Audio Engineering Society

Convention Paper 7963

Presented at the 128th Convention
2010 May 22–25 London, UK

A GPGPU Approach to Improved Acoustic Finite Difference Time Domain Calculations

Jamie A S Angus[1], Andrew Caunce[2]

School of Computing, Science and Engineering, University of Salford, Salford, Greater Manchester, M5 4WT, United Kingdom
[1]j.a.s.angus@salford.ac.uk
[2]a.j.caunce@student.salford.ac.uk

ABSTRACT

This paper shows how to improve the efficiency and accuracy of Finite Difference Time Domain acoustic simulation by both calculating the differences using spectral methods and performing these calculations on a Graphics Processing Unit (GPU) rather than a CPU. These changes to the calculation method result in an increase in accuracy as well as a reduction in computational expense. The recent advances in the way that GPU's are programmed (for example using CUDA on Nvidia's GPU) now make them an ideal platform on which to perform scientific computations at very high speeds and very low power consumption.

1. INTRODUCTION

Finite difference time domain methods were first conceived for electrodynamics problems in the 1960s by Yee [1] in order to solve Maxwell's equations for electrodynamics problems. His work was built upon by and the term FDTD coined by Alan Taflove in 1980 [2]. Meloney and Cummings first applied this method to acoustics in 1995, [3].

The finite difference time domain method involves discretising both time and space, and then solving the 2 dimensional wave equation by way of the leapfrog method. In order for this to work one must separate acoustic particle velocity (in both the x and y directions as particle velocity is a vector quantity) and pressure (which is scalar) onto separate points on the grid as shown in Figure 1.

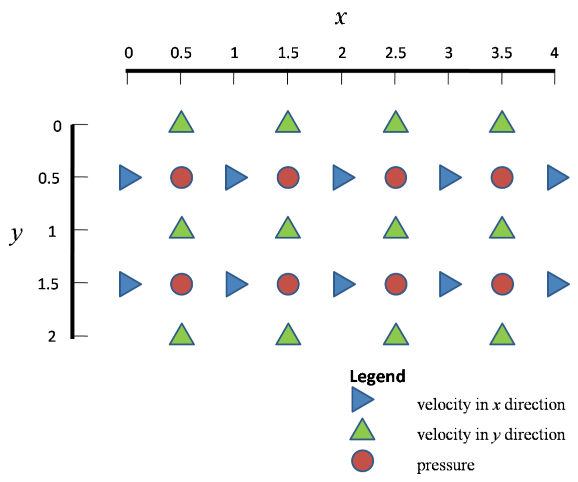

Figure 1 FDTD Grid used in simulations (Adapted from picture in [4])

1.1. Spectral Methods

Spectral Methods form one member of the three major technologies used in the solution of partial differential equations. These are the finite element method, the aforementioned finite difference methods, and spectral methods.

If one is solving a PDE where the initial data is smooth enough, then a spectral method offers very high accuracy or lower computation times [5]. Spectral methods for solving differential equations involve using the Fourier transform to perform the differentiation. This is achieved by taking the impulse response of a perfect differentiator and convolving it with the input data. This convolution can be done by multiplying the two 'signals' in the 'frequency' domain (or k domain in the case of the input FDTD data). This means that the whole extent of the input signal is taken into consideration when calculating the derivative as opposed to just a few surrounding points, thus removing the dispersion error created by the leapfrog method of solving the PDE's. This has the advantage that the grid would no longer have to be sampled at higher than the normal Nyquist limit and therefore could be sampled at just two times the highest frequency, reducing 2D calculations by up to 5 times and 3D calculations by up to 25 times.

1.2. GPU Computing

The architecture of the GPU differs greatly from that of the CPU in that the CPU will have up to 4 separate processing cores, whereas the GPU has up to (as of 2009) 200+. This means that if the problem can be naturally split into numerous operations that can be calculated in parallel (i.e. Each calculation can be calculated independently of the other calculations being carried out) then intuitively substantial speed-ups can be achieved.

This lends itself to the world of scientific computing where many problems can be effectively parallelized, a good example being the fast Fourier transform. This has achieved major speedups over the equivalent CPU algorithms [6]. This indicates a huge increase in computational efficiency meaning that higher degrees of accuracy can be achieved in more feasible amounts of time.

The major limitations of GPU programming are the time taken to access global memory and the size of the local memory and cache. This means that in order to achieve maximum efficiency accesses to the global or host memory should be kept to an absolute minimum (ideally eliminated altogether) this means that it is advantageous to recalculate values as they are needed as opposed to calculating once and then accessing the memory. In other words in order to maximise the efficiency of the program one should optimise the arithmetic intensity to minimise the memory accesses. Thus using the spectral methods means that although there are potentially more calculation steps ($N\log_2 N$) more to be exact, there should be a lot less memory accesses due to the much smaller grid size than is necessary for the straight leap frog formula.

1.3. CUDA

CUDA, Complete Unified Digital Architecture, is NVIDIA's implementation of GPGPU. CUDA is based around the industry standard C programming language and provides a simple way of programming for GPUs. The programming interface remains fairly high level with the compiler and runtime deciding which thread to run on which processor etc., meaning that the programmer is left with a very intuitive way to program [7].

2. STANDARD FDTD CALCULATION METHOD ON GPU

The Finite Difference Time Difference method, when applied to acoustics, is in essence a finite solution to the second order wave equation below, and Euler's equation to relate pressure to velocity for simplicities sake this will be explained in terms of the 1 dimensional case.

$$\text{1D Wave Equation:} \quad \frac{\partial^2 p}{\partial x^2} - \frac{1}{c^2}\frac{\partial^2 p}{\partial t^2} = 0 \tag{1}$$

$$\text{Differentiated Euler's Formula:} \; \rho_0 \frac{\partial}{\partial x}\left[\frac{\partial u}{\partial t}\right] = \frac{\partial^2 p}{\partial t^2} \tag{2}$$

The above equation (1 is then equated to Euler's formula, which has been differentiated with respect to space (2).

$$\rho_0 \frac{\partial}{\partial x}\left[\frac{\partial u}{\partial t}\right] = \frac{1}{c^2}\frac{\partial^2 p}{\partial t^2} \tag{3}$$

Integrating equation (3) with respect to time yields:

$$\frac{\partial p}{\partial t} = -\rho_0 c \frac{\partial u}{\partial x} \tag{4}$$

Expressing this equation and (2) in terms of both time and space then discretising both pressure and velocity onto separate points as shown in the Figure 1 yields:

$$p\left(x_{i+\frac{1}{2}}, t_{n+\frac{1}{2}}\right) = p\left(x_{i+\frac{1}{2}}, t_{n-\frac{1}{2}}\right) - \rho_0 c \frac{\Delta t}{\Delta x}\left[u(x_{i+1}, t_n) - u(x_i, t_n)\right] \tag{5}$$

$$u(x_i, t_n) = p(x_i, t_{n-1}) - \frac{1}{\rho_0}\frac{\Delta t}{\Delta x}\left[u\left(x_{i+\frac{1}{2}}, t_{n-\frac{1}{2}}\right) - u\left(x_{i-\frac{1}{2}}, t_{n-\frac{1}{2}}\right)\right] \tag{6}$$

These equations are now dependent only on the previous values pressure and velocity respectively.

2.1. Sources

There are three types of source that are useful in calculating FDTD when applied directly to a pressure node, these are:

1. Hard Source

2. Soft Source

3. Transparent Source

2.1.1. Hard Source

This is implemented by, at the desired point of excitation, the FDTD equation is negated and the pressure of that point (x_{sx}) is excited only by the required source equation.

$$p\left(x_{sx}, t_{n+\frac{1}{2}}\right) = S(x_{sx}) \tag{7}$$

This type of source is useful for the fact that the signal that inputs into the grid is identical to that desired. However because the FDTD equations have been negated, the source will cause reflections and scattering.

2.1.2. Soft Source

A soft source is implemented by adding the source data onto the calculated FDTD pressure in the form:

$$p\left(x_{sx}, t_{n+\frac{1}{2}}\right) = S(x_{sx}) + p\left(x_{i+\frac{1}{2}}, t_{n-\frac{1}{2}}\right) - \rho_0 c \frac{\Delta t}{\Delta x}\left[u(x_{i+1}, t_n) - u(x_i, t_n)\right] \tag{8}$$

This type of source has the advantage of not introducing reflections or scattering however the downfall is that the signal is changed to a degree by the grid. The reason for this is that the grid acts as a form of medium and as such has an impulse response that alters the signal being fed into the grid.

2.1.3. Transparent Source

The transparent source aims to reduce the warping caused by the impedance of the grid, by matching the impedance of the source to that of the grid. This is achieved by convolving the source signal with the impulse response of the grid. In two dimensions this is impossible due to the high frequency limit of the grid (Section 3.3.2); this means that a proper impulse response cannot be achieved, as a Dirac delta function will not give good enough data.

Due to the complexity of this method and the fact that due to the aforementioned limitations the effect of this source is to only reduce the warping, not eliminate it, this method is impractical for this proof of concept style project.

2.2. Boundaries

Boundaries in the calculation of FDTD are subject to a large amount of research [8] and can be of many levels of complexity. The only two to be used in this project are a perfectly hard boundary, and a perfectly matched layer.

2.2.1. Perfectly Hard Boundary

This is achieved by way of; at the specific node where a boundary is required the velocity is forced to be zero. This simulates a surface with a reflection coefficient of one and an absorption coefficient of zero. This is useful when trying to model the effects of diffraction and other low frequency effects although of limited use in practical simulations of real spaces.

2.2.2. Perfectly Matched Layer

A perfectly matched layer is a necessary piece of technology in any FDTD calculation, in that it is used to absorb any energy heading for the edge of the grid. It works to gradually absorb the energy of a wave propagating in any direction.

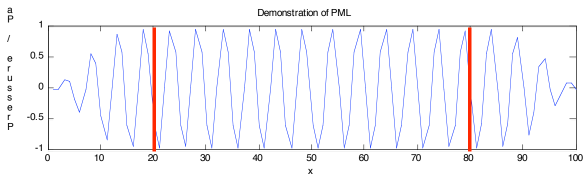

Figure 2 Demonstration of Perfectly Matched Layer

This means that it can be used to model perfectly anechoic conditions and also, when a boundary that is not totally absorbent is used, some energy is transmitted through the boundary, a PML can be used to absorb this and prevent spurious reflections from entering the grid.

The PML is implemented by way of an auxiliary parameter, which is defined for a PML at both edges of a 1D grid as;

$$\text{let } d = \text{PML Depth}$$

$$\alpha = \begin{cases} i < d & \alpha = \frac{1}{3}\left(\frac{i}{d}\right)^3 \\ d < i < N-d & \alpha = 0 \qquad\qquad i = 1,2,3,\ldots,N-1 \\ i > N-d & \alpha = \frac{1}{3}\left(\frac{N-i}{d}\right)^3 \end{cases} \tag{9}$$

Then the FDTD equations are in order to accommodate this auxiliary parameter thus:

$$p\left(x_{i+\frac{1}{2}}, t_{n+\frac{1}{2}}\right) = p\left(x_{i+\frac{1}{2}}, t_{n-\frac{1}{2}}\right)\left(\frac{1-\alpha_{i+\frac{1}{2}}}{1+\alpha_{i+\frac{1}{2}}}\right) - \rho_0 c\frac{\Delta t}{\Delta x}\left(\frac{1}{1+\alpha_{i+\frac{1}{2}}}\right)[u(x_{i+1},t_n) - u(x_i,t_n)] \tag{10}$$

$$u(x_i,t_n) = p(x_i,t_{n-1})\left(\frac{1-\alpha_i}{1+\alpha_i}\right) - \frac{1}{\rho_0}\frac{\Delta t}{\Delta x}\left(\frac{1}{1+\alpha_i}\right)\left[u\left(x_{i+\frac{1}{2}}, t_{n-\frac{1}{2}}\right) - u\left(x_{i-\frac{1}{2}}, t_{n-\frac{1}{2}}\right)\right] \tag{11}$$

2.3. Adaptations for Implementation in CUDA

The first major change is that instead of looping through the values in space, an individual thread is launched to calculate each point. For each time step there are 2 sets of threads launched, this is to ensure that all of the velocity values are calculated before the pressures begin to be calculated. This means that the pressure and velocity values of each point at each time step are calculated at the same time, greatly reducing the time taken iterating through the points in space.

For example in calculating the perfectly matched layer, the values of the coefficients are calculated in each thread, thus saving an allocation and transfer of three large 2D, NxN, arrays into the device memory. In order to reduce memory transfers between the host and the device only the output pressure after each time step will be copied back in to the host memory and output. However if velocity were required this would have to be copied as well as or in place of pressure.

2.4. Proof of Accurate Calculation

The following output clearly shows the FDTD working and showing diffraction working, the source was the first half of a sine wave (similar to a cosine window), to represent an impulse. The subsequent plots show that the wave is propagating across the grid and becoming incident upon a wall with holes in it.

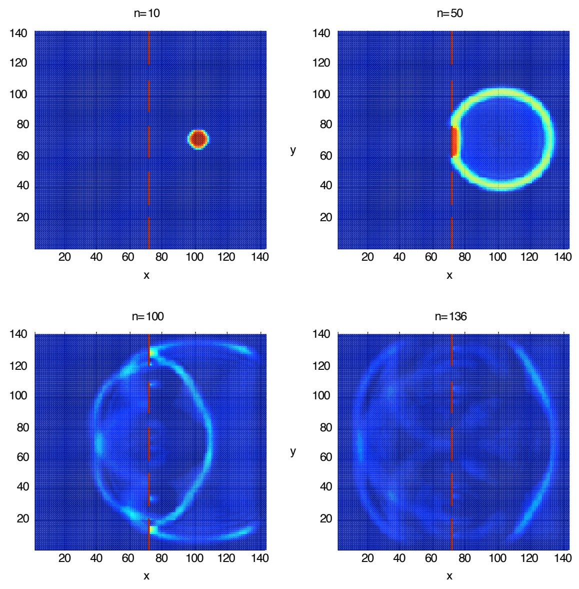

Figure 3 Plots of pressure versus position showing the propagation of the wave over time

Also shown in this plot is the PML, it is clearly visible that there are no spurious reflections from the boundaries as the wave is absorbed entirely by the PML, thus accurately modeling a perfectly reflection free environment.

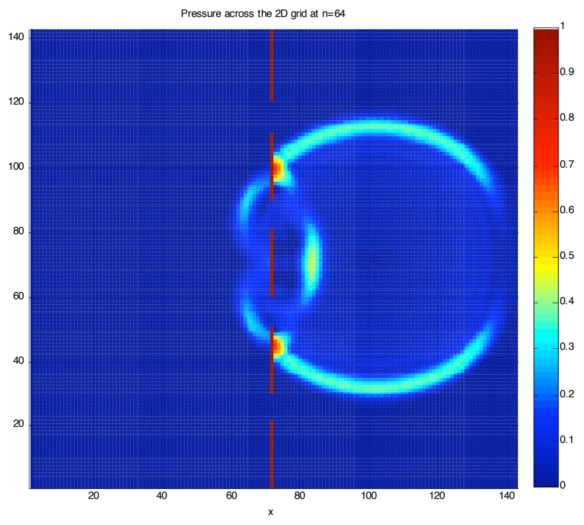

Figure 4 Output of FDTD program showing diffusion taking place.

It is clear from the above output that the FDTD is accurately modeling the diffraction of the wave around the edges of the barriers. The red lines indicated a perfectly hard surface (velocity forced to 0). A line has been drawn between the excitation point and the edge of one of the hard surfaces. This is to demonstrate that diffraction is taking place. Any energy within the white lines, i.e. in the shadow zone (shaded), can only be there due to diffraction, in this example we see plenty of energy in this region proving that diffraction is being modeled effectively.

2.5. Problems with this Method

The major problem with the standard method of calculating the FDTD is that there is a high frequency limit to the method. This is demonstrated in the figure 5. In order to accurately calculate the frequency response the impulse response of a low pass filter (sinc ($\omega_0 t$)) is used with the cut off frequency set to where the desired highest frequency is.

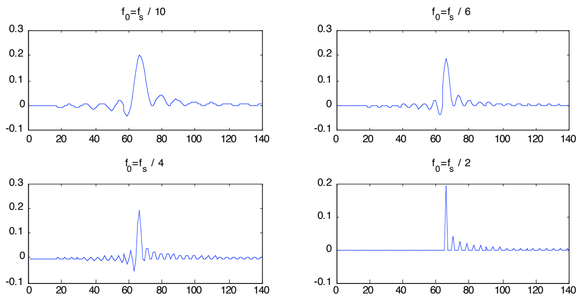

Figure 5 Demonstration of upper frequency limit of FDTD Grid

The above figures show that the grid begins to seriously distort the grid at frequencies approaching $0.2f_s$, this means that in order to model a range specific of frequencies, with a reasonable degree of accuracy, we would have to sample the grid at $5f_s$ in the case of the audible range this equates to sampling at >200*kHz*, this leads to a huge amount of data and calculation due to the dependency between the spatial and time discretisations, otherwise known as the Courant limit [9]. This states that for a square grid (i.e. *dx*=*dy*)

$$\Delta t \leq \frac{\Delta x}{c\sqrt{2}} \tag{12}$$

This shows the direct proportionality between Δx and Δt, thus the smaller Δt becomes the proportionally smaller Δx must become as well, thus in addition to more time steps being necessary as ($\Delta t = 1/f_s$) there are also more spatial points required.

3. IMPLEMEMTATION OF SPECTRAL METHODS IN CUDA

3.1. Theory

The most basic way of thinking about the spectral method used in this case is that rather than calculating the derivative of the velocity or pressure from the adjoining values one can replace this with the appropriate value from a differentiation matrix created from the spectral differentiation. This has the advantage of rather than evaluating just two points in order to calculate the derivative at that point, all of the points in the data set are being evaluated producing the theoretically most accurate derivative possible with the N points. In one dimension the frequency domain representation of the differentiation matrix takes the form:

$$w(i) = \begin{cases} i < \frac{N}{2} & y = i \\ i = \frac{N}{2} & y = 0 \\ i > \frac{N}{2} & y = i - N \end{cases} \qquad i = 0,1,2,\ldots,N-1 \tag{13}$$

This equation applied to when N=30 is shown in figure 6.

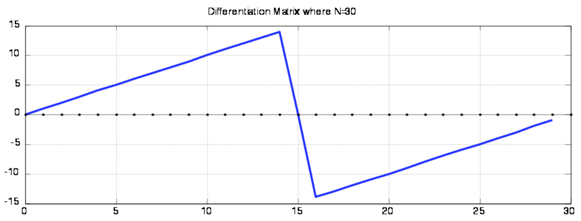

Figure 6 Differentiation Matrix for 1D spectral differentiation.

This impulse response of the perfect differentiator is then multiplied by the result of the Fourier Transform of the input ‘signal’ to form the differential of the input. This can then be used instead of the centre differences in the FDTD equations (5) and (6) in order to achieve a more accurate estimation of the state of the FDTD grid at that point:

$$p\left(x_{i+\frac{1}{2}}, t_{n+\frac{1}{2}}\right) = p\left(x_{i+\frac{1}{2}}, t_{n-\frac{1}{2}}\right)\left(\frac{1-\alpha_i}{1+\alpha_i}\right) - \rho_0 c \frac{\Delta t}{\Delta x}\left(\frac{1}{1+\alpha_i}\right) w_u(x_i) \tag{14}$$

$$u(x_i, t_n) = p(x_i, t_{n-1})\left(\frac{1-\alpha_i}{1+\alpha_i}\right) - \frac{1}{\rho_0}\frac{\Delta t}{\Delta x}\left(\frac{1}{1+\alpha_i}\right) w_p\left(x_{i+\frac{1}{2}}\right) \tag{15}$$

In order to validate the differentiation method, the derivative of a sine wave was calculated using this method, the results are shown in figure 7 plotted against the perfect derivative cos(*x*):

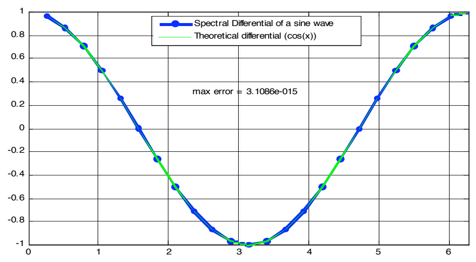

Figure 7 Graph showing the calculated spectral derivative and the theoretical perfect derivative.

This calculation was performed in MATLAB as opposed to in CUDA, therefore it has been performed with double precision floating point numbers as opposed to if the calculation was performed in CUDA it would have been performed in single precision floating point numbers. The error shown on the above plot is somewhere close to the limit of the accuracy of the floating-point algebra. This is phenomenally accurate differentiation more accurate than could have been achieved by way of centre differences. However the downfall with spectral methods is the accuracy of the result depends hugely on the smoothness of the function. For example see the spectral differentiation of a triangular window shown in figure 8.

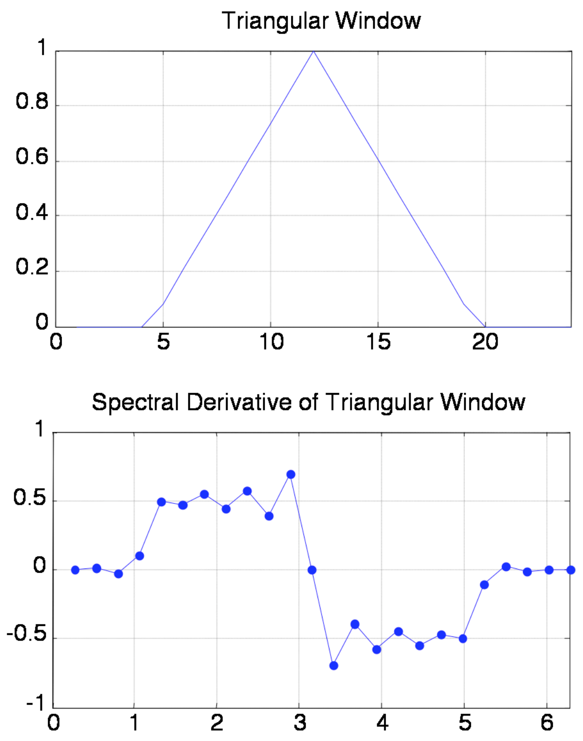

Figure 8 Graphs showing a triangular window and its spectral derivative (Based on Illustration in [5])

3.2. Considerations in CUDA

The implementation in CUDA is very similar in general to that for the more conventional method of performing FDTD calculations. In that there will be a **for** loop to iterate through the time steps then a set of CUDA kernels for both pressure and velocity.

The major difference is the addition of the FFTs. Rather than the author writing a bespoke FFT, the NVIDIA CUFFT library was used. This is a very general purpose FFT algorithm and as such is potentially not the most efficient FFT that could be used for this purpose, however the CUFFT library has been shown by benchmark tests to be more efficient for large transform sizes than the ubiquitous FFTW [6] The reason that the CUFFT has no benefit over small transform sizes is simply due to the fact that CPUs have larger registers and therefore can store the necessary parts of the problem more locally to the core than in the somewhat limited registers of a GPU core.

The arrays used in this type of calculation are potentially very large, especially when the possibility of 3D simulations is considered, therefore it was considered prudent to use the CUFFT library and accept the potential lack of speed on small scale simulations.

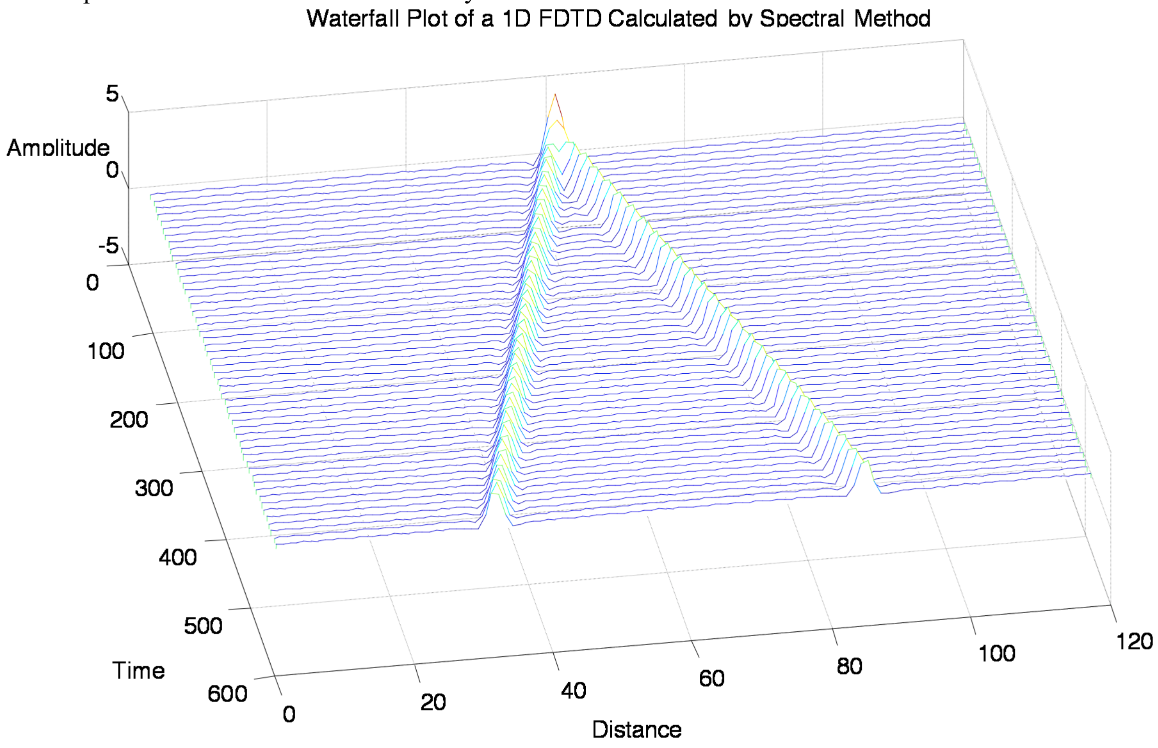

Figure 9 Waterfall plot of 1D FDTD calculated with Spectral Methods

3.3. Results

Figure 9 shows a lovely wave propagating across the grid with minimal ripples. This suggests that a 2D spectral FDTD would be more accurate and have fewer errors than the standard centre difference method of calculating FDTD.

4. CONCLUSIONS

4.1. Finite Difference Time Domain on GPGPU

Presented is a proof of concept that Finite Difference Time Domain calculations can be a carried out with ease on a Graphics Processing Unit thanks to the advent of CUDA. These calculations, although not always as accurate perhaps as their equivalent on a CPU due to the lack of double precision support (although this point is irrelevant for the latest GPUs that do support double precision), the calculations are carried out very well and the power of the method is in no way compromised.

4.2. Spectral Methods

Spectral methods are a phenomenally powerful way of solving partial differential equations. The results obtained in this, albeit, limited investigation are very impressive and the potential for this method is impressive. The power of this method definitely warrants further work.

5. FURTHER WORK

5.1. Spectral Methods

The further work required is to expand the spectral method into 2 or even 3 dimensions and to test that the high frequency limit is, as theorised, less restrictive than for the more general way to calculate FDTD.

5.2. GPGPU

The limited time available for this project meant that the decision was taken to take no time optimising the CUDA code and testing the speed at which the code executes. This is defiantly a project for further investigation as the code used to execute here was laid out in the most intuitive way as opposed to the potentially fastest and most efficient way.

6. REFERENCES

[1] K. Yee, *"Numerical solutions of initial boundary value problems involving Maxwell's equations in isotropic media,"* IEEE Transactions on Antennas and Propagation., vol. AP-14, pp. 302–307, 1966.

[2] A. Taflove, *"Application of the finite-difference time-domain method to sinusoidal steady-state electromagnetic penetration problems,"* IEEE Transactions on Electromagnetic Compatibility, vol. 22, pp. 191-202, Aug. 1980.

[3] J.G. Meloney and K.E. Cummings, *"Adaption of FDTD techniques to acoustic modelling",* 11th Annual Review Prog Applied Computational Electromagnetics, Vol.2, 724 (1995)

[4] Ostashev, V. E. & Wilson, D. K. & Liu, L. & Aldridge D. F. & Symons, N. P. & Marlin, D.: 2005 .*Equations for Finite-Difference, Time-Domain Simulation of Sound Propagation in Moving inhomogeneous Media and Numerical Implementation*., J. Acoust. Soc. Am. **117** (2).

[5] L. N. Trefethen, *Spectral Methods in MATLAB*, SIAM,Philadelphia, 2000.

[6] N. K. Govindaraju, et al. (2008). *`High performance discrete Fourier transforms on graphics processors'. In* SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, pp. 1-12, Piscataway, NJ, USA. IEEE Press.

[7] NVIDIA Corporation, *"CUDA Programming Guide"* NVIDA, Santa Clara CA, 2007, text available at www.nvidia.com/cuda.

[8] Hyok Jeong, Ian Drumm, Brian Horner, Yiu Wai Lam, "*The modeling frequency dependent boundary conditions in FDTD simulation of concert hall acoustics",* Proceedings of the 19th ICA, Madrid, Spain. 2007

[9] A. Taflove and S. C. Hagness, *Computational Electrodynamics: The Finite-Difference Time-Domain Method, 2nd ed.* Norwood, MA: Artech House, 2000.

7. APPENDICIES

7.1. 2D FDTD

```
//Include files
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include <cutil.h>
#include <math.h>

//Initialise CUDA
#if __DEVICE_EMULATION__

bool InitCUDA(void){return true;}

#else
bool InitCUDA(void)
{
      int count = 0;
      int i = 0;

      cudaGetDeviceCount(&count);
      if(count == 0) {
            fprintf(stderr, "There is no device.\n");
            return false;
      }

      for(i = 0; i < count; i++) {
            cudaDeviceProp prop;
            if(cudaGetDeviceProperties(&prop, i) == cudaSuccess) {
                  if(prop.major >= 1) {
                        break;
                  }
            }
      }
      if(i == count) {
            fprintf(stderr, "There is no device supporting CUDA.\n");
            return false;
      }
      cudaSetDevice(i);

      printf("CUDA initialized.\n");
      return true;
}

#endif


//Declare pressure and velocity ararys


const int N = 143;





__global__ void CalculateVelocities(int n,float* pDevice,size_t pitch,float* uxDev,float* uyDev,int
xSize,int ySize,int PMLDepth)
{
      //Declare Constants
      float rho0=1.21,c=343,fs=20000;
      float dt=1/fs;
      float dx=dt*c*sqrt(2.0);
      float pConst = rho0*pow(c,2)*(dt/dx);
      float uConst = (1/rho0)*(dt/dx);
      //Set up indexes
      int x = (blockIdx.x*blockDim.x+threadIdx.x)+1;
      int   y = (blockIdx.y*blockDim.y+threadIdx.y)+1;
      float alphaUx2,alphaUy2,alphaP2;


      if (x<PMLDepth)
```

```
              alphaUx2=(1/3.0)*pow(((float)(PMLDepth-x-1)/PMLDepth),3);
        else if (x>xSize+PMLDepth)
              alphaUx2=(1/3.0)*pow((float)(((float)x-(xSize+PMLDepth))/PMLDepth),3);
        else
              alphaUx2=0;

        //Calculate y velocity values for Alpha
        if (y<PMLDepth)
              alphaUy2=(1/3.0)*pow((float)((float)(PMLDepth-y-1)/PMLDepth),3);
        else if (y>ySize+PMLDepth)
              alphaUy2=(1/3.0)*pow((float)(((float)y-(ySize+PMLDepth))/PMLDepth),3);
        else
              alphaUy2=0;

        alphaP2 = (alphaUx2+alphaUy2)/2;

        if (n==1){
              uxDev[y*pitch/4+x]=0;
              uyDev[y*pitch/4+x]=0;}

  if (y==2)
  y=y;
  if((y<N-2&&x<N)&&(y>1&&x>1)){
        //Calculate Velocities
        uxDev[y*pitch/4+x]=uxDev[y*pitch/4+x]*((1.0-alphaUx2)/(1.0+alphaUx2))-
  uConst*(1.0/(1.0+alphaP2))*(pDevice[y*pitch/4+x]-pDevice[y*pitch/4+(x-1)]);
        uyDev[y*pitch/4+x]=uyDev[y*pitch/4+x]*((1.0-alphaUy2)/(1.0+alphaUy2))-
  uConst*(1.0/(1.0+alphaP2))*(pDevice[y*pitch/4+x]-pDevice[(y-1)*pitch/4+x]);

        ////barrier to diffract round
        if ((x<40&&y==40)||(x<40&&y==60)||(x==30&&y<60&&y>40)){
              uxDev[y*pitch/4+x]=0;
              uyDev[y*pitch/4+x]=0;
        }


  }
```

```
}

__global__ void CalculatePressures(int n,float* pDevice,size_t pitch,float* uxDev,float* uyDev,int
xSize,int ySize,int PMLDepth )
{

      float rho0=1.21,c=343,fs=20000;
      float dt=1/fs;
      float dx=dt*c*sqrt(2.0);
      float pConst = rho0*pow(c,2)*(dt/dx);
      float uConst = (1/rho0)*(dt/dx);
      //Set up indexes
      int x = blockIdx.x*blockDim.x+threadIdx.x;
      int   y = blockIdx.y*blockDim.y+threadIdx.y;

      float alphaUx2,alphaUy2,alphaP2;

      if (x<PMLDepth)
            alphaUx2=(1/3.0)*pow(((float)(PMLDepth-x-1)/PMLDepth),3);
      else if (x>xSize+PMLDepth)
            alphaUx2=(1/3.0)*pow((float)(((float)x-(xSize+PMLDepth))/PMLDepth),3);
      else
            alphaUx2=0;

      //Calculate y velocity values for Alpha
      if (y<PMLDepth)
            alphaUy2=(1/3.0)*pow((float)((float)(PMLDepth-y-1)/PMLDepth),3);
      else if (y>ySize+PMLDepth)
            alphaUy2=(1/3.0)*pow((float)(((float)y-(ySize+PMLDepth))/PMLDepth),3);
      else
            alphaUy2=0;

      alphaP2 = (alphaUx2+alphaUy2)/2;

      if (n==1)
            pDevice[y*pitch/4+x]=0;


      if((y<N-2&&x<N)&&(y>1&&x>1))
      {
            //Calculate Pressure Output
      ////Apply Excitation
      pDevice[y*pitch/4+x]=pDevice[y*pitch/4+x]*((1.0-alphaP2)/(1.0+alphaP2))
                                    -(pConst*(1.0/(1.0+alphaUx2))*(uxDev[y*pitch/4+(x+1)]-
uxDev[y*pitch/4+x])
                                    +pConst*(1.0/(1.0+alphaUy2))*(uyDev[(y+1)*pitch/4+x]-
uyDev[y*pitch/4+x]));
      if (x==71&&y==50)
      {
            if ((2.0*3.142*900*n*dt)<3.142)
                  pDevice[y*pitch/4+x]=3*sin(2.0*3.142*900*n*dt);///(2.0*3.142*900*n*dt);
            else
                  pDevice[y*pitch/4+x]=0;
      }
      }




}

extern "C" int Run2DFDTD(float pOutput2[N][N],int timeSteps)
{
      //Declare Constants
      const int PMLDepth = 21;
      const float rho0=1.21,c=343,fs=20000;
      const float dt=1/fs;
      const float dx=dt*c*sqrt(2.0);
      //float pOutput2[N][N];

      const int H=3,W=3;
      const int xSize = 101, ySize=101;

      float alphaP[N][N],alphaUx[N][N],alphaUy[N][N];                    //alpha matricies

      float *pDev,*uxDev,*uyDev;
      size_t pitch;

      //Calculate pressure alpha



//
//alphaP[1][1]=(alphaUx[1][1]+alphaUy[1][1])/2;

#pragma region Memory Allocations
      size_t arraySize = N*sizeof(float);
      cudaMallocPitch((void**)&pDev, &pitch,arraySize, arraySize);
      cudaMallocPitch((void**)&uxDev, &pitch,arraySize, arraySize);
      cudaMallocPitch((void**)&uyDev, &pitch,arraySize, arraySize);
#pragma endregion


      dim3 dimBlock(1,1);
      dim3 dimGrid(N/dimBlock.y, N / dimBlock.x);
      //loop through time steps
      int n = 0;
      for (n=1;n<timeSteps;n++)
      {
            //Invoke kernel to calculate the grid
            CalculateVelocities<<<dimGrid,dimBlock>>>(n,pDev,pitch,uxDev,uyDev,xSize,ySize,PMLDepth);
            CalculatePressures<<<dimGrid,dimBlock>>>(n,pDev,pitch,uxDev,uyDev,xSize,ySize,PMLDepth);

            float devMemOutput[N][N];

            printf("n=%i  Completed!!!\n",n);
            if (n==timeSteps-1)
            {
                  cudaMemcpy2D (devMemOutput, N*sizeof(float), pDev, pitch,
                                                      N*sizeof(float), N*sizeof(float),
cudaMemcpyDeviceToHost);
                  printf("Results Copy Complete!!");
```

```
                        for (int i = 0;i<N;i++)
                        {
                              for (int j=0;j<N;j++)
                              {
                                    pOutput2[i][j]=devMemOutput[i][j];
                              }
                        }
            }
      }


      cudaFree(pDev);
      cudaFree(uxDev);
      cudaFree(uyDev);


      return(1);
}
```

7.2. Spectral FDTD

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include <cutil.h>
#include <cufft.h>
#include <math.h>


const cufftComplex j={0,1};
const float pi = 3.142;
#define CUFFT_FORWARD -1
#define CUFFT_INVERSE 1
const float c = 343.0;
const float fs = 10000.0;
/************************************************************************/
/* Init CUDA                                                            */
/************************************************************************/
#if __DEVICE_EMULATION__

bool InitCUDA(void){return true;}

#else
bool InitCUDA(void)
{
      int count = 0;
      int i = 0;

      cudaGetDeviceCount(&count);
      if(count == 0) {
            fprintf(stderr, "There is no device.\n");
            return false;
      }

      for(i = 0; i < count; i++) {
            cudaDeviceProp prop;
            if(cudaGetDeviceProperties(&prop, i) == cudaSuccess) {
                  if(prop.major >= 1) {
                        break;
                  }
            }
      }
      if(i == count) {
            fprintf(stderr, "There is no device supporting CUDA.\n");
            return false;
      }
      cudaSetDevice(i);

      printf("CUDA initialized.\n");
      return true;
}

#endif
/***************************************************************************/
/* Kernels                                                                 */
/***************************************************************************/
__device__ cufftComplex CMul(cufftComplex z,cufftComplex x)
{
      cufftComplex answer;
      answer.x=z.x*x.x-z.y*x.y;
      answer.y=(z.x+z.y)*(x.x+x.y)-z.x*x.x-z.y*x.y;
      return(answer);
}
__global__ void SetArray(cufftComplex *a,bool isP,int N)
{
      int k = threadIdx.x;
      float h=2*3.142/(float)N;

            //if(isP)
            //    a[k].x=3*(float)exp(-5*pow((h*k-1)-1,2));
            //    //a[k]=fmax(0,1-fabs(h*k-3.142)/2);
            //else
                  a[k].x=0;
```

```
}
__global__ void ZeroComplex(cufftComplex *a)
{
      int k = threadIdx.x;
      a[k].x=0;
      a[k].y=0;
}

__global__ void CalculateWHat(float *diffMat,cufftComplex *aHat, cufftComplex *wHat)
{
      int k = threadIdx.x+1;
      cufftComplex j = {0,1};
      //cast diffmat as a complex number for the multiplication
      cufftComplex temp = {diffMat[k],0};
      //calculate wHat
      wHat[k]=CMul(CMul(j,aHat[k]),temp);
      k=k;
}

__global__ void CalcDiffMat(float *diffMatrix,int N)
{
      //Calculate and write the impulse response of a perfect differentiation matrix of size 1*N
      int k = threadIdx.x;
      if(k<N/2)
            diffMatrix[k]=k;
      if(k==N/2)
            diffMatrix[k]=0;
      if(k>N/2)
            diffMatrix[k]=k-N;



}
```

```
__global__ void CalcP(cufftComplex *p,cufftComplex *uDif,float pConst,int N,int i,int timeSteps,int
PMLDepth)
{
      int k = threadIdx.x;
      if(k==fabs(N/2.0))
            k=k;
      float alpha;
      int gridSize=N-2*PMLDepth;

      if (k<PMLDepth)
            alpha=(1/3.0)*pow(((float)(PMLDepth-k)/PMLDepth),3);
      else if (k>gridSize+PMLDepth)
            alpha=(1/3.0)*pow((float)(((float)k-(gridSize+PMLDepth))/PMLDepth),3);
      else
            alpha=0;


      p[k].x=p[k].x*((1.0-alpha)/(1.0+alpha))-pConst*(1.0/(1.0+alpha))*(uDif[k].x/(3.142*N));


      float pulseWidth=10,pulseCentreX=40,pulseCentreT=5;
      if(i==1&&k>(N/2-pulseWidth/2)&&k<(N/2+pulseWidth/2))
            //p[k].x=1;
            p[k].x=p[k].x+(1-(1.93*cos(2*pi*((k-(N/2-pulseWidth/2))/pulseWidth)))
                                  +1.29*cos(4*pi*((k-(N/2-pulseWidth/2))/pulseWidth))
                                  -0.388*cos(6*pi*((k-(N/2-pulseWidth/2))/pulseWidth))
                                  +0.0322*cos(8*pi*((k-(N/2-pulseWidth/2))/pulseWidth)));

}
__global__ void CalcU(cufftComplex *u,cufftComplex *pDif,float uConst,int N,int PMLDepth)
{
      int k = threadIdx.x+1;

      float alpha;
      int gridSize=N-2*PMLDepth;
      //calculate alpha value for this point
      if (k<PMLDepth)
```

```
            alpha=(1/3.0)*pow(((float)(PMLDepth-k)/PMLDepth),3);
      else if (k>gridSize+PMLDepth)
            alpha=(1/3.0)*pow((float)(((float)k-(gridSize+PMLDepth))/PMLDepth),3);
      else
            alpha=0;


      u[k].x=u[k].x*((1.0-alpha)/(1.0+alpha))-uConst*(1.0/(1.0+alpha))*(pDif[k].x/(3.142*N));


}

/**************************************************************************/
/* Main Functions                                                         */
/**************************************************************************/
int main(int argc, char* argv[])
{

      if(!InitCUDA()) {
            return 0;
      }
      unsigned int timer = 0;
      CUT_SAFE_CALL( cutCreateTimer( &timer));
      CUT_SAFE_CALL( cutStartTimer( timer));

      //Declare Physical Constants
      float rho0=1.21;

      float dt=1/(2*fs);      //-|conditions required to run
      float dx=2*dt*c;  //-|simulation at the courant limit

      float pConst = rho0*pow(c,2)*(dt/dx)*dt*c;
      float uConst = (1/rho0)*(dt/dx)*dt*c;

      //Setup Grid
      int PMLDepth = 30;
      float gridWidth = 2,endTime=0.03;
      const int N = fabs(gridWidth/dx)+2*PMLDepth;
      int timeSteps = fabs(endTime/dt);

#pragma region declare and allocate device variables
      float *diffMat;
      cufftComplex *uHat_d,*pHat_d,*p_d,*u_d,*uDifHat_d,*pDifHat_d,*uDif_d,*pDif_d;
      cudaMalloc((void**)&p_d,N*sizeof(cufftComplex));
      cudaMalloc((void**)&u_d,N*sizeof(cufftComplex));
      cudaMalloc((void**)&diffMat,N*sizeof(cufftReal));
      cudaMalloc((void**)&pHat_d,N*sizeof(cufftComplex));
      cudaMalloc((void**)&uHat_d,N*sizeof(cufftComplex));
      cudaMalloc((void**)&pDif_d,N*sizeof(cufftComplex));
      cudaMalloc((void**)&uDif_d,N*sizeof(cufftComplex));
      cudaMalloc((void**)&pDifHat_d,N*sizeof(cufftComplex));
      cudaMalloc((void**)&uDifHat_d,N*sizeof(cufftComplex));
      cufftComplex* p_h=(cufftComplex*)malloc(N*sizeof(cufftComplex));
#pragma endregion

#pragma region Set values of input variables
      dim3 dimBlock(1,1);
      SetArray<<<dimBlock,N>>>(p_d,true,N);
      ZeroComplex<<<dimBlock,N>>>(pHat_d);
      ZeroComplex<<<dimBlock,N>>>(uHat_d);
      ZeroComplex<<<dimBlock,N>>>(pDifHat_d);
      ZeroComplex<<<dimBlock,N>>>(uDifHat_d);
      ZeroComplex<<<dimBlock,N>>>(u_d);
#pragma endregion

      //Set up differentiation matrix
      CalcDiffMat<<<dimBlock,N>>>(diffMat,N);

      FILE* input;
      input = fopen("input.txt","w+");
      for (int n=0;n<N;n++)
            fprintf(input,"%f ",p_d[n]);
      fclose(input);

```

```
#pragma region Set cufft plans
      cufftHandle planR2C,planC2R,planC2C;
      cufftPlan1d(&planR2C,N,CUFFT_R2C,1);
      cufftPlan1d(&planC2R,N,CUFFT_C2R,1);
      cufftPlan1d(&planC2C,N,CUFFT_C2C,1);
#pragma endregion



      //Open file to record output
      FILE* file;
      file=fopen("output.txt","w+");
      //loop through time steps

      for (int i=0;i<timeSteps;i++)
      {
            //Take spectral Derivatives
            cufftExecC2C(planC2C, p_d, pHat_d,CUFFT_FORWARD);
            CalculateWHat<<<dimBlock,N-2>>>(diffMat,pHat_d,pDifHat_d);
            cufftExecC2C(planC2C, pDifHat_d, pDif_d,CUFFT_INVERSE);


//          Calculate new values of p & u
            CalcU<<<dimBlock,N>>>(u_d,pDif_d,uConst,N,PMLDepth);

            cufftExecC2C(planC2C, u_d, uHat_d,CUFFT_FORWARD);
            CalculateWHat<<<dimBlock,N-2>>>(diffMat,uHat_d,uDifHat_d);
            cufftExecC2C(planC2C, uDifHat_d, uDif_d,CUFFT_INVERSE);


            CalcP<<<dimBlock,N-2>>>(p_d,uDif_d,pConst,N,i,timeSteps,PMLDepth);


            //Write out p to output
            cudaMemcpy(p_h,p_d,N*sizeof(cufftComplex),cudaMemcpyDeviceToHost);

            //if(fmod(i,4.0)==0){
                  for (int n=0;n<N;n++)
                  {
                        fprintf(file,"%f  ",p_h[n].x);
                  }
                  fprintf(file,"\n");
             //}
            printf("time step n=%i Complete =D!!!\n",i);
      }
      fclose(file);
      fclose(input);





      CUDA_SAFE_CALL( cudaThreadSynchronize() );
      CUT_SAFE_CALL( cutStopTimer( timer));
      printf("Processing time: %f (ms)\n", cutGetTimerValue( timer));
      CUT_SAFE_CALL( cutDeleteTimer( timer));

      free(p_h);
      cudaFree(p_d);
      cudaFree(u_d);
      cudaFree(diffMat);
      cudaFree(pHat_d);
      cudaFree(uHat_d);
      cudaFree(pDif_d);
      cudaFree(uDif_d);
      cufftDestroy(planC2R);
      cufftDestroy(planR2C);
      CUT_EXIT(argc, argv);

      return 0;
}

```