

## **SOLVER DEVELOPMENT FOR LINEAR SYSTEMS BY STEEPEST DESCENT METHOD FOR PARALLEL PROCESSING ON GPU**

**Liliana de Ysasa Pozzo**

**Hugo Sakai Idagawa**

**Luiz Otávio Saraiva Ferreira**

*lipozzo@fem.unicamp.br*

*hugoidagawa@gmail.com*

*lotavio@fem.unicamp.br*

Department of Computational Mechanics, Mechanical Engineering Faculty,  
State University of Campinas, Campinas, SP, Brazil

**Abstract.** *Recent progresses in graphics processing unit (GPU) devices and its applications in general purpose computing on GPUs (GPGPU) brought new perspectives to numerical modeling of multidomain systems and, in particular, to electromechanical microsystems. Thus, the usage of this new technology can impart a great contribution to overcome computational costs limitations and dynamics modeling of conventional techniques. The majority of problems in engineering do not have analytical solutions, or those solutions have a high computational cost, therefore numerical methods are needed to get those solutions. Solution of linear systems with a large number of equations is a frequent and important problem; so we present an implementation of the Steepest Descent method for parallel processing on 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 difference method.*

**Keywords:** *GPU, parallel programming, Steepest Descent Method, graphics computing*

## 1. INTRODUCTION

Several engineering and scientific problems involve how 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). For example, a significant amount of systems of linear or nonlinear equations is the solution of differential equations, by discretization methods just as finite difference method or finite element method. 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 it imposed to 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 - Direct3D or OpenGL. Consequently, researchers who wanted to harness the GPUs processing power had to work with these APIs (Krüger and Westermann, 2003). The problem was that those individuals weren't necessarily experts in graphics programming, which seriously complicated access to the technology. Where 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, the release of those cards were totally programmable, and also appears 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 (GP GPU) 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, which its objective is to implement an iterative method known as the Steepest Descent to find the solution of those linear systems. 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).

When programmed through CUDA, the GPU is viewed as a compute device capable of

executing a very high number of threads in parallel. A kernel is a portion of an application that is executed on the GPU and can be isolated into a C language function, that is executed as many different threads.

A multiprocessor in GPU consists of 8 scalar processors with 16 kB shared memory and a total of 8192 registers. The NVIDIA GeForce 8800 GT graphics card has 112 stream processors and a 256-bit memory interface, allowing it to achieve a fill rate of 33.6 GigaTexels per second and a memory bandwidth of 57.6 GB/s. All in a slim, single-slot graphics card (NVIDIA, 2009).

## 2. THE METHOD

A System of linear equations has  $N$  equations with  $N$  unknowns. The matricial representation of a system of linear equations is  $Ax=B$ , where  $A$  is a matrix ( $N \times N$ ),  $B$  is vector of independent terms ( $N \times 1$ ), and  $x$  is a vector of unknowns. The basic approach to resolve 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. The majority of linear systems problems in engineering deal with large matrices, then direct methods become inappropriate. Another problem with direct methods is the difficulty in making them parallel, because each step of their algorithm depends on the prior results.

Iterative methods, such as Gauss-Seidel, Jacobi, Steepest descent, and Conjugated gradient, perform successive approximations to converge to the desired solution. Therefore, this work used an iterative method to explore the maximum power of the parallelization. In the steepest descent algorithm was noted operations among matrices and between matrices and vectors, thus the calculation between matrix/vector elements is done independently of the others elements, i.e. the computation can be done in parallel form. We chose the steepest descent method, because each iteration depends only on matricial and vectorial operations. This method can be easily understood, becoming appropriate for didactic studies in simulating large systems with GPU, like to find the solution for a 2D wave equation.

The Steepest Descent method is based on the minimization of a system of equations, starting from an initial approximation  $x_0$  (usually a zero vector), to reduce the error of the solution at each iteration ( $i$ ) of the method. For this, two things are necessary: the direction of greatest decrease of the system and the increment size that must be made in this direction (called the step  $\alpha$ ). With these two things is possible to calculate the next value of the solution ( $x_{i+1}$ ) and remake a new iteration of the method if the desired precision is not achieved.

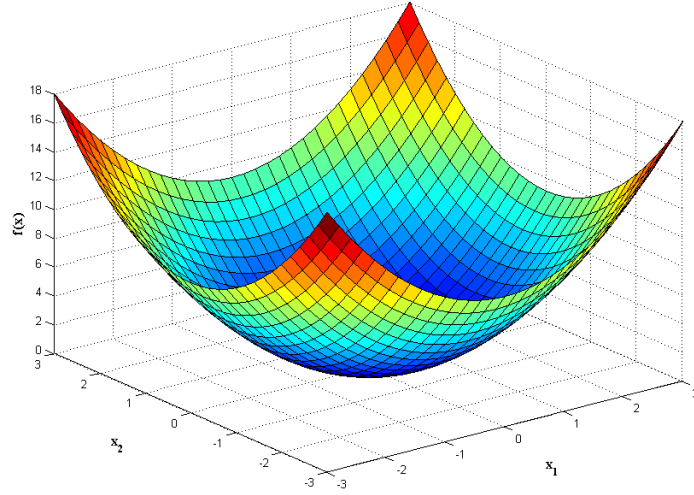
Figure 1 shows the quadratic form of  $f(x)$ . Equation 1 also represents  $f(x)$ , where  $A$  is a symmetric positive definite matrix.

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

This function is minimized when its gradient is zero.

$$\nabla f(x) = Ax - B \quad (2)$$

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 ( $x_i$ ), which can be used to check if the solution of the system is converging, and use it as a stop criteria to the method.



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

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 ( $\alpha$ ) in Eq. 3, which minimizes  $f(x_i)$  along the gradient. This is equivalent to finding  $\alpha$  for which the derivative of  $f(x_i)$  is zero (Shewchuk, 1994).

$$\alpha_i = \frac{r_i^T r_i}{r_i^T A r_i} \quad (3)$$

For the first step,  $r$ , the residue, is calculated using Eq. 4. For the subsequent steps,  $r$  must be calculated using Eq. 5.

$$r_0 = B - Ax_0 \quad (4)$$

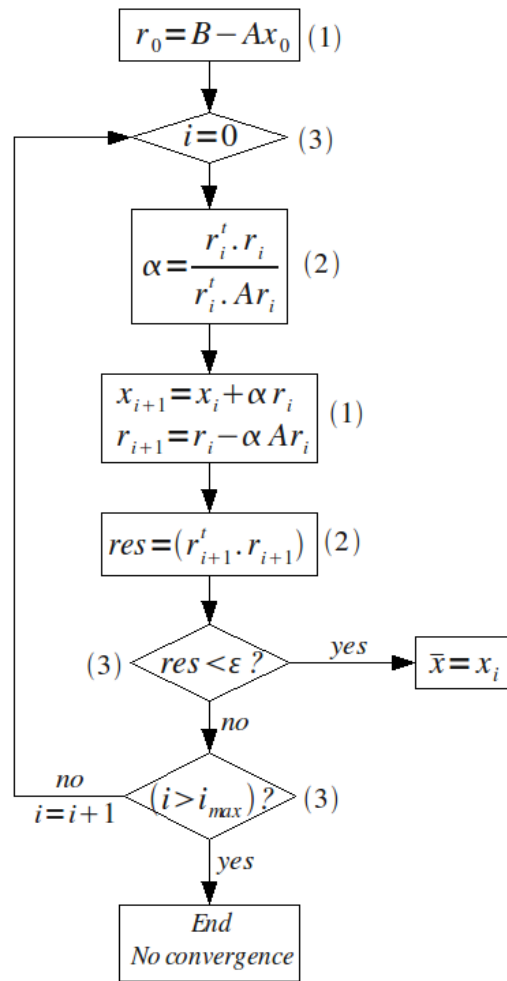
$$r_{i+1} = r_i - \alpha_i A r_i \quad (5)$$

With those two values ( $\alpha$  and  $r$ ), it is enough to update the solution ( $x_{i+1} = x_i + \alpha r_i$ ) and repeat the algorithm until the result reaches the wanted precision. Figure 2 presents in a more detailed way the Steepest Descent algorithm, while Fig. 3 shows how the method moves forward to converge. This method is easy to implement, but it possesses the inconvenience of assuring the convergence only if  $A$  is a symmetric positive definite matrix.

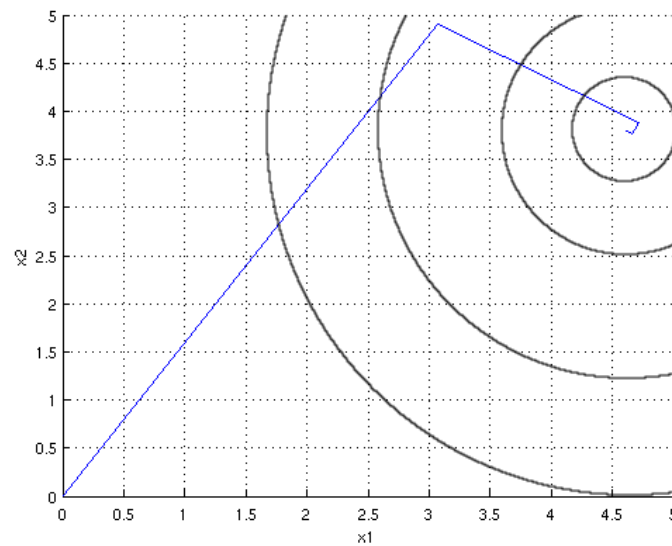
### 3. IMPLEMENTATION IN GPU

Using the algorithm described above, it can be divided in portions that can be parallelized. These portions, which are identified in Fig. 2 by the number (2), are basically composed by multiplications between two vectors (characterizing a scalar product) and between a matrix and a vector (which can be simplified as a group of scalar products).

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. 2 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 2:** Algorithm of the Steepest Descent method.



**Figure 3:** Representation of how the Steepest Descent Method converges.

Besides these two parts, there is also a third part of the algorithm, identified in Fig. 2 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. However, all the data preparation is made before the code execution, therefore data transfer between CPU and GPU is unnecessary (what would be extremely slow, decreasing the performance). The residue is the only information that is transferred 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 4 represents a simplified code of the parallel algorithm for the Steepest Descent method with some calls to the kernels.

```

scalarProd<<<grid, bloco>>>(Ax, A, x, N, N);
MultiAdd<<<grid2, bloco2>>>(B, Ax, d_r, -1.0f, N);

For i = 0 to #i_max
    scalarProd<<<grid3, bloco2>>>(rtr0, d_r, d_r, 1, N);
    scalarProd<<<grid, bloco>>>(Ar, A, d_r, N, N);
    scalarProd<<<grid3, bloco2>>>(rAr, d_r, Ar, 1, N);

    MultiAdd<<<grid2, bloco2>>>(x, d_r, x, alfa, N);
    MultiAdd<<<grid2, bloco2>>>(d_r, Ar, d_r, -alfa, N);

    scalarProd<<<grid3, bloco2>>>(residuo, d_r, d_r, 1, N);

    Check convergence
End For

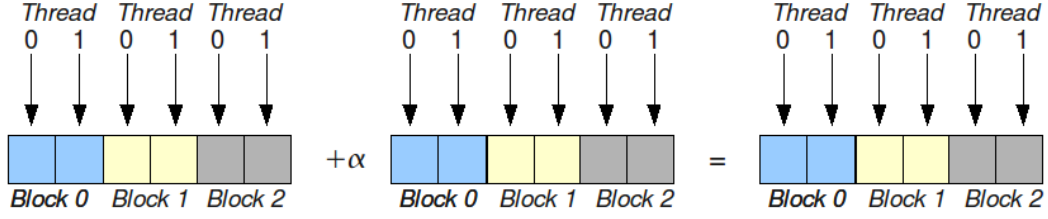
```

**Figure 4:** 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 able to calculate N scalar products by a second vector (scalar products of the type  $Ax$ ), thus one of its call arguments can be a matrix (NxN) or a vector (1xN), while the other argument must be a vector (Nx1). 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 in global memory. For this kernel an unidimensional or bidimensional block can be used. The grid, however, needs to be unidimensional.

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 5 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 ( $\sum r_i^2$ ), which is returned to the CPU to be used as a stopping criteria, as shown in Fig. 6.



**Figure 5:** Visualization of the threads and blocks for the *MultiAdd* kernel.

$$\text{Check convergence: } \sqrt{\sum r_i^2} < \varepsilon \quad \text{With } \varepsilon = \textit{precision}$$

**Figure 6:** Stopping criteria.

#### 4. 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 6 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.

$$c^2 \left( \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) = \frac{\partial^2 u}{\partial t^2} \quad (6)$$

To solve Eq. 6 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 7 shows the resulting stencil for this problem.

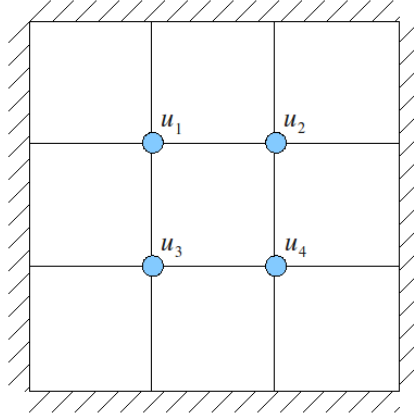
$$\begin{bmatrix} (4\alpha + 1) & -\alpha & -\alpha & -\alpha & -\alpha \end{bmatrix} \begin{bmatrix} u_{x,y}^{t+1} \\ u_{x+1,y}^{t+1} \\ u_{x-1,y}^{t+1} \\ u_{x,y+1}^{t+1} \\ u_{x,y-1}^{t+1} \end{bmatrix} = \gamma \quad (7)$$

In Eq. 7,  $\alpha$  is given by Eq. 8 and  $\gamma$  by Eq. 9.

$$\alpha = \frac{c^2(\Delta t)^2}{2\Delta x\Delta y} \quad (8)$$

$$\gamma = \alpha(u_{x+1,y}^t + u_{x-1,y}^t + u_{x,y+1}^t + u_{x,y-1}^t) + (2 - 4\alpha)u_{x,y}^t - u_{x,y}^{t-1} \quad (9)$$

So, as an example, if this problem is applied in a 2x2 grid (Fig. 7), the resulting linear system that needs to be solved is given by Eq. 10. This linear system must be solved for each



**Figure 7:** Example of a 2x2 grid for the 2D wave problem.

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.

$$\begin{bmatrix} (4\alpha + 1) & -\alpha & 0 & -\alpha \\ -\alpha & (4\alpha + 1) & -\alpha & 0 \\ 0 & -\alpha & (4\alpha + 1) & -\alpha \\ -\alpha & 0 & -\alpha & (4\alpha + 1) \end{bmatrix} \begin{bmatrix} u_1^{t+1} \\ u_2^{t+1} \\ u_3^{t+1} \\ u_4^{t+1} \end{bmatrix} = \begin{bmatrix} \gamma_1 \\ \gamma_2 \\ \gamma_3 \\ \gamma_4 \end{bmatrix} \quad (10)$$

## 5. RESULTS

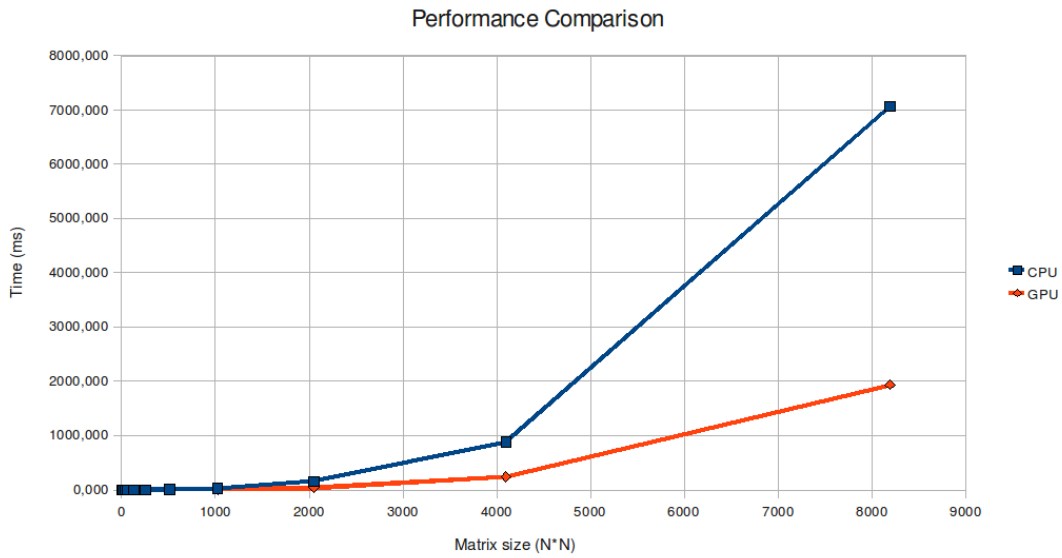
This paper presents a comparison between GPU and CPU algorithm performance. The hardware platform chosen was the personal computer equipped with a Intel Core 2 Duo E8400 processor 3 GHz with 4 GB memory, and a NVIDIA GeForce 8800 GT graphics card with 1 GB of memory GDDR3 PCI-E 16x - 256 bits. The software 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.

To test the performance of each implementation (parallel and serial), random linear systems with different sizes were generated, saved in text files and loaded inside each algorithm to solve. The results generated were then compared with the solution provided by MatLab to validate the algorithms.

Table 1 shows the results of these tests and Fig. 8 shows a graphic comparison of the performance difference between the CPU and the GPU. The first column of Tab. 1 represents the size of the linear system, i.e. the number of unknowns in the linear system. It can be seen that in both cases, the execution time increases jointly with N (order of the linear system). Nevertheless, the CPU time increased more quickly than the GPU time, which increases gradually. For N equals to 1024, the gap between the GPU and CPU becomes more visible. When we have a linear system bigger than 2048x2048, the performance speedup is around 3.6x. Thus, for big linear systems the GPU performance is better than the CPU.

**Table 1:** Tests results for different matrix sizes

Matrix Size (NxN)	CPU Iterations	CPU [ms]	GPU Iterations	GPU [ms]
4	11	0.003	11	0.944
8	17	0.003	17	1.023
16	20	0.011	20	1.559
32	20	0.025	20	1.583
64	21	0.096	21	1.971
128	22	0.411	22	2.254
256	23	1.640	23	2.895
512	25	6.959	25	5.393
1024	28	30.835	28	13.260
2048	33	159.081	33	40.694
4096	47	886.504	47	247.828
8192	95	7065.225	95	1941.227

**Figure 8:** Performance comparison (CPU vs. GPU).**Table 2:** Tests results for different grid sizes for the 2D wave equation problem

Grid size	CPU Iterations	CPU [ms]	GPU Iterations	GPU [ms]
4	5	0.004	5	0.489
8	6	0.034	6	0.571
16	6	0.526	6	0.883
32	7	9.287	7	5.161
64	7	165.539	7	54.456

Table 2 shows the results for the tests performed using the 2D wave equation. It must be noted that for a grid size  $N$ , the order of the linear system is equal to  $N \times N$ . So, for the grid size equals to 64, the algorithm is actually solving a linear system with  $(64 \times 64)$  unknowns. This explains why there are fewer tests in Tab. 2 than in Tab. 1. Just like the results for Tab. 1, it is noted that the GPU algorithm is faster than the CPU version for larger linear systems.

Since the GPU is capable of solving this linear system faster than the CPU for larger grids

at each time step, it provides the possibility to display the propagation of this wave at reasonable frame rates.

## 6. CONCLUSIONS

In this paper we proposed the parallel implementation method for the Steepest Descent 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 as the order of the linear system increases. The proposed method can be implemented without many difficulties, but the drawback of this method is the convergence requirement: that is, the matrix  $A$  must be symmetric and positive definite. Even with this inconvenience, a lot of modelling studies comply with this requirement, so we still get the speed advantage of the GPU. The algorithm shows that the GPU is around 3.6 times faster than a conventional CPU. 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.

Since there are a large number of engineering problems that require numerical methods with a solver for linear systems, we used the 2D wave problem to demonstrate a reduction in the simulation time for this type of problems when using the GPU. It shows that it can be a good alternative to reduce the simulation time and has a computational power not yet well explored. The next steps of that research is to implement the Conjugate Gradients method and compare it against the Steepest Descent method. Since we already have the basic kernels working (*MultiAdd* and *scalarProd*), the Conjugate Gradient method can be implemented faster. It is known that the Conjugate Gradients method provides a faster convergence than the Steepest Descent, but it requires a few more matrix/vector operations for each iteration. So, this could potentially result in a bigger convergence time than the Steepest Descent method for linear systems that requires only a few iterations to converge.

## Acknowledgements

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

## REFERENCES

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