University of Edinburgh
College of Humanities and Social Sciences
School of Arts, Culture and Environment

MSc in Acoustics and Music Technology
Final Research Project : Dissertation

Computing 3D Finite Difference schemes for acoustics – a CUDA approach.

Craig J. Webb
s0956654
Supervisor – Dr. Stefan Bilbao

19th Aug 2010

Declaration of own work

This sheet must be filled in (each box ticked to show that the condition has been met), signed and dated, and included with all assessments - work will not be marked unless this is done

Name: …… Matriculation Number……

Course/Programme:……

Title of work:……

I confirm that all this work is my own except where indicated, and that I have:

- Clearly referenced/listed all sources as appropriate □
- Referenced and put in inverted commas all quoted text of more than three words (from books, web etc) □
- Given the sources of all pictures, data etc. that are not my own □
- Not made any use of the essay(s) of any other student(s) either past or present □
- Not sought or used the help of any external professional agencies for the work □
- Acknowledged in appropriate places any help that I have received from others (e.g. fellow students, technicians, statisticians, external sources). □
- Complied with any other plagiarism criteria specified in the course literature □
- I understand that any false claim for this work will be penalised in accordance with the University regulations □

Signature……

Date……

Please note: If you need further guidance on plagiarism, you can

1. Consult your course literature, Programme Handbook or course WebCT page.
2. Speak to your course organiser or supervisor
3. Check out **http://www.aaps.ed.ac.uk/regulations/Plagiarism/Intro.htm**

JISC Anti-Plagiarism Software

The university subscribes to JISC plagiarism detection software. In cases of suspected plagiarism, you may be asked to supply an electronic version of your work for testing on this system. It is important, therefore, that you maintain an electronic file of your coursework.

Abstract

This project explores the use of parallel computing on Graphics Processing Units to accelerate the computation of 3D finite difference schemes. GPU computing offers the possibility of large speeds-ups over traditional serial execution. A comparison is made between codes performed in serial in Matlab and C, to CUDA codes executed by the latest FERMI architecture Nvidia Tesla cards. This shows speed-ups of 375 times over Matlab, and 90 times over C, at double precision. Achieving this speed-up allows the computation of realistic room acoustics in a reasonable time frame, using the 3D finite difference method.

Contents

1. Introduction

Physical modeling synthesis has developed since the 1970's as a method to create realistic and controllable models of musical instruments. From Karplus-Strong and digital waveguides, to modal synthesis and hybrid systems, the underlying philosophy is to give the realism and expressive capability that is missing from simple methods such as sample-replay. In seeking to achieve the ultimate in fidelity, direct numerical simulation can be employed to calculate the propagation of sound waves in a system. By creating a mathematical model that accurately describes the behaviour of acoustic waves, one can aim to produce highly realistic simulations.

Finite difference schemes are a method of calculating discrete approximations to wave equation systems. In the time domain, these schemes consist of a set of conditions and an update system, which calculates displacement based on previous time step values. 1D systems can be used to simulate strings, and 2D systems can be used to simulate objects such as plates. However, these are only abstractions from the actual behaviour of sound waves in musical instruments. In reality acoustic waves operate in 3D systems.

In dealing with 3D schemes, we can also go beyond strings and plates and consider acoustic waves propagating in real spaces. Architectural acoustics have long been a major topic of research in both the academic and commercial fields. The simulation of reverberation has developed from mechanical plates and springs, and analogue delays, to the popular use of convolution techniques in digital signal processing. Using 3D finite difference schemes, we can accurately simulate sound travelling through any defined space. Thus, it is possible to simulate reverberation in far greater detail than other methods.

However, there are two major issues when considering such an approach. One is the complexity of defining such physical systems and obtaining the mathematical derivation of the scheme. The second is the computational expense required to actually compute them. Even in 1D or 2D, the schemes can be computationally heavy. In 3D, the computation becomes immense.

The purpose of this project is to investigate the use of parallel computation for 3D finite difference schemes using Graphics Processing Units (GPU's). The use of GPU's for general purpose computing has become possible in recent years due to advances in the hardware capability and programming environments. CUDA is an architecture developed by Nvidia that enables their graphics cards to be used for general purpose computing as opposed to just calculating pixels on the screen. It also provides a programming model that uses extensions of the C language as the basis for coding programs.

Using this technology, this project seeks to accelerate the computation of 3D finite difference schemes to simulate room acoustics.

2. Background

This project combines two areas of research. Firstly, the study of 3D finite difference schemes for creating simulations of room acoustics, and secondly the use of GPU computing to accelerate computation. There are four sections in this background chapter. A literature review gives an outline of the current research being carried out in finite difference methods for room acoustics, as well as examples of the use of CUDA for GPU computing. An overview of finite differences scheme is given, along side a basic description of parallel computing using GPU's and CUDA.

2.1 Literature Review

Finite difference schemes have been used as a numerical tool since the 1920's. As such, there is a large volume of literature available on the subject. It has applications in many fields, including electromagnetism, seismology, as well as acoustics. Similarly, the study of architectural acoustics has a long and well-studied history.

In terms of acoustic modeling, current research looks at two different approaches. Firstly we have the numerical methods, such as finite difference time domain (FDTD) and digital waveguide mesh. In the area of FDTD, Kowalczyk and Van Walstijn [16] have presented work using compact, explicit 3D schemes that focus on boundary conditions and modeling of the medium. Southern, Murphy and Wells [17] have shown auralisation techniques that enable walk-throughs of acoustic environments. This uses 3D FDTD to calculate multiple impulse responses and convolution for rendering. In a similar vein, Raghuvanshi, Narain and Lin [13] have used rectangular decomposition to accelerate the computation of 3D FDTD, exploiting the analytical solution of the wave equation in a rectangular domain.

Aside from FDTD, acoustic modeling can be simulated using a digital signal processing methodology. Savoija, Karjalainen and Takala [14] have presented the use of digital waveguide mesh structures that use nodes connected by individual waveguides of unity delay.

Secondly, there is the raytracing approach. This seeks to use 3D graphics rendering technology, but applied to sound waves instead of light. Rober, Kaminski and Masuch [15] have used raytracing techniques to render auralisations of virtual scenes. By approximating sound waves as rays, they then calculate the reflections over all objects and boundaries in the virtual space. Drumm [21] uses a similar approach, known as adaptive beam tracing. However, whilst these systems accurately model reflections, they do not allow for the diffraction effect of sound waves. The wavelength of sound is a huge order of magnitude greater then that of light, and the diffraction effect is just as pervasive as reflection for sound propagating in a physical space such as a room. This is the advantage of numerical methods, which seek to model both physical characteristics.

In terms of CUDA and 3D finite difference schemes, most research looks at the application of GPU's to accelerate schemes for electromagnetics and seismology. Since the release of CUDA in 2007, many groups have studied its application to solving Maxwell's equations using the Yee formulation of FDTD. Adams et al. [18] and Liuge et al. [19] have all shown acceleration of computation using CUDA. The work of Micikevicius [11] in the realm of seismology has recently been included in the SDK for CUDA as an example project. This uses 8^{th} and 10^{th} order stencils for the 3D FDTD scheme, so is more complex than that considered here. The focus of the example project is the use of shared memory for optimisation, which will be discussed later in this report.

The parallelization of finite difference schemes is not restricted to GPU implementations. Motuk, Woods and Bilbao [20] presented the parallelization of 2D schemes for physical modeling synthesis using FPGA's, and research into this approach is ongoing.

2.2 Finite Difference schemes

Finite difference schemes can be used to provide discrete approximations to the partial differential equations that describe the propagation of acoustic waves. For the purpose of this project we will not be concerned with the mathematical derivations of such schemes, but rather the mechanics of how they are computed. Finite difference schemes are iterative systems. Starting from a defined initial state, the system recursively calculates new states based on previously calculated values. In terms of acoustic waves in the time domain, the system is updated over time steps. The time step, *k*, is the inverse of the sampling rate of the system. So, in the case of a sampling rate of 44.1kHz, the system updates 44,100 times for every second of audio produced.

The system describes wave propagation as discrete displacement values over a dimensional space using a grid. A 1D scheme describing a string will use a 1D array representing the length of the string. Each value of the array is a displacement value at that point on the string. A 2D scheme will use a 2D grid of points, representing displacements over a surface. A 3D scheme will use a 3D grid representing displacements in 3D space. The grid spacing is chosen for numerical stability, and is related to the sample rate.

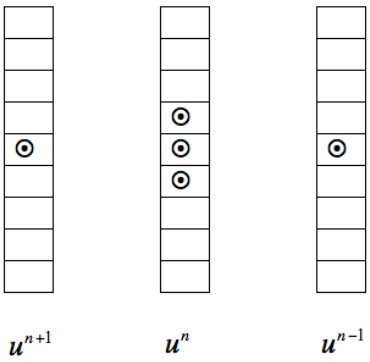
In order to reference previously calculated values, three grids are used. u^{n+1} for the new state of the scheme, u^n for the state one time step ago, and u^{n-1} for the state two time steps ago. At every time step, we now calculate the value of each point of the grid based on an *update equation*. For the simplest 1D wave equation, the general form is :

$$u_l^{n+1} = Cu_l^n + D(u_{l-1}^n + u_{l+1}^n) - u_l^{n-1}$$

where C and D are constants based on the physical properties of the system.

This makes use of the current point in the grid u^n plus its two neighbours, and the current point in grid u^{n-1}, as shown in the *stencil* pattern here :

Fig 2.1 – 1D grid scheme

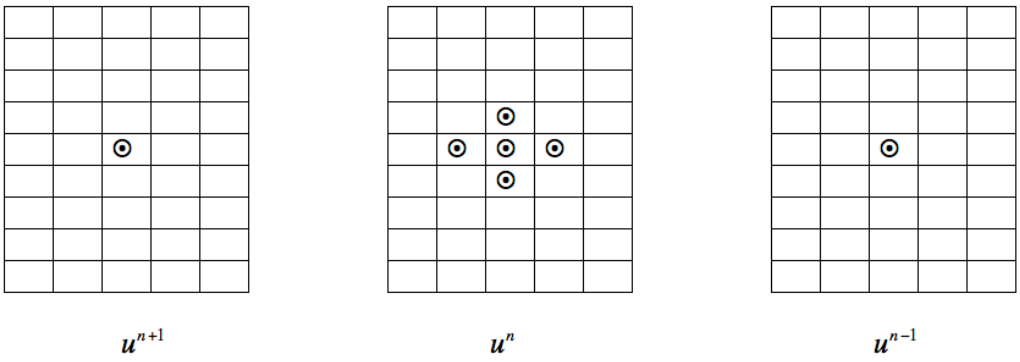


In the case of the 2D wave equation, the update equation becomes :

$$u_{l.m}^{n+1} = Cu_{l.m}^{n} + D(u_{l-1,m}^{n} + u_{l+1,m}^{n} + u_{l,m-1}^{n} + u_{l,m+1}^{n}) - u_{l,m}^{n-1}$$

This references four neighbours from the grid u^n.

Fig 2.2 – 2D grid scheme

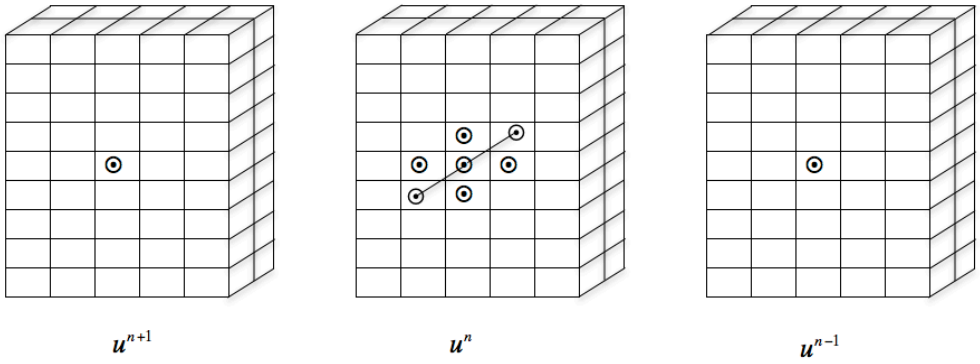


For the 3D case, the update equation becomes :

$$u_{l.m,n}^{n+1} = Cu_{l.m,n}^{n} + D(u_{l-1,m,n}^{n} + u_{l+1,m,n}^{n} + u_{l,m-1,n}^{n} + u_{l,m+1,n}^{n} + u_{l,m,n-1}^{n} + u_{l,m,n+1}^{n}) - u_{l,m,n}^{n-1}$$

which references the six neighbours from the grid u^n.

Fig 2.3 – 3D grid scheme



As well as the update equation, there are two further elements that make up the scheme. Firstly, the initial state and input excitation need to be defined. In the 1D case of a string this could be a pluck or bow. Then there are the boundary conditions, which define the behaviour at the edges of the grid. Algorithmically, all of the above schemes are composed of the same parts:

1. Define the parameters and initialise memory for the grids.
2. Set the initial conditions.
3. Loop from time step sample one to end of duration.
   3.1 Update every point in u^{n+1} based on values from u^{n} and u^{n-1}, applying the boundary conditions at the exterior.
   3.2 Swap the memory pointers around, to shift the grids for the next time step.

The above schemes are in their simplest forms, as they only refer to the nearest neighbours in grid u^{n}. This can be extended to reference further neighbouring points. Taking this to its extreme leads to the basis of *spectral methods*, where every other point in the grid is used in the update equation for a point.

Computationally, the execution time of step 3 of the algorithm, its *kernel*, is based on :

a) The number of samples in the duration of the scheme.
b) The number of floating-point operations performed in the update equation.
c) The number of points in a grid.

In terms of the update equation, there are two multiplications, a subtraction, and 2, 4 or 6 additions depending on the dimensions of the scheme. Whilst this is a simple calculation, it must be performed on every grid point, at every time step.

In a typical simulation of a 1D string of length 1 metre at 44.1kHz, the 1D grids would contain approximately 80 points each. For a 2D scheme, a 1 metre by 1 metre plate would now require grids of 80 x 80 = 6,400 points. For a 3D scheme of a 1m cubic space, the grids would be of size 80 x 80 x 80 = 512,000 points.

So, that is half a million update equations performed 44,100 times for every one second of audio output. This is a large amount of computation, and for only a one cubic metre space. When considering actual room acoustics, the number of update operations is in the order of hundreds or thousands of millions of update equations at every time step. As we will see, trying to perform this magnitude of calculations in Matlab on a standard computer takes a very, very long time indeed. However, one interesting property of the update equation is that it is applied consistently and independently to all of the points on the interior of the grid.

2.3 Parallel Computing

There are two ways to reduce the execution time of a serial program without refactoring. One is to increase the number of operations performed every second, by using a faster processor. Throughout the 1980's and 90's, this was the expected trend as CPU clock speeds rose every year. However, since 2003 this year-on-year increase has dramatically slowed. Around the 4GHz mark is a brick wall, which current processor technologies cannot overcome due to issues such as energy consumption and heat dissipation [1].

The second method is to compute parts of the serial program simultaneously in *parallel*. By dispatching *n* number of threads to *n* number of processor cores, we can potentially speed-up the execution without increasing the clock speed. This speed-up is described by Amdahl's law :

$$Speed - Up = \frac{1}{1 - P}$$

where P is the fraction of the program which can run in parallel.

So, if 50% of the execution of a serial code can be made parallel, then the maximum speed-up is only 2 times. If 90% can be made parallel, then the maximum speed-up is 10 times. At 99%, the speed-up becomes 100 times, given enough cores to process all threads simultaneously.

Returning to the finite difference schemes, the kernel presents a very high level of *data parallelism*. All interior points can be computed independently from one another for a given time step. Also, the kernel takes the majority of the execution time, as the setup overhead is very small. Thus, the finite difference scheme is a good potential candidate for showing speed-up with parallel computation, with respect to the size of the grids. The demands of the update equation mean that thread synchronisation is required every time step, so one loop over the time domain is still required. But potentially all of the grid point updating can be performed in parallel each time step, depending on the boundary condition requirements.

From a hardware perspective, there are many alternatives for actually performing parallel computing. Conceptually we can :

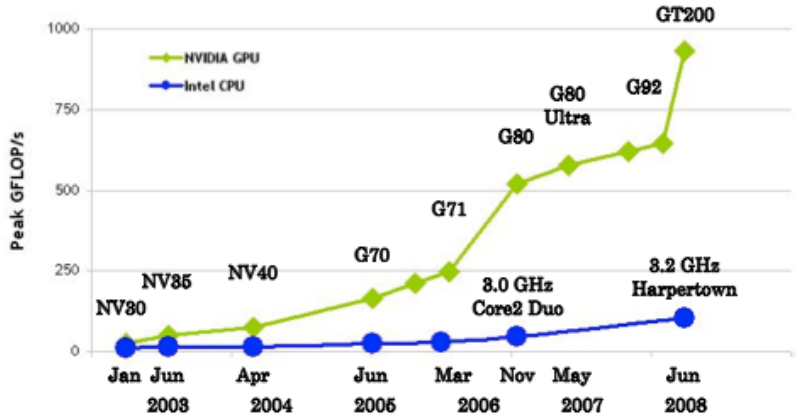
a) Use multiple CPU's in a single computer, as is the case with supercomputers such as the CRAY and BlueGene.
b) Use a number of computers with single CPU's, either physically connected in a single system such as a cluster, or distributed over a networked grid.
c) Use multiple cores on a single CPU, as has become commonplace in standard desktop and workstation computers.

Methods *a* and *b* are mostly confined to scientific laboratories and can be hugely expensive. Method *c* is limited by the current small number of cores on commercial machines, with 2 or 4 cores being the norm. A further alternative, and the focus of this project, is the use of GPU's.

2.4 GPU computing and CUDA

Whilst the multicore architecture of processors such as Intel's quad core i7 seeks to maximise the execution speed of sequential programs, the many-core architecture of GPU's seeks to maximise the throughput of parallel programs [1]. Cards such as the Nvidia Tesla C1060 and GT200 contain 240 cores, each of which is highly multithreaded. This allows for far greater floating-point performance over traditional CPU's, as can been seen here :

Fig 2.4 – GPU vs CPU flop speeds [2]

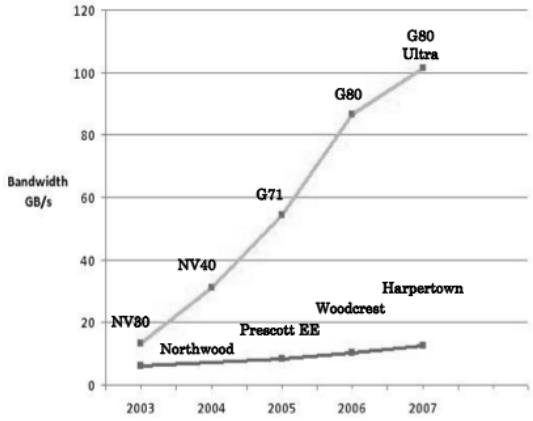


This increased performance is due to the different hardware designs used by GPU's. Whilst multicore CPU's provide large caches and implement full x86 instruction sets per core, GPU's contain many smaller cores dedicated to floating-point throughput.

Fig 2.5 – Architecture of CPU vs GPU [2]

Performance is also increased by the memory bandwidth. Graphics cards operate at around ten times the bandwidth of contemporary CPU chips, with current models capable of speeds in the order of 100 Gb/s.

Fig 2.6 – GPU vs CPU memory bandwidth [1]



This trend for increased floating-point calculation makes GPU computing a viable alternative for performing highly intensive computations that exhibit data parallelism.

2.4.1 Development of GPU computing

The earliest research into general computation on graphics based processors dates back to the late 1970's [10] with machines like the Ikonas [4], and the Pixel Machine [5] in 1989. With the advent of dedicated graphics cards in the late 1990's, a small number of developers experimented with using GPU's to perform scientific calculations. Trendall and Stewart [3] presented an overview of such techniques in 2000. This research became more prominent, and was known as GPGPU, General-Purpose computing on GPU's. However, both hardware and software limitations meant that the usability of such systems was limited. Writing applications was a laborious and complex process. For instance, '*to get results from one section of computation to the next meant writing all results to a pixel frame buffer, and then using that frame buffer as a texture map input to the pixel fragment shader of the next stage of computation [1]'*. This was hardly very efficient from a programming perspective.

In 2007, Nvidia released the first version of its CUDA technology. Along with Apple's OpenCL released in 2008, this allowed the general science community to access the potential of GPU computing. OpenCL is implemented within Apple's latest operating system, Snow Leopard, and can be programmed directly with C, C++, or Objective C [7]. CUDA requires an SDK and uses extensions to the C programming language. It is these advances in the programming environments that have lead to the current level of interest in GPU computing. This project makes use of the CUDA technology, due to the availability of dedicated Tesla units for high performance computing at the University of Edinburgh's EPCC department.

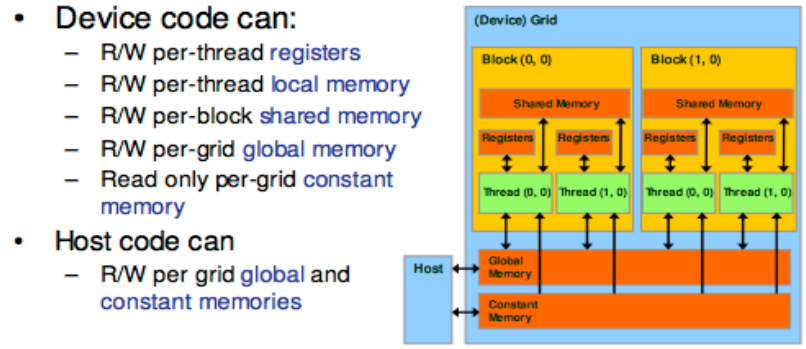
2.4.2 CUDA GPU Architecture

The hardware of CUDA-enabled GPU's consists of an array of highly threaded *streaming multiprocessors* (SM's). Each SM has a number of *streaming processors* (SP's) that share control logic and instruction cache. In the case of the Tesla C1060, there are 30 SM's each with 8 SP's, giving the total of 240 cores. Each SP is massively threaded, and can run thousands of threads per application.

The actual execution of threads starts with *blocks* being assigned to SM's, where each block can contain up to 512 threads. Although we can assign 240 blocks simultaneously most applications will use many thousands of blocks. The runtime system maintains a list of blocks that need to execute and assigns new blocks to SM's as they become available. Once a block of threads is assigned to an SM, it is further divided into 32 thread units called *warps*. The warp is the unit of thread scheduling in SM's at a hardware level. 32 warps can reside in an SM at any one time, giving a total of 1,024 threads. As a block can only accommodate a maximum of 512 threads, this extra headroom allows for latency hiding in reading from global memory.

There are several different types of memory. Each core has registers and shared memory, which are on-chip and very fast access. Then there is constant memory, which is also fast but read only (useful for storing parameter constants). Finally there is global memory, the slowest of the types but the largest in size. The Tesla C1060 has 4Gb of global memory, 65Kb of constant memory, and 16Kb of register and shared memory per SM.

Fig 2.7 – Memory types [1]



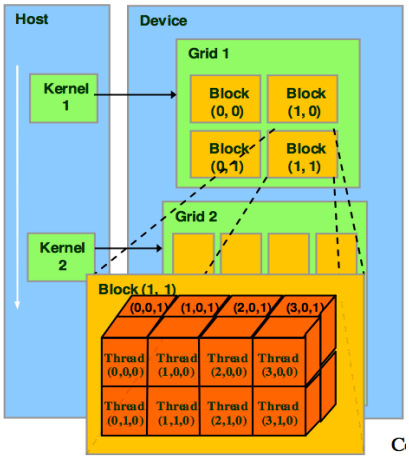
Understanding the hardware architecture of GPU's is far more important than in general programming for CPU's. In most high-level languages for CPU programming, hardware is abstracted and hidden from the programmer entirely, especially with memory management. With CUDA, the hardware architecture is crucial to understanding performance issues. However, the programming model does provide some level of abstraction from dealing directly with SM's and warps.

2.4.3 CUDA Threading model

From a programming perspective, there are three basic operations required to run a kernel on a GPU. Firstly, one must initialise and transfer data from the *host* DRAM to the *device* DRAM (global memory). With the data on the GPU card, a kernel is executed *n* times that launches the required number of *n* total threads on the device. When all threads have completed, data can then be transferred back from the device to the host. The programming model for threads is analogous to a wall of blocks.

Firstly, threads are organised into blocks, which can be up to three dimensional in shape. Within a block, threads are referred to by their `threadIdx.x`, `threadIdx.y` and `threadIdx.z`. A block has a maximum size of 512 threads. When more than 512 threads are required, we form a two dimensional wall of blocks known as a *grid*. Blocks are referred to by their `blockIdx.x` and `blockIdx.y`. So, the grid contains all threads executing the given kernel operation.

Fig 2.8 – Thread blocks and grid formation [1]



The launch code for a kernel takes the form :

```
Test_Kernel<<<dimGrid, dimBlock>>>(parameters…)
```

Where `dimGrid` is a struct of two integers defining the grid size, and `dimBlock` a struct of up to 3 integers defining the block size.

The associated kernel has the form :

```
__global__void Test_Kernel(parameters)
{
      kernel code
}
```

which is executed by every thread.

A key point to note here is that each thread can only use its threadId's and blockId's to reference the relevant parts of data that it needs to operate on. Data sets have to be *tiled* into appropriate sub-sections for this purpose. Whilst 2D data sets fit well into the block and grid abstraction, dealing with 3D data sets is slightly trickier. The maximum size of a 3D block is 512 threads in total, so 3D data sets need to be sub-sectioned into multiple layers. This maximum of 512 is for Nvidia GPU's of *compute capability* 1.x. The latest FERMI architecture cards implement compute capability 2.0, allowing a maximum of 1,024.

2.4.4 Performance limitations of CUDA

It was noted earlier that the memory bandwidth of GPU's is very high compared with typical CPU's, around 100Gb/s. However, when dealing with parallel threading this is still a limiting factor. The global memory access is still the slowest part of the system, and so the *Compute to Global Memory Access Ratio* (CGMA) is an important factor. This is defined as the number of floating-point calculations performed by a kernel for each access to global memory.

An example from Nvidia is as follows. '*The G80 card supports 86.4 Gb/s of global memory access bandwidth. The highest achievable floating-point calculation throughput is limited by the rate at which the input data can be loaded from global memory. With 4 bytes in each single precision float, one can expect to load no more than 21.6 Gb of floats per second. With a CGMA ratio of 1.0, we can execute no more than 21.6 billion floating-point operations per second. Whilst this seems reasonable, it is only a fraction of the 370 gigaflops that is possible on this card*.' [1]

Maximising the CGMA is only one of the factors for generating optimal speeds using CUDA. The list of other optimisation factors recommended by Nvidia is :

'a) Minimise data transfer between the host and device, even if it means running some kernels that do not show performance gains.

b) Ensure global memory accesses are coalesced whenever possible.

c) Minimise the use of global memory, preferring shared memory access.

d) Avoid different execution paths within the same warp.

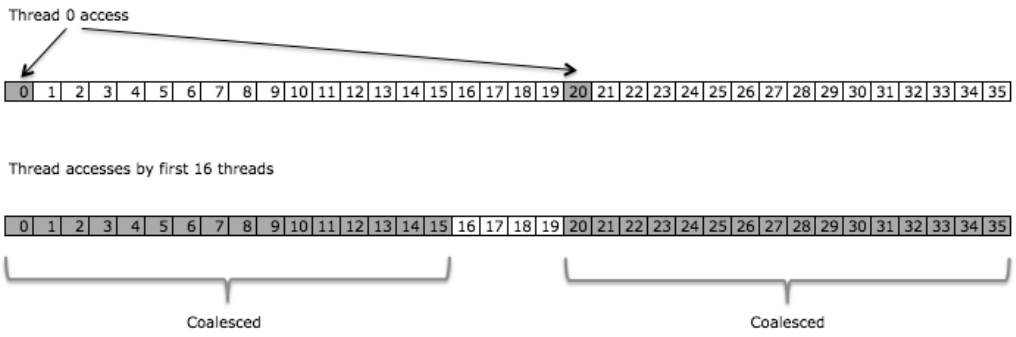
e) To hide latency arising from register dependencies, maintain at least 20% occupancy (on devices with compute capability of 1.2 and above).

f) The number of threads per block should be a multiple of 32, as this provides optimal computing efficiency and facilitates coalescing.' [6]

There are two terms here that require explanation, *memory coalescing* and *occupancy*. To achieve anything close to peak performance from DRAM memory, a kernel must arrange its data accesses so that each request is for a large number of consecutive memory locations. This is vitally important for CUDA architectures with compute capability less then 2.0, as they do not cache DRAM access. By organising the memory access of threads into favourable patterns, we can achieve coalesced access that allows a program to reach its greatest efficiency. To do this, we make use of the fact that threads in a single warp execute the same instruction at a given time. When all threads in a half-warp access data from consecutive global memory locations, the hardware coalesces these into *one single* access to DRAM.

Note that this refers to 'all threads in a half-warp'. This does not mean that individual threads need to only access consecutive memory locations, but rather that each group of 16 ordered threads makes use of blocks of contiguous memory. For example, take a kernel that uses its thread Id to access two memory locations. One is at the location of the Id, and one 20 locations after.

Fig 2.9 – Memory coalescing



Here the first half-warp of threads accesses two contiguous blocks of memory of 16 elements, and so can be loaded in two coalesced transactions rather than 32 individual ones.

Whilst this coalescing is crucial on devices with compute capabilities 1.x, the very latest FERMI based GPU's use compute capability 2.0. This architecture provides some global memory cache on-chip, and so reduces the necessity for data pattern arrangement

for coalescing [8]. The performance benefits of the FERMI architecture are shown throughout the testing results of this project.

Occupancy is another key term in performance criteria. To achieve peak performance it is necessary to get as many cores doing as much work as possible. The occupancy is the ratio of active warps to the maximum possible number of warps for each core. High occupancy rates also help in hiding the latency in accessing global memory.

Nvidia provides an 'Occupancy Calculator' spreadsheet that is a basic tool for profiling kernel performance. The calculator uses compile time information concerning the threads, registers and shared memory usage, viewed by using the 'Xptxas –v' flags.

Fig 2.10 – Compile time profile information

```
[cjwseven@nessgpu1 ~]$ nvcc -arch=sm_13 -Xptxas -v lrooms.cu -o lrooms.x
ptxas info    : Compiling entry function '_Z5inoutPdS_S_iS_'
ptxas info    : Used 10 registers, 40+16 bytes smem, 48 bytes cmem[0]
ptxas info    : Compiling entry function '_Z12UpDateSchemePdS_S_ddd'
ptxas info    : Used 18 registers, 48+16 bytes smem, 48 bytes cmem[0], 20 bytes cmem[1]
```

This information is then used to determine the occupancy rate, and gives further profile data, as shown here :

Fig 2.11 – Profile Data

1.) Select Compute Capability (click):	**1.3**
2.) Enter your resource usage:	
Threads Per Block	256
Registers Per Thread	18
Shared Memory Per Block (bytes)	64
(Don't edit anything below this line)	
3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	**768**
Active Warps per Multiprocessor	**24**
Active Thread Blocks per Multiprocessor	**3**
Occupancy of each Multiprocessor	**75%**
Physical Limits for GPU:	**1.3**
Threads / Warp	32
Warps / Multiprocessor	32
Threads / Multiprocessor	1024
Thread Blocks / Multiprocessor	8
Total # of 32-bit registers / Multiprocessor	16384
Register allocation unit size	512
Shared Memory / Multiprocessor (bytes)	16384
Warp allocation granularity (for register allocation)	2
Allocation Per Thread Block	
Warps	8
Registers	4608
Shared Memory	512
These data are used in computing the occupancy data in blue	
Maximum Thread Blocks Per Multiprocessor	Blocks
Limited by Max Warps / Multiprocessor	4
Limited by Registers / Multiprocessor	3
Limited by Shared Memory / Multiprocessor	32
Thread Block Limit Per Multiprocessor highlighted	**RED**

This data can then be used to optimise code in terms of varying block sizes, and the use of shared memory and registers.

One final performance consideration is the issue of floating-point precision. GPU's are (not surprisingly) primarily designed for performing vector graphics calculations. These types of calculations on polygons are generally performed using single precision floating-point numbers, and hence GPU's are primary designed to achieve peak performance using this precision. Whilst GPU's are able to perform double precision floating-point arithmetic, they do so at far less speed. Again, the latest FERMI architecture GPU's provide far better support for double precision.

Having considered the background to finite difference schemes and parallel computing using CUDA, we are in a position to combine the two and give detail of experimental results. The implementation of this project was carried out in three distinct phases.

1. A simple 2D finite difference scheme.

2. A simple 3D finite difference scheme.

3. An advanced 3D finite difference scheme that simulates room acoustics.

Rather than explain all of the implementation and then the results, each phase is described with its associated results over the next four chapters, beginning with the methodology.

3. Methodology

The purpose of this project is test the use of GPU's for computing finite difference schemes for acoustics. *Testing* here has two components. Firstly, examining the execution times of programs, and secondly the accuracy of their output. In order to compare and contrast results, a consistent methodology was applied at each phase.

1. Create a Matlab script and optimise where possible using vectorization. Run benchmark times.

2. Create a C port, and verify its correctness against the output of the Matlab file. Optimise where possible, and run benchmark times.

3. Create a CUDA port, and verify its correctness against the output of the Matlab file. Also compare to the output of the C port. Optimise and run benchmark times.

The decision to use Matlab for creating initial models and for benchmarking correctness is two-fold. Firstly, the simple scripting language allows for fast prototyping and testing of results, and it is designed for matrix arithmetic. Secondly, it implements the full IEEE 754-2008 standard for double precision floating-point arithmetic, and the accuracy of its results are widely accepted in the scientific community.

The use of a C port as an interim stage is also two-fold. CUDA programming is C based, differing only in the key word extensions that implement the kernels and data transfers. So, much of the C code will port directly to the later CUDA code. Secondly, C is generally considered as a fast tool for running computationally heavy programs. It is therefore interesting to test just how fast C codes run in comparison to Matlab, and ultimately to CUDA. Although it is possible to run both Matlab and C on parallel processors, the benchmarking will be performed on single core processors as serial code.

3.1 Testing variables

Each of the phases used a standard test model. As the parallelization in CUDA is over individual time steps, variations in grid size were used as the independent variable for testing. The dependent variable outcome of each test was execution time. This was taken as the difference between two clock statements. One placed at the start of the code, and the other at the end of the kernel (including the device to host transfer in CUDA). Thus, execution time represents the time taken to complete the entire process of the simulation. As a matter of interest, further clock statements were also used to measure the setup times in CUDA.

Every simulation used a sample rate of 44.1kHz, and had a duration of 44,100 time steps, producing one second of audio output.

3.2 Inputs and outputs

Each scheme was tested twice, firstly using an impulse, and secondly using a short vocal sample of ¾ second duration. Both sets of outputs were used for correctness testing. For the execution times, only the vocal sample was used.

The inputs were applied at a source position in the grid, and an output taken from a read position some distance away. Correctness tests analysed the output from a single position on the grid, not the grid as a whole.

3.3 Benchmarking and hardware

The hardware used for both the Matlab and C port tests was a single core of a 2GHz Intel processor with 2Gb of DRAM. Whilst this is by far not the fastest processor available today, it does represent a common configuration found in many laptop and desktop machines. The Matlab was run in Windows Vista OS, whilst the C codes were executed from within the Xcode environment in Apple OSX 10.6 ‘Snow Leopard’. Both were performed on the same computer.

The CUDA codes were initially tested on a Tesla C1060. This is a single GPU card with a compute capability of 1.3 and 4Gb of DRAM. Towards the end of the project, the very latest FERMI architecture cards became available and testing was also performed on a Tesla C2050. This is a single GPU card with compute capability of 2.0 and 3Gb of DRAM.

3.4 Testing correctness

Correctness testing was used to examine differences in the outputs from the C and CUDA ports to the base Matlab output. These were considered in terms of sample-by-sample differences. The magnitude of the input was adjusted to give a normalised output waveform with maximum amplitudes -1 ~ 1. Two tests were performed at each phase, for a single grid size.

1. The straight A - B differences between each sample of the outputs of two schemes.

2. The normalised differences between each sample of the outputs of two schemes. This was calculated as (A-B) / (A+B) for each pair.

For the most part, all tests were performed using double precision floating-point numbers, with the exception of the following.

3.5 Floating-point precision

At the commencement of the project, the only GPU available for testing was the Tesla C1060. This has a compute capability of 1.3, meaning that support for double precision floating-point calculation is very poor. It was known that the next generation of FERMI architecture Nvidia GPU's were arriving soon, and would have far superior double precision support. It was indicated that the likely execution times of double precision code on these FERMI cards would by comparable to the times of single precision codes on the older C1060. Thus, some testing was performed using single precision floats.

Though the initial purpose of this was to obtain comparative timing results, the correctness testing results proved interesting in themselves. The propagating errors

induced by using single precision lead to markedly different results compared to double precision outputs. This is examined further in section 4.6.

In terms of the timing results shown here, it was not considered necessary to run full testing at single precision, as the FERMI card became available to use. Thus, all timing data is given for double precision codes, to show a comparison of the double precision support of the new FERMI card compared to the older versions.

4. Phase 1 – Basic 2D scheme

Whilst the objective of the project is to simulate 3D room acoustics, for the first phase of implementation a simple 2D finite difference scheme was used. This was to understand the issues involved in programming the schemes in the different environments, before making the jump to 3D.

4.1 Update scheme

The following update scheme was coded during this phase :

$$u_{l,m}^{n+1} = coeff\,(u_{l-1,m}^{n} + u_{l+1,m}^{n} + u_{l,m-1}^{n} + u_{l,m+1}^{n}) - u_{l,m}^{n-1}$$

where *coeff = 0.32*.

Fixed boundary conditions were used. This scheme represents the very simplest form in 2D, using just a single coefficient, the four neighbours from u^n and the point from u^{n-1}.

4.2 Matlab

The kernel in Matlab is straightforward, with one loop over the time steps. The grid point updating is performed in a single vectorized statement.

```
for n = 1:NF
    % update scheme
    u(2:Nx-1,2:Ny-1)=coeff*(u1(1:Nx-2,2:Ny-1)
                           +u1(3:Nx,2:Ny-1)
                           +u1(2:Nx-1,1:Ny-2)
                           +u1(2:Nx-1,3:Ny))
                      -u2(2:Nx-1,2:Ny-1);

    % read output at listener position
    out(n) = u(rpsy,rpsx);
```

```
    % shift memory pointers
    u2 = u1;
    u1 = u;
end
```

The fixed boundary conditions are implemented by setting a layer of zeros around the edges of the 2D grids. The points in the grid are then only updated from elements two to edge-1, thus picking up the zero boundaries as the neighbouring points.

The declaration of 2D arrays in Matlab uses a standard row, column format using : `u = zeros(Height, Width);` which also initialises all elements to zero.

4.3 C port

The transition from Matlab to C requires some refactoring of the code.

1. The first part of refactoring concerns de-vectorizing. Matlab is designed to use vector arithmetic for efficiency of execution and coding. The C port will need to loop over the grid points at each time step.

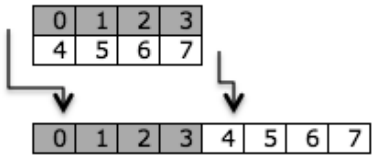
2. Whilst standard array notation of `u[row][col]` can be used in C, the declaration of such arrays does not allow us to test if there is enough memory in the DRAM stack to initialise it. This becomes important when dealing with very large array sizes. Also, CUDA requires that arrays use a single contiguous block of linear memory.

So, rather than declaring a 2D array in C, a single linear block of memory can be used and referenced as if it were a 2D array. The *calloc* function also initialises all elements to zero. This functions returns a null pointer if it cannot allocate a linear block of sufficient size.

The *row-major* format was employed to reference the linear array in a 2D abstraction. Each row of the 2D array is set out end-to-end, and referenced using :

$$Linear\ address = row * 2D\ width + column$$

Fig 4.1 – 2D array decomposition



Thus, data is addressed in code as `u[row*Width+col]`.

The kernel for the C port is :

```
for(n=0;n<NF;n++)
{
   // update scheme
   for(q=1;q<Ny-1;q++){
     qx = q*Nx;
      for(r=1;r<Nx-1;r++){
         u[qx+r] = coeff*(u1[qx-Nx+r]
                         + u1[qx+Nx+r]
                         + u1[qx+r-1]
                         + u1[qx+r+1])
                   - u2[qx+r];
      }
   }

   // read output
   out[n] = u[rpsy*Nx+rpsx];
   // update pointers
   dummy_ptr = u2;
   u2 = u1;
   u1 = u;
   u = dummy_ptr;
}
```

Note that C uses array indexing that starts from position *0* and goes to *end - 1*, unlike Matlab that begins with position *1* and goes to *end*.

The only optimisation used was to remove a *redundant calculation*. By declaring `qx = q*Nx` before the start of the loop over the columns (Nx), this saves six multiplications from within the update equation. Removing redundant calculations within loops can have a significant impact on timings.

4.4 CUDA port

The transition from C to CUDA port requires both additional coding, as well as refactoring of the kernel. Additional code is necessary to initialise memory on the device, and to deal with the transfers of data to the device and back to the host after the simulation has been completed. To remove any data transfers during the simulation, the input and output arrays are both loaded into the global memory and referenced directly from there. The grid arrays are left on the device after the time loop has completed, and just the output is transferred back to the host.

The kernel is refactored from inside of the time loop. Instead of the loops over the rows and columns of the grids, a *threading kernel* is executed as follows :

```
for(n=0;n<NF;n++)
{
   UpDateScheme<<<dimGrid,dimBlock>>>(parameters);
   cudaThreadSynchronize();

   inout<<<1,1>>>

   // update pointers
   dummy_ptr = u2_d;
   u2_d = u1_d;
   u1_d = u_d;
   u_d = dummy_ptr;

}
```

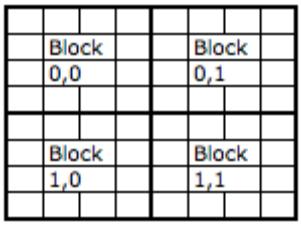
The call to `UpDateScheme<<< >>>` starts *n* number of threads of the following code, where *n* is the total number of threads in the grid.

```
__global__ void UpDateScheme(parameters)
{
   int q = blockIdx.y * bh + threadIdx.y;
   int r = blockIdx.x * bw + threadIdx.x;

   if(!(q==0||q==Nx-1||r==0||r==Nx-1))
   {
      u[q*Nx+r] = coeff*(u1[(q-1)*Nx+r]
                       + u1[(q+1)*Nx+r]
                       + u1[q*Nx+(r-1)]
                       + u1[q*Nx+(r+1)])
                  -u2[q*Nx+r];
   }
}
```

There are several differences in the coding approach of the threading kernel compared to the inner loops of the C port.

1. The threading model in CUDA arranges threads into blocks of a maximum of 512 (or 1,024 for FERMI). For simulating room acoustics, we will require grids containing many millions of points. So, each of the three grids here is *tiled* into a series of 2D blocks.

Fig 4.2 – Grid blocking



The occupancy calculator showed a 16 x 16 = 256 block size gives the optimum occupancy rate in this case (100%). In order to use the same row-major data arrangement as the C port, each thread needs to find its row and column in the 2D grid. This is the purpose of lines two and three of the threading kernel, which calculates the row and column based on the threadId and blockId.

2. In the C port the inner loops do not operate over the outer edges of the grids, to implement the fixed boundary conditions. In CUDA, the threading and blocking indexes start from zero and are in 2D, so an IF statement is required to ensure that the edges of the grid are not updated. Whilst this is not optimal in terms of threading behaviour, it is still the case that the majority of the threads have identical paths.

3. After the threads are synchronised in the time loop, a single thread is executed that updates the input and output arrays. This is necessary, as host code does not have direct access to the grids on the device.

4.5 Test Results

With a baseline Matlab and a C and CUDA port completed, we are now in a position to test the outputs for correctness and execution times.

4.5.1 Correctness testing

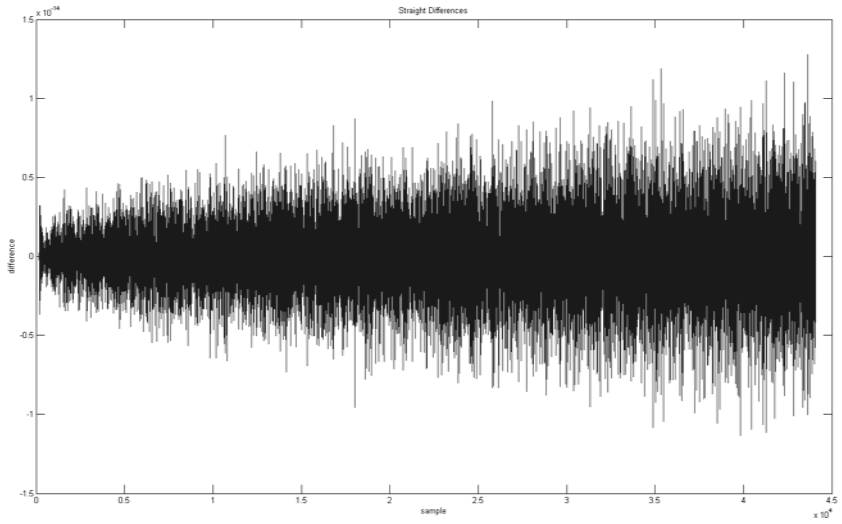
Eight output files were produced for the correctness testing of the basic 2D scheme, all using double precision floats. For both an impulse input, and the vocal sample input, there are a :

a) Matlab file.
b) C file.
c) CUDA file run on the original Tesla (C1060).
d) CUDA file run on the FERMI Tesla (C2050).

Each was produced for a grid size of 256 x 256 = 65,536 points. Comparing the Matlab files to the C files, there were absolutely no differences between the outputs, down to the last significant digit. This was also true of the outputs of the two Tesla's. The FERMI card produced exactly the same output as the original Tesla.

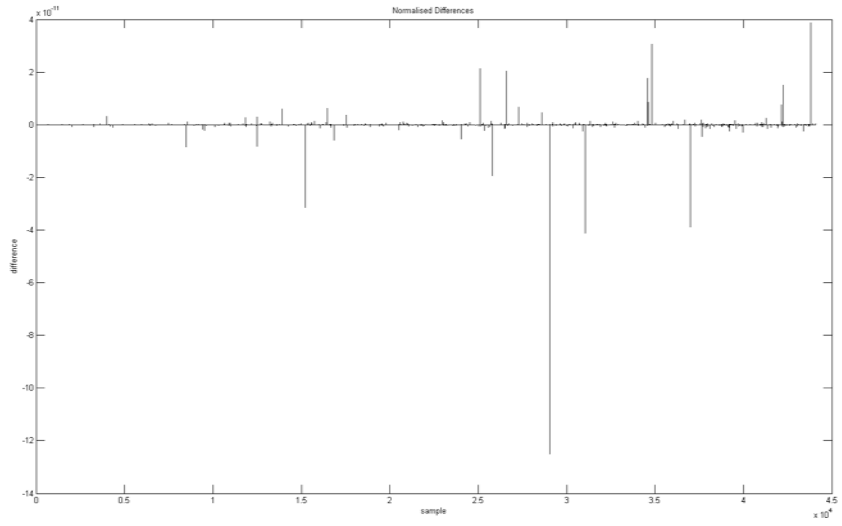
This is not the case when comparing the outputs of the Tesla's to the Matlab (or C as they are identical). Below is a plot of the straight A-B differences between Matlab and the Tesla for the impulse input :

Fig 4.3 – Straight differences between Matlab and Tesla for impulse input



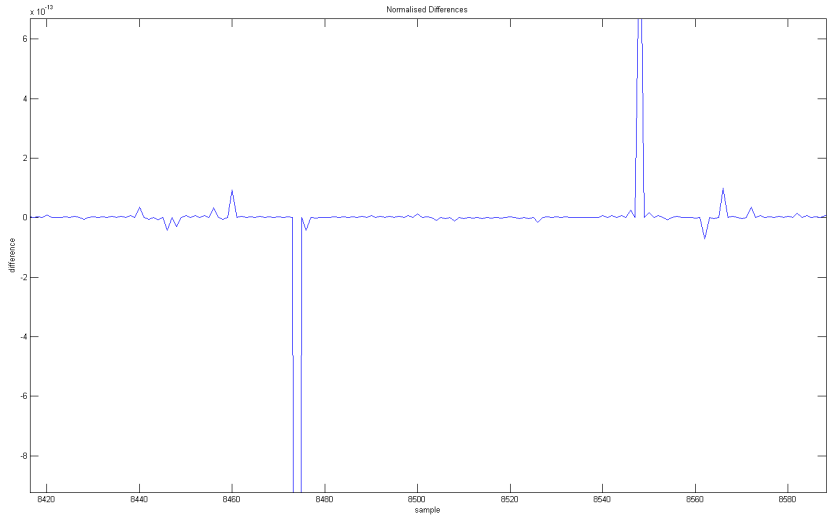
The differences are in the order of E-14. A more revealing graph is that of the normalised differences, taken as (A-B) / (A+B), as shown here :

Fig 4.4 – Normalised differences between Matlab and Tesla for impulse input



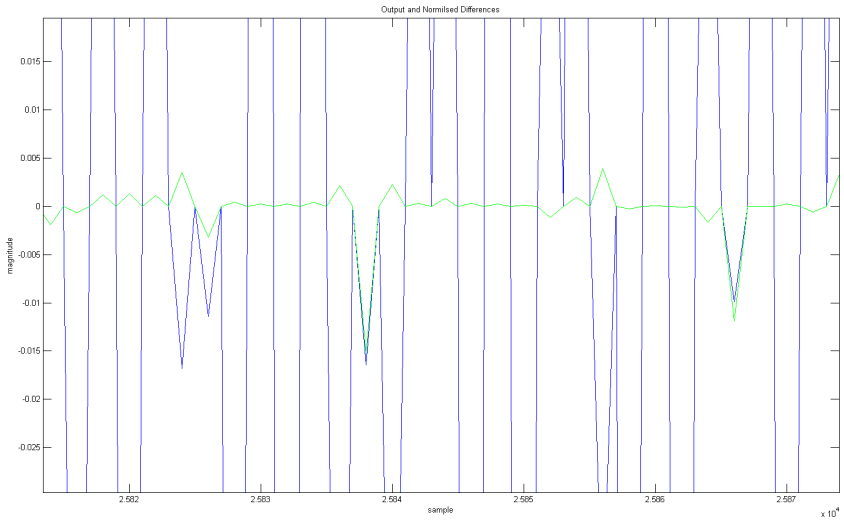
This shows a small number of spikes in the order of E-10, against a background of differences in the order E-14. This is more easily seen on a magnified view here :

Fig 4.5 – Magnified view of Fig 4.4



These spikes last for a single sample, and then return to the background level. Over-laying the output waveform of Matlab onto the normalised differences multiplied by E14 produced the following plot. The output waveform is in blue, and differences in green.

Fig 4.6 – Over-lay plot of output and normalised differences

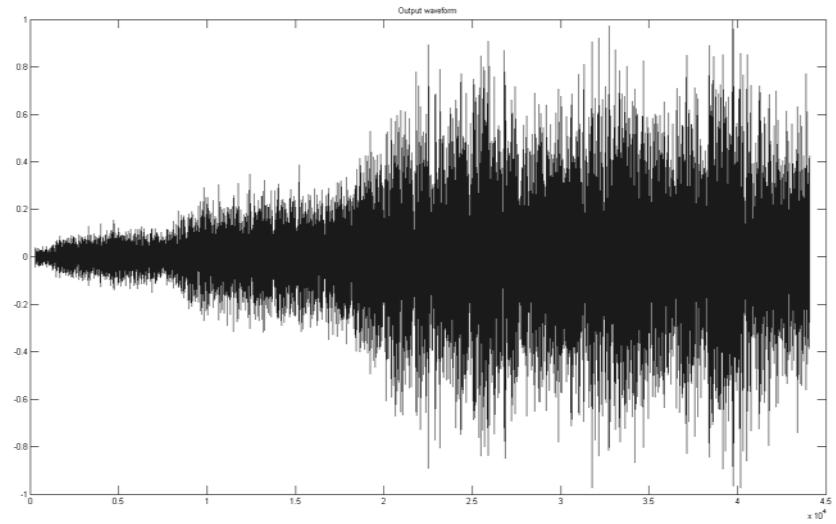


This shows the normalised differences at times mirroring the output signal, and at other times being reversed.

The same behaviour was found when analysing the outputs from the CUDA files using the vocal sample, compared to the Matlab output. Again, the Matlab and C were

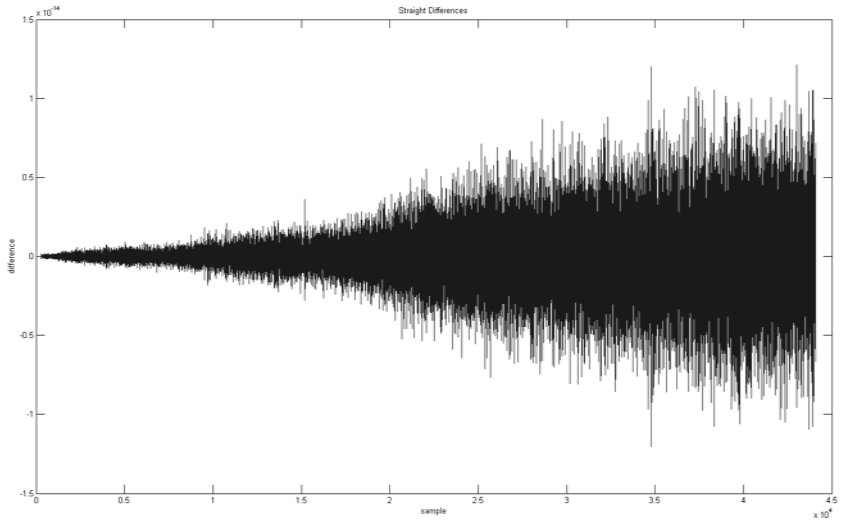
identical, as were both of the CUDA files. The following plot shows the output waveform from Matlab.

Fig 4.7 – Output waveform for vocal input



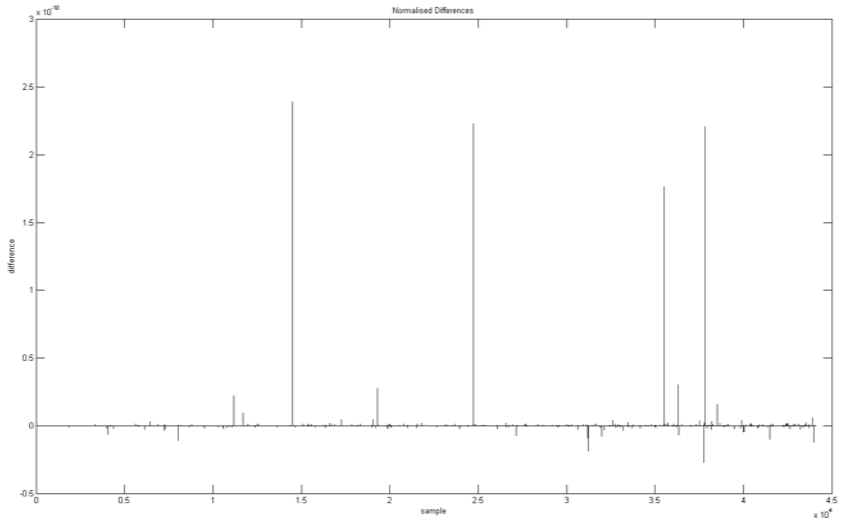
The straight A-B differences between Matlab and the Tesla outputs are again in the order of E-14.

Fig 4.8 – Straight differences between Matlab and Tesla for vocal input



The normalised differences show the same spike behaviour as per the impulse files. There are four significant spikes in the order of E-10, against a background of variations in the order of E-14.

Fig 4.9 – Normalised differences between Matlab and Tesla for vocal input



Whilst we may expect some differences between Matlab, C and CUDA at a machine level of E-18, it appears that errors are propagating above this level. It is interesting that Matlab and C both produce identical output, while the Tesla cards produce the variations. The exact differences shown in the CUDA files are *reproducible*. The outputs from both the original Tesla and the FERMI Tesla were identical down to last significant digit.

The background differences of E-14 may be due hardware implementation of floating-point arithmetic, but it is not clear where the spikes at E-10 come from, and why they themselves do not propagate into subsequent samples. Further investigation is required into the nature of these results, and also how they vary depending on the grid size.

What is clear from this testing is that both the C and CUDA code are performing correctly in terms of implementing the algorithm. Any errors in the codes would produce differences in the outputs of much higher magnitude.

4.5.2 Timing results

The correctness testing is interesting in it own right, but the exciting part is the timing results. The following table shows the execution times for the 2D scheme running at double precision, over various grid sizes.

Fig 4.10 – 2D timing results

Array Width	Grid Points	Matlab	C	Tesla	Tesla FERMI
256	65,536	213 (3.5m)	51	3.5	3.6
512	262,144	1,034 (17m)	210 (3.5m)	9.4	6.7
1,024	1,048,576	4,320 (72m)	848 (14m)	32.6	17.3
2,048	4,194,304	17,302 (288m)	3,408 (57m)	125.0	58.1

Times in seconds (minutes)

Viewing this as speed-up times over Matlab gives :

Fig 4.11 – Speed-ups over Matlab, 2D scheme

Array Width	Grid Points	C	Tesla	Tesla FERMI
256	65,536	4.2	60.9	59.2
512	262,144	4.9	110.0	154.3
1,024	1,048,576	5.1	132.5	249.7
2,048	4,194,304	5.2	138.4	297.8

So, both the Matlab and C show nearly linear increases in execution time. For every four-times increase in the grid size, their times increase by a factor of four. The C code shows a five times speed-up over Matlab.

The Tesla's show super-linear speed-ups. The initial four-fold increases in grid size take only two or three times longer. This increases to 3.5 and 3.8 for larger sizes. Even the older Tesla C1060 shows a 138 times speed-up at 4.2 million grid points. The improvement of the FERMI card is huge, showing a 300 times speed-up over Matlab. This is the difference between nearly 5 hours in Matlab, to 1 minute on the Tesla. The speed-ups over C are around 60 times.

The above testing ceased at 4.2 million as the Matlab testing was taking a very long time. The next step up to 16 million points would take almost 20 hours to run. However, even at this size, the Tesla's are only just getting warmed up.

4.5.3 Taking it to the limit

Section 2.2 showed a rough approximation of the number of grid points required to model 3D room acoustics. Every 1m cubic space requires around half a million points, at a sample rate of 44.1kHz. So, even a small room of 5m x 5m x 4m will require around 50 million grid points. The FERMI Tesla C2050 has 3Gb of DRAM. As the finite difference scheme used three arrays, this allows 1Gb per array. With 8 bytes in a double precision float, the maximum array size will be around 125 million points each, less space for the inputs and outputs (for the advanced 3D scheme, the code was refactored to use just two arrays, see section 6.5.3).

Testing was performed at an array width of 10,592 x 10,592 = 112,190,464 grid points, on both of the Tesla's. As the Matlab times are roughly linear, we can make a (very rough) approximation of the Matlab execution time. The 112 million point test is 26.7 times larger than the last 4.2 million test. So, the Matlab execution time would be around 7,700 minutes. This is 5.4 days…

The Tesla C1060 took 53.1 minutes. The FERMI Tesla C2050 took 26.7 minutes, again a 300 times speed-up over the Matlab, and 60 times over C.

4.6 Issues with floating-point precision

Before moving to the 3D schemes, we digress somewhat to consider the issue of floating-point accuracy. As was mentioned in section 3.5, the original intention was to run single precision codes on the Tesla C1060 to give an approximate idea of the possible timing results using FERMI architecture cards. As the FERMI card actually became available, this became a redundant test. However, single precision accuracy was tested using a C port of a 2D finite scheme that simulates a plate reverb. This allowed a comparison of single to double precision results in a realistic situation.

4.6.1 Floating-point implementation

Computers are very good at doing binary calculations, where as mathematics and physics generally takes place using *real* numbers. The floating-point format is used to

fit the real numbers used in situations such as physical modeling into a binary format that computers can work with. Floating-point numbers take the form : $\pm(1+f).2^e$, where f is the mantissa and e the exponent. The fraction f must satisfy $0 \leq f \leq 1$ and be representable in binary format. The exponent must be an integer value. Any real number that does not meet these conditions must be approximated to fit. The *precision* is given by the number of bits used in the mantissa. The *range* is given by the size of the exponent.

Two types of floating-point numbers are typically used in programming, single precision *floats* and double precision *doubles*. Single precision requires 32 bits (4 bytes) of storage. 24 bits are used for the mantissa, and 8 bits for exponent. This gives an accuracy of around 7 decimal digits. Double precision requires 64 bits (8 bytes) of storage. 53 bits are used for the mantissa, and 11 for the exponent. This gives an accuracy of around 16 decimal digits.

This large difference in the representable numbers has a significant impact on mathematical calculations. Nine decimal digits of information is lost when dealing with single precision, and when considering iterative schemes such as finite difference, this introduces large errors in the results.

4.6.2 Advanced 2D finite difference scheme

To test the degree of error, a 2D scheme was ported to C from a Matlab script provided by Dr. Bilbao. This allowed a comparison of the double precision in Matlab, to the double and single precision of an equivalent C code. The scheme accurately simulates a plate reverb, with coefficients calculated to represent the real physical properties of a plate. Realistic boundaries conditions are also implemented, that allow the pivoting of each edge about at zero point one position inwards. The update scheme itself is :

$$u_{l,m}^{n+1} = bu_{l,m}^{n} - cu_{l,m}^{n-1} + b_1(u_{l+1,m}^{n} + u_{l-1,m}^{n} + u_{l,m+1}^{n} + u_{l,m-1}^{n})$$
$$+b_2(u_{l+1,m+1}^{n} + u_{l+1,m-1}^{n} + u_{l-1,m+1}^{n} + u_{l-1,m-1}^{n})$$
$$+b_3(u_{l+2,m}^{n} + u_{l-2,m}^{n} + u_{l,m+2}^{n} + u_{l,m-2}^{n})$$

This is more complex then the simple 2D scheme considered so far. It makes use of a wider stencil pattern in u^n, and more coefficients.

4.6.3 Correctness testing

The following plots show the normalised differences between Matlab and C.

Fig 4.12 – Normalised difference, Matlab and double precision C

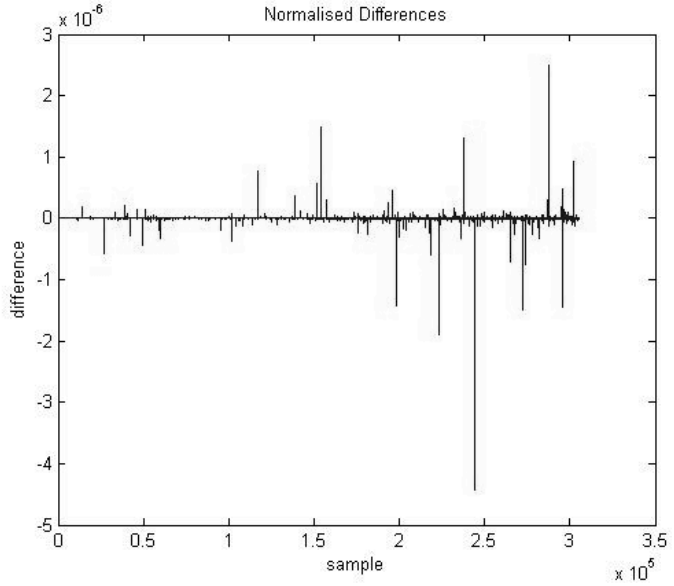
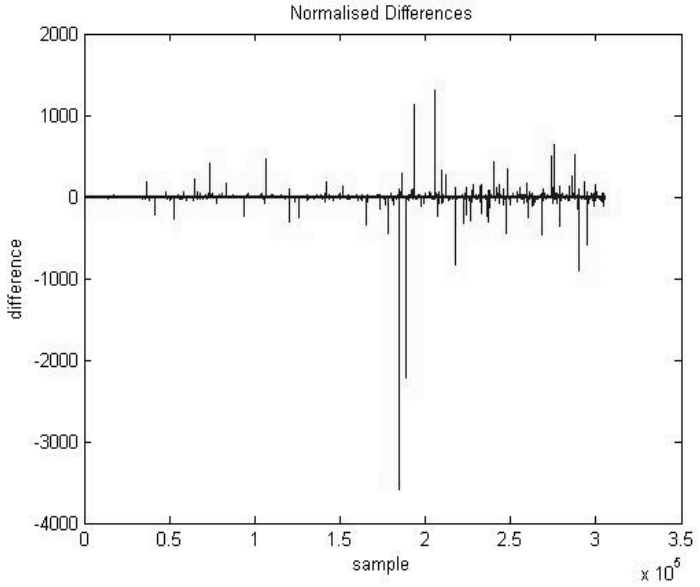


Fig 4.13 – Normalised difference, Matlab and single precision C



Whilst the basic 2D scheme used previously showed no differences at all between Matlab and double precision C, the situation now is different. With double precision there are normalised errors that spike at E-6, with background levels around E-10. It would seem that the use of realistically calculated coefficients, along with the use of addition neighbour points, has introduced differences between Matlab and C. The C code shows the same 'spiking' behaviour as was found with basic 2D scheme running on the Tesla's.

When the code was run using single precision floats for *all* non-integer variables, the resulting differences are now very large. Spikes occur in the order of E3, nine times larger than for doubles. Intuitively this is to be expected, as by using single precision floats we have lost 9 decimal digits of information.

5. Phase 2 – Basic 3D scheme

The second implementation phase concerned the move from 2D arrays to 3D arrays. A simple finite difference scheme was used, as the focus of the work was to understand the issues in dealing with 3D data sets.

5.1 Update scheme

The update equation is an extension of the basic 2D scheme. It simply uses the two extra neighbours from the *z* dimension of u^n.

$$u_{l.m,n}^{n+1} = coeff\,(u_{l-1,m,n}^{n} + u_{l+1,m,n}^{n} + u_{l,m-1,n}^{n} + u_{l,m+1,n}^{n} + u_{l,m,n-1}^{n} + u_{l,m,n+1}^{n}) - u_{l,m,n}^{n-1}$$

Again, fixed boundary conditions were used and the *coeff* is 0.32.

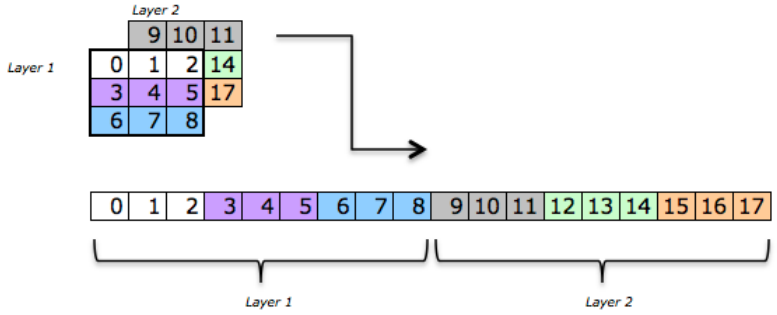
5.2 Matlab

The only change to the Matlab is to address the grid arrays with a third dimension.

5.3 C port

The jump to 3D in C requires more extensions to the code. As the C port is using a single linear block of memory, the 3D array needs to be abstracted to a single dimension. The *z* dimension layers are placed side-by-side, whilst using the 2D row-major format within each layer.

Fig 5.1 – 3D data arrangement



With this configuration, the linear data is accessed using :

*Linear address = (dep*width*height)+(row*width+col)*

The kernel for the C port now appears as :

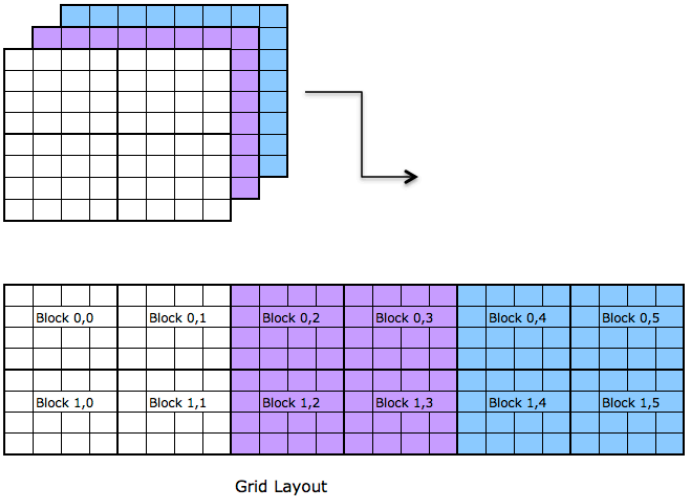
```
for(n=0;n<NF;n++)
	{
		// update u matrix
		for(p=1;p<Nz-1;p++){
			pa = p*area;
			for(q=1;q<Ny-1;q++){
				qx = pa + q*Nx;
				qm = qx-area;
				qa = qx+area;
				qn = qx-Nx;
				qt = qx+Nx;
				for(r=1;r<Nx-1;r++){

					u[qx+r] = coeff*(u1[qm+r]
							+u1[qa+r]+u1[qn+r]
							+u1[qt+r]+u1[qx+r-1]
							+u1[qx+r+1])
						-u2[qx+r];
				}
			}
		}

		// read output
		out[n] = u[(rpsz*area)+(rpsy*Nx+rpsx)];

		// sum in source at srcx,y,z
		u[(srcz*area)+(srcy*Nx+srcx)] += audio[n];

		// update pointers
		dummy_ptr = u2;
		u2 = u1;
		u1 = u;
		u = dummy_ptr;
	}
```

Note the removal of redundant calculations at levels one and two of the three inner loops. This leaves the actual update code with just one or two additions for each data access, and provided a two times speed-up over the code with redundant calculations.

5.4 CUDA port

The CUDA port requires one further conceptual leap. As threads are arranged into a grid of blocks, we require a method of *tiling* the 3D data sets. As thread blocks can be up to three-dimensional, this gives many different possible ways of laying tiles over the grids. In order to keep the thread indexing as simple as possible, each layer of the 3D grid was placed side-by-side into the thread grid, as follows :

Fig 5.2 – 3D blocking



This allowed the use of 2D block sizes, and reduced the complexity of calculating the position of a thread in the 3D data set. The threading kernel is now :

```
__global__ void UpDateScheme()
{
      int q = blockIdx.y * bh + threadIdx.y;
      int colt = blockIdx.x * bw + threadIdx.x;
      int p = colt/Nx;
      int r = colt - (p*Nx);
```

```
        int pa = p*area;
        int qx = pa + q*Nx;
        int qm = qx-area;
        int qa = qx+area;
        int qn = qx-Nx;
        int qt = qx+Nx;

        if(!(q==0||q==Nx-1||r==0||r==Nx-1||p==0||p==Nz-1))
        {
              u[qx+r] = coeff*(u1[qm+r]+u1[qa+r]
              +u1[qn+r]+u1[qt+r]
              +u1[qx+r-1]+u1[qx+r+1])
              -u2[qx+r];
        }
}
```

Lines 2 ~ 5 obtain the layer, row and column of the thread. From there, data access is performed in exactly the same way as for the C port.

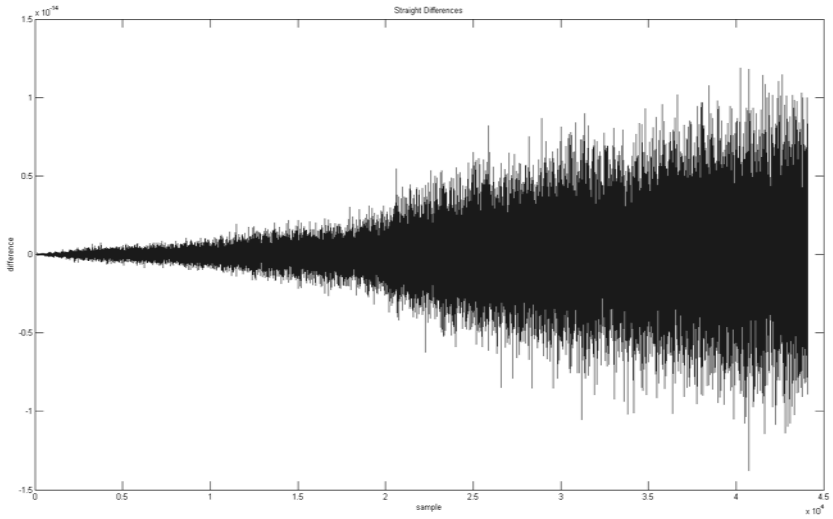
5.5 Test Results

The extension from 2D to 3D in Matlab and C required minimal refactoring of the code. One would assume that the 3D codes would behave in the same manner as their 2D counterparts. Testing was again performed at a grid size of 65,536 points.

5.5.1 Correctness testing

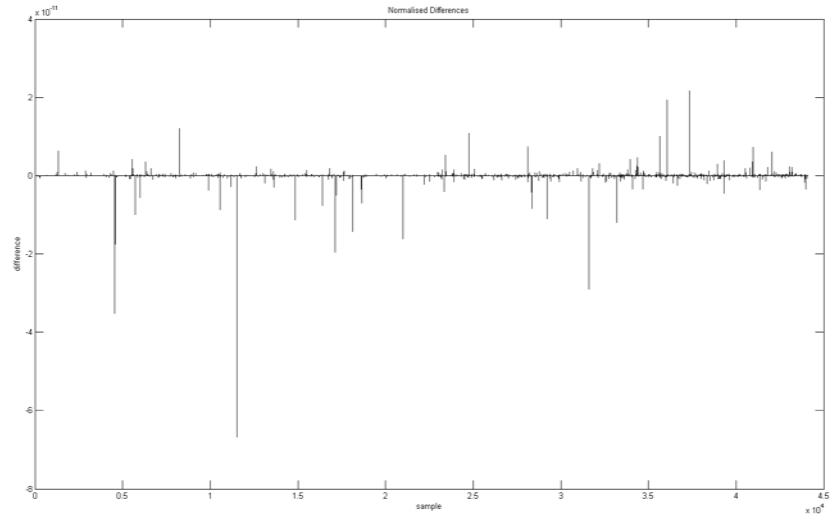
For the basic 2D code, there were no differences at all between the Matlab and C ports. This is not the case now.

Fig 5.3 – Straight differences, Matlab to c



The straight differences are in the order of E-14. The normalised differences are :

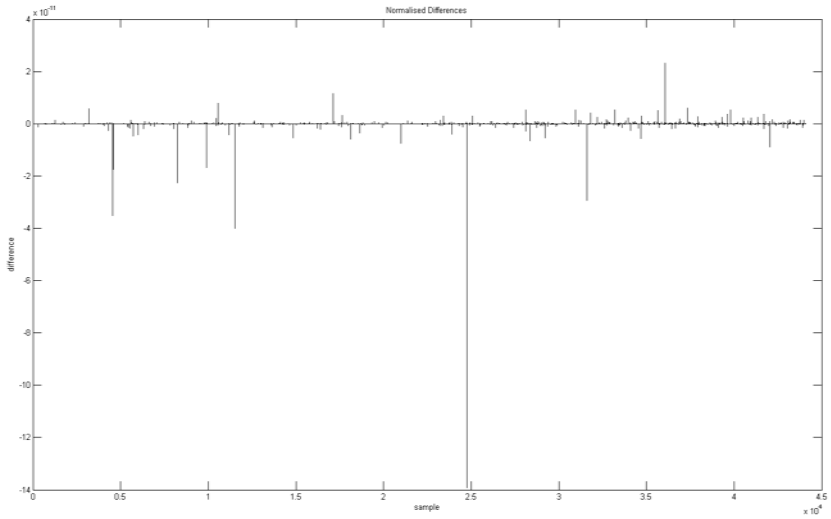
Fig 5.4 – Normalised differences, Matlab to C



There are spikes in the order of E-11. The introduction of the two extra neighbour points has produced variations in the outputs.

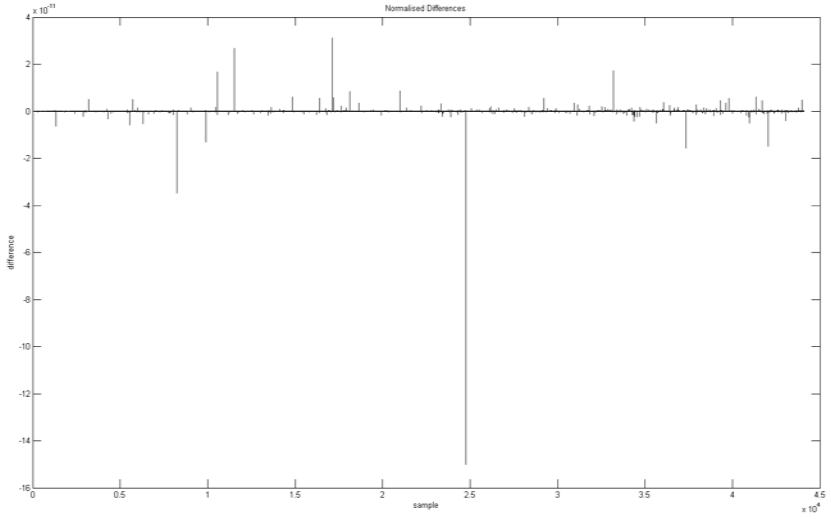
The differences between the Matlab and the Tesla's output are :

Fig 5.5 – Normalised differences, Matlab to Tesla



Again, showing spikes up to E-11. This, however, is the comparable to the differences found in the basic 2D case. As then, both Tesla's produced exactly the same outputs. The above plots show that the C and Tesla differences are not the same. The differences here are :

Fig 5.6 – Normalised differences, C to Tesla



Again, theses are in order of E-11. To summarise, the C differs from the Matlab, the Tesla differs from the Matlab, and the C differs from the Tesla. The introduction of differences between Matlab and C confirms the results of the advanced 2D scheme. The addition of further neighbouring points produces differences where previously there were none.

5.5.2 Timing results

The following table shows the execution times for the basic 3D scheme :

Fig 5.7 – Basic 3D execution times

Grid Points	Matlab	C	Tesla	Tesla FERMI
65,536	264 (4.4m)	47.6	4.5	4.1
262,144	1,148 (19m)	197 (3.2m)	10.4	7.0
1,048,576	5,304 (88m)	927 (15.4m)	48.1	18.5
4,194,304	21,249(354m)	3,925 (65m)	185.5	71.9

Times in seconds (minutes)

Which as speed-ups over the Matlab are :

Fig 5.8 – Speed ups over Matlab

Grid Points	C	Tesla	Tesla FERMI
65,536	5.6	58.7	64.4
262,144	5.8	110.4	164.0
1,048,576	5.7	110.3	286.7
4,194,304	5.4	114.5	295.5

Firstly, the Matlab code runs slower than for the 2D scheme for the same number of grid points, by a factor of around 1.1 ~ 1.2. The additional index in the arrays takes longer to compute. The same is true of the C code, and indeed the Tesla's. Overall, the same speed-up of nearly 300 times is shown for the FERMI card over Matlab at 4 million points.

Comparing the original Tesla to the FERMI card, as the grid sizes increase the FERMI card becomes progressively faster. The original C1060 seems to quickly settle at around 110 ~ 130 times speed-up over Matlab, whereas the FERMI Tesla keeps on increasing up to around the 300 mark.

5.5.3 Taking it to the limit

As with the 2D scheme, we can run maximum DRAM tests on the Tesla's in very little time. The 3D grid dimensions were set to give 12 million points in each grid. The results mirrored those for the smaller dimensions, being a factor of 1.2 slower than the 2D tests.

Tesla C1060 : 5,349s, 89.1 minutes.
Fermi Tesla C2050 : 1,932 s, 32.2 minutes.

This still maintains a 300 times speed-up over the projected Matlab time for this grid size.

5.6 Optimisation tests

It is generally expected that initial 'naive' CUDA ports will run slower than their original serial C code. In order to get the promised speed-ups from GPU computing, a considerable amount of refactoring and optimisation testing is required. The use of the occupancy calculator to adjust thread block sizes is a usual first step. With all of the CUDA tests, a block size of 256 threads was found to produce the optimum execution times, even on the FERMI Tesla.

The second stage of optimisation is the use of shared memory. The Tesla C1060 does not provide any DRAM cache, and so using the shared memory as a block cache can achieve significant speed-ups. A second CUDA port of the basic 3D code was used to test this. However, the implementation of a shared memory version was complicated by the requirements of the update scheme.

Recall that the 3D scheme uses two-dimensional tiling over each layer of the 3D data set. By loading a 2D array of data into shared memory, we can potential access four of the six neighbour points from there. However, in order for the threads at the very edges of each tile to access their correct neighbour points, the 2D array in shared memory needs to have an extra layer of data around each side. For the 16 x 16 block size, the array in shared memory needs to be 18 x 18. This creates a problem, as the only way to

load shared memory is by using the threads themselves. So, it becomes necessary to use IF statements to ensure that the 18 x 18 shared memory array is loaded by just 16 x 16 threads.

As a result, the tests performed on the shared memory version ran slightly slower than the original, by a factor of 1.1. This could possible be improved by the use of 3D block tiling. By using the extra dimension, all six neighbour points could be accessed from shared memory. However, 3D block tiling would require far more complex de-referencing of data access, and so is not expected to provide any overall benefits.

The reason for the speed-ups shown by the original CUDA code is that we are achieving a high level of data transfer from global memory, due to memory coalescing. The latency is also hidden by the high occupancy rates. In any case, the use of the FERMI architecture Tesla negates the benefits of shared memory usage, due to the caching of global memory access on-chip.

6. Phase 3 – Advanced 3D scheme

The final phase of this project implements an advanced 3D finite difference scheme that simulates acoustic waves propagating in a room. The model defines a 3D space where each opposing wall is an equally sized rectangle. The height, width and depth of the space are allowed to vary independently. A loss system is used in the boundary conditions to simulate wall reflections.

The scheme is derived from the 3D wave equation :

$$\delta_{tt}u = \gamma^2(\delta_{xx}u + \delta_{yy}u + \delta_{zz}u)$$

The stability condition is :

$$\frac{\gamma k}{h} \leq \frac{1}{\sqrt{3}} \quad [9]$$

The finite difference scheme was provided by Dr. Bilbao, and is as follows.

6.1 Update scheme

The update equation used for the interior points is :

$$u_{l,m,n}^{n+1} = \left(2 - 6\frac{c^2k^2}{h^2}\right)u_{l,m,n}^{n} + \frac{c^2k^2}{h^2}S - u_{l,m,n}^{n-1}$$

where :

$$S = u_{l-1,m,n}^{n} + u_{l+1,m,n}^{n} + u_{l,m-1,n}^{n} + u_{l,m+1,n}^{n} + u_{l,m,n-1}^{n} + u_{l,m,n+1}^{n}$$

$$c = 345m/s$$

$$k = \frac{1}{SampleRate}$$

$$h = ck\sqrt{3}$$

At the boundaries, the update equation becomes :

$$u_{l,m,n}^{n+1} = \frac{1}{1+\lambda\alpha}\left(\left(2 - K\frac{c^2k^2}{h^2}\right)u_{l,m,n}^{n} + \frac{c^2k^2}{h^2}S - (1-\lambda\alpha)u_{l,m,n}^{n-1}\right)$$

where :

K = 5 at a face, 4 at an edge, 3 at a corner and the exterior neighbours outside of the grid are dropped.

$\lambda = \frac{ck}{h}$

$\alpha =$ *loss factor, > 0*

The two loss coefficients $\frac{1}{1+\lambda\alpha}$ and $1-\lambda\alpha$ were not used at the edges or corners.

6.2 Matlab

There are two approaches to implementing the code for the more complex boundary conditions. To keep the vectorized format that was used in the basic 2D and 3D models, the *K* value and two loss coefficients have to be stored in 3D arrays of the grid size. There are initialised prior to the time loop, and then can be accessed point-wise in the update equation. However, this is not efficient from a memory usage perspective. The second option is to switch to a non-vectorized loop version, which mimics the layout of the C port. Memory usage is now the same as for the basic 3D case. Both options were tested, and the vector version was 1.14 times faster. This was used for the timing results.

6.3 C port

The kernel of the C port is now :

```
for(n=0;n<NF;n++)
{
   for(p=1;p<Nz-1;p++){
      pa = p*area;
      for(q=1;q<Ny-1;q++){
         qx = pa + q*Nx;
         qm = qx-area;
```

```
        qa = qx+area;
        qn = qx-Nx;
        qt = qx+Nx;

        for(r=1;r<Nx-1;r++){
              K = (0||(r-1))+(0||(Nx-2-r))+(0||(q-1))
                 +(0||(Ny-2-q))+(0||(p-1))+(0||(Nz-2-p));

              if(K==5){
                 fcc = fac;
                 fcc2 = fac2;}
              else {
                 fcc = 1.0;
                 fcc2 = 1.0;}

              main_coeff = 2 - K*coeff;
              S = u1[qm+r]+u1[qa+r]+u1[qn+r]
                 +u1[qt+r]+u1[qx+r-1]+u1[qx+r+1];
              // update scheme
              u[qx+r] = fcc*(main_coeff*u1[qx+r]
                            + coeff*S - fcc2*u2[qx+r]);
        }
     }
  }
  // sum in source at srcx,y,z
  u[sco] += audio[n];
  // read output
  out[n] = u[rpo];
  outR[n] = u[rpo2];
  // update pointers
  dummy_ptr = u2;
  u2 = u1;
  u1 = u;
  u = dummy_ptr;}
```

Here, the value of *K* is calculated using an OR statement at the start of the innermost loop. An IF statement is then used to set the two loss coefficients at the faces. A layer of zeros around the grid is used to drop the exterior neighbour points. Note that two read-out values are taken, to give a stereo output. The read-out positions were set 0.3m apart to simulate binaural listening.

6.4 CUDA port

The CUDA port was updated in a similar manner to the C port. The threading kernel was given the same treatment as the C version, and code was added to give a stereo output. The full codes for the Matlab, C and CUDA advanced schemes are found in the Appendix.

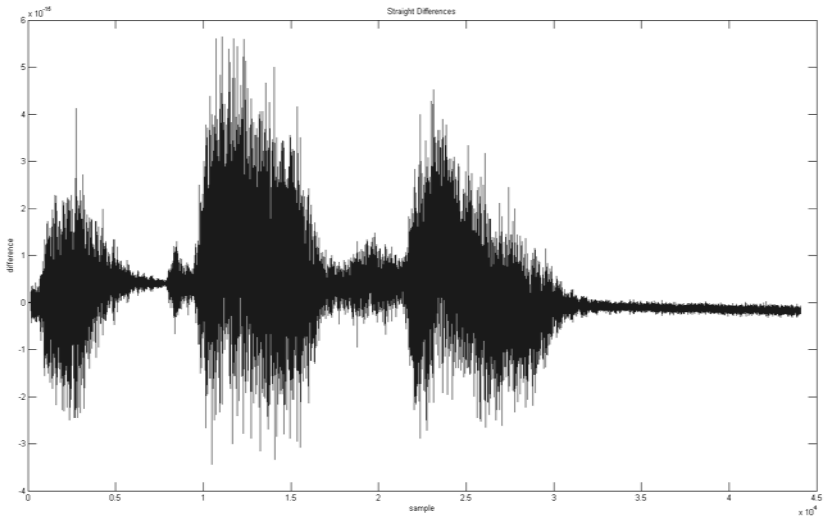
6.5 Test Results

Initial testing was performed as per phases one and two, to verify the ports and obtain comparison timing results. A maximum DRAM test was run on the FERMI Tesla to obtain a realistic simulation of an enclosed room. Further acoustical analysis could then be carried out to assess the physical properties of the simulation.

6.5.1 Correctness testing

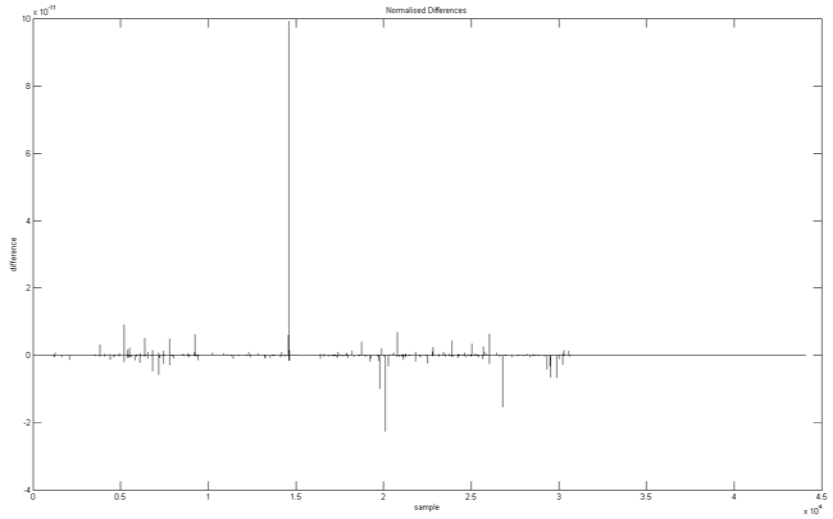
The straight A-B differences between Matlab and C are in the order of E-15, as shown here :

Fig 6.1 – Straight differences, Matlab to C



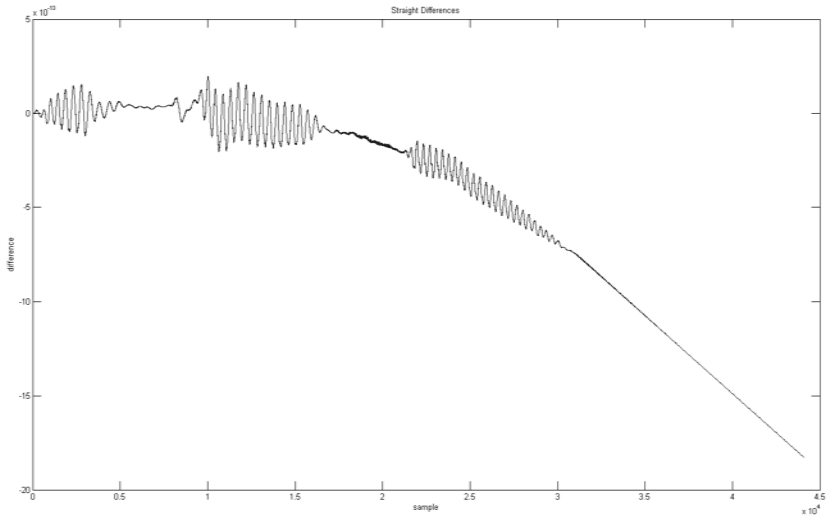
The normalised differences show the occasional spikes of E-11 against a background of E-15.

Fig 6.2 – Normalised differences, Matlab to C



This compares favourably to the results from the basic 3D case. The straight differences between Matlab and the Tesla's is in the order of E-13.

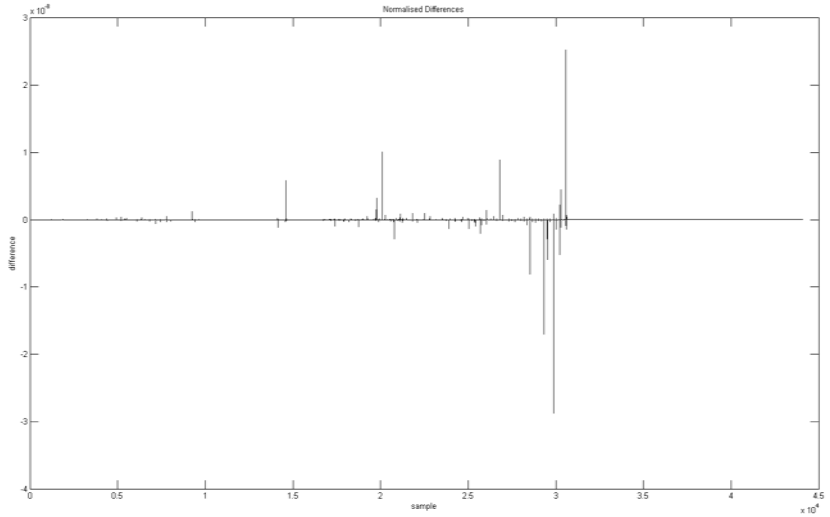
Fig 6.3 – Straight differences, Matlab to Tesla



Both Tesla's produced exactly the same output. However, they show a different pattern of differences compared to the basic 3D scheme. Here, we have a drift over the duration of the test.

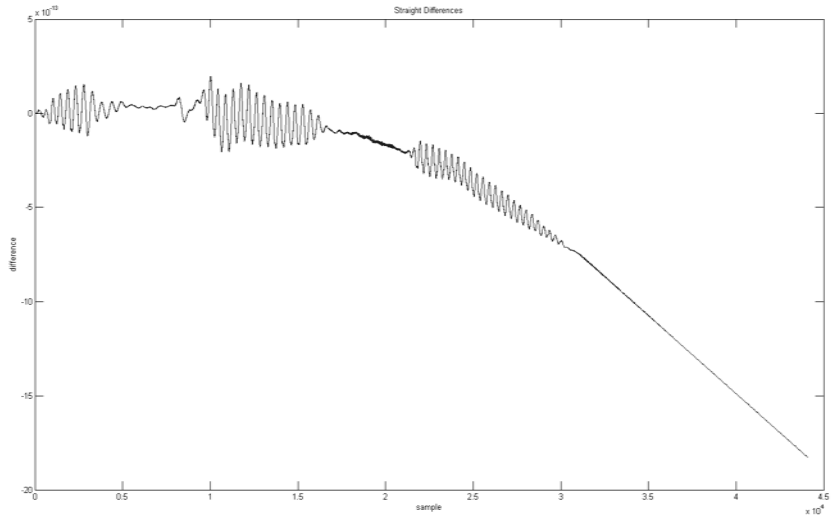
The normalised differences show spikes in the order E-8, as shown here :

Fig 6.4 – Normalised differences, Matlab to Tesla



The Tesla's show the same pattern of differences when compared to the C. This is expected, as the Tesla's differences are of an order of E3 higher than the differences of C to Matlab.

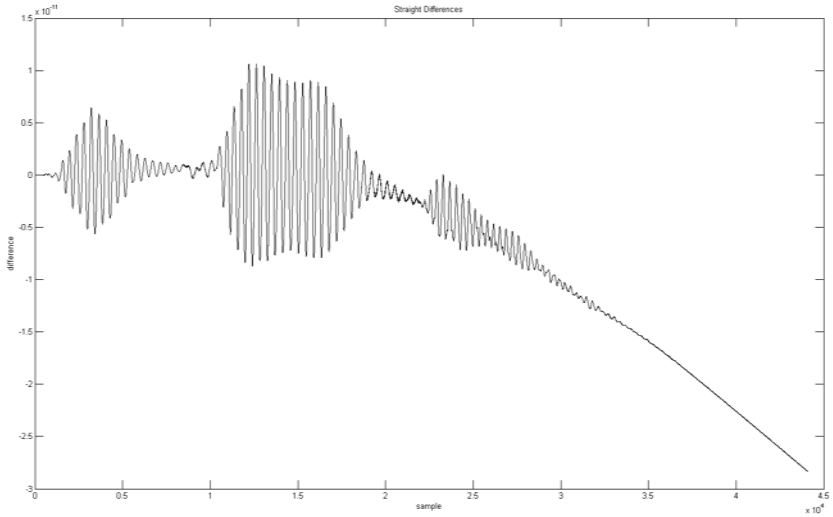
Fig 6.5 – Straight differences, C to Tesla



A further test was performed at a grid size of 1 million points to verify the drift behaviour.

The straight differences between C and the Tesla are :

Fig 6.6 – Straight differences, C to Tesla at 1 million points



This shows the same pattern as for the smaller test, but the order of magnitude has increased to E-11.

6.5.2 Timing results

The execution times for the advanced 3D scheme are as follows :

Fig 6.7 – Execution times for advanced 3D scheme

Grid Points	Matlab	C	Tesla	Tesla FERMI
65,536	360 (6m)	80	4.8	4.1
262,144	1,526 (25.4m)	330 (5.5m)	12.5	7.2
1,048,576	6,792 (113m)	1,495 (25m)	57.4	19.6
4,194,304	28,625 (477m)	6,882 (115m)	218.2	76.2

Times in seconds (minutes)

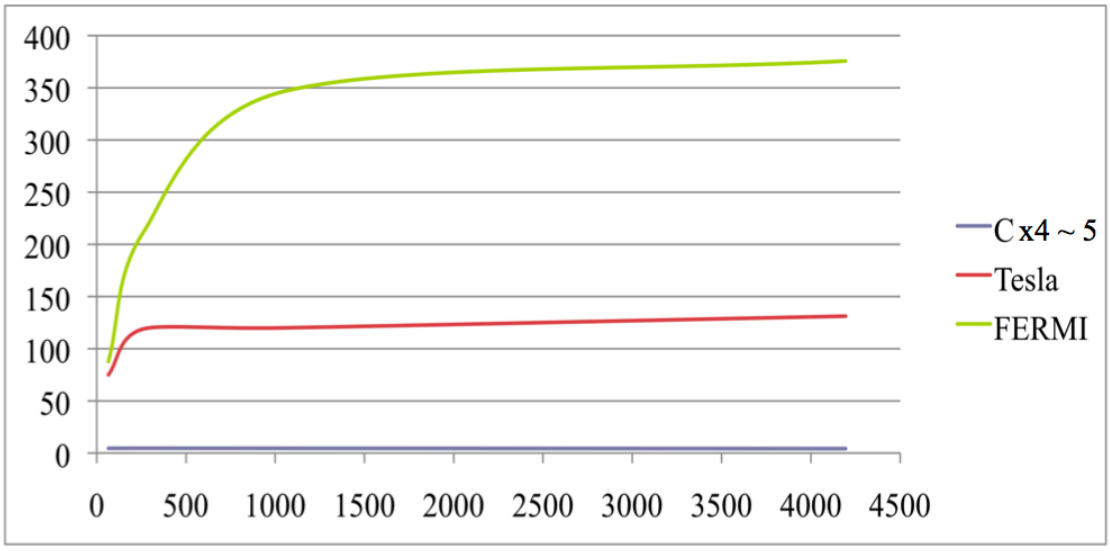
As speed-ups over Matlab :

Fig 6.8 – Speed-ups over Matlab for advanced 3D scheme

Grid Points	C	Tesla	Tesla FERMI
65,536	4.5	75.0	87.8
262,144	4.6	122.1	211.9
1,048,576	4.5	118.3	346.5
4,194,304	4.2	131.2	375.7

Plotting speed-ups against grid size gives :

Fig 6.9 – Plot of speed-ups over Matlab for advanced 3D scheme



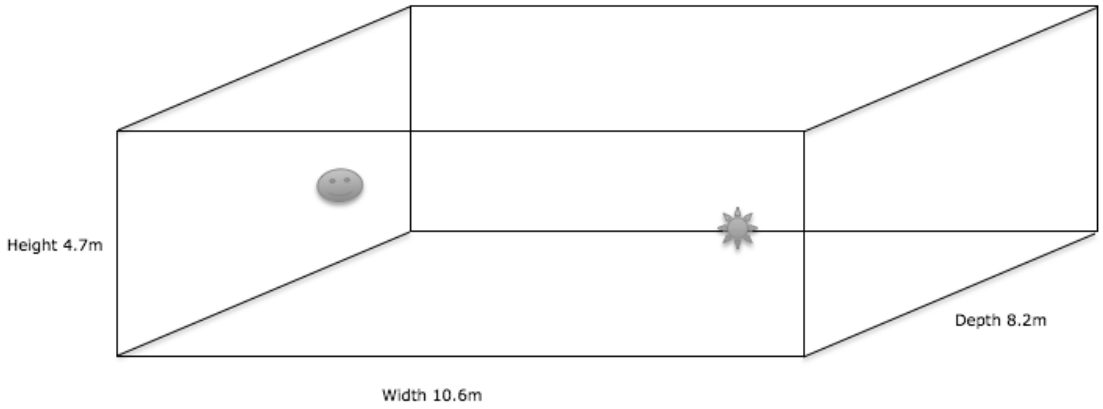
Compared to the basic 3D scheme, Matlab is 1.4 times slower at the 4 million grid size with the extra calculations required for the advanced scheme. The C code is similar at 1.5 times slower. The Tesla's fare much better. At the 4 million grid size the C1060 runs 30 seconds slower than the basic 3D scheme, and the FERMI Tesla just 4 seconds slower. This now gives a 375 times speed-up over Matlab. The extra computation of the advanced scheme is easily dealt with in the parallel threads of the CUDA code.

6.5.3 Simulating a realistic room

With the correctness and comparison timings completed, the final stage of this project was to create a simulation of a realistic room. To make the most of the 3Gb of DRAM on the FERMI Tesla, the CUDA code was refactored. Up till now, the codes have been using three arrays for the grids, u^{n+1}, u^{n} and u^{n-1}. At the start of every time step, u^{n+1} holds data which is from *three* times steps ago. This is redundant data and gets over-written. So, it is possible to use just *two* arrays in memory to hold all the data required. At the start of each time step, u^{n+1} can hold the data from two times steps ago. Before updating each point, we simply store the current value in a temporary variable, and then

use this in the update equation. By using two arrays instead of three, we can increase the maximum size of the grids by 50%. This allowed the simulation of a room of size 10.6m x 4.7m x 8.2m = 417 cubic metres.

Fig 6.10 – Room dimension of simulation



At 44.1kHz this gives a grid size of : 784 x 352 x 608 = 167,788,544 points.
The timing results for 1 second of output were :

Tesla C1060 : 7,784s, 129 minutes.
Fermi Tesla C2050 : 3,102s, 51 minutes.

At this grid size, the set-up time for the Tesla's is 2.9 seconds. This is the time taken from the start of the code to the start of the kernel. At the smaller grid sizes used for the comparison testing, the set-up times were around 0.6 seconds. In terms of full execution time, the FERMI Tesla is over 2.5 times faster than the C1060.

The source and stereo read positions were set at either end of the room, approximately 1m inside the facing wall. Some experimentation was required to find useful settings of *alpha*, the loss factor of the update scheme. At this grid size, a value of 0.01 gave a realistic amount of reverberation given the size of the physical space.

6.5.4 Acoustic testing of the room simulation

The acoustic testing was performed on a room size of 7.15m x 3.90m x 9.54m, which gives grids of 107 million points each. The reverberation was analysed in terms of RT60 times for sine wave inputs of 250, 500 and 1kHz. The times varied between 1.0 ~ 1.1 seconds. As the losses in the room are not frequency dependent, we would expect the RT60 times to be very similar.

Aside from the reverberation qualities of the output, a key acoustical property is the room modes. Any enclosed space will create standing waves across the parallel surfaces of the walls. The modes are given by :

$$f_{l,m,p} = \frac{c}{2}\sqrt{\frac{l^2}{Lx^2} + \frac{m^2}{Ly^2} + \frac{p^2}{Lz^2}}$$

The first three modes for the room are :

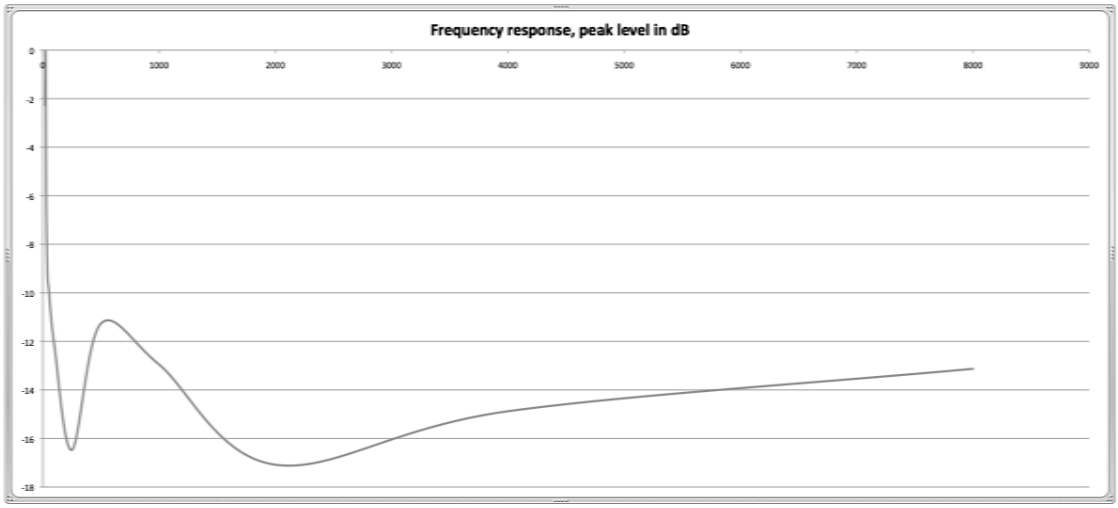
1. *l,m,p = 0,0,1* : 18.0 Hz

2. *l,m,p = 1,0,0* : 24.1 Hz

3. *l,m,p = 1,0,1* : 30.1 Hz

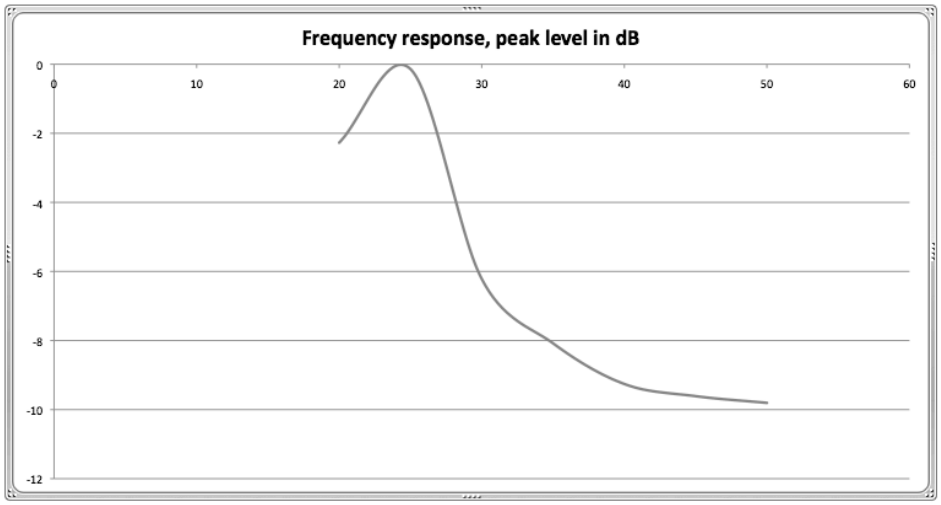
The frequency response of the room was analysed by testing with ½ second sine wave inputs at various frequencies. The sine wave inputs were all of amplitude 1. The maximum amplitude of the output, at each frequency, is plotted here :

Fig 6.11 – Frequency response, in dB scale compared to maximum



The low frequency part is magnified here :

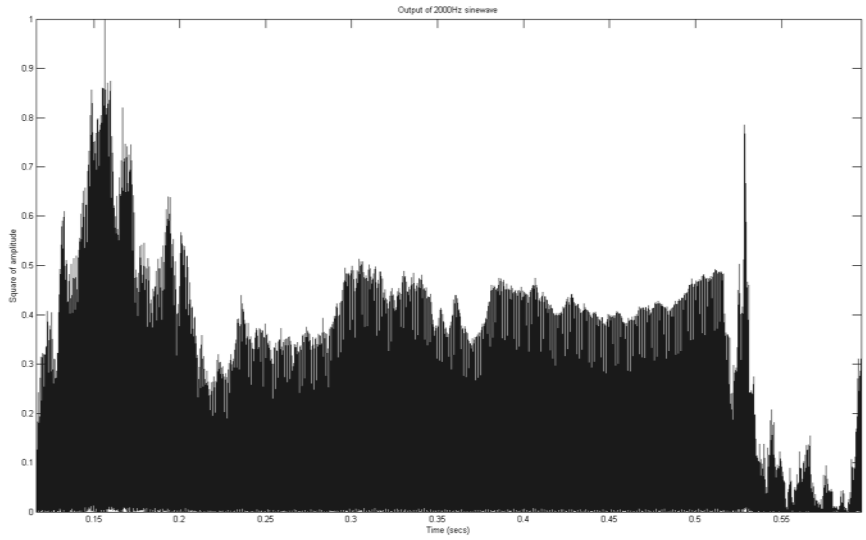
Fig 6.12 – Low frequency response



Between 20 and 30Hz there is a large peak, with a maximum at around 25Hz. This is as expected, with the first three modes of the room in this region.

The sine wave testing also revealed an echo that is clearly audible on the outputs. The plot of the squared amplitude levels for the 2000Hz sine wave is :

Fig 6.13 – 2000Hz sine wave output amplitude



The initial peak is at around 0.15 seconds. The output drops just after 0.5 seconds as the input ceases. At around 0.53 seconds there is a second peak, an echo of the initial.

7. Summary and conclusions

The three-phase implementation of this project allowed the comparison of results for a simplistic 2D finite difference scheme through to an advanced 3D scheme for simulating room acoustics. This has shown many variations in terms of the correctness of the outputs at each stage, along side the exceptional speed-ups provided by the Tesla GPU's.

7.1 Summary of correctness results

The first observation is that the outputs from both C and CUDA show differences from the output of Matlab, with the exception of the basic 2D case comparing Matlab to C. In some respects this seems unusual. All three implement double precision floating-point arithmetic, and perform the same calculations. However, in terms of the Tesla's there are known discrepancies between the IEEE 754-2008 standard and that implemented on the GPU [6]. These differences are small, but then so are the resulting differences shown in the results.

All of the testing phases for the Tesla's showed spikes of individual sample errors for the normalised differences, against a background in a smaller range. These varied from spikes at E-10 for the basic 2D case, to E-8 for the advanced 3D case. The background differences are E-3 or E-4 beneath this. As GPU processors do not have the same error checking capability as CPU's, these spikes could be produced by individual bit errors. However, in all the testing the outputs of both Tesla cards was exactly the same, down to the last significant digit. If random bit errors were causing the spikes, then this exact matching would not be seen.

The fact that these same spikes are shown in the C code implies that the differences are a product of the iterative nature of the schemes. Both the advanced 2D case and the 3D schemes used a wider stencil pattern, and this seems to have introduced more differences. Further examination of this behaviour is required to understand how they come about. However, even in the advanced 3D scheme the differences are reasonable. The C differs from the Matlab in the order of E-15, and the Tesla outputs differ by E-13,

on straight differences. These are *very* small, and certainly within an acceptable bound for the purpose of digital audio. The relationship between differences and the size of the grids also requires further analysis. The testing case performed at a grid size of 1 million points showed an increase in the order of the normalised errors from E-13 to E-11.

7.2 Summary of timing results

Whilst the correctness testing has show results that were unexpected, the results of the timings were certainly what were hoped for. The objective of the project was to try to accelerate the computation of 3D finite schemes, and this has been shown.

Firstly, the C ports gave a speed-up of 4 ~ 5 times over Matlab. This in itself is substantial, and it is not overly complicated to port from Matlab. Both Tesla's show far greater performance. The older C1060 card showed a speed-up of 120 ~ 130 times over Matlab, and 25 times over C. The benefits of the new FERMI architecture are clear, showing 375 times speed-up compared to Matlab, and 90 times to C, for the advanced 3D scheme.

This is an important result. Execution times have a big impact on the way experimental research is carried out. There is a major difference between waiting 7 days for a result from Matlab, compared to just 30 minutes for the FERMI Tesla.

7.3 Appraisal of final room simulation

The testing of the advanced scheme at maximum DRAM on the GPU's gave audio outputs that simulate realistic room acoustics. With the vocal sample input, the sound clearly has the qualities of a source emanating from a large, reflective room. The use of stereo listening positions gave some indication of spatial positioning in the horizontal plane. The reflections exhibit a large amount of energy at high frequency, which results in a 'whistle' that is slightly unnatural. Further improvements to the boundary condition losses should improve this.

7.4 Concluding remarks and scope for future work

In the domain of finite difference schemes, GPU computing has the potential for vast speed-ups over traditional implementations using Matlab or C. As the CUDA environment matures, along with further increases in GPU performance, the speed-ups available in future should continue to increase, along with becoming easier to program. Whilst the correctness of their outputs needs to be examined in detail, the advantages offered in efficiency are difficult to ignore.

This project presented initial steps in simulating actual room acoustics. Its focus has been on accelerating the execution time in order to calculate realistic room sizes in a reasonable time frame. There is much scope for further research continuing on from this project, in areas such as :

1. Analysis of correctness in terms of the origination and propagation of errors, and the effect of increased grid sizes and duration times.

2. The use of multiple GPU cards, benefiting from increased DRAM availability and extending parallelization. All CUDA codes in this project were executed on a single GPU card, but multiple cards are available for simultaneous use. However, there is a difficulty in using multiple cards, as device-to-host memory transfers would be required at every time step.

3. Improvements in the model itself, in terms of its physical qualities. This could be in the areas of spatialisation and binaural listening, improved boundary conditions for realistic reflections, the shape and dimensions of the space, and indeed the contents of the room itself.

References

[1] D.Kirk and W.Hwu. *Programming Massively Parallel Processors*. Morgan Kaufmann Publishers, 2010.

[2] NVIDIA. *CUDA C Programming Guide 3.1, version 3.1.1*. CUDA Zone. 2010. (http://www.nvidia.com/object/cuda_home_new.html).

[3] C.Trendall and A.J.Steward. *General Calculations using Graphics Hardware, with Applications to Interactive Caustics*. In Proceedings of Eurogaphics Workshop on Rendering 2000, Springer, 287- 298. 2000.

[4] J.N.England. *A system for interactive modeling of physical curved surface objects*. In Proceedings of SIGGRAPH 78 1978, 336-340. 1978.

[5] M.Potmesil and E.M.Hoffert. *The Pixel Machine: A Parallel Image Computer*. In Proceedings of SIGGRAPH 89 1989, ACM, 69-78. 1989.

[6] NVIDIA Corp. *CUDA C Best Practices Guide, version 3.1*. CUDA Zone. 2010. (http://www.nvidia.com/object/cuda_home_new.html).

[7] Apple Inc. *OpenCL programming guide for OSX*. 2009. (http://developer.apple.com/mac/library/documentation/Performance/Conceptual/OpenCL_MacProgGuide/Introduction/Introduction.html)

[8] NVIDIA Corp. *NVIDIA Fermi Compute Architecture Whitepaper*. CUDA Zone. 2010. (http://www.nvidia.com/object/cuda_home_new.html).

[9] S.Bilbao. *Numerical Sound Synthesis*. John Wiley and Sons, Chichester, UK, 2009.

[10] M.Harris, G.Coombe, T.Scheuermann, and A.Lastrahttp. *Physically-based visual simulation on graphics hardware.* In Proceedings of the ACM Siggraph /Eurographics conference on Graphics hardware. Germany, 2002.

[11] P.Micikevicius. *3D finite difference computation on GPUs using CUDA*. Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, p.79-84, March 08-08, 2009, Washington, D.C.

[12] S. Adams, J. Payne, R. Boppana. *Finite Difference Time Domain (FDTD) Simulations Using Graphics Processors*. hpcmp-ugc, pp.334-338, 2007 DoD High Performance Computing Modernization Program Users Group Conference, 2007.

[13] Nikunj Raghuvanshi, Rahul Narain, Ming C. Lin. *Efficient and Accurate Sound Propagation Using Adaptive Rectangular Decomposition.* IEEE Transactions on Visualization and Computer Graphics, vol. 15, no. 5, pp. 789-801, Sep./Oct. 2009, doi:10.1109/TVCG.2009.28

[14] L. Savioja, M. Karjalainen, and T. Takala. *DSP Formulation of a Finite Difference Method for Room Acoustics Simulation*. In NORSIG'96 IEEE Nordic Signal Processing Symposium (Espoo, Finland), pp. 455-458, Sept. 1996

[15] N. Rober, U. Kaminski, and M. Masuch. *Ray acoustics uisng computer graphics technology*. In Proceedings of the 10th International Conference on Digital Audio Effects, Bordeaux, France, September 10--15 2007.

[16] K. Kowalczyk and M. Van Walstijn. *Room acoustics simulations using 3-D compact explicit FDTD schemes*. In IEEE Transactions on Audio, Speech and Language Processing. Issue:99,p1. April 2010.

[17] Alex Southern, Damian Murphy and Jeremy Wells. *Rendering walk-through auralisations using wave-based acoustical models*. The 17th European Signal Processing Conference (EUSIPCO 2009), Glasgow, UK, August, 2009

[18] S. Adams, J. Payne, R. Boppana. *Finite Difference Time Domain (FDTD) Simulations Using Graphics Processors*. hpcmp-ugc, pp.334-338, 2007 DoD High Performance Computing Modernization Program Users Group Conference, 2007

[19] D.Liuge, L.Kang, K.Fanmin. *Parallel 3D Finite Difference Time Domain Simulations on Graphics Processors with CUDA*. In proceedings of Computational intelligence and Software Engineering, CiSE 2009.

[20] E. Motuk, R. Woods, S. Bilbao. *Implementation of finite difference schemes for the wave equation on FPGA*. In proceedings of IEEE International Conference ICASSP ’05 Vol 3, p237~240. 2005.

[21] I. A. Drumm. *The Application of Adaptive Beam Tracing and Managed DirectX for the Visualisation and Auralisation of Virtual Environments*. Information Visualisation (IV05) Conference Proceedings 2005.

Appendix

1. Matlab code for advanced 3D scheme

```
%-------------------------------------------------
% File name     : room_fdtd_10.m
% Description   : 3D Room Reverb, FDTD scheme
%               : With loss
% Author        : Craig J. Webb
% Date created  : 19/07/10
%-------------------------------------------------

clear; close all;
tic

%-------------------------------------------------
% set global variables
SR = 44100;                             % sampling rate
c = 345;                                % speed of sound in air (m/s)
L = [0.87 0.87 0.22];                   % room dimensions, L, W, H (m) 3.48 0.87
TF = 1;                                 % duration of scheme (s)
alpha = 0.01;                           % loss factor, > 0

% derived parameters
k = 1/SR;                               % time step
NF = floor(SR*TF);                      % duration of scheme (samples)
h = sqrt(3)*c*k;                        % grid spacing
Nx = floor(L(1)/h);                     % x grid points
Ny = floor(L(2)/h);                     % y grid points
Nz = floor(L(3)/h);                     % z grid points
src = [Nx-19 (Ny/2)+1 (Nz/4)+1]
rps = [21 (Ny/2)+1 (Nz/2)+1]

grid = [Nx,Ny,Nz]                       % display grid dimensions
total_points = grid(1)*grid(2)*grid(3) % display total number grid points
lambda = c*k/h;                         % coefficient
coeff = c^2*k^2/h^2;                    % coefficient
fac = 1/(1+lambda*alpha);
fac2 = 1-lambda*alpha;
fcc = 1;
fcc2 = 1;

%-------------------------------------------------
% Source excitation
file1 = fopen('vocal_44_d.bin','r');
if file1==-1
    error('Cannot open file1...');
end

% Source excitation
si = fread(file1,'double');
si = [si;zeros(NF-length(si),1)];
fclose(file1);

% initialise grid arrays
u = zeros(Ny,Nx,Nz);
u1 = zeros(Ny,Nx,Nz);
u2 = zeros(Ny,Nx,Nz);

K = zeros(Ny,Nx,Nz);
fcc = zeros(Ny,Nx,Nz);
fcc2 = zeros(Ny,Nx,Nz);
main_coeff = zeros(Ny,Nx,Nz);

% initialise K, fcc and fcc2 and main_coeff
for dep = 2:Nz-1
    for col = 2:Nx-1
        for row = 2:Ny-1
            K(col,row,dep) = (0||(row-2))+(0||(Ny-1-row))+(0||(col-2)) ...
                   +(0||(Nx-1-col))+(0||(dep-2))+(0||(Nz-1-dep));
                if K(col,row,dep) == 5
                    fcc(col,row,dep) = fac;
                    fcc2(col,row,dep) = fac2;
```

```
                else
                    fcc(col,row,dep) = 1;
                    fcc2(col,row,dep) = 1;
                end
                main_coeff(col,row,dep) = 2-K(col,row,dep)*coeff;
        end
    end
end

% initialise output array
out = zeros(NF,1);

%-----------------------------------------------
% calculate scheme
for n = 1:NF

   u(2:Nx-1,2:Ny-1,2:Nz-1) = fcc(2:Nx-1,2:Ny-1,2:Nz-1)
                             .*(main_coeff(2:Nx-1,2:Ny-1,2:Nz-1)
                             .*u1(2:Nx-1,2:Ny-1,2:Nz-1) ...
                             - fcc2(2:Nx-1,2:Ny-1,2:Nz-1).*u2(2:Nx-1,2:Ny-1,2:Nz-1) ...
                             + coeff*(u1(1:Nx-2,2:Ny-1,2:Nz-1) ...
                               +u1(3:Nx,2:Ny-1,2:Nz-1) ...
                               +u1(2:Nx-1,1:Ny-2,2:Nz-1) ...
                               +u1(2:Nx-1,3:Ny,2:Nz-1) ...
                               +u1(2:Nx-1,2:Ny-1,1:Nz-2) ...
                               +u1(2:Nx-1,2:Ny-1,3:Nz)));

   % calc scheme update
   %u(row,col,dep) = fcc*(main_coeff*u1(row,col,dep) + coeff*S - fcc2*u2(row,col,dep));

    % sum in the source impulse at grid point src[]
    u(src(2),src(1),src(3)) = u(src(2),src(1),src(3)) + si(n)*75;

    % read output at listener position
    out(n) = u(rps(2),rps(1),rps(3));

    % update matrices
    u2 = u1;
    u1 = u;

end
toc

% write to file
%fid = fopen('m_lrooms.bin','wb');
%fwrite(fid,out,'double');
%fclose(fid);
```

2. C code for advanced 3D scheme

```
/*
 Author       : Craig J. Webb
 Date         : 20th July 2010
 Description  : Room 3D FDTD code, audio input
 Notes        : Offset parameters (dep*width*height)+(row*width+col)
 : (dep*Nx*Ny)+(row*Nx+col)
 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <time.h>
#define ReaL double

int main () {

        // start clock
        clock_t t1=clock();
        //-----------------------------------------
        // read in audio data
        ReaL *in_file;
        FILE *file_ptr;
        size_t bytes_read;
        int sf = 32000;
```

```
if(sizeof(ReaL)==sizeof(double))
{
	// DOUBLE FILE
	file_ptr = fopen("vocal_44_d.bin","rb");
	in_file = (ReaL *)calloc(sf,sizeof(ReaL));
}
else
{
	// FLOAT FILE
	file_ptr = fopen("vocal_44_s.bin","rb");
	in_file = (ReaL *)calloc(sf,sizeof(ReaL));
}

if(file_ptr != NULL)
{
	bytes_read = fread(in_file,sizeof(ReaL),(size_t)sf, file_ptr);
	(void)fclose(file_ptr);
	printf("\n%d samples from audio file...\n",(int)bytes_read);

}
else
{
	printf("\n\nFile open failed...\n\n");
	exit(EXIT_FAILURE);
}

//-----------------------------------------
// variable declarations
int Nx = 256;                                   // grid dimensions
int Ny = 256;
int Nz = 16;
int TF = 1;                                     // duration of scheme, seconds

ReaL c = 345.0;                                 // speed of sound
ReaL SR = 44100.0;                              // sample rate
ReaL alpha = 0.01;                              // loss factor

ReaL *u;                                        // pointer to 3D grid u
ReaL *u1;                                       // pointer to 3D grid u1
ReaL *u2;                                      // pointer to 3D grid u2;
ReaL *out;                                      // pointer to output 1D array
ReaL *outR;
ReaL *audio;                                    // audio data
ReaL *dummy_ptr;                                // dummy pointer
int n,p,q,r;                                    // loop counters
int K;                                          // coefficient
ReaL S;                                         // sum of neighbours
ReaL main_coeff;                                // main coefficient
int pa,qx,qm,qa,qn,qt;

// derived parameters
ReaL k = 1/SR;                                  // time step
int NF = SR*TF;                                 // duration in samples
ReaL h = sqrt(3)*c*k;                           // grid spacing
int area = Nx*Ny;                               // grid area
int srcx = Nx-20;
int srcy = Ny/2;
int srcz = Nz/4;
int rpsx = 20;
int rpsz = Nz/2;
int rpsz2 = rpsz+2;
ReaL coeff = (c*c*k*k)/(h*h);
ReaL lambda = c*k/h;
ReaL fac = 1/(1+lambda*alpha);
ReaL fac2 = 1-lambda*alpha;
ReaL fcc = 1.0;
ReaL fcc2 = 1.0;
int sco = (srcz*area)+(srcy*Nx+srcx);
int rpo = (rpsz*area)+(srcy*Nx+rpsx);
int rpo2 = (rpsz2*area)+(srcy*Nx+rpsx);

//-----------------------------------------
// initialise arrays
u = (ReaL *)calloc(Nx*Ny*Nz,sizeof(ReaL));
u1 = (ReaL *)calloc(Nx*Ny*Nz,sizeof(ReaL));
u2 = (ReaL *)calloc(Nx*Ny*Nz,sizeof(ReaL));
```

```
out = (ReaL *)calloc(NF,sizeof(ReaL));
outR = (ReaL *)calloc(NF,sizeof(ReaL));

// check allocation
if((u==NULL)||(u1==NULL)||(u2==NULL)||(out==NULL)||(outR==NULL))
{
        printf("\nMemory allocation failed...\n");
        exit(EXIT_FAILURE);
}

// copy over audio data input
audio = (ReaL *)calloc(NF,sizeof(ReaL));
memcpy(audio, in_file, sf*sizeof(ReaL));

// print grid size
printf("\nGrid size : %d x %d x %d = %d\n",Nx,Ny,Nz,Nx*Ny*Nz);
printf("Grid size : %.1f x %.1f x %.1f\n",h*Nx,h*Ny,h*Nz);

// print source and read positions
printf("\nSource : %d,%d,%d\n",srcx,srcy,srcz);
printf("Read : %d,%d,%d and %d\n",rpsx,srcy,rpsz,rpsz2);

//-----------------------------------------
// calculate scheme
for(n=0;n<NF;n++)
{
        // update u matrix
        for(p=1;p<Nz-1;p++){
                pa = p*area;
                for(q=1;q<Ny-1;q++){
                        qx = pa + q*Nx;
                        qm = qx-area;
                        qa = qx+area;
                        qn = qx-Nx;
                        qt = qx+Nx;
                        for(r=1;r<Nx-1;r++){
                                K = (0||(r-1))+(0||(Nx-2-r))+(0||(q-1))
                                   +(0||(Ny-2-q))+(0||(p-1))+(0||(Nz-2-p));

                                if(K==5){
                                        fcc = fac;
                                        fcc2 = fac2;
                                }
                                else {
                                        fcc = 1.0;
                                        fcc2 = 1.0;
                                }

                                main_coeff = 2 - K*coeff;
                                S = u1[qm+r]+u1[qa+r]+u1[qn+r]+u1[qt+r]
                                   +u1[qx+r-1]+u1[qx+r+1];
                                // update scheme
                                u[qx+r] = fcc*(main_coeff*u1[qx+r]
                                        + coeff*S - fcc2*u2[qx+r]);
                        }
                }
        }

        // sum in source at srcx,y,z
        u[sco] += audio[n]*75.0;

        // read output
        out[n] = u[rpo];
        outR[n] = u[rpo2];

        // update pointers
        dummy_ptr = u2;
        u2 = u1;
        u1 = u;
        u = dummy_ptr;
}

//-----------------------------------------
// print process time
clock_t t2=clock();
printf("%.1lf seconds of processing.\n", (t2-t1)/(double)CLOCKS_PER_SEC);
```

```
        //print last 6 samples
        for(n=NF-7;n<NF;n++)
        {
                printf("Sample %d : %.10f\n",n,out[n]);
        }

        // write to file
        size_t bytes_written;

        file_ptr = fopen("c_lroom_dL.bin","wb");
        if(file_ptr != NULL)
        {
                bytes_written = fwrite(out,sizeof(ReaL),(size_t)NF, file_ptr);
                (void)fclose(file_ptr);
                printf("\n%d samples written to file...\n",(int)bytes_written);
        }
        else
        {
                printf("\n\nFile open failed...\n\n");
        }
        /*
        file_ptr = fopen("c_lroom_dR.bin","wb");
        if(file_ptr != NULL)
        {
                bytes_written = fwrite(outR,sizeof(ReaL),(size_t)NF, file_ptr);
                (void)fclose(file_ptr);
                printf("\n%d samples written to file...\n",(int)bytes_written);
        }
        else
        {
                printf("\n\nFile open failed...\n\n");
        }
        */
        free(u);free(u1);free(u2);free(out);free(in_file);free(audio);free(outR);
    exit(EXIT_SUCCESS);
}
```

3. CUDA code for advanced 3D scheme

```
/*
------------------------------------------------
File        : threed_6.cu
Author      : Craig J. Webb
Date        : 20/07/10
Description : Room 3D FDTD scheme, audio input, loss
Notes       : constants array :
              [0] - Nx
              [1] - block_width
              [2] - rpsz
              [3] - srcx
              [4] - srcy
              [5] - block_height
              [6] - area
              [7] - Nz
              [8] - srcz
              [9] - rpsx
              [10] - Ny
              [11] - rpsz2;
------------------------------------------------
*/

#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <math.h>
#include <string.h>

#define ReaL double

// constant memory arrays
__constant__ int dcons[12];

//-----------------------------------------------
// Kernel code
```

```
__global__ void UpDateScheme(ReaL *u,ReaL *u1,ReaL coeff,ReaL fac,ReaL fac2)
{
	ReaL K,S,main_coeff,tempu;
	int q = blockIdx.y * dcons[5] + threadIdx.y;		// Y position, Row
	int colt = blockIdx.x * dcons[1] + threadIdx.x;
	int p = colt/dcons[0];					// Z position, Dep
	int r = colt - (p*dcons[0]);				// X position, Col

	ReaL fcc = 1.0;
	ReaL fcc2 = 1.0;
	int qx = (p*dcons[6]) + (q*dcons[0]);

	K = (0||(r-1))+(0||(dcons[0]-2-r))+(0||(q-1))+(0||(dcons[10]-2-q))+(0||(p-
	1))+(0||(dcons[7]-2-p));
	if(K==5){
		fcc = fac;
		fcc2 = fac2;
	}

	main_coeff = 2 - K*coeff;

	if(!(r==0||r==dcons[0]-1||q==0||q==dcons[10]-1||p==0||p==dcons[7]-1))
	{
		S = u1[qx-dcons[6]+r]+u1[qx+dcons[6]+r]+u1[qx-
		dcons[0]+r]+u1[qx+dcons[0]+r]+u1[qx+r-1]+u1[qx+r+1];
		// update scheme
		tempu = u[qx+r];
		u[qx+r] = fcc*(main_coeff*u1[qx+r]  + coeff*S - fcc2*tempu);
	}
}

//------------------------------------------------
// read output and sum in input
__global__ void inout(ReaL *u, ReaL *out, ReaL *audio, const int nn, ReaL *outR)
{
	// sum in source at srcx,y,z
	u[(dcons[8]*dcons[6])+(dcons[4]*dcons[0]+dcons[3])] += audio[nn]*75.0;

	// read output
	out[nn] = u[(dcons[2]*dcons[6])+(dcons[4]*dcons[0]+dcons[9])];
	outR[nn] = u[(dcons[11]*dcons[6])+(dcons[4]*dcons[0]+dcons[9])];
}

//------------------------------------------------

int main(int argc, const char * argv[])
{
    // start clock
	clock_t t1 = clock();

	//-----------------------------------------
	// read in audio data
	ReaL *in_file;
	FILE *file_ptr;
	size_t bytes_read;
	int sf = 32000;

	if(sizeof(ReaL)==sizeof(double))
	{
		file_ptr = fopen("vocal_44_d.bin","rb");
		in_file = (ReaL *)calloc(sf,sizeof(ReaL));
	}
	else
	{
		file_ptr = fopen("vocal_44_s.bin","rb");
		in_file = (ReaL *)calloc(sf,sizeof(ReaL));
	}

	if(file_ptr != NULL)
	{
		bytes_read = fread(in_file,sizeof(ReaL),(size_t)sf, file_ptr);
		(void)fclose(file_ptr);
		printf("\n%d samples from audio file...\n",(int)bytes_read);

	}
	else
	{
```

```
        printf("\n\nFile open failed...\n\n");
        exit(EXIT_FAILURE);
}

// ----------------------------------------
// variable declarations
int Nx = 784;
int Ny = 352;
int Nz = 608;
int TF = 3;
ReaL c = 345.0;
ReaL SR = 44100;
ReaL alpha = 0.01;

int block_width = 16;
int block_height = 16;
ReaL *u_h, *u1_h, *out_h, *outR_h;
ReaL *u_d, *u1_d, *out_d, *outR_d;
ReaL *dummy_ptr, *audio_h, *audio_d;
int n;

//-----------------------------------------------
// derived parameters
ReaL k = 1/SR;
int NF = (int)SR*TF;
ReaL h = sqrt(3)*c*k;

int grid_width  = Nx/block_width*Nz;
int grid_height = Ny/block_height;
int area = Nx*Ny;
int srcx = Nx-80;
int srcy = Ny/2;
int srcz = 100;
int rpsx = 100;
int rpsz = Nz/2;
int rpsz2 = rpsz+20;
ReaL coeff = (c*c*k*k)/(h*h);
ReaL lambda = c*k/h;
ReaL fac = 1/(1+lambda*alpha);
ReaL fac2 = 1-lambda*alpha;

size_t mem_size = Nx*Ny*Nz*sizeof(ReaL);
dim3 dimBlock(block_width,block_height,1);
dim3 dimGrid(grid_width,grid_height);

printf("\nArray size : %d x %d x %d = %d",Nx, Ny, Nz, Nx*Ny*Nz);
printf("\nArray size : %.1f x %.1f x %.1f\n",h*Nx, h*Ny, h*Nz);
printf("\nBlock size : %d x %d =
%d",block_width,block_height,block_width*block_height);
printf("\nGrid size : %d x %d = %d
blocks\n",grid_width,grid_height,grid_width*grid_height);

//-----------------------------------------
// allocate constant memory arrays
int hcons[12];
hcons[0] = Nx;hcons[1] = block_width;hcons[2] = rpsz;hcons[3] = srcx;hcons[4] =
srcy;hcons[5] = block_height;
hcons[6] = area;hcons[7]=Nz;hcons[8]=srcz;hcons[9]=rpsx;hcons[10]=Ny;hcons[11] =
rpsz2;
cudaMemcpyToSymbol(dcons,hcons,12*sizeof(int),0,cudaMemcpyHostToDevice);

if(cudaGetLastError() != cudaSuccess)
{
        printf("\nConstants failed...\n");
        exit(-1);
}

// allocate memory on host
u_h = (ReaL *)calloc(mem_size,1);
u1_h = (ReaL *)calloc(mem_size,1);
out_h = (ReaL *)calloc(NF,sizeof(ReaL));
outR_h = (ReaL *)calloc(NF,sizeof(ReaL));
audio_h = (ReaL *)calloc(NF,sizeof(ReaL));

// check allocation
if((u_h==NULL)||(u1_h==NULL)||(out_h==NULL)||(audio_h==NULL)||(outR_h==NULL))
{
```

```
            printf("\nHost memory allocation failed...\n");
            exit(1);
    }

    // allocate memory on device
    cudaMalloc((void**)&u_d, mem_size);
    cudaMalloc((void**)&u1_d, mem_size);
    cudaMalloc((void**)&out_d, NF*sizeof(ReaL));
    cudaMalloc((void**)&outR_d, NF*sizeof(ReaL));
    cudaMalloc((void**)&audio_d, NF*sizeof(ReaL));

    // check allocation
    if(cudaGetLastError() != cudaSuccess)
    {
            printf("\nDevice memory alloc failed...\n");
            exit(-1);
    }

    // copy audio data input to audio array
    memcpy(audio_h, in_file, sf*sizeof(ReaL));

    //-----------------------------------------
    // print positions
   printf("Source : %d,%d,%d\n",srcx,srcy,srcz);
    printf("Read : %d,%d,%d and %d\n",rpsx,srcy,rpsz,rpsz2);

   //-----------------------------------------
    // copy arrays to device
   cudaMemcpy(u_d, u_h, mem_size, cudaMemcpyHostToDevice);
    cudaMemcpy(u1_d, u1_h, mem_size, cudaMemcpyHostToDevice);
    cudaMemcpy(out_d, out_h, NF*sizeof(ReaL), cudaMemcpyHostToDevice);
    cudaMemcpy(outR_d, outR_h, NF*sizeof(ReaL), cudaMemcpyHostToDevice);
    cudaMemcpy(audio_d, audio_h, NF*sizeof(ReaL), cudaMemcpyHostToDevice);
   if(cudaGetLastError() != cudaSuccess)
    {
            printf("\nHost -> Dev failed...\n");
            exit(-1);
    }

// start clock2
    clock_t t2 = clock();
    printf("\n%.1lf seconds of setup.\n", (t2-t1)/(double)CLOCKS_PER_SEC);
//-----------------------------------------
// calculate scheme
for(n=0;n<NF;n++)
    {

            UpDateScheme<<<dimGrid,dimBlock>>>(u_d,u1_d,coeff,fac,fac2);
            cudaThreadSynchronize();

            // perform read in out
            inout<<<1,1>>>(u_d,out_d,audio_d,n,outR_d);

            // update pointers
            dummy_ptr = u1_d;
            u1_d = u_d;
            u_d = dummy_ptr;

    }

// copy result back from device
    cudaMemcpy(out_h, out_d, NF*sizeof(ReaL), cudaMemcpyDeviceToHost);
    cudaMemcpy(outR_h, outR_d, NF*sizeof(ReaL), cudaMemcpyDeviceToHost);
   if(cudaGetLastError() != cudaSuccess)
    {
            printf("\nDev -> Host failed...\n");
            exit(-1);
    }

//-----------------------------------------
    // print process time
    clock_t t3=clock();
    printf("\n%.1lf seconds of processing.", (t3-t2)/(double)CLOCKS_PER_SEC);
    printf("\n%.1lf seconds total.\n\n", (t3-t1)/(double)CLOCKS_PER_SEC);

    //print last 6 samples
    for(n=NF-7;n<NF;n++)
```

```
	{
		printf("Sample %d : %.10f\n",n,out_h[n]);
	}

	// find max
	double maxy = 0;
	for(n=0;n<NF;n++)
	{
		if(fabs(out_h[n])>maxy)
		{
			maxy = fabs(out_h[n]);
		}
	}

	printf("\nMax : %.10f\n",maxy);

	// write to file
	size_t bytes_written;

	file_ptr = fopen("cu_lrooms_dL.bin","wb");
	if(file_ptr != NULL)
	{
		bytes_written = fwrite(out_h,sizeof(ReaL),(size_t)NF,file_ptr);
		(void)fclose(file_ptr);
		printf("\n%d samples written to file L...\n",(int)bytes_written);
	}
	else
	{
		printf("\n\nFile L open failed...\n\n");
	}

	file_ptr = fopen("cu_lrooms_dR.bin","wb");
	if(file_ptr != NULL)
	{
		bytes_written = fwrite(outR_h,sizeof(ReaL),(size_t)NF,file_ptr);
		(void)fclose(file_ptr);
		printf("%d samples written to file R...\n",(int)bytes_written);
	}
	else
	{
		printf("\n\nFile R open failed...\n\n");
	}

	// Free memory
	free(u_h); free(u1_h); free(out_h); free(audio_h); free(in_file);free(outR_h);
	cudaFree(u_d); cudaFree(u1_d); cudaFree(out_d);
	cudaFree(audio_d);cudaFree(outR_d);

}
```