CONEM 2010
VI Congresso Nacional de Engenharia Mecânica

ABCM COMPROMETIDA COM A PROMOÇÃO DO DESENVOLVIMENTO DA ENGENHARIA E DAS CIÊNCIAS MECÂNICAS
ASSOCIAÇÃO BRASILEIRA DE ENGENHARIA E CIÊNCIAS MECÂNICAS

VI CONGRESSO NACIONAL DE ENGENHARIA MECÂNICA
VI NATIONAL CONGRESS OF MECHANICAL ENGINEERING
18 a 21 de agosto de 2010 – Campina Grande – Paraíba - Brasil
August 18 – 21, 2010 – Campina Grande – Paraíba – Brazil

IMPLEMENTATION OF THE CONJUGATE GRADIENT METHOD FOR SPARSE MATRIX USING GPU

Liliana de Ysasa Pozzo, lipozzo@fem.unicamp.br[1]
Hugo Sakai Idagawa, idagawa@fem.unicamp.br[1]
Luiz Gustavo Turatti, turatti@fem.unicamp.br[1]
Luiz Otávio Saraiva Ferreira, lotavio@fem.unicamp.br[1]

[1]Department of Computational Mechanics, Mechanical Engineering Faculty, State University of Campinas, Campinas, SP, Brazil.

Abstract. *A vast number of problems in engineering do not have analytical solutions or those solutions have a high computational cost, therefore numerical methods are needed to get to those solutions. A significant amount of linear or nonlinear systems of equations are obtained by discretization methods, such as the finite difference method or the finite element method, applied to differential equations. In general, those systems are very large and have scattered characteristics that help in its numerical solution. However, those linear systems have large dimensions and they require a huge computational power to achieve the solution in a suitable time interval. In this work, we present an implementation of the Conjugate Gradient method with sparse matrix for parallel processing on graphics processing unit (GPU), reducing the computational cost to get the solution for that type of equations. After this algorithm was implemented, tested and validated, it was used to solve the problem of the 2D wave equation which was discretized using the finite differences method. Finally, the method was compared in two distinct implementations (CPU - serial form and GPU - parallel form).*

Keywords: *GPU, parallel programming, Conjugate Gradient Method, graphics computing*

1. INTRODUCTION

Several engineering and scientific problems needs to determine the solution of a system of linear equations. Estimations indicate that three in every four computational problems are converted in a solution of a system of equations (de Castro Cunha, 1993). In general, those systems are very large and have scattered characteristics that help in its numerical solution. However, those linear systems have large dimensions and they require a huge computational power to achieve the solution in a suitable time interval.

Recent progresses in graphics processing unit (GPU) devices and its applications in general purpose computing on GPUs (GPGPU) brought new perspectives to numerical modeling. In 2007 an extensive bibliographical review was published about GPU applications in general purposes computation (Owens et al., 2007). It showed an increase of the GPU calculation power in much higher rates than CPU. That fact shows a tendency that reinforces the community's interest in the use of GPU for physical and numeric simulations.

Until very recently the use of graphics processors (GPU) for numerical calculations demanded the usage of specific application programming interfaces (APIs) for graphics. Those APIs were inadequate for processing general applications, which demanded the programmers a laborious work to adapt their algorithms to those inadequate APIs. The only way to get access to the GPU's resources in 2003 was to use one of the two graphics APIs available - Direct3D or OpenGL. Consequently, researchers who wanted to harness the GPUs processing power had to work with these APIs (Kruger and Westermann, 2003). The problem was that those individuals weren't necessarily experts in graphics programming, which seriously complicated access to the technology. While 3D programmers talk in terms of shaders, textures and fragments; specialists in parallel programming talk about streams, kernels, scatter, and gather. In addition, the GPU hardware was less flexible and segmented in specific calculation units for each step of the 3D graphics processing, implicating in the partial use of the hardware for numerical processing.

The new generation of NVIDIA GeForce 8 series GPUs changed that paradigm. NVIDIA adopted a unified architecture hardware, with all processors capable to execute any type of numerical or graphics operations. In addition, those cards were totally programmable and it was also released a new software and hardware architecture called CUDA (Computes Unified Device Architecture). So the CUDA development team created a set of software layers to communicate with the GPU. This work was entirely developed with this new architecture.

CUDA has several advantages over traditional general purpose computation on GPUs (GPGPU) using graphics APIs. It has scattered reads, i.e. code can read from arbitrary addresses in memory. CUDA exposes a fast shared memory region (16 kB in size) that can be shared amongst threads. This can be used as a user-managed cache, enabling higher bandwidth than is possible using texture lookups (NVIDIA, 2008). Faster downloads and readbacks to and from the GPU. CUDA offers full support for integer and bitwise operations including integer texture lookups.

This present paper was developed in a suitable time context problem, with the objective to implement an iterative method known as the Conjugate Gradient to find the solution of linear systems using sparse matrices. We did the parallel implementation of the method by the usage of the CUDA programming language (Compute Unified Device Architecture). Furthermore, the code was implemented in CPU (Intel core 2 duo E8400 3 GHz cache L2 6 MB). So the method can be compared in two distinct implementations (CPU - serial form and GPU - parallel form).

2. CONJUGATE GRADIENT METHOD

A System of linear equations has N equations with N unknowns. The matricial representation of a linear system is Ax=B, where A is a matrix (NxN), B is vector of independent terms (Nx1), and x is a vector of unknowns. The basic approach to solve these types of systems is by direct and iterative methods. Direct methods, such as Gauss-Jordan Elimination, LU-decomposition, and Cramer Method, are the ones where the solution is achieved by a well defined number of arithmetic operations, but they are unsuitable for large matrices that are common in engineering problems. Another problem with direct methods is the difficulty in making them parallel, because each step of their algorithm depends on the previous results.

Iterative methods, such as Gauss-Seidel, Jacobi, Steepest descent, and Conjugated gradient, perform successive approximations to converge to the desired solution. These methods can be applied to sparse matrices,that are too big to be solved using direct methods such as the Cholesky decomposition method. Therefore, this work used an iterative method that explores the maximum power of the parallel hardware of GPUs. The Conjugate Gradient algorithm is made of matrix-matrix operations and matrix-vectors operations, which allows a lot of calculations to be performed in parallel. This method can be easily understood, becoming appropriate for didactic studies in simulating large systems using a GPU, like finding the solution for a 2D wave equation. The Conjugated Gradient method was developed to solve systems of linear equations iteratively that converges in a finite number of iterations, assuming there is no numerical errors in the computation of each iteration. This method uses two conjugated vectors to determine the search direction for the solver at each iteration. The Conjugate Gradient method is an algorithm for finding the nearest local minimum of a function of n variables which presupposes that the gradient of the function can be computed.

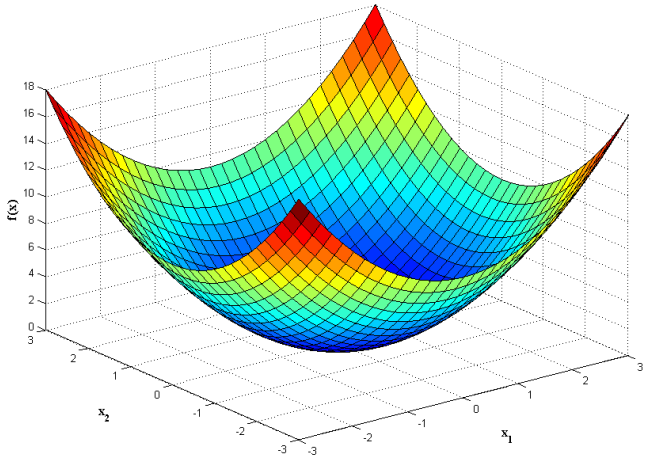

Figure 1. Graph of a quadratic form f(x). The minimum point of this surface is the solution to Ax=B.

Figure 1 shows the quadratic form of f(x). Equation 1 also represents f(x), where A is a symmetric positive definite matrix.

$$f(x) = \frac{1}{2}xAx - Bx \tag{1}$$

This function is minimized when its gradient is zero.

$$\nabla f(x) = Ax - B \tag{2}$$

This way we get Ax=B, that is the representation of a system of linear equations. With this definition, the direction of greatest decrease of the system corresponds to the negative gradient of $f(x_i)$ (Shewchuk, 1994). We can also observe that this gradient is the error (or residue r_i) between the real solution and the approximate solution (x_i).

Each new residual is orthogonal to all the previous residuals and search directions; and each new search direction is constructed (from the residual) to be T-orthogonal to all the previous residuals and search directions (figure2).

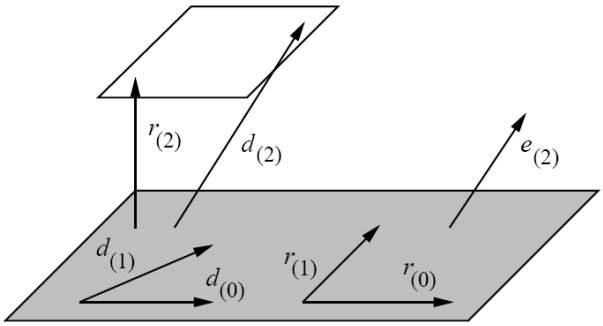

Figure 2. The endpoints of $r_{(2)}$ and $d_{(2)}$ lie on a plane parallel to D_2 (the shaded subspace). In CG, $d_{(2)}$ is a linear combination $r_{(2)}$ of and $d_{(1)}$ (Shewchuk, 1994).

At each new iteration i, a vector d_i is computed, which is linearly independent from the previous d_0 ... d_{i-1} and it represents the direction that minimizes the function f(x). Using this vector, a new solution x_{i+1} can be calculated from eq. 3.

$$x_{(i+1)} = x_{(i)} + \alpha_{(i)} d_{(i)} \tag{3}$$

Knowing the direction which the solution must advance, the next step is to determine how much should be advanced. To do this, simply find the value (α) in Eq. 4, which minimizes $f(x_i)$ along the gradient. This is equivalent to finding α for which the derivative of $f(x_i)$ is zero (Shewchuk, 1994).

$$\alpha_{(i)} = \frac{r_{(i)}^T r_{(i)}}{d_{(i)}^T A d_{(i)}} \tag{4}$$

For the first step, *d*, is calculated using Eq. 5. For the subsequent steps, *r* and *d* must be calculated using Eq. 6.

$$d_{(0)} = r_{(0)} = b - A x_{(0)} \tag{5}$$

$$r_{(i+1)} = r_{(i)} - \alpha_{(i)} A d_{(i)} \tag{6}$$

The search directions in equation 8 are updated using the residuals.

$$d_{(i+1)} = r_{(i+1)} + \beta_{(i+1)} d_{(i)} \tag{7}$$

Where,

$$\beta_{(i+1)} = \frac{r_{(i+1)}^T r_{(i+1)}}{r_{(i)}^T r_{(i)}} \tag{8}$$

Figure 4 presents in a more detailed way the Conjugate Gradient algorithm. This method is easy to implement, but it possesses the inconvenience of assuring the convergence only if A is a symmetric positive definite matrix.

3. ELLPACK-R METHOD

It is very common to solve systems of linear equations comprised of sparse matrices when discretizing differential equations, but this requires a high computational cost. The matrix-vector multiplication (SPMV) is the operation that requires the most computational time. The storage of sparse matrices has the objective to accelerate the linear algebra operations. A high performance code demands a compact data structure that benefits from the properties of sparse matrices. There is a wide range of data structures designed to store sparse matrices, each one is adequate for specific applications and matrices. The one that was chosen for this work is the ELLPACK-R, which is derived from the traditional ELLPACK storage method.

ELLPACK-R consists of two arrays, A[ ] (float) and j[ ] (integer) with dimensions N × M; and, moreover, an additional integer array called rl[ ] with dimension N (i.e. the number of rows) is included with the purpose of storing the actual length of every row, regardless of the number of the zero elements padded. An important point is the fact that the arrays store their elements in column-major order (figure 3) .

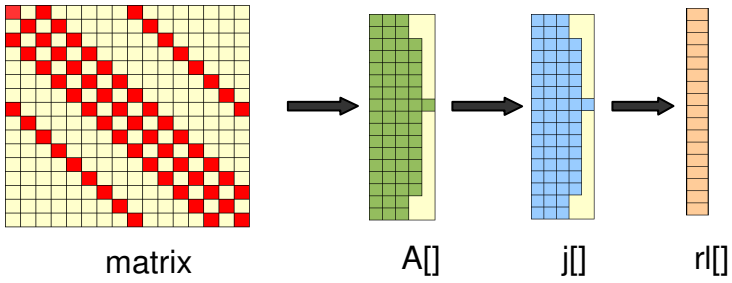

Figure 3. ELLPACK-R format

4. IMPLEMENTATION IN GPU

Using the algorithm described above, it can be divided in portions that can be parallelized. These portions, which are identified in Fig. 4 by the number (2), are basically composed of multiplications between two vectors (characterizing a scalar product) and between a sparse matrix and a vector (SpMV). Another operation type that can be identified in the method is a sum between two vectors where one of them is multiplied by a scalar (this is identified in Fig. 4 by the number (1)). These two portions, which represent the most arithmetic intensive parts of the algorithm, can be implemented in the GPU using just a few kernels (code executed in parallel for GPU).

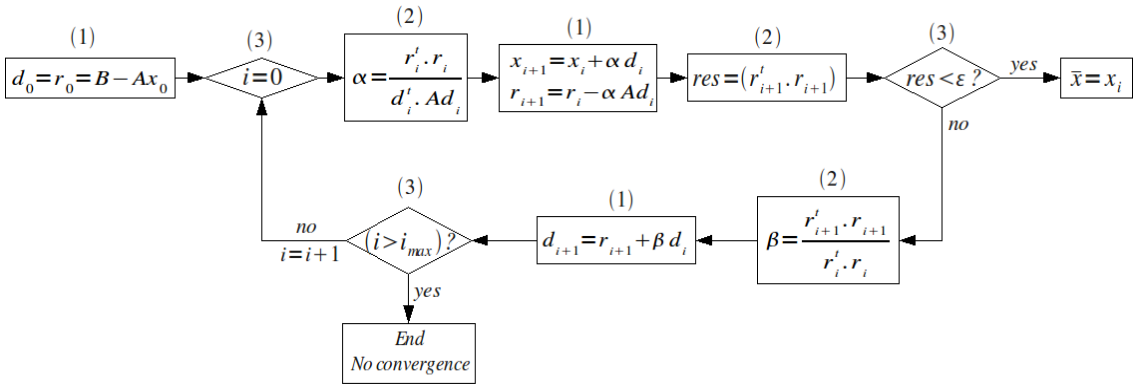

Figure 4. Algorithm of the Conjugate Gradient method.

Besides these two parts, there is also a third part of the algorithm, identified in Fig. 4 by the number (3), that consists basically in flow control and it is controlled by the CPU. This part is responsible to check the convergence of the algorithm and quit if this is achieved.

The code that executes the method is composed by consecutive calls from different kernels. The residue is the only information that is transfered from the GPU to the CPU at each iteration (this happens because it is necessary to check if the result is already inside the desired precision). However, since this transfer is equivalent to a 32-bit floating-point number, this does not represent a big delay to the GPU. Figure 5 represents a simplified code of the parallel algorithm for the Conjugate Gradient method with some calls to the kernels.

```
spmv_ellpack<<<grid, bloco>>>(Ax, A, x, N, N);
MultiAdd<<<grid2, bloco2>>>(B, Ax, d_r, -1.0f, N);
d_d = d_r;
scalarProd<<<grid3, bloco2>>>(rtr0, d_r, d_r, 1, N);

For i = 0 to #i_max
        spmv_ellpack<<<grid, bloco>>>(Ad, A, d_d, N, N);
        scalarProd<<<grid3, bloco2>>>(dAd, d_d, Ad, 1, N);

        MultiAdd<<<grid2, bloco2>>>(x, d_d, x, alfa, N);
        MultiAdd<<<grid2, bloco2>>>(d_r, Ad, d_r, -alfa, N);
        scalarProd<<<grid3, bloco2>>>(residuo, d_r, d_r, 1, N);

        Check convergence

        MultiAdd<<<grid2, bloco2>>>(d_d, d_d, d_r, beta, N);
```

Figure 5. Parallel algorithm with calls to the kernels.

MultiAdd kernel performs the addition operation between two vectors, where the second vector must be multiplied by a constant in advance. This kernel is invoked using an unidimensional grid and block. The scalarProduct kernel is used calculate a scalar products between two vectors. Within this kernel three operations are performed: first, each thread performs the multiplication term by term between the two input vectors and stores the result in a local temporary vector, then the reduction process is performed within this vector which produces the sum of all elements of the temporary vector and stores this sum in the first position, and finally the results of each block are saved back to global memory. For this kernel an unidimensional block and grid is used. The spmv_ellpack kernel in fig.6 calculates the product of a sparse matrix (NXN) ,structured by the method ELLPACK-R, with a vector (1XN). In this kernel, the thread identified by index x accesses the elements in the x row: A[x+iN] with { $0 \leq i < rl[x]$ } where i is the column index and rl[x] is the total number of non-zeros in row x. Consequently, two threads x and x + 1 access to consecutive memory address, thereby fulfilling the conditions of coalesced global memory access (F Vãzquez et al., 2009).

For both kernels, each thread in the block is responsible for one data of the vector/matrix. This way, we could achieve a better parallelism in the algorithm, while taking advantage of a memory coalescing access pattern. Figure 6 shows how the threads operate for the MultiAdd kernel.

In the end of the algorithm, the scalarProd kernel is also used to compute the sum of squared residues (Σr_i^2), which is returned to the CPU to be used as a stopping criteria, as shown in Fig. 7.

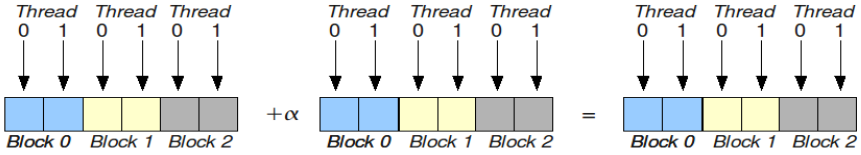

Figure 6. Visualization of the threads and blocks for the MultiAdd kernel.

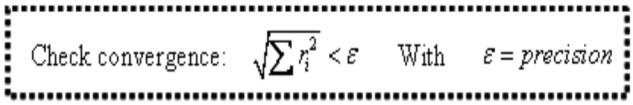

Figure 7. Stopping criteria.

5. APPLICATION EXAMPLE

Both versions of our solver (CPU and GPU) were tested to solve the problem of the 2D wave equation applied to a membrane with boundary conditions of four sides fixed. Equation 10 shows the partial differential equation (PDE) needed to be solved. In this equation, u represents the membrane displacement, c is the wave speed, t refers to the time and x and y is the 2D spatial domain of the problem.

$$c^2\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) = \frac{\partial^2 u}{\partial t^2} \tag{10}$$

To solve Eq. 10 it was discretized using finite-difference equations, which replaced the partial derivatives with a central-difference approximation. In order to make the solver unconditionally time stable, it was used the Crank-Nicholson Scheme which averages two time steps of the time discretization. Equation 11 shows the resulting stencil for this problem.

$$\begin{bmatrix}(4\alpha+1) & -\alpha & -\alpha & -\alpha & -\alpha\end{bmatrix}\begin{bmatrix}u_{x,y}^{t+1}\\ u_{x+1,y}^{t+1}\\ u_{x-1,y}^{t+1}\\ u_{x,y+1}^{t+1}\\ u_{x,y-1}^{t+1}\end{bmatrix} = \gamma \tag{11}$$

In Eq. 11, α is given by Eq. 12 and γ by Eq. 13.

$$\alpha = \frac{c^2(\Delta t)^2}{2\Delta x \Delta y} \tag{12}$$

$$\gamma = \alpha(u_{x+1,y}^t + u_{x-1,y}^t + u_{x,y+1}^t + u_{x,y-1}^t) + (2-4\alpha)u_{x,y}^t - u_{x,y}^{t-1} \tag{13}$$

So, as an example, if this problem is applied in a 2x2 grid (Fig. 8), the resulting linear system that needs to be solved is given by Eq. 14. This linear system must be solved for each time step if it is desired to observe the propagation of the 2D wave, but for our tests, it was solved for only one time step using different grid sizes.

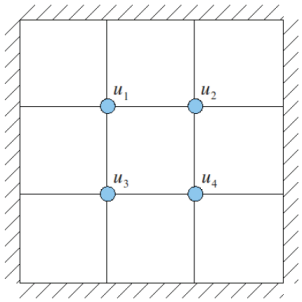

Figure 8. Example of a 2x2 grid for the 2D wave problem.

$$\begin{bmatrix} (4\alpha+1) & -\alpha & 0 & -\alpha \\ -\alpha & (4\alpha+1) & -\alpha & 0 \\ 0 & -\alpha & (4\alpha+1) & -\alpha \\ -\alpha & 0 & -\alpha & (4\alpha+1) \end{bmatrix} \begin{bmatrix} u_1^{t+1} \\ u_2^{t+1} \\ u_3^{t+1} \\ u_4^{t+1} \end{bmatrix} = \begin{bmatrix} \gamma_1 \\ \gamma_2 \\ \gamma_3 \\ \gamma_4 \end{bmatrix} \tag{14}$$

6. RESULTS

This paper presents a comparison between GPU and CPU algorithm performance. The hardware platform chosen was the personal computer equipped with a Intel Core 2 Duo E8400 processor 3 GHz with 4 GB memory, and a NVIDIA GeForce 8800 GT graphics card with 1 GB of memory GDDR3 PCI-E 16x - 256 bits. The software platform was the CUDA (Compute Unified Device Architecture) version 2.1. CUDA is a parallel computing architecture developed by NVIDIA, which offers a C++ compiler, libraries, linear algebra on GPU, and the intermediate layers of software that carries out the implementation on GPU. Linux operating system (Ubuntu 9.04 distribution) was used because it offers greater versatility and reliability than Windows operating system.

In order to test the performance of each code (parallel and serial) four situations where proposed, like showed in Table 1. The results vary from each device (CPU or GPU) where data was generated (sparse matrix A, vector x and vector B) thus the execution order. The results of all the experiments are plotted in Figure 8.

Table 1 . Possible cases studied for simulation

Case 1	Case 2	Case 3	Case 4
Data generation in CPU	Data generation in GPU	Data generation in GPU	Data generation in CPU
Execution in CPU	Execution in GPU	Execution in GPU	Execution in CPU
Copy from CPU to GPU	Copy from GPU to CPU	Data generation in CPU	Data generation in GPU
Execution in GPU	Execution in CPU	Execution in CPU	Execution in GPU

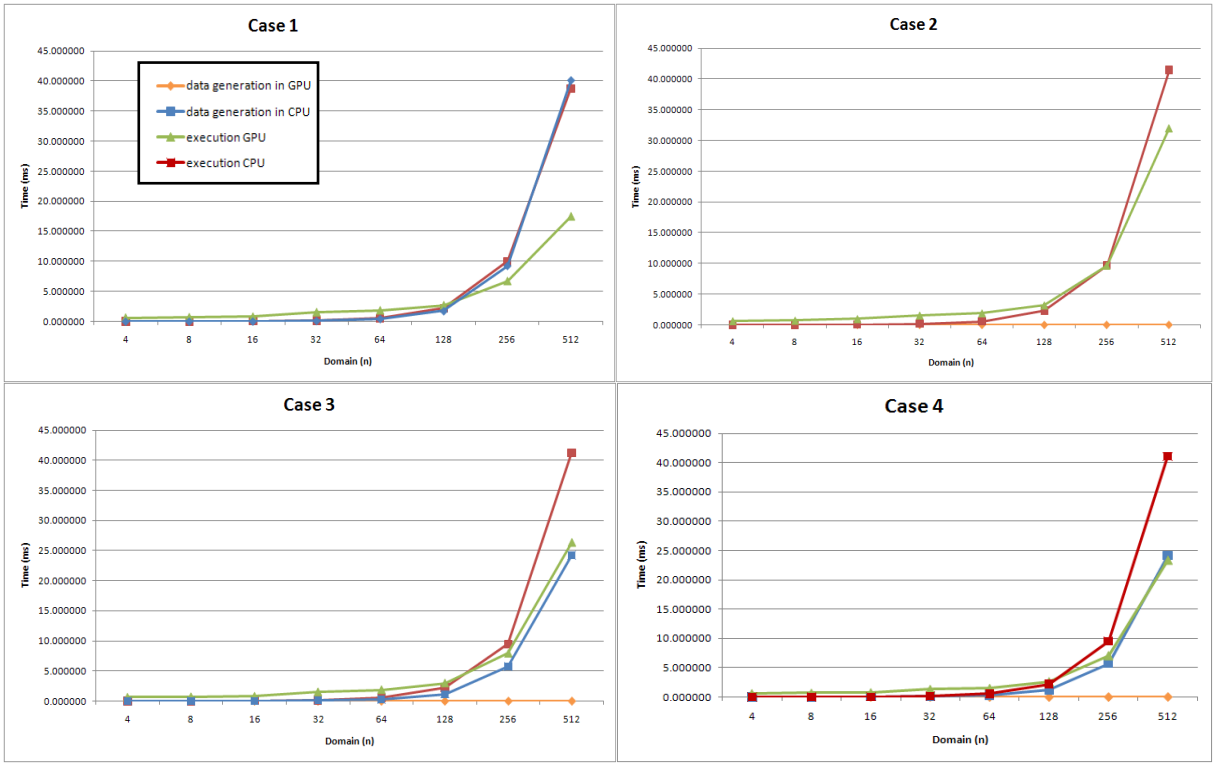

Figure 8. Simulations results.

Comparing cases 1 and 2, the results shows that when the data is generated in just one source (CPU and GPU respectively) the performance gain for the GPU is better than when data is generated by the CPU. When the GPU generates all the data, it uses the graphics main memory reducing the performance gain, which results in a increased time to solve the proposed problem. In every situation cited here there is always an advantage to process all the data in the GPU (3 times faster in case 1 and 1.4 times faster in case 2).

In cases 3 and 4, it is noted that the data generation in the GPU is a lot faster than the CPU (15000 times faster), just as the GPU processing time is twice faster in both cases.

Analyzing the proposed possibilities, the data generation is performed in just one of the devices (CPU or GPU) for cases 1 and 2, which demonstrated better performance in case 2. When both forms of data generation (CPU and GPU) where probed, the better results were obtained for case 3. This is the case where the entire process was executed in the GPU.

In general, the results showed that the GPU is about 2.8 times faster than the CPU, as can be observed in Table 2.

Table 2. Results obtained with the maximum domain supported (2048). The details of each case shows the regular execution order.

	[1] DG CPU (ms)	[2] DG GPU (ms)	[3] EX CPU (ms)	[4] EX GPU (ms)	Total Time(ms)
Case 1 (1,3,4)	717.3399±0.0511	None	814.2865±0.1076	262.5792±0.1584	1794,2056
Case 2 (2,4,3)	None	0.0323±0.0003	901.3375±0.7197	639.2529±0.2234	1540,6227
Case 3 (2,4,1,3)	406.4061±0.3489	0.0269±0.0002	865.1089±1.0797	452.6854±0.2589	1724,4694
Case 4 (1,3,2,4)	403.1616±0.0247	0.0269±0.0002	893.0198±0.0868	448.8870±0.0799	1745,0953

7. CONCLUSIONS

In this paper we proposed the parallel implementation method for the Conjugate Gradient algorithm using CUDA. To solve linear systems of large order requires a very high computational cost to a CPU, so the implementation in GPU provided a good speedup since it was made on a GPU Geforce 8800 GT graphics card, that has much more computational power than a CPU and at a lower cost. We compared the results for the same method implemented in the both CPU and GPU. The results showed an advantage for the GPU around 3 times faster than a conventional CPU in global time. We noted that data generated in CPU, when "copied" to the GPU just made memory links between pointers because the number of interactions made with the second solver was 1 every time in cases 1 and 2. The limit for our algorithm domain was 2048, with a matrix composed by 17592186044416 of elements. The results also showed the advantages and the easiness to work with CUDA, which uses C programming language rather than operation with APIs, like OpenGL, to communicate with the GPU.

8. ACKNOWLEDGEMENTS

The authors would like to acknowledge CNPq for the financial support.

9. REFERENCES

de Castro Cunha, M. C., 1993. Métodos numéricos para as engenharias e ciências aplicadas. Campinas, SP: UNICAMP, 1993.

F Vãzquez, E M. Garzõn , J. A. Martiñez, J J. Fernãndez, The sparse matrix vector product on GPUs, Computer Architecture and Electronics Dep., University of Almeria, Jun 2009.

Kruger, J. & Westermann, R., 2003. Linear algebra operators for gpu implementation of numerical algorithms. ACM Transactions on Graphics, vol. 22, pp. 908–916.

NVIDIA, 2008. NVIDIA CUDA Programming Guide 2.1.

NVIDIA. NVIDIA, 2009. Technical brief: NVIDIA GeForce 8800 GPU architecture overview.http://www.nvidia.com/page/geforce8.html.

Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Kr¨uger, J., Lefohn, A. E., Purcell, T. J., et al., 2007. A survey of general-purpose computation on graphics hardware. Computer Graphics Forum, vol. 26, n. 1, pp. 80–113.

Shewchuk, J. R., 1994. An introduction to the conjugate gradient method without the agonizing pain. Technical report, Carnegie Mellon University.

10. RESPONSIBILITY NOTICE

The authors are the only responsible for the printed material included in this paper.