid computers.

Parallel environments such as MPI have been extensively used in numerical methods to simulate large complex problems that describe the scattering acoustic waves simulation (Ramadan et al., 2004), (Kern and Méfire, 1996), (Hammonds et al., 2007).

GPU Computing has become an important choice for many parallel computational problems, since the GPUs are potentially more powerful for massively parallel computations than the CPUs. The reason behind the discrepancy in floating-point capability between CPUs and GPUs is that the GPU is specialized for compute-intensive, highly parallel computation, since this is typically required on graphics rendering. Therefore, its architecture is designed in such a way that more transistors are devoted to data processing than data caching and flow control (NVIDIA, 2008).

Many different non-graphical computation, simulation and numerical problems, including Protein Structure Prediction (Langdon and W.Banzhaf, 2008), Solution of Linear Equation Systems (Bolz et al., 2003), Options Pricing (Abbas-Turki and Lapeyre, 2009), Flow Simulation (Rozen et al., 2008), Wave Propagation (Balevic et al., 2008), (Michea and D.Komatitsch, 2010), have been solved in GPUs.

Few years ago, scientific computing based on GPU architecture was developed using shader languages in combination with some graphics APIs, with all the vertex and pixel shader limitations and idiosyncrasies. Recently, the architectures of GPUs have been rearranged in order to make it easier to program general purpose problems using such devices, starting a new paradigm for scientific computing, known as GPU Computing.

GPUs have a hierarchical memory structure divided as: global, texture and shared. The shared memory has a large velocity of access, almost equivalent to the velocity of accessing the registers. Texture memory is slower than the shared one, but is almost twice as faster as global memory. On the other hand, it follows a read only paradigm. Global memory is readable and writeable and can be accessed by any thread and processor at any time, which is important for communication between different threads.

Two of the most popular languages for programming in parallel GPU Computing paradigm are CUDA (Compute Unified Device Architecture) from nVidia (NVIDIA, 2008) and OpenCL (Open Computing Language) (NVIDIA, 2003).

Finite-difference Methods (FDM) in the time domain (FDTD) are widely used to solve the problem of simulating the scattering of acoustic waves in semi-infinite non-homogeneous media

and some approaches about this technique are already modeled using GPUs (Balevic et al., 2008), (Michea and D.Komatitsch, 2010), (Micikevicius, 2009).

This paper presents a parallel implementation for the scattering of 2-D acoustic waves in semi-infinite non-homogeneous medium in heterogeneous cluster based on GPUs.

The paper is organized as follows. Acoustic wave equation and some aspects about Finite Difference Method (FDM) are described in Section 2. The GPU architecture and CUDA paradigm are discussed in Section 3. Section 4 presents the cluster architecture and some aspects about the computational implementation in CUDA for tge FDM. Numerical results are presented in Section 5. Section 6 includes the conclusions and some directions for future work.

2 ACOUSTIC WAVE EQUATION

The wave equation is a second order linear differential equation which describes the behavior of sound waves over time, amongst other types of waves, where all of them describe a medium perturbation. The acoustic wave field is described by $P(x, y, z, t)$ and $u(x, y, z, t)$, where P is the pressure field and u is the particle's displacement. The relation between pressure and particle's displacement is given by $P(x, y, z, t) = -k\nabla u(x, y, z, t)$ with k representing the volumetric compression module. One of the hypotheses of the model considered here is that the pressure field is invariable in z-axis, which implies the partial derivative in relation to z is zero. Thus, the (2-D) wave equation with a constant volumetric compression is given by:

$$\frac{\partial^2 P}{\partial t^2} = c^2(x, y)\nabla^2 P + f(x, y, t) \tag{1}$$

where, $P = P(x, y, t)$, x and y are cartesian coordinates, t is time, c is the velocity acoustic wave and $f(x, y, t)$ is the source term.

In the classical approach, the change of velocity field $c(x, y)$ represents the change of medium and it allows to generate both reflection and diffraction of waves.

To numerically solve the partial differential equation in (Eq.1), we first discretize it into a set of finite-difference equations by replacing partial derivatives with central differences. A central-difference approximation can be derived from the Taylor series (Mitchell, 1969). Thus, using a second order approximation for space and time, assuming $h = \Delta x = \Delta y$ and $t = n\Delta t$, Eq.(1) is rewritten as:

$$P^{n+1}_{(i,j)} = 2P^{n}_{(i,j)} - P^{n-1}_{(i,j)} + A\left[P^{n}_{(i-1,j)} + P^{n}_{(i+1,j)} - 4P^{n}_{(i,j)} + P^{n}_{(i,j-1)} + P^{n}_{(i,j+1)}\right] \tag{2}$$

where, $A = \left(\frac{c(x,y)\Delta t}{h}\right)^2$ and $n = 1, 2, ...$ represents slice time.

In general, approximations more accurate for the derivative imply that more neighboring points are required and therefore more expensive the calculation may become. However, not only the precision must be analyzed, we must also consider the stability criteria that are important to ensure the convergence for certain width of the spatial mesh.

2.1 Numerical Stability

The stability of the explicit FDM for second order 2-D wave propagation is given by the Courant−Friedrichs−Lewy condition (CFL condition). This condition requires that the do-

main of dependence of the PDE must lie within the domain of dependence of the finite difference scheme for each mesh point of an explicit finite difference scheme for the acoustic wave equation. Thus, the CFL condition that guarantees the stability in this case is (Mitchell, 1969):

$$\sqrt{A} = \left(\frac{c\Delta t}{h}\right) \leq \frac{1}{\sqrt{s}} \tag{3}$$

where s is the domain dimension, here $s = 2$.

2.2 Semi-Infinite Domains

The scattering wave equation is theoretically solved in a non-finite medium. However in computing models the size of memory computer is limited which requires that artificial frontiers must be introduced to the model. This non-realistic contour implies in wave reflection of borders of domain. The solution, in the case of punctual source, is a explicit approximation. This approach consist in increasing the domain, so that artificial reflections do not appear in the simulation. However this solution implies in a high computational cost. For instance, in geophysics' simulations, the domain dimension is very large, thus the process can become infeasible.

Aiming to simulate semi-infinite domains, this work considers the boundaries conditions proposed by Reynolds (1978). Such conditions are determined by decomposing the unidimensional scalar wave equation, obtaining the product of two terms, each one representing the spread of the wavefront in one direction. Equation 4 represents this condition when the wave propagation occurs to the right. The boundaries conditions for the left direction can be obtained analogously (Reynolds, 1978). The conditions in the top and the bottom of model are given by Dirichlet conditions (Golub and Ortega, 1991).

$$\frac{\partial P}{\partial \vec{n}} = \frac{1}{c}\frac{\partial P}{\partial t} \tag{4}$$

where $\vec{n}$ is the normal vector.

2.3 Source Function

The choice of an appropriate source function is essential to FDM. The frequence of pulse affects diretly the numerical dispersion of the method (Bording and Lines, 1997). Thus, the pulse is prescribed in according to Eq. 5:

$$f(x, y, t) = 1 - 2 \times \left(t \times \sqrt{\left(\frac{1}{2}\right)} \times \pi \times CF\right)^2 \tag{5}$$

where CF represents the cut frequency and t is the time instant.

3 CUDA AND GPU COMPUTING

CUDA enables inexpensive multi-threaded GPUs programming. One of the advantages of programming in CUDA instead of conventional Shader language programming is that it allows one to work with familiar programming concepts while developing software that can run on a GPU, avoiding the performance overhead of graphics layer APIs by compiling the software directly to the hardware (Dobbs, 2008).

CUDA has several advantages over traditional general purpose computation on GPUs, such as: scattered reads - code can read from arbitrary addresses in memory; shared memory - CUDA

exposes a fast shared memory region which can be shared among different threads; faster downloads and read-backs to and from the GPU and full support for integer and bitwise operations, including integer texture lookups (NVIDIA, 2008). Besides all this memory facilities of GPU, CUDA does not cater memory access by CPU. This constraint requires the CPU to previously copy the data from its memory to the GPU.

The GPU used for running CUDA is based on an unified architecture, *i.e.*, there is a set of general stream processors for both vertex and pixel programs. Due to the fact that CUDA organizes this set of stream processors in a set of multiprocessor cores, it is possible to run multiple threads concurrently. The threads are arranged in blocks, which have their own shared memory space for sharing among their threads, as well as each multiprocessor can process one block 1. Therefore, the level of parallelism is enhanced through proper arrangement of the GPU model, blocks and threads per blocks. The block set is named grid.

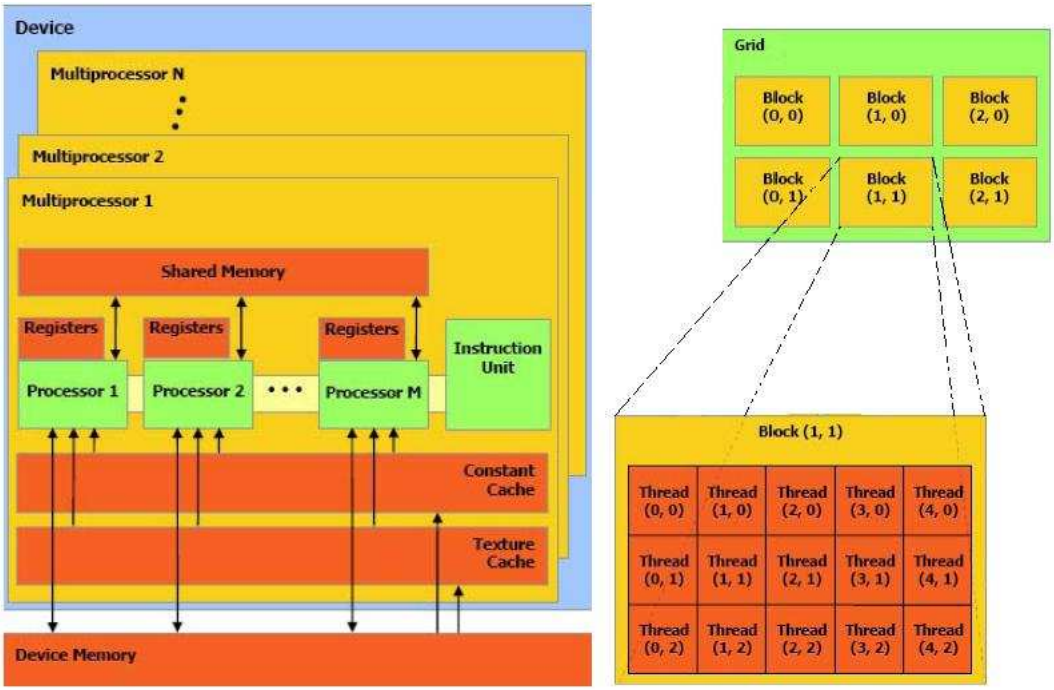

Figure 1: CUDA's execution configuration (Balevic et al., 2008).

CUDA enables application programming in an extension of the C language (C for CUDA). In fact, a CUDA program is a set of C functions, named kernels, that can be invoked by the host, and are executed on the deviceś *n* instance of the kernel in parallel. In CUDA architecture, CPU is defined as host a GPU as a device.

When a kernel starts running based on the execution configuration and according to the function arguments, the host continues to the next line of code, after the kernel launches. At this point, both the CUDA device and host are simultaneously running their separate programs. Nevertheless, it is possible to artificially synchronize both host and kernel programs.

A typical structure of a CUDA program is composed by the steps illustrated in Fig.2:

1. copy of data from the main host memory to the GPU memory;

2. the host invokes the kernel;

3. the kernel runs concurrently in different threads in the device;

4. the results stored in GPU memory are copied back to the CPU.

Not all problems can be solved efficiently in CUDA. Problems which are computation-intensive and that involve processing large data sets using the same code (Single Instruction Multiple Data paradigm) are natural candidates to be parallelized on GPUs. Developing an appropriate parallel algorithm that suits the CUDA programming model may be very difficult. Besides, there are two main challenges in modelling an efficient CUDA program: breaking the problem appropriately into many sub-problems that can run concurrently in several independent threads and dealing appropriately with CUDA memory hierarchy, so that no overhead is introduced due to an inappropriate choice of data distribution, order of access and communication strategies.

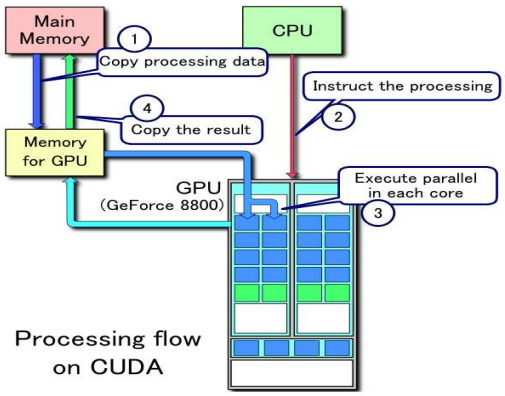

Figure 2: Processing Flow on Cuda (HBeonLabs, 2009).

4 GPU-BASED CLUSTER

In practical problems that numerical methods are employed, in particular the finite difference method, there are a lot of points to be calculated, in some cases this quantity can reach in a billions order, for instance in seismic modeling. The simulation of such problems requires a computer powerful, in many cases supercomputers. The memory demand is another requirement of these problems, which can become unfeasible the solution on GPU. However, a cluster based on GPUs is a cheap alternative for this.

In this context, this work presents a approach of a cluster architecture, where each node (CPU) has, at least, one GPU. In this hybrid-cluster the GPUs are responsible for all mathematical processing of the simulation, while MPI is used to provide the communication between the nodes.

Although the GPUs are different in performance and memory size, all of them have to be of the same brand, because CUDA language is a solution only to nVidia GPUs, on the other hand the CAL language runs only on ATI GPUs. Another solution is the OpenCL language, which is a framework for writing programs that execute across heterogeneous platforms, as ATI, NVIDIA and CPUs. In this work, the CUDA language is adopted for the cluster is constituted only by Nvidia GPUs.

The hybrid-cluster is composed by two parts. The first is responsible for simulating the numerical method, *i.e.*, calculating the value of each domain point. This is done on GPUs with the CUDA language. The second is responsible for guaranteeing the correct communication among the nodes, thus MPI is employed. Thus, for each time instant the GPUs calculate the unknown values and then the communication phase, takes place. In the latter, each node sends to others the values required for the next time instant.

The architecture of a GPU is organized by CUDA to work as a GPU cluster inside a CPU. This architecture defines CPU as a host and GPU as a device. Furthermore, the memory is defined as a host memory (CPU memory) and as a memory device (GPU memory). The GPU stream processors is tided in a set of multiprocessor and it can process a kernel instance, i.e., a thread. The kernel code is written for GPU (in CUDA language) (NVIDIA, 2008).

The number of stream processors depends on GPU model and the parallelism level is directly related with it. Since the instances of the kernel (threads) are organized in groups called blocks, each one is processed by one multiprocessor. The threads of blocks are processed in batch.

The communication bandwidth and space in memory are always bottlenecks (Zamith et al., 2010). All data must be sent to the host memory before processing by GPU. GPU memory is small when compared with hostś memory. In addition to that, part of the memory is allocated for visualization tasks in some GPU models, except for Tesla which is completely dedicated to GPU Computing. The memory of the GPU has a hierarchy, which describes features in size and time access. As mentioned earlier, it is arranged in texture, constant, global share and local memory. Although the shared memory access is as fast as a register, the sharing occurs only among threads in the same block. Moreover, the maximum size of shared memory allowed is 16Kb per block (NVIDIA, 2010). Thus, the strategy here adopted is the use of shared memory, so as to improve the time processing. The use of shared memory implies that, in order to copy data from global memory to shared memory, a synchronization command immediate follows the copy operation, so that all threads can access data correctly. Basically, the kernel is composed by two steps: the first is the synchronization step between shared and global memory, as described above; the second one is responsible for defining the new value of the unknown.

Indeed, hierarchy structure is applied to optimize the deviceś memory access. The strategy adopted is to use three matrices to represent each time slice, plus vectors that represent the outline boundaries. The model presented in this work is $O(h^2, \Delta t^2)$. i.e., uses five neighbors at time step $t-1$ and only one at $t-2$. In oder to minimize the global memory access, only the data from time instant $t-1$ are copied from global to shared memory.

Since each node must know the outline boundaries of its neighbors, vectors of the informations in its neighbors are defined. In each step of simulation the part of the domain in device memory is copied to host memory followed by a synchronizing step, where these vectors are sent to neighbors and others are received through MPI APIs. The Fig. 3 illustrates it.

Despite the fact that MPI offers several synchronization statements, the barrier function is adopted. Furthermore, CUDA has only one sync object and it is invoked to synchronize threads of same block. This procedure is done in three steps. The first one is the synchronize between device and host, *i.e.*, the copying of data from device to the host. The second is the MPI synchronization and the last one is a copy from host to device. Thus, each GPU processes a slice of the domain. Fig. 3 shows an illustration of this communication scheme.

4.1 Finite Difference Method on GPU

Such described, the kernel with shared memory is more efficient than the other based on texture memory.The former is divided in two steps: The first copies the pressure values in the

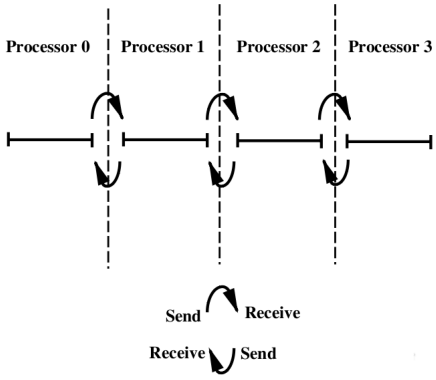

Figure 3: MPI - Send / receive messages

current time $P^n_{(i,j)}$ from the global memory to the shared memory; the second one uses Eq.2 to calculate the values in the next time $P^{n+1}_{(i,j)}$. Besides, according to Eq. 2, only points in time instant t need to be loaded in shared memory.

The approach considered has each new value $P^{n+1}_{(i,j)}$ computed by only one thread. Hence, the 2-D domain is divided in blocks and threads per blocks, observing the constraints described in NVIDIA (2010). The maximum number of blocks that can be allocated is $65,535$ with 512 threads for each one, giving a total of $33,553,920$ points.

Some experiments were done to identify the bottlenecks as well as the best arrangement of threads and threads per blocks of the simulation. We detected that a blocksize of 32×16 gives a better result than a blocksize of 16×32. Such difference in processing time is due to coalesced memory accesses and the number of threads within the warp (Zamith et al., 2010).

The kernel configuration defines a 2-D block with 32 threads in one dimension and 16 threads in the other.

Shared memory's size is based on the number of threads plus a region called buffer border whose size is two times the neighborhood size corresponding to the 2-D stencil's size. One should be aware that there is a $16kb$ limit in the size of the shared memory per block.

Each thread copies its data in the current time from the global memory to the shared memory. Besides, the threads at the border of a block also copy data corresponding to the data of its two neighbors which belong to a neighboring block. By doing this, the shared memory structure has all data necessary to compute new values of the points inside a block without any additional access to the global memory. In other words, it guarantees that all threads are able to access the memory addresses corresponding to instant t from shared memory necessary to define the new point value.

By using the additional buffer region the computation can be done with optimal memory access time.

One should be aware that this strategy has a limitation that depends on the neighborhood considered in the finite difference discretization method. A larger neighborhood suggests a smaller number of threads to account for the limit of available shared memory depending on the hardware specification.

5 NUMERICAL RESULTS

The test network is composed by three computers. One of them uses an Intel Quad Core with 2.4GHz, 4GB and nVidia Tesla C1060 GS graphic card. The other two computers use an Intel Core 2 dual with 2.4 GHz, 2GB of memory and nVidia GeForce 8800 GTS graphic card. All computers use the Ubuntu 9.10 operation system (64bits) and OpenMPI. The network bandwidth is 1.0 Gbits. The server computer is the one with the Quad Core processor, while the others act as clients.

Steps	32x16	16x16	16x32	X	Y	Gsample 32x16	Gsample 16x16	Gsample 16x32
10000	3,8716400	4,1563500	5,0819700	1024	1024	2,71	2,52	2,06
50000	19,2819900	20,7243800	25,3448770	1024	1024	2,72	2,53	2,07
100000	38,5077900	41,4066700	50,6085400	1024	1024	2,72	2,53	2,07

Table 1: Simulations to determine of the best arrangement of blocks and threads per block.

The first tests are applied to identify which is the best arrangement of blocks and threads per block. It is a single core CPU, *i.e.*, without MPI. Thus, three tests are done with arrangement of threads per block: 32×16, 16×16 and 16×32. The best results are shown by arrangement of 32×16, as presented in table 1. The hardware used is the Tesla 1060 and CPU quadcore. One can notice that the best arrangement is 32×16 since this one exhibits the lowest time of processing and the highest gigasamples, *i.e.*, this configuration gives 2.7 gigasample. A gigasample is the number of calculations that can be performed per second, that is, $10^{-9} \times (number\_points\_X \times number\_points\_Y \times number\_points\_T)/(computing\_time)$.

X per node	X	Y	Number of points	Simulation step	Processing time (s)	Gsample(s)
1024	3072	1024	3145728	1000	29,83	0,11
2048	6144	2048	12582912	1000	62,01	0,2
4096	12288	4096	50331648	1000	151,84	0,33
1024	3072	1024	3145728	2000	59,87	0,11
2048	6144	2048	12582912	2000	123,25	0,2
4096	12288	4096	50331648	2000	300,48	0,34
1024	3072	1024	3145728	3000	91,99	0,1
2048	6144	2048	12582912	3000	184,42	0,2
4096	12288	4096	50331648	3000	455,01	0,33
1024	3072	1024	3145728	4000	120,29	0,1
2048	6144	2048	12582912	4000	248,17	0,2
4096	12288	4096	50331648	4000	602,62	0,33
1024	3072	1024	3145728	5000	149,15	0,11
2048	6144	2048	12582912	5000	306,82	0,21
4096	12288	4096	50331648	5000	755,76	0,33
1024	3072	1024	3145728	6000	184,21	0,1
2048	6144	2048	12582912	6000	374,83	0,2
4096	12288	4096	50331648	6000	900,38	0,34
1024	3072	1024	3145728	7000	210,89	0,1
2048	6144	2048	12582912	7000	430,13	0,2
4096	12288	4096	50331648	7000	1052,76	0,33
1024	3072	1024	3145728	8000	239,8	0,1
2048	6144	2048	12582912	8000	491,5	0,2
4096	12288	4096	50331648	8000	1213,4	0,33
1024	3072	1024	3145728	9000	280,33	0,1
2048	6144	2048	12582912	9000	586,89	0,19
4096	12288	4096	50331648	9000	1385,04	0,33
1024	3072	1024	3145728	10000	310,03	0,1
2048	6144	2048	12582912	10000	624,83	0,2
4096	12288	4096	50331648	10000	1541,74	0,33

Table 2: Results of scalability with MPI-GPU approach.

The second test presents the scalability given by MPI and combination of this technique with GPU. The network latency and bandwidth are the bottlenecks (Zamith et al., 2010). The MPI can be used for breaking the GPU constraint of domain size, though. This is illustrated by table 2.

6 CONCLUDING REMARKS AND PROSPECTIVE WORK

Although GPU can be a powerful hardware applicable to massively parallel mathematical problems, its own hardware such as memory, bandwidth as well as the own language adopted, is limited. Since CUDA is applied for GPUs from NVIDIA, which this work employed, the source code is not portable for other platforms. For instance, ATI developed its own language, called CAL (Compute Abstraction Layer). Nowadays, OpenCL is a framework that provides cross platform heterogeneous programming targetting GPUs and multicore-CPUs as well, with the same source code. The domain scale can be worked out with many other GPUs, either by employing one single CPU or by employing several CPUs. This latter was the case in the work presented herein. In the first case, all GPUs are plugged on the same motherboard whereas in the latter a network structure is employed. In this case the connection speed should be carefully chosen in order not to harm the overall computing time.

As a forthcoming work the use of adaptive time is to be employed. This strategy shall be valuable in order to save computing time. This occurs because in some regions of the domain, the CFL condition can be served with smaller time steps than in others, due to heterogeneities. An additional and related feature to be discussed is the choice of efficient and effective load balancing heuristics for domain partition, in the context of time adaptivity.

Summarizing, this work employed a scientific problem of industrial interest, namely the scattering of acoustic waves in heterogeneous media, as a testbed for studying the efficient application of GPU Computing for FDM discretization of mathematical problems. The use of MPI provided an efficient way for reaching domain scalability.

Acknowledgements

The authors gratefully acknowledge Petrobrás, CNPq, CAPES and FAPERJ for the financial support of this work.

REFERENCES

Abbas-Turki L.A. and Lapeyre B. American options pricing on multi-core graphic cards. *International Conference on Business Intelligence and Financial Engineering*, 0:307–311, 2009.

Balevic A., Rockstroh L., Tausendfreund A., Patzelt S., Goch G., and Simon S. Accelerating simulations of light scattering based on finite-difference time-domain method with general purpose gpus. *Computational Science and Engineering, IEEE International Conference on*, 0:327–334, 2008.

Bolz J., Farmer I., Grinspun E., and SchrÃűoder P. Sparse matrix solvers on the gpu: conjugate gradients and multigrid. In *ACM Transactions on Graphics: Proceedings of ACM SIGGRAPH*, pages 917–924. 2003.

Bording R.P. and Lines L.R. *Seismic Modeling and Imaging with the Complete Wave Equation*. Course Notes Series Society of Exploration Geophysicist, 8 edition, 1997.

Dobbs D. Cuda, supercomputing for the masses: Part 1. *Avalible at: http://www.ati.com/developer/techpapers.html*, pages 1–7, 2008.

Fernandes K., NETO A.S., Tenembaum R.A., and STUTZ L. A damage assessment strategy

based on a sequential algebraic algorithm. In *17 ICSV: Proceedings of the 17th International Congress on Sound and Vibration*. 2010.

Golub G. and Ortega J. *Scientific Computing and Differential Equations: an Introduction to Numerical Methods*. Academic Press, 1 edition, 1991.

Hammonds J.S., Saied F., and Shannon M.A. Solving coupled 3-d paraxial wave and thermal diffusion equations with mixed-mode parallel computations. *Parallel Computing*, 33(1):43–53, 2007.

HBeonLabs. Cuda– compute unified device architecture. *Avalible at: http://www.hbeonlabs.com/detailcuda.htm*, 2009.

Kern M. and Méfire S.M. Parallel solution of the wave equation using higher order finite elements. In D. Kaeli and M. Leeser, editors, *MPIDC' 96: Proceedings of the Second MPI Developers Conference*, page 125. 1996.

Langdon W.B. and W.Banzhaf. A simd interpreter for genetic programming on gpu graphics cards. In *Lecture Notes in Computer Science: Genetic Programming*, pages 73–85. Springer Berlin-Heidelberg, 2008.

Michea D. and D.Komatitsch. Accelerating a three-dimensional finite-difference wave propagation code using gpu graphics cards. *Geophysical Journal International*, 182:389âĂŞ402, 2010.

Micikevicius P. 3d finite difference computation on gpus using cuda. In D. Kaeli and M. Leeser, editors, *ACM, Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 79–84. 2009.

Mitchell A. *Computational Methods in Partial Differential Equations*. John Wiley and Sons, 1969.

NVIDIA. Opencl. *Avalible at: http://developer.nvidia.com/object/opencl.html*, 2003.

NVIDIA. Nvidia - cuda compute unified device architecture. *Programming guide, NVIDIA*, 2008.

NVIDIA. *NVDIA - CUDA Programming Guide*. NVIDIA, 2010.

Ramadan O., Akaydin1 O., Salamah M., and Oztoprak A.Y. Parallel implementation of the wave-equation finite-difference time-domain method using the message passing interface. In *Computer and Information Sciences: ISCIS 2004*, pages 810–818. Springer Berlin-Heidelberg, 2004.

Reynolds A. Boundary condition for the numerical solution of wave propagation problems. *Geophysics*, 43(1):1099–1110, 1978.

Rozen T., Boryczko K., and Alda W. A gpu-based method for approximate real-time fluid flow simulation. *Machine Graphics and Vision International Journal*, 17(3):267–278, 2008.

Zamith M.P., ao D.B., Madeira D., Clua E., Kischinhevsky M., Leal-Toledo R., Montenegro A., and ao A.B. Performance evaluation of optimized implementations of finite difference method for wave propagation problems on gpu architecture. *To appear in Proceedings of First Workshop on Applications for Multi and Many Core Architectures - WAMMCA*, 2010.