# Project 1

## Finite Difference Solution of a Vibrating 2D Membrane on a GPU

Shawn Hinnebusch

October 30, 2020

University of Pittsburgh

Professor Senocak

Parallel Computing for Engineers

# Introduction:

As researchers try to push the limits on solving complex problems, computers and hardware can become limits as many problems are very computationally expensive. Simulations can take days, weeks, and even months to run. CPUs are designed to handle complex tasks which will complete many of these simulations. However, GPUs are meant to do one thing and do it very well. They can handle repetitive low-level tasks which is exactly what is needed in many computational fields. This is where the GPU shines compared to the CPU. If you consider a pipeline of tasks, the GPU splits the tasks into multiple stages where it can complete these simultaneously. [1] This process gives a much faster speed up as the architecture is much different than CPUs which are meant for security, running operating systems, and many other things.

CUDA is produced by NVIDIA specifically API programming for user interface into using computational coding to feed into their GPU cards. Some of the main languages they support are C, C++, Fortran, along with integration into Python and many other languages. It is a rapidly growing program used in computation along with machine learning. CUDA can be used across multiple cards along with super computers to make use of lots computational power to solve larger problems that were before impossible to complete.

# Part 1: Initial Displacement of the membrane

The first part of this project is creating the matrices and initializing the starting conditions. To first create the matrices, the function calloc was called to initialize the variables to zero. CudaMallocManaged was used for the GPU counterparts. From here, equation 1 was used to fill the current values of the initiation. The results were printed in MATLAB 2020a which was used with the rest of the figures in this report. Figure 1 shows a surface plot showing that it starts in a dome configuration.

$$\phi(x, y, t) = 0.1(4x - x^2)(2y - y^2) \quad (1)$$

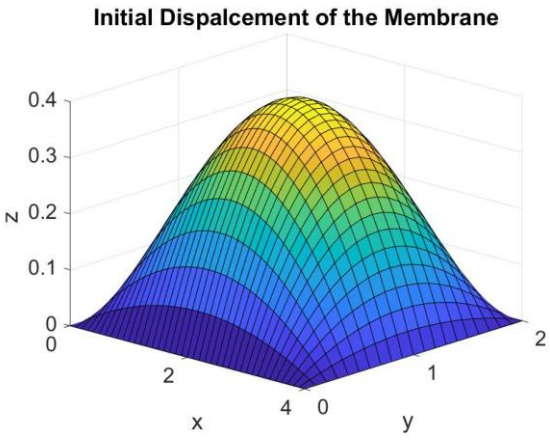


*Figure 1: Initial condition plotted of phi at time = 0*

## Part 2: C program

A two dimensional wave of a membrane is represented by the linear partial differential equation in equation 2. Phi is the height of the wave with x and y being the spatial coordinates. $C^2$ is a constant which in this project is fixed at 5. t is time.

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

The boundary conditions for this wave is that it is clamped on all the 4 edges. To solve equation 2, a second order accurate central finite difference scheme is used. This scheme is shown in equation 3. In each case of the problems in this project, the number of elements in the x and y direction were chosen to keep $\Delta x$ and $\Delta y$ the same. This simplifies the equation slightly as terms can be combined. Part of the initial condition was used in solving equation 3 to come up with the previous iteration before stepping forward in time.

$$\frac{\phi_{i,j}^{t+1} - 2\phi_{i,j}^{t} + \phi_{i,j}^{t-1}}{\Delta t^2} = c^2\left(\frac{\phi_{i+1,j}^{t} - 2\phi_{i,j}^{t} + \phi_{i-1,j}^{t}}{\Delta x^2} + \frac{\phi_{i,j+1}^{t} - 2\phi_{i,j}^{t} + \phi_{i,j-1}^{t}}{\Delta y^2}\right) \tag{3}$$

Since the finite difference scheme steps forward in time, a time step must be chosen. To make sure there is a numerical stability, a time step shown in equation 4 is used. In the actual code a $\Delta t = h * c$ was used. Since this value does not fit the criteria, it was divided by 10 to ensure it was always smaller.

$$\frac{c\Delta t}{h} < 1.0 \tag{4}$$

# Part 3: Computing the analytical solution

To check the finite difference on the CPU and the GPU, the analytical solution was used for a comparison. This equation is shown in equation 5. This was calculated and stored in memory along with the CPU and GPU version.

$$\phi(x,y,t) = \sum_{m=1,odd}^{\infty} \sum_{n=1,odd}^{\infty} \left( \frac{1}{m^3 m^3} \cos\left( t \frac{\sqrt{5}\pi}{4} \sqrt{m^2 4n^2} \right) \sin\left( \frac{m\pi x}{4} \right) \sin\left( \frac{n\pi y}{2} \right) \right) \quad (5)$$

Figures 2 and 3 show the results of the analytical solution along with the CPU and GPU results. In the top left, are 3 constant values of x plotted for y against phi. The top right is a constant value of x = 0.25L of the analytical solution, CPU and GPU. Bottom left shows the 3 constant values of x plotted for the CPU. The bottom right shows the GPU results also plotted as the 3 constant values. Figure 2 are all results for a time of 0.1 seconds. Figure 3 shows the results of time at 0.5 seconds.

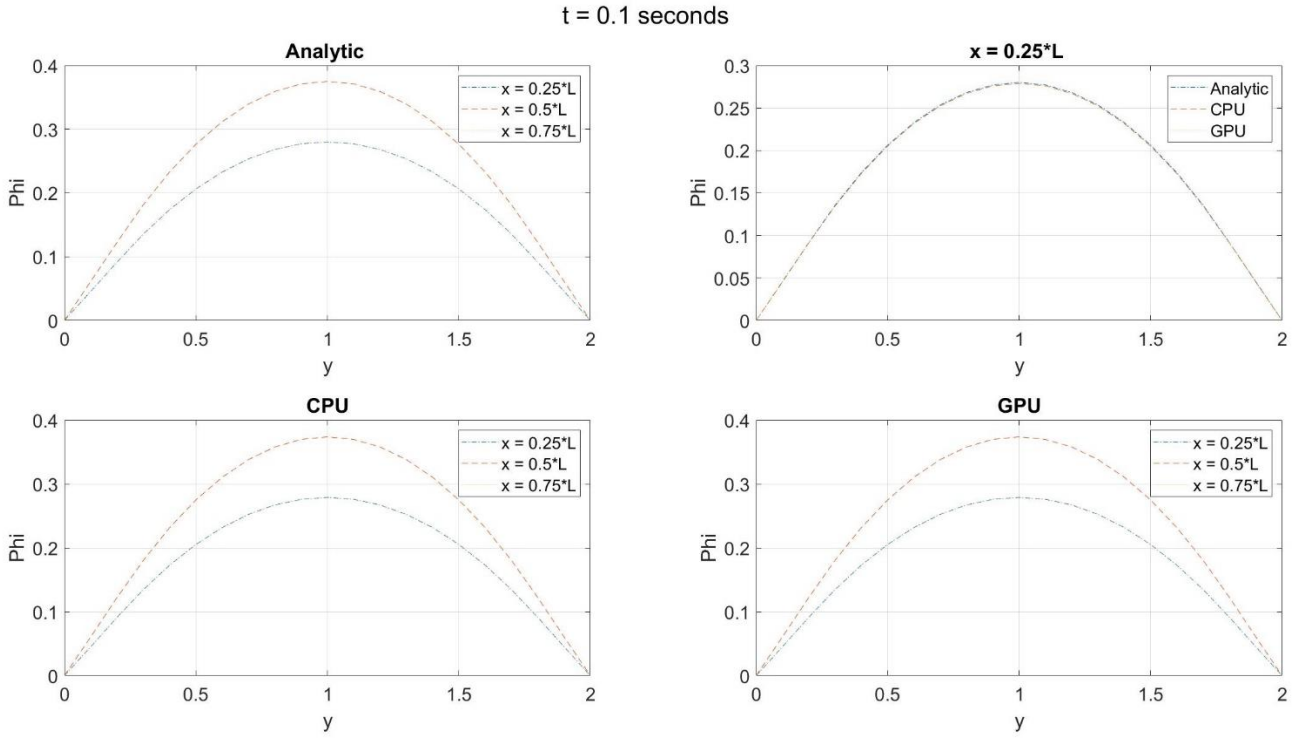


*Figure 2: Time = 0.1 seconds. Comparison of the analytical solution to the CPU and the GPU solutions. Top left is the analytic solution. Bottom left is the CPU results. Bottom right is the GPU results at x = 0.25L, 0.5L, and 0.75L. Top left is the analytical solution, CPU, and GPU at x = 0.25L.*

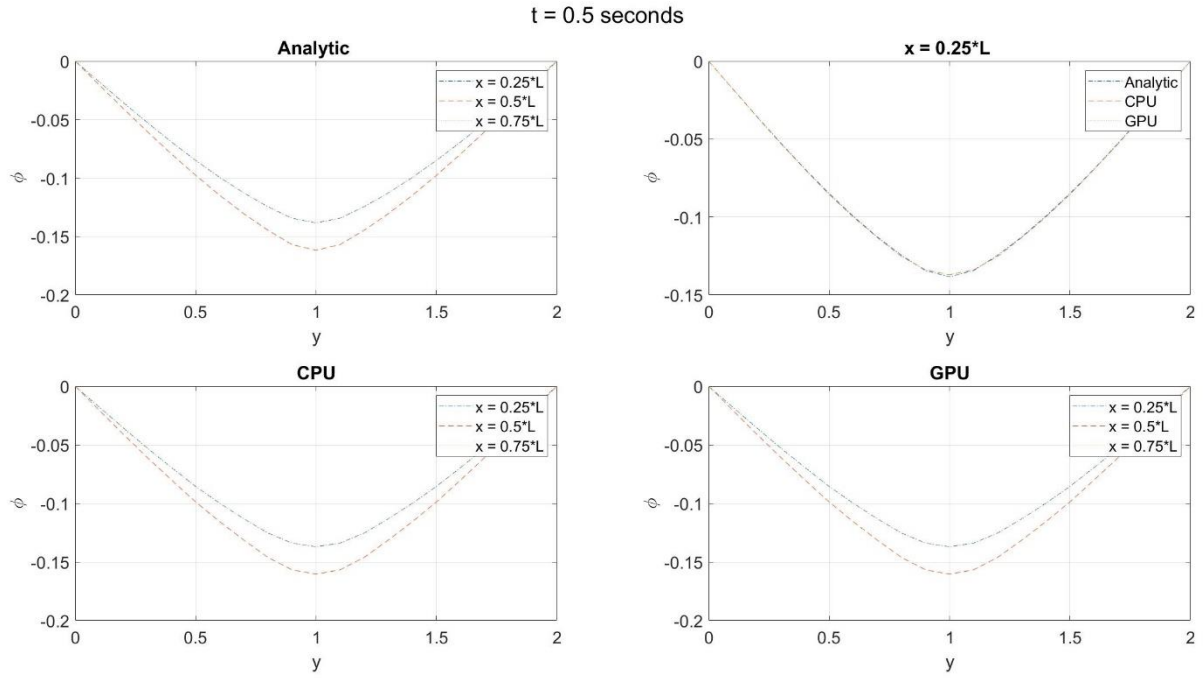


*Figure 3: Time = 0.5 seconds Comparison of the analytical solution to the CPU and the GPU solutions. Top left is the analytic solution. Bottom left is the CPU results. Bottom right is the GPU results at x = 0.25L, 0.5L, and 0.75L. Top left is the analytical solution, CPU, and GPU at x = 0.25L*

The last part of comparison is showing the analytical solution at 4 different time references of 0.1, 0.5, 1.2, and 1.5 seconds. The various wave profiles are shown in figure 4.

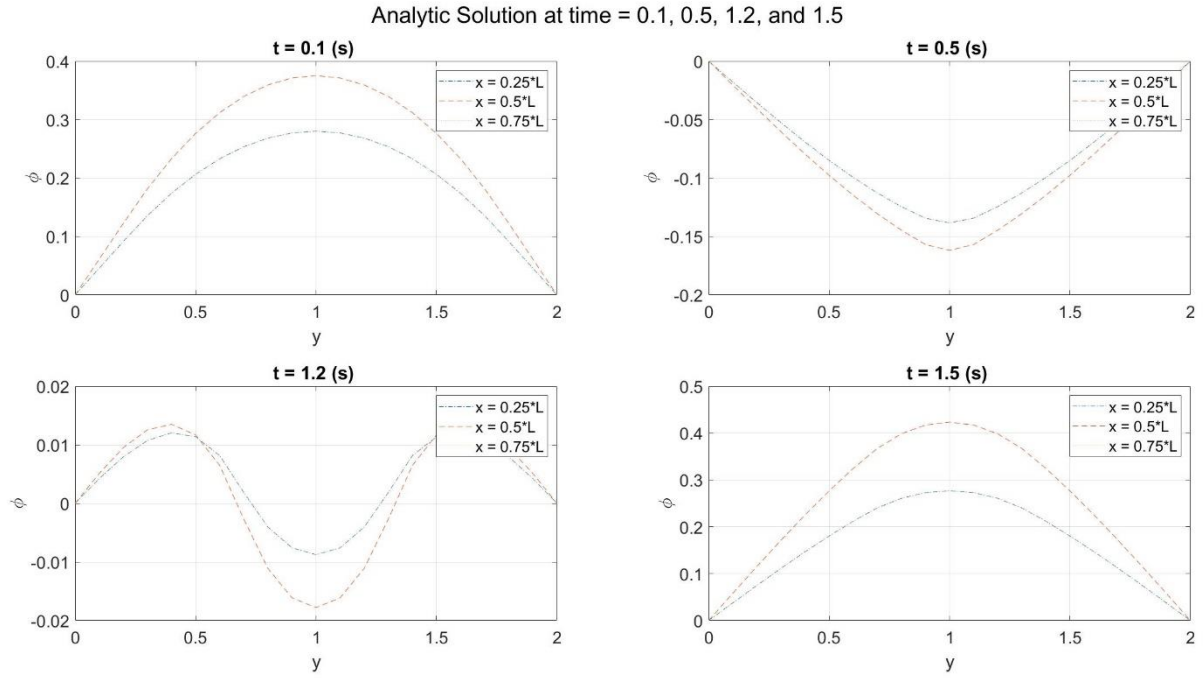


*Figure 4: Wave profiles for instances of x = 0.25L, x = 0.5L, and x = 0.75L.*

## Part 4: 3D surface contour of the wave near its maximum amplitude

To find the wave near its maximum amplitude, a function was created to loop through the current value of phi and find the maximum value. This code was run on a small mesh for time up to 5 seconds to find the max. The maximum was found to be around 1.5 seconds. The surface plot is shown on the bottom right, but a few other time steps were included to show some of the main wave transformations that were found.

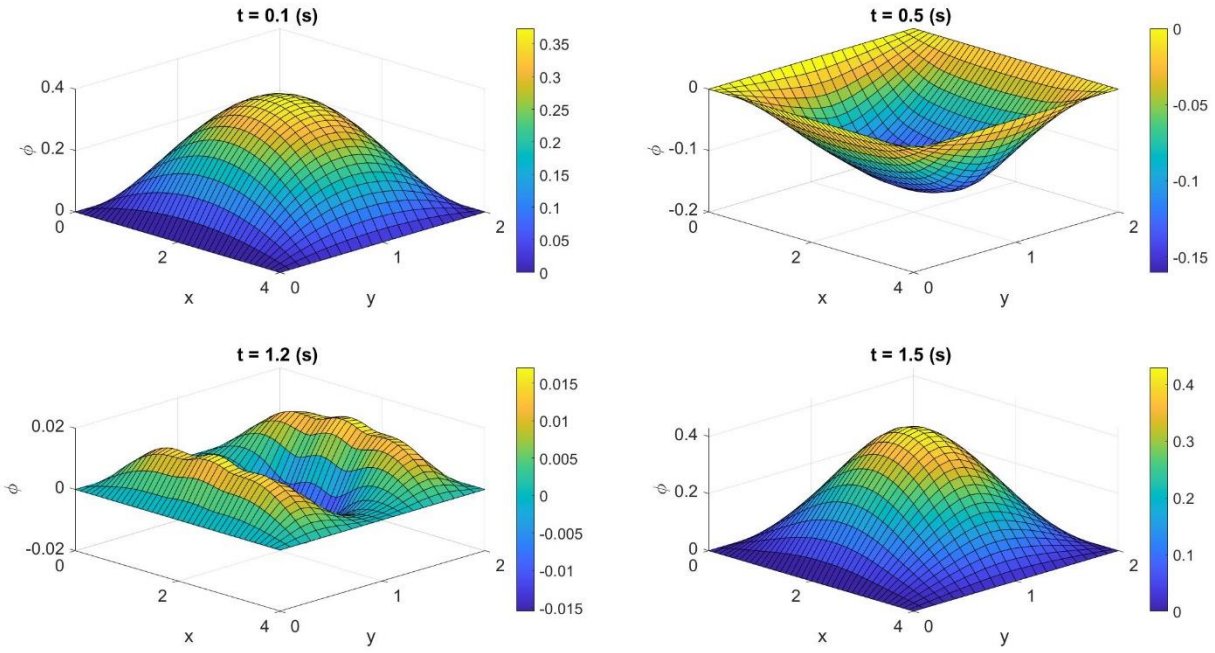


*Figure 5: Surface plots of the finite difference results for various time steps*

## Part 5: Parallel performance relative to serial CPU

The last part of this project is comparing the speed of the CPU and the GPU. To accomplish this, various block sizes were run through the code and timed to get the results shown in table 1. The time was set to 1 second to keep the loop size the same. All the initialization was completed outside of the timing function to allow only the time stepping forward to be calculated.

*Table 1: Speed comparison for the CPU and GPU*

| | CPU | GPU | | | |
|---|---|---|---|---|---|
| **Mesh Size** | 1 core (s) | 8x8 (s) | 16x16 (s) | 32x16 (s) | 32x32 (s) |
| **513x257** | 183.412 | 16.341 | 13.823 | 13.4267 | 12.5922 |
| **1026x512** | 1,523.51 | 67.8216 | 49.5555 | 48.127 | 50.1309 |
| **2049x1025** | 13,265.87 | 366.2786 | 293.2203 | 291.8103 | 294.0038 |

From these timing results, it was noted that for the GPU a block size of 8x8 was the slowest with the rest of the block sizes vary close in timing. Since the time always fluctuates, it is difficult to tell which block size is the fastest. To future visualize the results, the data of the CPU and the GPU of block size 16x16 were plotted of mesh size against the time.

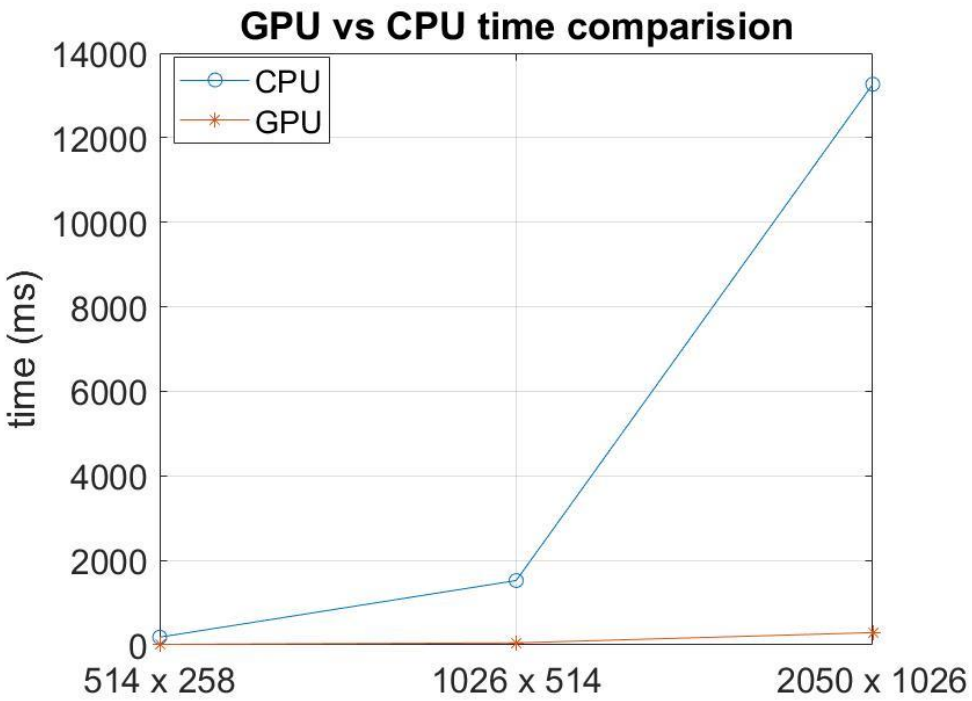


*Figure 6: GPU and CPU time plotted for various mesh sizes.*

## Conclusions:

In the end of this project, the wave equation was compared using the analytical solution, the finite difference for GPU and CPU and a time comparison to see the speed up effects. The CPU and GPU results matched identical and those results matched very closely with the analytical solution. There was a slight error on the analytical to finite difference, but this is expected. The error grew more as time went on in the time stepping. The speed comparison shows a massive reduction in time it takes utilizing the GPU instead of just the CPU. If the time was included to generate and initialize the mesh, the speed up overall would not be as great, but still very worth while in the end for large mesh sizes. For small mesh sizes, the time to copy the data to and from the GPU can actually slow the time than using just the CPU.

## References:

J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone and J. C. Phillips, "GPU Computing," in Proceedings of the IEEE, vol. 96, no. 5, pp. 879-899, May 2008, doi: 10.1109/JPROC.2008.917757.

“Performance Development.” *TOP500*, www.top500.org/statistics/perfdevel/.