FDTD/K-DWM SIMULATION OF 3D ROOM ACOUSTICS ON GENERAL PURPOSE GRAPHICS HARDWARE USING COMPUTE UNIFIED DEVICE ARCHITECTURE (CUDA)

J Sheaffer School of Computing, Science and Engineering, the University of Salford, UK.
B Fazenda School of Computing, Science and Engineering, the University of Salford, UK.

1 INTRODUCTION

Computer-based modelling is becoming standard practice in modern room acoustics. Its applications range from prediction of design parameters, visualization of sound propagation and auralisation [1]. Traditional architectural acoustic modelling methods, such as Ray-Tracing [2] and Image-Source models [3], draw on geometrical assumptions and therefore may be confidently applied only to an explicit class of design problems. Wave-based methods, on the other hand, attempt to solve the acoustic wave equation under specific boundary conditions, and are therefore more precise in predicting sound propagation, albeit at a much higher computational cost. Common methods such as Boundary Element [4] and Finite Element [5] analyses are considered highly accurate; however they impose two major limitations: a) their computational requirements grow exponentially with frequency, thus rendering them inadequate for wide band problems; and b) traditionally, their solution correspond to a steady-state analysis which neglects discrete time-domain transient responses. Throughout the past decade much attention has been shifted towards numerical time domain methods, particularly the Finite Difference Time Domain (FDTD) [6] and the Digital Waveguide Mesh (DWM) [7]. Even as being able to provide discrete time-space approximations to the N-dimensional wave equation, these methods have two major drawbacks. First, the entire acoustic space must be modelled resulting in large computational domains, hence higher calculation times. Second, as finite difference approximations are employed, these methods exhibit dispersion errors that increase with frequency and vary with direction of propagation [8]; thus imposing a high-frequency calculation limit. Various methods for reducing these errors have been studied, most noticeably by considering non-rectangular mesh topologies [9], [10], by spatial interpolation and frequency warping [11]. One feasible trade-off is to oversample the mesh, consequently extending the high frequency limit at the expense of a considerably large computational problem. Evidently, a way to efficiently solve such large problems at reasonable calculation times would diminish the aforementioned restrictions.

As grid-based time-stepping methods are easily parallelizable, multi-core implementations for the digital waveguide mesh have been suggested in the past [12]. However, commonly available CPUs are still based on dual or quad core architectures, with octa-core processors only recently introduced. Nevertheless, graphics processing units (GPUs) employ a many-core scheme, with typical high-end consumer devices providing up to 240 cores. Harnessing the power of graphics processors for general purposes (GPGPU), is gaining high popularity in many computational sciences, with recent evidence of successful finite difference applications in electro-dynamics [13] and seismic modelling [14]. In acoustics, an implementation for a 2D waveguide mesh has been recently suggested by Southern et al. [15], and a 1D spectral method has been demonstrated by Angus and Caunce [16]. The introduction of nVidia's CUDA platform [17] provides an attractive solution for implementing naïve code on GPGPUs at a comfortable learning curve; thus allowing scientists and engineers to easily and quickly adapt their codes to run on graphics hardware.

In this paper we present a GPGPU algorithm for solving the 2^{nd} order acoustic wave equation in three dimensions using the FDTD/K-DWM method. We evaluate two different boundary formulations and suggest two methods for implementing 3D data structures on the native 2D grid provided by the GPU. The methods are benchmarked at different sampling rates and tested against a CPU reference algorithm. Emphasis is given to computational costs, memory requirements and accuracy. It is worth noting that aside from general GPU programming 'thumb-rules', mostly

described in this text, no particular optimisation is performed on the GPU code; as we wish to show that even a naïve GPU implementations can considerably improve calculation speed. This is an important point as it emphasises the fact that adapting algorithms for GPU execution is simple enough for users having even little programming experience. As one of the goals of this paper is to encourage acousticians to appreciate the advantages of recent GPU technology, we wish to show that writing GPU code using CUDA is extremely beneficial relative to the little amount of time required to learn coding for the platform.

2 FINITE DIFFERENCE METHODS FOR THE WAVE EQUATION

2.1 Propagation of Sound

Under the assumption of inviscid-adiabatic flow of energy, propagation of sound waves in air may be described using Euler's equations, namely the conservation of mass and conservation of momentum [18], given by

$$\nabla p = -\rho \frac{\partial \boldsymbol{u}}{\partial t} \tag{1}$$

$$\nabla \cdot \boldsymbol{u} = -\kappa \frac{\partial p}{\partial t} \tag{2}$$

Where ∇ and $\nabla\cdot$ are the gradient and divergence operators respectively, p is the acoustic pressure, ρ is the density of air, t is time, c is the velocity of sound, $\boldsymbol{u}$ is the particle velocity vector, and $\kappa = 1/\rho c^2$. Traditional FDTD methods involve a discretisation of equations (1) and (2); however if evaluating particle velocity is not necessary, they may be combined to obtain the 2nd order wave equation given by

$$\frac{\partial^2 p}{\partial t^2} = c^2 \nabla^2 p \tag{3}$$

Where ∇^2 is the Laplace operator.

2.2 Spatio-Temporal Discretisation

In order to establish a computer model, the domain must be sampled in space and the problem solved in discrete time steps. If the physical distance between each spatial sample along the *x* axis is denoted δx, and along the *y* and *z* axes respectively, we can form a discrete coordinate system using a set of space indices (i, j, k) defined as

$$i \triangleq \frac{x}{\delta x} \ , \ j \triangleq \frac{y}{\delta y} \ , \ \ k \triangleq \frac{z}{\delta z} \tag{4}$$

Similarly, let *n* denote the state index related to the temporal period δt by

$$n \triangleq \frac{t}{\delta t} \tag{5}$$

For simplicity we assume uniform space discretization steps on the grid, therefore $\delta x = \delta y = \delta z = \delta g$. The spatial and temporal periods are related to each other through the Courant variable λ denoting the numerical stability of the algorithm [8]. Values must adhere to the Courant criterion given by

$$\lambda = c \frac{\delta t}{\delta g} \leq \frac{1}{\sqrt{D}} \tag{6}$$

Where *D* denotes the dimensionality of the problem. In order to minimize dispersion it is customary to choose the Courant variable at its upper stability limit, namely for a 3D case $\lambda = 1/\sqrt{3}$. As previously mentioned, phase velocity errors caused by numerical dispersion are the foremost handicap of finite difference methods, which limit the accuracy of the algorithm at high frequencies. For more rigorous analyses of numerical dispersion and its effects, readers are referred to [8].

2.3 The Kirchhoff Digital Waveguide Mesh (K-DWM)

Following the conventions described in (2.2) and employing central finite differences, a discrete update equation can be derived; representing the finite difference scheme for equation (3). In this case, pressure samples are given on a 3D rectangular mesh as shown in Figure (1). The corresponding update equation is given by [19]

$$p|_{i,j,k}^{n+1} = \lambda^2\left(p|_{i-1,j,k}^{n} + p|_{i+1,j,k}^{n} + p|_{i,j-1,k}^{n} + p|_{i,j+1,k}^{n} + p|_{i,j,k-1}^{n} + p|_{i,j,k+1}^{n}\right) + 2(1-3\lambda^2)p|_{i,j,k}^{n} - p|_{i,j,k}^{n-1} \tag{7}$$

This equation is used to calculate the acoustic pressure at the different air nodes inside the discretised domain. However, at the edges of the mesh propagation should be terminated while satisfying the boundary conditions enforced on the wave equation. A simple 1D termination is often assumed at the boundaries resulting in the classical boundary update equations as formulated by Savioja et al.[20], given by

$$p|_{B}^{n+1} = (1+R)p|_{B-1}^{n} - R \cdot p|_{B}^{n-1} \tag{8}$$

Where p_B is the pressure on the boundary node, p_{B-1} is the pressure on the adjacent air node parallel to the boundary, and *R* is the acoustic reflection factor. An improved formulation of boundary conditions based on locally reactive surfaces (LRS) theory has been recently proposed by Kowalczyk and Van-Walstijn [19]. The update equations for right faces, right-back edges and the right-up-back corner of the mesh are given in equations (9), (10) and (11) respectively. Derivation of similar update equations for other types of faces, edges and corners is straightforward.

$$p|_{B}^{n+1} = \left[\begin{matrix}\lambda^2\left(2p|_{i-1,j,k}^{n} + p|_{i,j-1,k}^{n} + p|_{i,j+1,k}^{n} + p|_{i,j,k-1}^{n} + p|_{i,j,k+1}^{n}\right) \\ +2(1-3\lambda^2)p|_{B}^{n} + \left(\frac{\psi}{R} - 1\right)\end{matrix}\right] \Big/ \left(\frac{\psi}{R} + 1\right) \tag{9}$$

$$p|_{B}^{n+1} = \left[\begin{matrix}\lambda^2\left(2p|_{i-1,j,k}^{n} + 2p|_{i,j-1,k}^{n} + p|_{i,j,k-1}^{n} + p|_{i,j,k+1}^{n}\right) \\ +2(1-3\lambda^2)p|_{B}^{n} + \left(\frac{\psi}{R_x} + \frac{\psi}{R_y} - 1\right)\end{matrix}\right] \Big/ \left(\frac{\psi}{R_x} + \frac{\psi}{R_y} + 1\right) \tag{10}$$

$$p|_{B}^{n+1} = \left[\begin{matrix}\lambda^2\left(2p|_{i-1,j,k}^{n} + 2p|_{i,j-1,k}^{n} + 2p|_{i,j,k-1}^{n}\right) \\ +2(1-3\lambda^2)p|_{B}^{n} + \left(\frac{\psi}{R_x} + \frac{\psi}{R_y} + \frac{\psi}{R_z} - 1\right)\end{matrix}\right] \Big/ \left(\frac{\psi}{R_x} + \frac{\psi}{R_y} + \frac{\psi}{R_z} + 1\right) \tag{11}$$

Where R_x, R_y, R_z are the components of the acoustic reflection factor in the x, y and z directions respectively, $\psi = \rho c \lambda$ and all other variables as before. Although this method can be viewed as a finite difference scheme for the 2nd order wave equation, it can also be derived from a network of digital waveguides employing Kirchhoff variables [21], hence its name.

2.4 Computational and Memory Requirements

The computational costs (in floating point operations per cubic meter of model per second of simulation time) and memory requirements (in megabytes per cubic meter of model) for running a

3D K-DWM model in single precision arithmetic are depicted in Figure (2). Clearly the growth is exponential in both cases (observe that the y axes are given in a base-2 log scale).

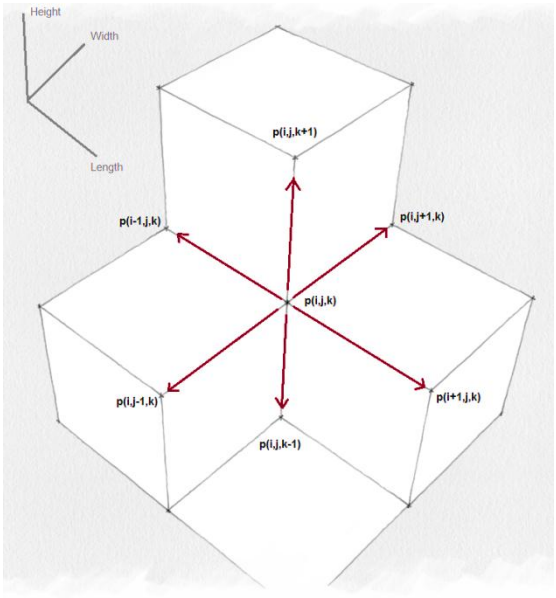

Figure 1: A spatial section of an FDTD/K-DWM grid

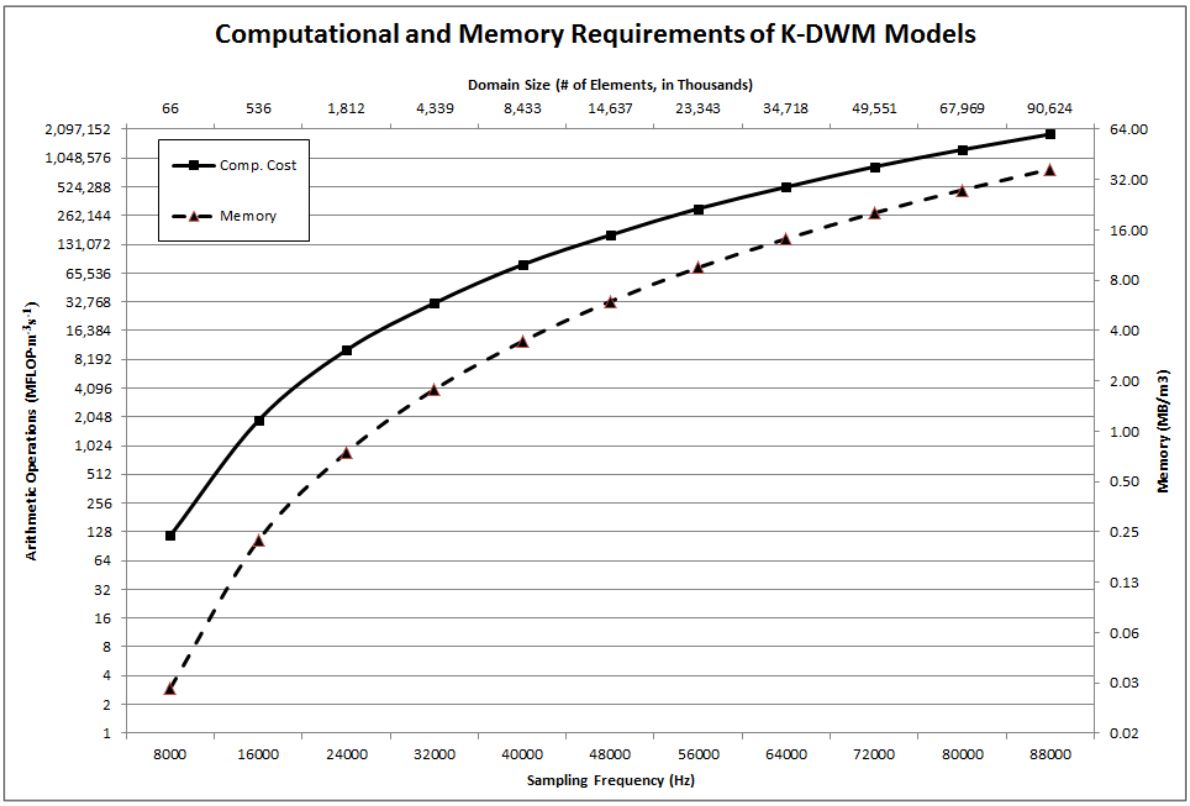

Figure 2: Computational and memory requirements of K-DWM models. Costs are given in millions of floating point operations per second per cubic meter. Memory requirements are given in MB per cubic meter. All values refer to single precision arithmetic accuracy.

3 GENERAL PURPOSE MODELLING ON GPU

3.1 Compute Unified Device Architecture (CUDA)

CUDA, or “Compute Unified Device Architecture”, is a parallel computing framework designed to be employed on nVidia® graphics hardware. One of CUDA’s main advantages is its accessibility to a wide range of programmers; as using it does not require prior knowledge or experience in developing applications for graphics hardware. Any CUDA-enabled GPU can be programmed using a high-level language termed “CUDA C”, which is essentially C with GPU extensions and some restrictions. Programmers may opt to write their code directly in CUDA-C or make use of wrappers, binders and plugins currently available for Python, Matlab and Fortran amongst others.

Figure (3) portrays a CUDA program flow for a time-iterative algorithm such as the K-DWM. First, memory is allocated on the host computer alongside any other preparations needed. Once data have been prepared it is copied onto the device’s global memory for further processing. Then, a loop is initiated on the host, which further executes a kernel for each time iteration. Once the entire computation cycle is completed, the required data is copied back to the host memory for further post-processing and output. In this model, space computations are performed in parallel by the kernel on the device whereas state computations are serially controlled by the host. This matches the requirements of finite difference schemes in which space variables, such as pressure or particle velocity, are dependent on their state counterparts (i.e. their previous time values).

3.2 Thread Hierarchy

In CUDA, a prototype function termed ‘kernel’ is defined for each series of computations. Whilst calculations inside a kernel are performed serially, a launched kernel activates a group of threads which are executed in parallel. A kernel can therefore be seen as a set of instructions and an active thread would correspond to that set of instructions performed on a designated part of data stored in memory. Since threads are executed in parallel, a larger amount of data would be processed at an instance in comparison to a traditional CPU implementation. In an ideal parallel environment, the number of possible active threads would be comparable to the size of the data, i.e. one thread per data point. However, this is not possible due to technological constraints and in reality, the amount of active threads is explicitly limited by hardware architecture; hence requiring a mechanism to schedule and synchronize their operation. The amount of threads executed at a time can be seen from both logical and physical perspectives.

From the programmer’s viewpoint, threads are grouped into ‘thread-blocks’ which are further clustered in grids. When a kernel is launched, data in a grid is processed block by block, each invoking its own group of threads; each having a unique identifier code. Nonetheless, in order to write robust and optimized code, one should also consider the process from a physical point of view. The CUDA platform is predicated on an architecture termed SIMT, or “Single-Instruction, Multiple-Thread”, which is employed by a scalable array of Streaming Multiprocessors. Once a kernel has been launched, active threads in a block execute concurrently on a multiprocessor. The multiprocessor is therefore in charge of administering thread execution; a process which is done in groups of 32 threads, or simply “warps”. Whilst warps may execute independently, their operation is controlled by a warp scheduler on the multiprocessor. An important point in case is that warps are designed to execute a single common instruction, so maximum computation efficiency is achieved when all threads in a warp follow a unanimous operation scheme. Divergence and branching due to thread-distinctive instructions results in serial execution, which in turn reduces efficiency.

3.3 Memory Hierarchy

Memory management is a key aspect in GPGPU development. In traditional CPU programming paradigms computational operations are thought of as 'expensive' whereas memory access is 'cheap'. That is, programmers prefer to pre-calculate any elements that are intended for reusability and store them in memory until needed. Conversely, un-cached GPU memory usage may exhibit a latency of up to 400-600 clock cycles; hence memory transactions are considerably more time-expensive than most typical computations. It is important to identify four types of memory; *Host* memory is the term used to describe the RAM provided by the hosting computer, *Global* memory is the RAM available on the GPU, *Shared* memory is a specifically dedicated amount of cached memory that can be shared between threads inside the same block, and *local* memory is a small amount of cached per-thread memory.

It is often necessary to control memory usage on a thread or warp level to ensure maximum performance. Memory coalescing is a common technique to minimize memory latency for a given set of instructions, allowing access to 16 words in a single memory transaction. In general, three conditions must be met [22]: 1. Data should be stored in 4, 8 or 16-byte words; 2. All accessed words must reside in the same memory segment; and 3. Words must be sequentially accessed by threads in a half-warp. Nevertheless, as CUDA technology progresses memory coalescing is becoming less and less of a concern, with newer "Compute Capability 2.x"[1] devices providing cached access even to global memory.

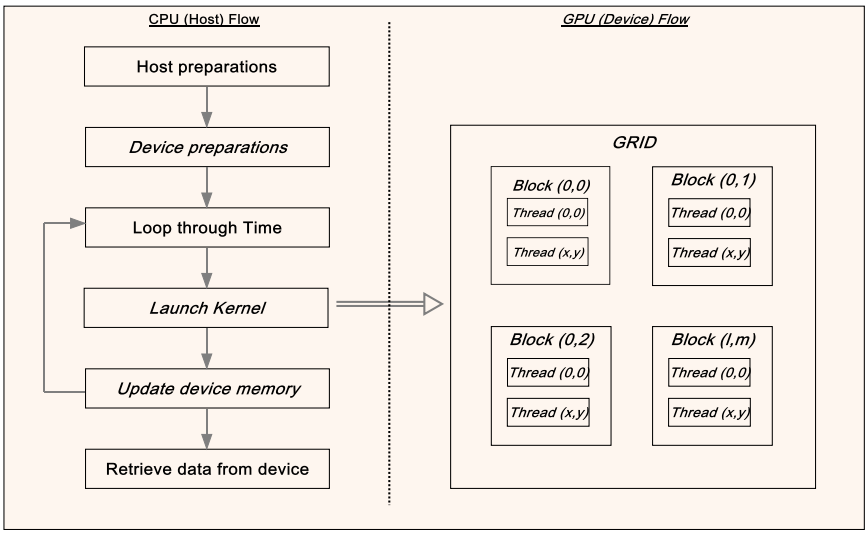

Figure 3: Program flow of a time-iterative algorithm in CUDA. Operations performed on the GPU are marked in italics.

3.4 Implementation of 3D Data Structures

Whilst thread blocks may be one, two or three dimensional, grids may have only up to two dimensions. This imposes a difficulty in orchestrating ties between thread indices and data indices, as three-dimensional arrays cannot be naïvely represented in a respective CUDA grid, at least for the time being. Nevertheless, it is not a requirement for block and grid dimensions to exactly match their data counterparts, as long as correct index translation is exercised. This entails a computationally effective method to compute 3D data using a 2D threading system. In this section we suggest two methods to overcome this obstacle. To avoid confusion, the term "data index" will be used to identify an arbitrary point in one processed array, and the term "thread index" will be used to identify an arbitrary thread within a block or grid.

[1] Compute Capability is the term used by nVidia to identify the different architectures supported by their GPUs.

Let us consider a computational domain with N_i, N_j, N_k denoting the discrete dimensions of the array, and *(i,j,k)* denoting the data index of a point in the array. Similarly, let Dg_x, Dg_y and Db_x, Db_y denote the dimensions of the GPU grid and thread-block respectively. The goal is to map the problem in such way that a given thread with a 2D index will point to its corresponding 3D indexed data point. It should be noted that it is not necessary for the mapping scheme to follow a logical pattern, as long as indices are repeatedly translated correctly.

The straightforward way would be to slice the volume, for example along its k-dimension, resulting in a group of 2D slices whose amount corresponds to N_k. The problem may then be computed slice-by-slice, where each slice is considered an individual 2D array. This can be rather simply implemented in finite difference methods, as calculated values depend only on their predecessors; hence read-after-write hazards are naturally avoided. With this method it is possible to create a logical 1:1 map between thread and data indices by setting $Dg_x = N_i$ and $Dg_y = N_j$, as shown in Figure (4a). In other words, grid dimensions are designed to match data dimensions of each k-slice. This method will be further referred to in this text as the "Slicing Method".

There are two main advantages to using this method: 1) it is logical and therefore easy to implement; and 2) as a 1:1 map is employed no computations of index translation are required. Nevertheless, it has a major handicap as a considerable amount of parallelism is lost due to serial computation of the k-slices. It is worth noting that variations in performance are observed when slicing along different dimensions. This is because 3D data structures are stored in linear memory, and access patterns vary with different stride orderings which in turn affect memory coalescing. The sliced dimension should therefore be chosen based on the highest stride variable. For example, if the amount of memory offset is calculated by

$$iN_jN_k + jN_k + k \tag{12}$$

Then slicing should be applied by looping over the i-dimension.

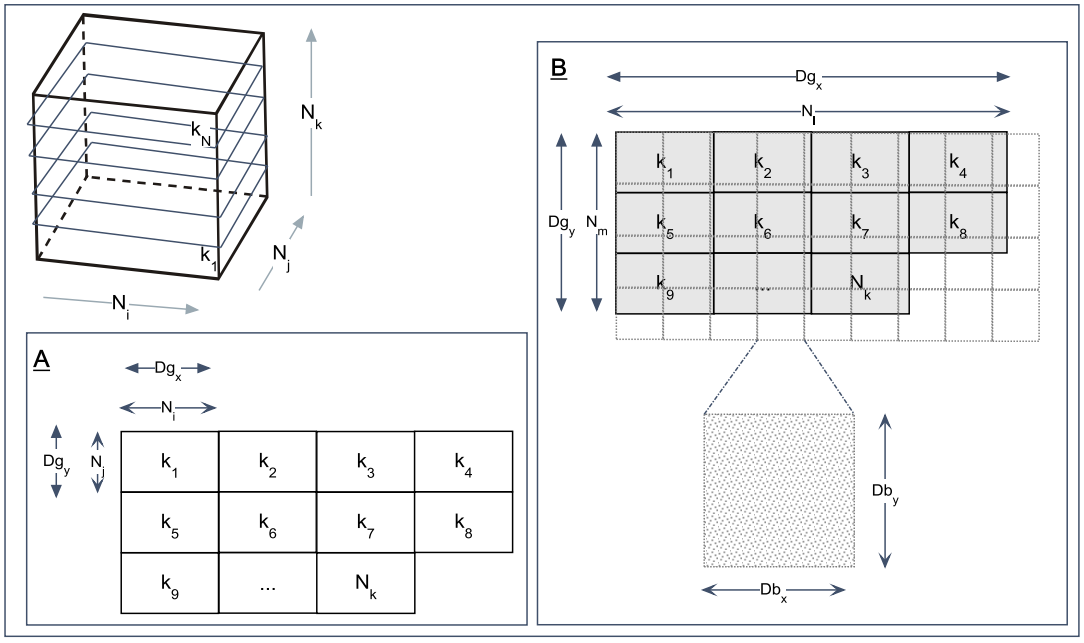

Figure 4: 3D Data Structure sliced along its k-dimension. (A) Slicing Method, where the data is processed slice-by-slice; (B) Tiling Method, where the data is processed as a 2D plane in a single pass.

An alternative approach is to flatten the volume into a tiled plane [23], where each tile would correspond to a specific slice of the k-dimension as shown in Figure (4b). Essentially, the 3D volume is now reduced to a 2D plane; however the natural data indices *(i,j,k)* should be translated

to their 2D equivalents. Let S_l and S_m denote the number of slices tiled on the width and height of the 2D plane, respectively. To avoid confusion we shall use the indexing notation *(l,m)* to indicate the location of an arbitrary data point on the 2D plane. The number of data points on the *l* and *m* dimensions can be calculated from $N_l = S_l N_i$ and $N_m = S_m N_j$ respectively. Now, a 1:1 map can be established between thread and data indices, such that $Dg_x = N_l$ and $Dg_y = N_m$. Once a kernel is invoked and the *(l,m)* thread indices are given, their corresponding data indices *(i,j,k)* may be obtained from

$$i = l \; mod \; N_i \tag{13}$$

$$j = m \; mod \; N_j \tag{14}$$

$$k = S_m \left\lfloor \frac{l}{N_i} \right\rfloor + \left\lfloor \frac{m}{N_j} \right\rfloor \tag{15}$$

It should be borne in mind that since block dimensions would not normally match the dimensions of each tile (i.e. $Db_x \neq N_i$ and $Db_y \neq N_j$), some threads may point to values outside the bounds of the data array. Such threads should be identified and terminated. Also worth noting is that equations (13)-(15) can be altered to represent different memory or slicing ordering, depending on the programming environment or memory storage method. The main advantage of this method is that the entire computational domain can be calculated in a single pass per time sample hence maximizing the parallel capabilities of the GPU. However, as seen from equations (13)-(15) index translation should be performed for each individual thread. Moreover, division and modulo are the most computationally expensive arithmetic operations for the GPU [22], requiring up to 16 clock cycles per operation. As such, we propose pre-calculating the quantities $(1/N_i, 1/N_j)$ and using built-in integer conversion functions to perform flooring and find the quotient, in order to improve efficiency.

4 METHODOLOGY

A computer model of a typical small room is used to evaluate the benefits of GPGPU implementations. Nevertheless, results are also shown in terms of computational problem size, and can therefore be induced on any room volume with respect to different sampling rates.

4.1 Room Description

We have chosen to model a 28 cubic meters room, with a length of 3.82m, width of 2.7m, and height of 2.72m, for which we have acoustic measurements. The floor, walls and ceiling are made of 30cm reinforced concrete, with a standard size steel door and a single 12mm PVB laminated window. Such room represents an idealised case of a rigid-reflective construction, and therefore serves as a standard reference for our models. The room was measured at 48,000 Hz using a 5.9 seconds weighted sweep, with a studio loudspeaker and a pressure transducer situated at opposite tri-corners. Since the investigation of accuracy will be limited to lower frequency modelling, the loudspeaker radiation may be adequately assumed omni-directional.

4.2 Computer Model

Based on the details presented above, we have created computer models of the room, for each of the methods presented in this paper. We have chosen to write our codes directly in CUDA-C, in order to avoid any performance bottlenecks that may be introduced by third party wrappers. The reference algorithm is implemented in plain C, in order to maximize the compatibility between the CPU and GPU programs, which have both been compiled with nVidia's NVCC compiler. The room was modelled using the aforementioned methods, with uniform frequency independent boundary conditions analogous to a real acoustic impedance of $Z_w \approx 80{,}000 \; rayl$. The chosen source function is a Gaussian pulse, band-limited to 1/8th of the sampling frequency, and injected into the mesh as a filtered 'soft-source' [21].

4.3 Technical Specifications

Models are executed on an AMD Opteron 2210 based SUN workstation with 8GB of RAM, running Windows XP professional x64 edition. The computer is attached to an nVidia Tesla S1070 computing system, featuring four Tesla T10 GPGPUs with a total of 16GB of RAM, and 4.14 TFLOP/s computational capabilities. The GPU programs are restricted to make use of only one of the T10 GPUs, allowing for results to be analogous to more widely accessible hardware, such as the Tesla C1060 or the Geforce GTX285/295. All models are tested using both 'tiled' and 'sliced' 3D data structures, using both types of boundary formulations, in single precision accuracy. The CPU reference program is a single-core implementation running on the same hosting computer as the GPU programs. The CPU and GPU algorithms were coded according to their respective programming paradigms; however no special optimisation was performed aside from the basic thumb rules provided in this document. Special care was given to the way the two algorithms treat coefficients and access multidimensional array structures. Nevertheless, no changes were made to the data structure in order to improve memory coalescing, since as previously mentioned, with new device architectures this is becoming less crucial.

5 RESULTS

A comparison of model computation times (in minutes) is provided in Figure (5). It is evident that significant accelleration can be achieved even for naïve code, with speed-up factors ranging between x20 to x30, depending on boundary formulation and data access structure. The average speed-up factor for models having at least 1 million elements, is x3.8 for the Sliced method and x25.5 for the Tiled method.

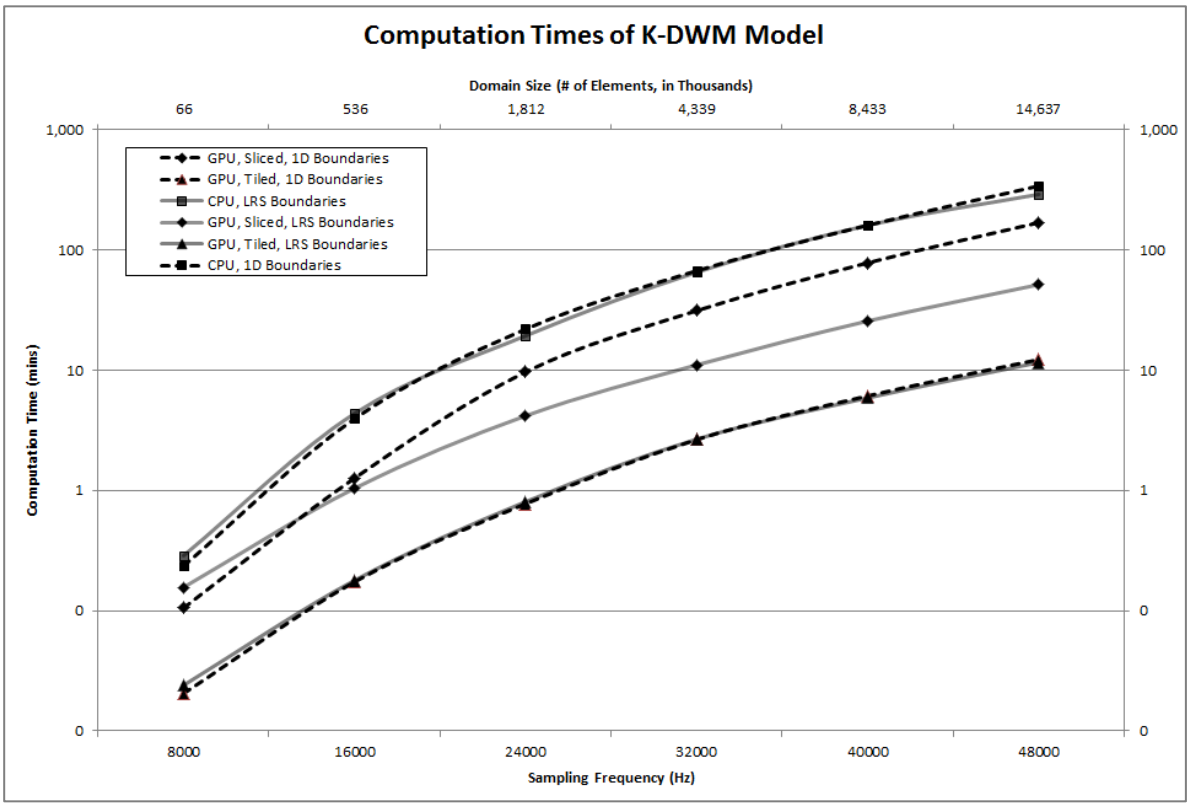

Figure 5: Benchmark of different devices, boundary formulations and 3D data structures.

It can also be seen that for the CPU and GPU-Tiled algorithms there is no significant difference between the two boundary formulations. However, the GPU-Sliced algorithm shows substantial

divergence in computation time between the two boundary formulations, for models having over ca. 250,000 elements. Although we have benchmarked up to a sampling rate of 48,000 Hz, the Tiled GPU method was further tested in order to demonstrate the potential of GPGPU computing. Using this method we were able to model the room at 88,200 Hz (over 90 million elements) in a computation time of 2.32 hours, which is reasonable for most scientific applications.

An accuracy comparison is provided in figures (6) and (7). As expected the K-DWM model follows measured results with slight differences in level and peak/trough depth; which may well be a result of differences in modelled vs. measured surface impedances. It is also clear that there are no differences between the two boundary formulations (mostly due to the fact that only reflective surfaces were modelled), and that the differences between the CPU and GPU algorithms are negligible.

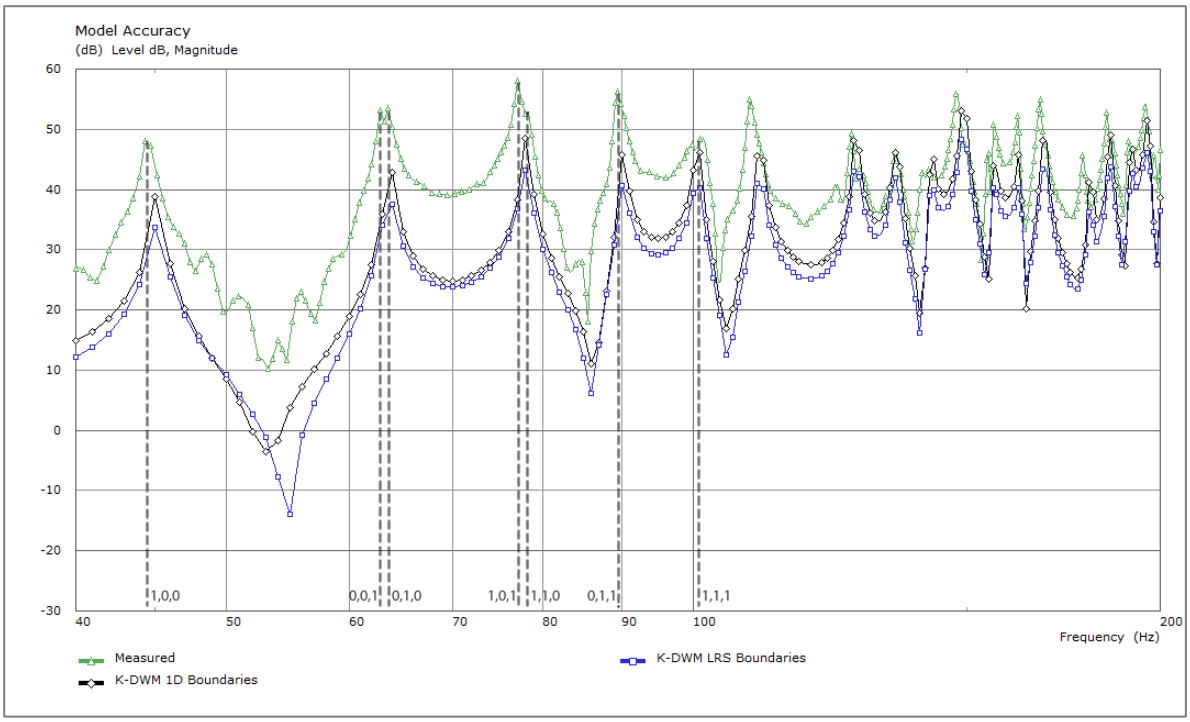

Figure 6: Low frequency comparison of modelled vs. measured response. Analytically calculated first order modes are indicated in dashed lines.

6 CONCLUSIONS

In this study we have shown that the K-DWM method is well suited for parallel computation. Substantial acceleration in computation time can be achieved even with naïve GPU code; which can be designed with simple understanding of the GPU programming paradigm. We have suggested and benchmarked two methods for processing 3D data structures in CUDA, which have clearly indicated that the Tiled method is superior in terms of computation time; whereas the performance of the Sliced method is more sensitive to variations in data structures.

Although benchmarks were conducted on a professional GPGPU, for this research its capabilities were limited to processing power equivalent to a high-end consumer GPUs; hence showing that good results can be achieved even with lower-cost hardware. Nevertheless, we expect a significant improvement had these tests been performed using the entire computational power of the GPU. Moreover, with new GPGPU devices on the market and the introduction of the new Fermi architecture, even lower calculation times can be expected for naïve code; mostly due to hardware-level memory caching.

In sum, we hope to encourage acousticians to embrace GPGPU computing as a simple, robust, and cost effective solution for accelerating finite difference algorithms; which in turn will allow for the modelling of rooms in higher sampling rates, hence lower dispersion errors.

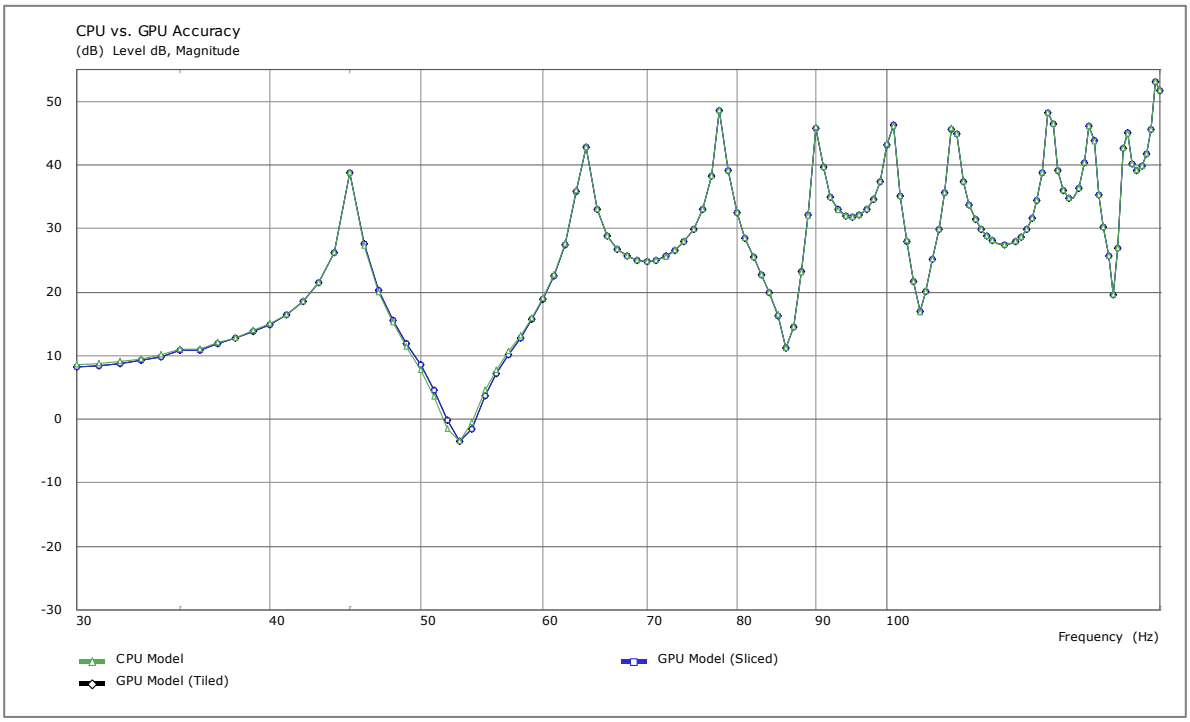

Figure 7: Accuracy comparison of the CPU and two GPU algorithms.

ACKNOWLEDGEMENTS

The authors would like to thank Jamie Angus and Eldad Klaiman for the insightful discussions on GPGPU computing, and John O'Hare for his technical assistance with the Tesla systems at the University of Salford.

REFERENCES

[1] M. Kleiner, B. I. Dalenbäck, and P. Svensson, "Auralization-an overview," *Journal-Audio Engineering Society*, vol. 41, pp. 861–861, 1993.

[2] A. Kulowski, "Algorithmic representation of the ray tracing technique," *Applied Acoustics*, vol. 18, no. 6, pp. 449–469, 1985.

[3] J. B. Allen and D. A. Berkley, "Image method for efficiently simulating small-room acoustics," *J. Acoust. Soc. Am*, vol. 65, no. 4, pp. 943–950, 1979.

[4] S. Kirkup, *The Boundary Element Method in Acoustics*. Integrated Sound Software, 1998.

[5] M. J. Crocker, *Handbook of Acoustics*, 1st ed. Wiley-Interscience, 1998.

[6] D. Botteldooren, "Finite-difference time-domain simulation of low-frequency room acoustic problems," *The Journal of the Acoustical Society of America*, vol. 98, no. 6, pp. 3302-3308, Dec. 1995.

[7] S. A. Van Duyne and J. O. Smith, "Physical modeling with the 2-D digital waveguide mesh," in *Proceedings of the International Computer Music Conference*, pp. 40–40, 1993.

[8] S. D. Bilbao, *Wave and scattering methods for numerical simulation*. Wiley, 2004.

[9] L. Savioja and V. Valimaki, "Reduction of the dispersion error in the triangular digital waveguide mesh

using frequency warping," *IEEE Signal Processing Letters*, vol. 6, no. 3, 1999.

[10] S. A. Van Duyne and J. O. Smith III, "The tetrahedral digital waveguide mesh," in *IEEE ASSP Workshop on Applications of Signal Processing to Audio and Acoustics, 1995.*, pp. 234–237, 1995.

[11] L. Savioja and V. Valimaki, "Interpolated 3-D digital waveguide mesh with frequency warping," in *IEEE INTERNATIONAL CONFERENCE ON ACOUSTICS SPEECH AND SIGNAL PROCESSING*, vol. 5, 2001.

[12] G. Campos and S. Barros, "The Meshotron: A Network of Specialized Hardware Units for 3-D Digital Waveguide Mesh Acoustic Model Parallelization," in *Audio Engineering Society Convention 128$}$*.

[13] S. Adams, J. Payne, and R. Boppana, "Finite difference time domain (FDTD) simulations using graphics processors."

[14] P. Micikevicius, "3D finite difference computation on GPUs using CUDA," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pp. 79–84, 2009.

[15] A. Southern, D. Murphy, G. Campos, and P. Dias, "Finite Difference Room Acoustic Modelling on a General Purpose Graphics Processing Unit," in *Audio Engineering Society Convention 128$}$*.

[16] J. A. Angus and A. Caunce, "A GPGPU Approach to Improved Acoustic Finite Difference Time Domain Calculations."

[17] C. Nvidia, "Compute Unified Device Architecture Programming Guide," *NVIDIA: Santa Clara, CA*, 2007.

[18] H. Kuttruff, *Room Acoustics*, 5th ed. Taylor & Francis, 2009.

[19] K. Kowalczyk and M. van Walstijn, "Formulation of Locally Reacting Surfaces in FDTD/K-DWM Modelling of Acoustic Spaces," *Acta Acustica united with Acustica*, vol. 94, pp. 891-906, Nov. 2008.

[20] L. Savioja, T. Rinne, and T. Takala, "Simulation of room acoustics with a 3-D finite difference mesh."

[21] M. Karjalainen and C. Erkut, "Digital waveguides versus finite difference structures: Equivalence and mixed modeling," *EURASIP Journal on Applied Signal Processing*, pp. 978–989, 2004.

[22] C. NVIDIA, *Best Practices Guide Version 3.0. nVidia Corporation, 2701 San Tomas Expressway Santa Clara*. 2010.

[23] M. Pharr and R. Fernando, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, 2005.