Large-scale Virtual Acoustics Simulation at Audio Rates Using Three Dimensional Finite Difference Time Domain and Multiple GPUs

Craig J. Webb*[1,2] and Alan Gray[2]

[1]Acoustics Group, University of Edinburgh
[2]Edinburgh Parallel Computing Centre, University of Edinburgh

To be presented at the 21st International Congress on Acoustics, Montréal, Canada, 2013

Abstract

The computation of large-scale virtual acoustics using the 3D finite difference time domain (FDTD) is prohibitively computationally expensive, especially at high audio sample rates, when using traditional CPUs. In recent years the computer gaming industry has driven the development of extremely powerful Graphics Processing Units (GPUs). Through specialised development and tuning we can exploit the highly parallel GPU architecture to make such FDTD computations feasible. This paper describes the simultaneous use of multiple NVIDIA GPUs to compute schemes containing over a billion grid points. We examine the use of asynchronous halo transfers between cards, to hide the latency involved in transferring data, and overall computation time is considered with respect to variation in the size of the partition layers. As hardware memory poses limitations on the size of the room to be rendered, we also investigate the use of single precision arithmetic. This allows twice the domain space, compared with double precision, but results in phase shifting of the output with possible audible artefacts. Using these techniques, large-scale spaces of several thousand cubic metres can be computed at 44.1kHz in a useable time frame, making their use in room acoustics rendering and auralization applications possible in the near future.

*C.J.Webb-2@sms.ed.ac.uk

INTRODUCTION

High fidelity virtual room acoustics can be approached through direct numerical simulation of wave propagation in a defined space. Unlike ray-based [1] or image source [2] techniques, this approach seeks to model the entire acoustic field within the simulation domain. Three-dimensional Finite Difference Time Domain (FDTD) schemes can be employed, however at audio sample rates such schemes are extremely computationally expensive [3], prohibitively so for serial computation. Recent advances in graphics processing unit (GPU) architectures allow for general purpose computation to be performed on these forms of highly parallel hardware. Whilst central processing units (CPUs) may contain a small number of cores, such as four or eight , GPUs contain hundreds of processing cores that can be used to perform parallel computation. Using this architecture, the data independence of FDTD schemes can be leveraged to gain significant acceleration over single-threaded implementations [4], and this allows large-scale simulations to be computed in time scales that are actually useable for performing research.

For scientific computing, Nvidia's Tesla GPUs are typically used in a workstation or compute node that can be configured with four GPUs connected across the same PCIe bus. This paper examines the simultaneous use of these four-GPU systems to render virtual acoustic simulations. This allows greater acceleration of existing models, or the combined use of all available memory across four GPUs to render large-scale domains containing billions of grid points. Recent versions of the CUDA language facilitate this process [5], without recourse to MPI programming techniques.

The first section details the FDTD schemes being used, followed by an outline of the CUDA programming model for the simultaneous use of multiple GPUs. We then describe the implementation of the schemes using both non-asynchronous and asynchronous approaches. Finally, we detail experimental testing in terms of floating-point precision and overall computation times for various configurations, including large-scale simulations that use maximum memory.

VIRTUAL ACOUSTICS USING FINITE DIFFERENCE METHOD

The starting point for acoustic FDTD simulations is the 3D wave equation, which in second order form is given by:

$$\frac{\partial^2 \Psi}{\partial t^2} = c^2 \nabla^2 \Psi \tag{1}$$

Here Ψ is the target acoustical field quantity, c is the wave speed in air, ∇^2 is the 3D Laplacian. Simple first-order boundary conditions are used, where:

$$\frac{\partial \Psi}{\partial t} = c\beta \mathbf{n} \cdot \nabla \Psi \tag{2}$$

Here $\mathbf{n}$ is a unit normal to a wall or obstacle, and β is an absorption coefficient. The standard FDTD discretisation leads to the following update equation, which includes boundary loss terms using a single reflection coefficient,

$$w^{n+1}_{l,m,p} = \frac{1}{1+\lambda\beta}\left((2-K\lambda^2)w^n_{l,m,p} + \lambda^2 S^n_{l,m,p} - (1-\lambda\beta)w^{n-1}_{l,m,p}\right) \tag{3}$$

where $w_{l,m,p}$ is the discrete acoustic field, K is 6 in free space, 5 at a face, 4 at an edge and 3 at a corner, $\lambda = \frac{cT}{X}$, β the coefficient for boundary reflection losses, and $S^n_{l,m,p}$ is $(w^n_{l+1,m,p} + w^n_{l-1,m,p} + w^n_{l,m+1,p} + w^n_{l,m-1,p} + w^n_{l,m,p+1} + w^n_{l,m,p-1})$.

The stability condition for the scheme follows from von Neumann analysis [6], such that for a given time step T the grid spacing X must satisfy:

$$X \geq \sqrt{3c^2T^2} \tag{4}$$

The basic scheme can be extended to include the effect of viscosity, which gives a frequency dependent damping [4].

$$\frac{\partial^2 \Psi}{\partial t^2} = c^2 \nabla^2 \Psi + c\alpha \nabla^2 \frac{\partial \Psi}{\partial t} \tag{5}$$

Here α is a viscosity coefficient. This leads to an update equation of the form:

$$w^{n+1}_{l,m,p} = \frac{1}{1+\lambda\beta}\left((2-K\lambda^2)w^{n}_{l,m,p} + \lambda^2 S^{n}_{l,m,p} - (1-\lambda\beta)w^{n-1}_{l,m,p} + ck\alpha(S^{n}_{l,m,p} - S^{n-1}_{l,m,p})\right) \tag{6}$$

Note that this update uses the nearest neighbours from two time steps ago, thus requiring the use of three data grids. The basic scheme, which only uses the centre point from two time steps ago, can be implemented using only two grids and a read, then overwrite procedure. These systems are referred to as the "2 Grid" and "3 Grid" schemes throughout this paper.

PARALLEL COMPUTING AND USE OF MULTIPLE GPUS IN CUDA

CUDA is Nvidia's programming architecture for implementing highly-threaded GPU code. In a serial implementation of equation 3, loops would be used to iterate over the computation domain, applying the update equation to each grid point. In CUDA, we issues a large number of *kernel* threads that implement the SIMD (Single Instruction Multiple Data) operation, and these threads are scheduled to execute using a large number of parallel processing cores. The program code contains a mixture of *host* serial C code, and *device* CUDA code. Whilst the host uses standard CPU memory, the device has multiple memory types. Global memory is the core data store, and is typical in the range of 3 to 6Gb per GPU. CUDA threads make use of local, register memory, and also have a small amount of shared memory per thread block. They can also communicate directly with the global and (read-only) constant memory. Each type has different access speeds, with global memory being the slowest, and shared and constant being fast. With a four GPU server, we have four instances of this memory model which are independent. The GPUs are connected in a pair-wise manner over the PCIe bus, as shown in figure 1.

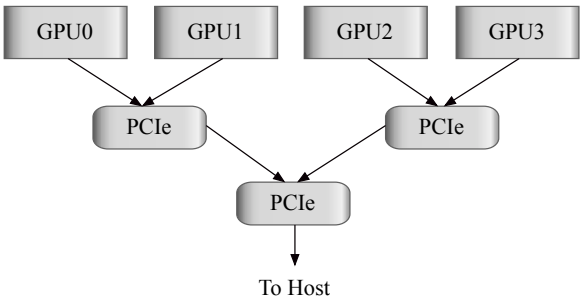

FIGURE 1: PCIe connections for four GPUs in a single compute node.

Version four and above of the CUDA architecture contains functionality that allows multiple GPUs that are connected in such a manner to be used concurrently [5]. Peer-to-peer communication allows data transfer between GPUs that bypasses the host altogether (transferring data from device to host, then back to another device is an expensive operation). This can be combined with the use of multiple streams of execution and asynchronous scheduling to achieve scalable speedups when using multiple GPUs.

IMPLEMENTATION OF THREE-DIMENSIONAL FDTD SCHEMES

This section gives a detailed description of the implementation of the basic 2 Grid FDTD scheme with its first-order boundaries. We start with a single GPU version, then extend this to an initial multiple GPU implementation. This is then developed to include the use of asynchronous data transfers of the partition halos.

Single GPU implementation

The CUDA programming model makes use of threads that are grouped first into *blocks*, and then into a *grid*. Both of these objects can be one, two, or three-dimensional in shape. Given a three-dimensional data domain, there are many possible approaches to mapping the threading model over the domain. Prior to the FERMI architecture, the standard approach was to utilise shared memory by issuing threads to cover a 2D layer of data. Each thread itself would then iterate over the final dimension, reusing data and shared memory [7]. Post FERMI, the caching system negates the benefits of this approach, and one can simply issue threads that cover the whole data domain [8]. Three-dimensional threads blocks can be used, for example 32 x 4 x 2, and a three-dimensional thread grid placed over the data. The time loop for the simulation contains a single kernel launch, then the input/output is processed, followed by swapping the pointers to the data grids, as shown in listing 1.

LISTING 1: Time loop for single GPU implementation.

```
1  for(n=0;n<NF;n++)
2  {
3     UpDateScheme<<<dimGridInt,dimBlockInt>>>(u_d,u1_d);
4     // perform I/O
5     inout<<<dimGridIO,dimBlockIO>>>(u_d,out_d,ins,n);
6     // update pointers
7     dummy_ptr = u1_d; u1_d = u_d; u_d = dummy_ptr;
8  }
```

The thread kernel itself implements both the interior and boundary update in a single SIMD operation, as shown in listing 2.

LISTING 2: Kernel code for single GPU implementation.

```
1  __global__ void UpDateScheme(real *u, real *u1){
2     // Get X,Y,Z from 3D thread and block Id's
3     int X = blockIdx.x * Bx + threadIdx.x;
4     int Y = blockIdx.y * By + threadIdx.y;
5     int Z = blockIdx.z * Bz + threadIdx.z + 1;
6
7     // Test that not at halo, Z block excludes Z halo
8     if((X>0) && (X<(Nx-1)) && (Y>0) && (Y<(Ny-1))){
9           // Calculate linear centre position
10          int cp   = Z*area+(Y*Nx+X);
11          int K    = (0||(X-1) + (0||X-(Nx-2)) + (0||Y-1) + (0||Y-(Ny-2)) + (0||Z-1) + (0||Z-(Nz-2));
12          real cf  = 1.0;
13          real cf2 = 1.0;
14          // set loss coefficients if at a boundary
15          if(K<6){ cf = cf_d[0].loss1; cf2 = cf_d[0].loss2; }
16          // Get sum of neighbour points
17          real S = u1[cp-1]+u1[cp+1]+u1[cp-Nx]+u1[cp+Nx]+u1[cp-area]+u1[cp+area];
18          // Calculate the update
19          u[cp] = cf*( (2.0 - K*cf_d[0].l2)*u1[cp] + cf_d[0].l2*S - cf2*u[cp] );
20    }
21 }
```

The kernel keeps the use of conditional statements to a minimum. A layer of non-updated "ghost" points is used around the data domain, and so line 8 employs a conditional to check for this. A single further conditional is used at line 15, to load the coefficients used at a boundary. The logical expression at line 11 computes boundary position in an efficient manner, without the need for a lengthy *IF-ELSEIF* statement.

Non-asynchronous implementation using multiple GPUs

In transitioning from a single GPU to the use of four GPUs, the data domain needs to be partitioned. The individual GPUs have discrete memory, and so the 3D data needs to be separated into four segments. A further complication is that the FDTD scheme requires neighbouring points in all dimensions, and so overlap halos will be required. The 3D data itself is decomposed using a row-major alignment for each layer of the Z dimension, with consecutive layers in series. In this format, each layer occupies contiguous memory locations. Thus, the most natural partitioning is across the Z dimension, as shown in figure 2. The overlap halos are individual Z layers, and so can be transferred as a single contiguous block of memory.

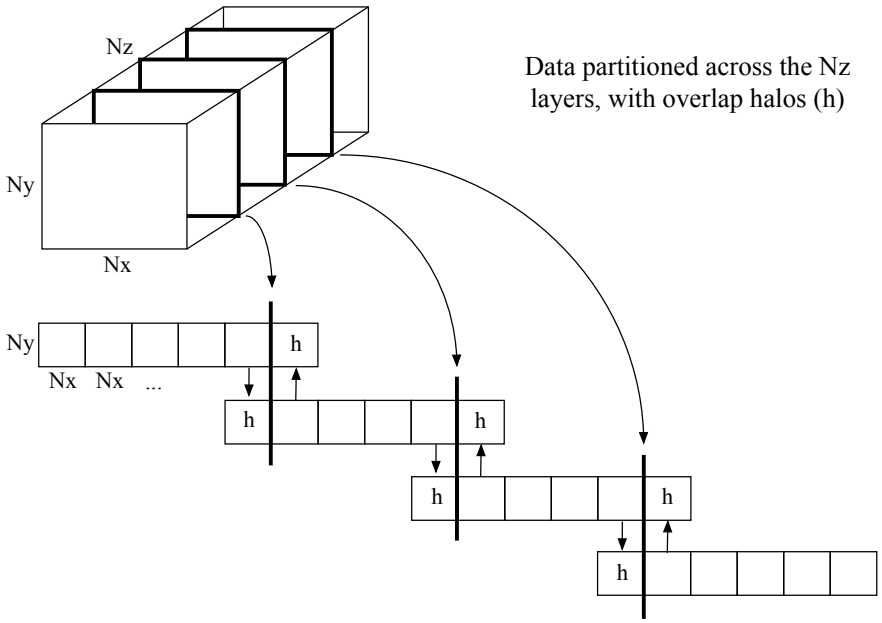

FIGURE 2: Data partitioning across the Z dimension using four GPUs, with overlap halos.

Whilst the domain partitioning is straightforward, the CUDA code itself requires many extensions compared to the single GPU case. In terms of the pre-time loop setup code, individual commands such as *cudaMalloc( )* become embedded in loops over the four GPUs. A call is made to *cudaSetDevice( )* at each iteration, to perform the operation on individual GPUs. Single pointers to device memory become arrays of pointers, and constant memory has to be allocated to each GPU. The halo offset locations have to be calculated as linear positions across memory, and finally the peer-to-peer access has to be initialised. In a non-asynchronous implementation, the time loop operates as follows:

1. Loop over the GPUs, issuing a kernel launch to compute the data on that GPU.
2. Synchronize all GPUs.
3. Perform peer-to-peer data transfer for overlap halos.
4. Perform input/output.
5. Synchronize and swap data pointers.

Each GPU computes its data simultaneously, but only when all have completed do we then perform the data transfers of the individual overlap halos.

Implementation using asynchronous data transfers

The above implementation contains an inherent time lag, as the GPUs are idle during the data transfers across the PCIe bus. To eliminate this, we can make use of asynchronous behaviour and *streams*. The approach used is based on that outlined by Nvidia [9], but is extended here to operate with the large-scale halo layers that occur in the 3D case. An individual Z layer can contain millions of floating-point values, and six layers have to be transferred between GPUs at each time step. A stream is simply a sequence of CUDA events that occur in series. However, multiple streams can be used so that events can execute in a concurrent and asynchronous manner.

As the FDTD scheme is data-independent at each time step, the overlap halo layers can be computed and the data transfers performed at the same time as the larger interior data segments on each GPU. This is accomplished by using one stream of events for the halos and transfers on each GPU, whilst a second

stream is used for the interior. The streams are identified in the kernel launches, as shown in the time loop code detailed in listing 3.

LISTING 3: Time loop for asynchronous four GPU implementation.

```
for(n=0;n<NF;n++)
{
   // Compute halo layers, then interior
   hp = 0;
   for(i=0;i<num_gpus;i++){
        cudaSetDevice(gpu[i]);
        UpDateHalo<<<dimGridHalo,dimBlockHalo,0,stream_halo[i]>>>(u_d[i],u1_d[i],hpos[hp]);
        hp++;
        if(i>0 && i<num_gpus-1){
             UpDateHalo<<<dimGridHalo,dimBlockHalo,0,stream_halo[i]>>>(u_d[i],u1_d[i],hpos[hp]);
             hp++;
        }
        cudaStreamQuery(stream_halo[i]);
        UpDateInterior<<<dimGridInt,dimBlockInt,0,stream_int[i]>>>(u_d[i],u1_d[i],i);
   }
   // Exchange Halos
   cudaMemcpyPeerAsync(u_d[1],gpu[1],&u_d[0][pos[0]],gpu[0],area_size,stream_halo[0]);
   ...
   // perform I/O
   ...
   // Synchronise and update pointers
   ...
   }
}
```

Initially, we iterate over the GPUs and launch the kernels required for the halos using *stream_halo*. Note that GPUs 0 and 3 have a single halo, whilst GPUs 1 and 2 contain two halos. Then the main interior data kernels are launched, using *stream_int*. The data transfer events are then pushed into *stream_halo*, which will execute when the halos have been computed. In this manner, the data transfers proceed at the same time as the interior computation is being performed. The GPUs are then synchronized before swapping the data pointers.

EXPERIMENTAL TESTING

Initial testing is performed using data grids containing 100 million points each. At double precision this requires 0.8Gb of data per grid (1.6Gb for the whole 2 Grid simulation), and so allows a comparison to be made between single GPU and four GPU implementation using Tesla C2050 GPUs that have 3Gb of global memory. Using a sample rate of 44.1kHz, the domain size is $244m^3$.

Computation times

The 2 Grid scheme is used to compare computation times for the single GPU, basic (non-asynchronous) four GPU, and asynchronous four GPU implementations. The simulations are computed for 4,410 samples in each case, at 44.1kHz, and for both single and double precision floating-point accuracy. Table 1 shows the resulting times. The data grids are of size Nx : 960 points, Ny : 396 points, and Nz : 264 points.

TABLE 1: Computation times and speedups for double (DP) and single (SP) precision.

Setup	DP Time (sec)	Speedup	SP Time (sec)	Speedup
Single GPU	145.3	-	89.5	-
Basic four GPU	59.4	x2.4	36.1	x2.5
Asynch four GPU	48.1	x3.0	29.5	x3.0

The basic four GPU implementation only achieves a speedup of x 2.5, whilst the asynchronous version gets to x3. The grid sizes for these initial tests contain a very large Z layer (960 x 396 = 380,160 points). As six overlap halos of this size have to be transferred between GPUs at each time step, this is still a limiting factor. To test the effect of the Z layer size, the double precision simulation is performed for decreasing

sizes whilst keeping the same overall domain size of $244m^3$, ranging from 380,160 points down to 76,032 points. Figure 3 shows the effect in terms of the dimensions of the space.

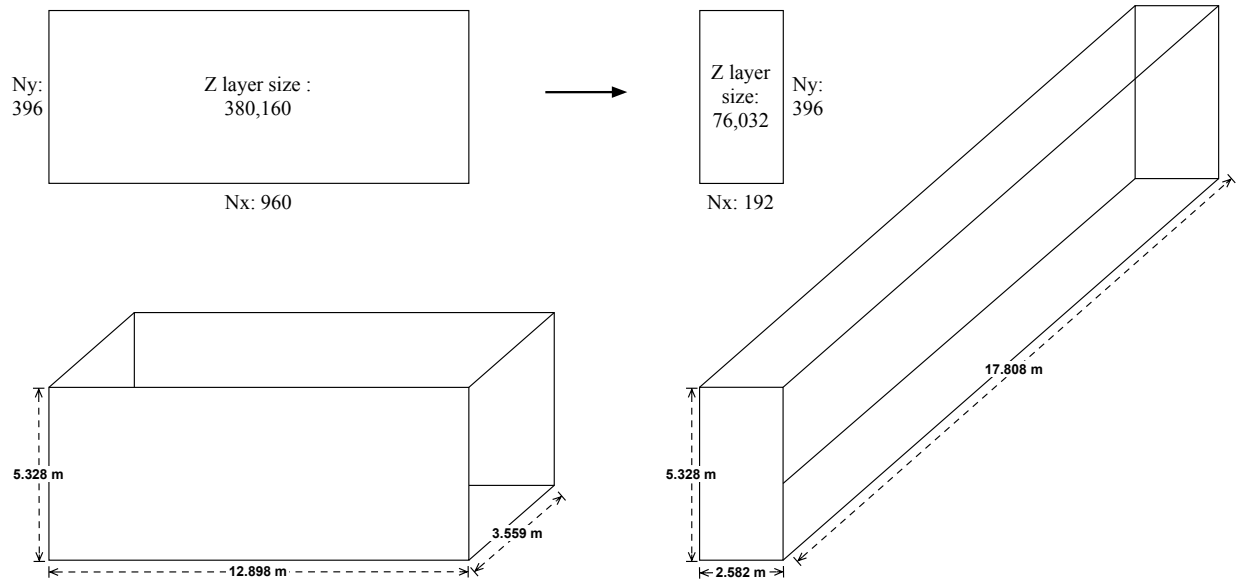

FIGURE 3: Variations in Z layer sizes for a domain of $244m^3$.

Table 2 shows the timing results. As the Z layer size decreases we get closer to the x4 scalable speedup.

TABLE 2: Effect of variation in the size of the Z layer on computation time.

Z layer size (points)	Time (sec)	Speedup over single GPU
380,160	48.1	x3.02
329,472	46.0	x3.16
278,784	45.1	x3.22
228,096	44.1	x3.29
177,408	43.4	x3.35
126,720	42.8	x3.39
76,032	41.3	x3.52

Floating-point precision

Single precision floating-point variables require 32 bits of memory compared to double precision which requires 64 bits. So, using single precision we can effectively double the size of the computation domain using the same amount of memory. There is also an additional benefit, as GPUs offer greater peak performance at single precision. However, testing on the 100 million point domain reveals stability issues when running at, or very close to, the Courant limit for the scheme. Figure 4 shows the outputs for a 40,000 time step simulation at 44.1kHz using a DC-blocked audio input and grid spacing set at the Courant limit. The single precision output (blue) is stable initially, but shows phase and amplitude differences compared to the double precision (red). After 30,000 samples, the single precision output begins to diverge, and finally becomes unstable after 40,000 samples. Backing away from the Courant limit by around 0.05% ensures stability in single precision, at the cost of introducing greater dispersion.

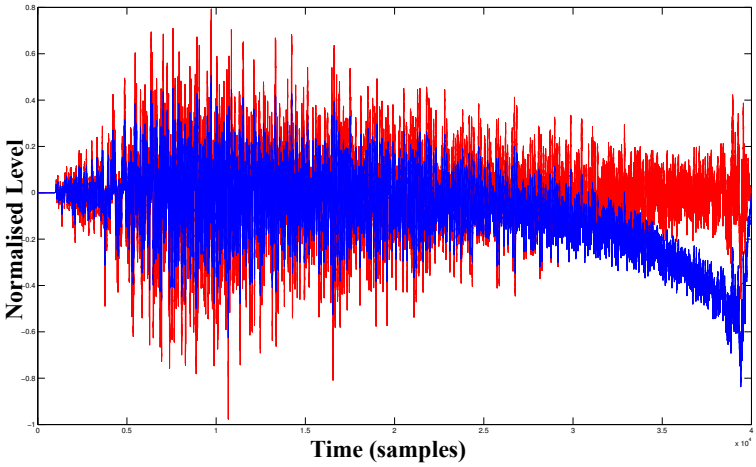

FIGURE 4: Double (red) vs Single (blue) precision at the Courant limit over 40,000 time steps.

LARGE-SCALE ACOUSTIC SIMULATIONS

Having detailed the efficiency of the asynchronous four GPU implementation, we can now consider the use of maximum available memory to perform large-scale simulations. Nvidia's Tesla GPUs come with various amounts of global memory, and so table 3 shows the maximum simulation sizes for various configurations, at a sample rate of 44.1kHz. Note that the GPUs have less available memory than is actually labelled, for example a 3Gb C2050 has a useable global memory of around 2.8Gb. Whilst the table

TABLE 3: Maximum simulation sizes in points per grid (millions) and cubic metres, at 44.1kHz.

GPU	2 Grid SP	m^3	2 Grid DP	m^3	3 Grid SP	m^3	3 Grid DP	m^3
3Gb	352	865	175	430	235	584	118	290
5Gb	595	1,461	297	729	395	983	199	490
6Gb	722	1,774	361	886	480	1,193	240	589
4 x 3Gb	1,409	3,460	702	1,722	940	2,336	473	1,160
4 x 5Gb	2,380	5,844	1,189	2,918	1,582	3,934	798	1,960
4 x 6Gb	2,889	7,096	1,444	3,546	1,920	4,775	960	2,357

shows the maximum sizes for both the 2 Grid and 3 Grid schemes, in practice this has to be reduced to allow for storage of audio output arrays and, in the four GPU case, overlap halos of variable size. Four Tesla C2050 GPUs are used for the testing here, each of which has 3Gb of global memory. Thus for the basic 2 Grid scheme at single precision we can compute simulations using 1.4 billion grid points, and a resulting simulation size of 3,350 m^3. For the 3 Grid scheme including viscosity, the grids contain just under a billion points. Table 4 shows the computation times for maximum memory simulations, running for 44,100 samples at 44.1kHz.

TABLE 4: Maximum memory computation times for 44,100 samples at 44.1kHz.

Simulation	Size (m^3)	Time (min)
2 Grid DP (double precision)	1,682	44.6
2 Grid SP (single precision)	3,350	53.1
3 Grid DP (double precision)	1,112	48.5
3 Grid SP (single precision)	2,257	52.7

CONCLUSIONS

The use of asynchronous data transfers and concurrent execution allows multiple GPUs to be used effectively to achieve near-scaleable speedups in three-dimensional FDTD schemes, typically ranging from x3 to x3.5 when using four GPUs, depending on the size of the overlap halos. By using all available memory on a four GPU compute node, we can perform virtual acoustic simulations using billions of grid points. At audio rates such as 44.1kHz, this allows the modelling of large rooms and halls, of several thousand cubic metres. Stability becomes an issue when running at single precision to maximise memory usage. Computing schemes at the Courant limit using single precision can lead to instability over time, although they may appear stable initially. Backing away from the Courant limit with a small increase in the spatial resolution resolves this behaviour.

Computation times for large-scale maximum memory simulations are around forty to fifty minutes per second at 44.1kHz, using four Tesla C2050 GPUs. Initial testing on the latest Kepler architecture GPUs shows a near two-fold speedup over the FERMI Tesla GPUs used here, and so should bring computation times down to under half an hour.

ACKNOWLEDGEMENTS

This work is supported by the European Research Council, under Grant StG-2011-279068-NESS.

REFERENCES

[1] N. Rober, U. Kaminski, and M. Masuch, "Ray acoustics using computer graphics technology", in *Proc. of the 10th Int. Conf. on Digital Audio Effects* (DAFx-07, Bordeaux, France) (2007).

[2] E. Lehmann and A. Johansson, "Diffuse reverberation model for efficient image-source simulation of room impulse responses", in *IEEE Transactions on Audio, Speech and Language Processing*, volume 18(6), 1429–1439 (2010).

[3] L. Savioja, D. Manocha, and M. Lin, "Use of GPUs in room acoustic modeling and auralization", in *Proc. Int. Symposium on Room Acoustics* (Melbourne, Australia) (2010).

[4] C. Webb and S. Bilbao, "Computing room acoustics with CUDA - 3D FDTD schemes with boundary losses and viscosity", in *Proc. of the IEEE Int. Conf. on Acoustics, Speech and Signal Processing* (Prague, Czech Republic) (2011).

[5] Nvidia, "Cuda C programming guide", CUDA toolkit documentation.[Online][Cited: 8th Jan 2013.] http://docs.nvidia.com/cuda/ (2012).

[6] J. Strikwerda, *Finite Difference Schemes and Partial Differential Equations* (Wadsworth and Brooks/-Cole Advanced Books and Software, Pacific Grove, California) (1989).

[7] P. Micikevicius, "3D finite difference computation on GPUs using CUDA", in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, 79–84 (New York, NY, USA) (2009).

[8] C. Webb and S. Bilbao, "Virtual room acoustics: A comparison of techniques for computing 3D FDTD schemes using CUDA", in *Proc. 130th Convention of the Audio Engineering Society (AES)* (London, UK) (2011).

[9] P. Micikevicius, "Multi-GPU Programming", Nvidia Cuda webinars. [Online][Cited: 6th Jan 2013.] http://developer.download.nvidia.com/CUDA/training/ (2011).