Efficient GPU-Based Solver for Acoustic Wave Equation

Ravish Mehra [1] Nikunj Raghuvanshi [1,2] Ming C. Lin [1] Dinesh Manocha [1]

[1] UNC at Chapel Hill [2] Microsoft Research

Abstract—We present an efficient algorithm to solve the acoustic wave equation that is used to model the propagation of sound waves through a material medium. Our approach assumes that the speed of sound is constant in the medium and computes an adaptive rectangular decomposition(ARD) of the environment. We map the algorithm to many-core GPU architectures by performing discrete cosine transforms (DCTs) inside each rectangular volume along with a sixth order interface handling at the boundaries. The entire solution to the second order PDE is computed on the GPU and we highlight many techniques to accelerate its performance by exploiting the features of GPU architectures. We highlight the performance of our algorithm in terms of computing impulse responses and sound propagation in complex 3D models. In practice, we observe a performance gain of more than 500X over finite-difference time-domain(FDTD) methods. The use of GPUs also results in almost one order of magnitude improvement over CPU-based ARD algorithms. To the best of our knowledge, this is the fastest method for solving the acoustic wave equation.

Index Terms—Computational acoustics, GPU-based algorithms, adaptive decomposition.

1 Introduction

Computational modeling and simulation of acoustic spaces is fundamental in many scientific and engineering applications [**?**]. The demands vary widely, from interactive simulation in computer games and virtual reality, to highly accurate offline room acoustic computations for musical halls and other man-made architectural or computer-aided design structures. Acoustic spaces may include large multi-room spaces or aircraft cabins with complex geometric shape and material properties, as well as outdoor spaces corresponding to urban models and open landscapes. For example, airplane and jet manufacturers deploy large noise engineering laboratories for measurements related to aircraft noise [**?**]. Other designers of man-made structures such as urban layouts, habitation [**?**], and automobile manufacturers [**?**] utilize acoustic simulation technologies for improving design and reducing needs for physical prototypes.

Computational acoustics has been an area of active research and has developed in conjunction with diverse fields, such as seismology, geophysics, meteorology, etc. for almost half a century. Nevertheless, achieving good acoustics in large complex structures remains a major computational challenge [**?**]. The most common approach to acoustic simulation is a two-stage process: the computation of impulse responses (IRs) representing an acoustic space, and the convolution of the impulse responses with dry (i.e. anechoically recorded or synthetically generated) source signals. The IR computation relies on an accurate calculation for modeling the time-varying spatial sound field. The sensation of sound is due to small variations in air pressure and the variations are governed by the three-dimensional *wave equation* which relates the temporal and spatial derivatives of the pressure field [31].

Computational Challenges: One of the key challenges in acoustic simulation are the computational requirements of an accurate solver. Depending on the particular method used (e.g. finite element or finite difference methods), the spatial discretization to solve the wave equation needs $6-10$ nodes per wavelength in order to resolve the frequencies faithfully [31]. If the entire frequency range of human hearing needs be simulated (i.e. up to $22kHz$), then the spacing between the nodes would need to be $1.5-2.5$mm. As a result, a cubic meter of acoustic space needs to be filled with $64-300$ million nodes and the complexity increases proportionally with the volume of the acoustic space. As a result, prior numerical solvers for the acoustic wave equation are limited to rather simple or small spaces and are often regarded as time-consuming. Most current acoustic simulation systems use geometric acoustic techniques, which are only accurate for higher frequencies and early reflections and may not be able to model the diffraction effects.

Main Results: We present a GPU-based algorithm for numerically solving the wave equation that performs adaptive rectangular decomposition (ARD) of the acoustic space. Our approach assumes a homogeneous medium in which the speed of sound is constant. Our formulation based on ARD results in no dispersion errors inside the rectangular cells, as compared to prior methods that are based on FDTD (finite-difference time domain) methods. We exploit the computational capabilities of many-core GPUs to accelerate the computations, by mapping all the component of the algorithm to current GPU architectures by using a high number of par-

allel threads. Moreover, we take into account the memory hierarchies of current GPUs to design appropriate methods to obtain high throughput. The number of cells used in ARD increases as the third power of frequency and our algorithm is able to obtain a speedup of over 500X over FDTD methods on current high-end GPUs (e.g. NVIDIA Quadro FX 5800), as long as the entire decomposition can fit into GPU memory. Moreover, we demonstrate that it is possible to effectively parallelize all stages of such a simulator, including boundary layer treatments, on current GPU architectures. In particular, ours is the first algorithm that can run simulations on scenes with volumes in the range of 14,000m^3 and generate impulse responses on 1kHz sources for auralization within 10 minutes on a desktop computer.

We analyze the performance of our algorithm and show that its various components map well to the current GPU architectures and are able to exploit the computational capabilities of high number of cores. Furthermore, GPU-based ARD algorithm is about an order of magnitude faster than CPU-based ARD algorithms.

Organization: The rest of the paper is organized in the following manner. We give a brief overview of prior work on different solvers for acoustic wave equations and GPU-based numerical algorithms in Section 2. We describe the adaptive rectangular decomposition (ARD) algorithm in Section 3 and highlight its benefits over prior acoustic solvers. Section 4 describes our GPU-based algorithm to solve the acoustic wave equation based on ARD. We describe our implementation in Section 5 and analyze its performance on different benchmarks in Section 6.

2 Related Work

Wave Equation The physics for room acoustics, as well as many other areas, can be described with good accuracy by the well known Wave Equation in the time-domain, which we will henceforth refer to simply as the Wave Equation –

$$\frac{\partial^2 p}{\partial t^2} - c^2 \nabla^2 p = f(\mathbf{x}, t). \quad (1)$$

The Wave Equation models sound waves as a time-varying pressure field, $p(\mathbf{x}, t)$. While the speed of sound in air (denoted c) exhibits slight fluctuations within a room due to changes in temperature and humidity, these can be ignored to good accuracy for most cases [18], as we do in this paper. We chose a value of $c = 340ms^{-1}$ corresponding to dry air at 20 degrees centigrade. Volume sound sources in the scene are modeled by the forcing field denoted $f(\mathbf{x}, t)$ on the RHS in the Equation 1. The operator $\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}$ is the Laplacian in 3D. The Wave Equation succinctly captures wave phenomena such as interference and diffraction that are observed in reality.

Numerical solvers for the Wave Equation Accurate high-frequency wave propagation is a very challenging computational problem because the smallest wavelengths govern the grid resolution and the scene can be thousands of wavelengths long in each dimension [39]. There is a large body of existing work on solving the Wave Equation developed over the past few decades. These methods may be roughly classified into Finite Element Methods (FEM) [38], [34], Boundary Element Method (BEM) [7], Finite Difference Time Domain (FDTD) [39], [32] and spectral methods [6]. Besides acoustic spaces, the underlying numerical methods are also useful for seismic forward acoustic modeling [22].

FEM solves for the field on an unstructured volumetric mesh, which works well for conforming to the (possibly complex) shape of the domain boundary. Mesh quality is critical to the quality of the solution and generating meshes of good quality is a central concern with FEM methods. BEM, achieves a one-dimension reduction, requiring only a discretization of the boundary of the domain. The field in the interior of the domain is expressed in terms of a boundary integral formulation. FEM and BEM have traditionally been employed mainly for the steady-state wave equation, as opposed to the full time-domain Wave Equation, with FEM applied mainly to interior and BEM to exterior scattering problems [16].

In contrast, the FDTD method was explicitly designed for solving the time-domain Wave Equation by Yee [37]. FDTD is an established technique for electromagnetic simulation [32] and seismic forward modelling [8], [14]. The FDTD method for room acoustics solves for the time-dependent pressure field on a Cartesian grid by making discrete approximations of the spatial derivative operators and using an explicit time-stepping scheme. Initial work was limited to small scenes in 2D due to computational limitations[4], [5]. Recently, FDTD has been applied to medium-sized scenes in 3D for room acoustic computations[29], [28], [30]. The implementation can take days of computation on a small cluster for medium-sized 3D scenes.

Spectral techniques achieve much higher accuracy than FEM/BEM/FDTD by expanding the field in terms of global basis functions [6]. In the context of room acoustics, the Pseudo-Spectral Time Domain algorithm [20] has been proposed in the past. However, temporal discretization errors are still present due to explicit time-stepping.

Geometric methods for the Wave Equation In the limit of infinite frequency, the Wave Equation can be expressed in terms of the geometric approximation (GA) – expressing wave propagation as rays of energy quanta. The history of GA methods for acoustics goes back roughly four decades [17], [1]. Most present-day room acoustics software use GA [27].

2.1 GPU based algorithms

GPUs have been widely used to accelerate many scientific, geometric, database and imaging computations [10], [23]. These include numerical algorithms for FFTs [11], [12] and Linear Algebra Computations [36], [3] on the

GPU. Currently GPUs are being used to solve different PDEs that arise in solving the electromagnetic wave equation [15], the Navier-Stokes equation [33], FEM simulation [19] and many other scientific applications, including seismic simulations [22], fluid dynamics [35], molecular dynamics [2], weather forecasting [21], etc. on a desktop machine.

3 Adaptive Rectangular Decomposition

In this section, we give an overview of Adaptive Rectangular Decomposition (ARD) solver [25] and highlight its benefits over prior solvers for the acoustic wave equation for uniform medium. Our GPU-based wave equation solver is based upon the ARD solver.

3.1 Background

Numerical errors in wave simulators arise from the discrete approximation of the differential operators in time and space. For finite-difference methods such as FDTD, these errors manifest as numerical dispersion – all frequencies don't travel with the same speed leading to accumulative errors that eventually destroy the waveform being propagated. The ARD technique avoids such dispersion errors by decomposing the scene into non-overlapping *rectangular* partitions. Within each partition, the field is expanded in a spectral basis. Owing to the rectangular shape, the basis can be chosen to satisfy the Wave Equation directly. Assuming sound-hard walls for the rectangles, the basis functions turn out to be Cosine functions. Mathematically, the pressure field $p(x,y,z,t)$ on the rectangular region $[0,l_x]\times[0,l_y]\times[0,l_z]$ is expressed as –

$$p(x,y,z,t)=\sum_{i=(i_x,i_y,i_z)} m_i(t)\,\Phi_i(x,y,z), \qquad (2)$$

where m_i are the time-varying spectral (mode) coefficients to solve, and Φ_i are the basis functions, given by –

$$\Phi_i(x,y,z)=\cos\left(\frac{\pi i_x}{l_x}x\right)\cos\left(\frac{\pi i_y}{l_y}y\right)\cos\left(\frac{\pi i_z}{l_z}z\right).$$

The global solution is composed by coupling the interior solutions via sixth-order accurate, finite-difference transmission operators at the artificial interfaces. These operators are local to *reduce computation* and they can lead to some small amount of low-amplitude, fictitious reflections at the interfaces.

Due to the accuracy gained by solving the wave equation analytically, ARD can propagate sounds for distances equal to thousands of wavelengths on very coarse grids close to the Nyquist limit without destroying the waveform. On the other hand, FDTD requires much finer grids and is thus slower. Both FDTD and ARD operate on a Cartesian grid in the volume of the scene – the crucial parameter being the number of samples per wavelength, s. The grid cell size is thus given by $h=\lambda/s$ where λ is the wavelength. To model absorbing scene boundary, Perfectly Matched Layer (PML) [26] absorbing layers are employed.

	s	# cells $N=V/h^3$	# steps t/dt	# FLOPS per cell	Total cost (TeraFLOPS)
FDTD	10	254 Million	17000	55	**237.5**
ARD	2.6	4.5 Million	4500	120	**2.4**

TABLE 1
FLOPS comparison of FDTD vs ARD on a scene of volume $V=10,000m^3$ with maximum simulation frequency $v_{max}=1kHz$ for the duration $t=1$ sec. Theoretically, ARD which uses s=10 is nearly hundred times efficient than FDTD($s=2.6$) on account of using a much coarser grid.

3.2 Accuracy and Computational Aspects

In this section, we highlight some benefits of ARD over prior methods. A direct theoretical comparison of performance of FDTD vs ARD for the same amount of error is difficult since both techniques introduce different kinds of errors. However, it is possible to compare them by assuming *tolerable* errors with both the techniques. Since the final goal in room acoustics is to auralize the sounds to a human listener, it is natural to set these error tolerances based on their auditory perceivability, as we briefly discuss below.

For FDTD, we assume $s=10$, as is common practice in FDTD applied to room acoustics [29]. Using $s=2.6$ with ARD [25], the fictitious reflection errors can be kept at a low level of $-40dB$. This means that for a complex scene with many interfaces, the overall loudness of the fictitious interface errors is $40dB$ below the level of the ambient sound field, rendering them imperceptible as demonstrated with auralizations in [24]. Therefore, tolerable errors are achieved with ARD with a much coarser sampling ($s=2.6$) compared to FDTD ($s=10$).

Computational expenditure: Table 1 shows a theoretical performance comparison of FDTD and ARD. For illustrative purposes, we consider a point source that emits a broadband Gaussian pulse band-limited to a frequency of $\nu=1kHz$, corresponding to a wavelength of $\lambda=c/\nu=34cm$. We further assume that the scene has an air volume of $V=10,000m^3$ and a 1 second long simulation is performed. The number of cells with either technique is given by $N=V/h^3$. The simulation time-step is restricted by the CFL condition $dt\le h/c\sqrt{3}$, smaller cell sizes require proportionally smaller time-steps. The performance thus scales as ν^4, ν being the maximum simulated frequency.

The update cost for sixth-order accurate FDTD in 3D is about 40 FLOPS per cell (plus the cost of boundary treatment (PML), which is the same as for ARD). The total cost for ARD can be broken down as: DCT and IDCT[1] – $4NlgN$, mode update – 9N, interface handling – $300\times6N^{2/3}$ and boundary treatment (PML) – $390\times6N^{2/3}$. The $6N^{2/3}$ term approximates the surface area of the scene by that of a cube with equivalent volume. As can be seen from Table 1, theoretically ARD is nearly 100 times more efficient than FDTD. In practise, the CPU-based ARD is 50-75X faster than FDTD implementation, as discussed in detail in Section 6. Also, ARD is highly

1. Assuming a DCT and IDCT take $2NlgN$ FLOPS each.

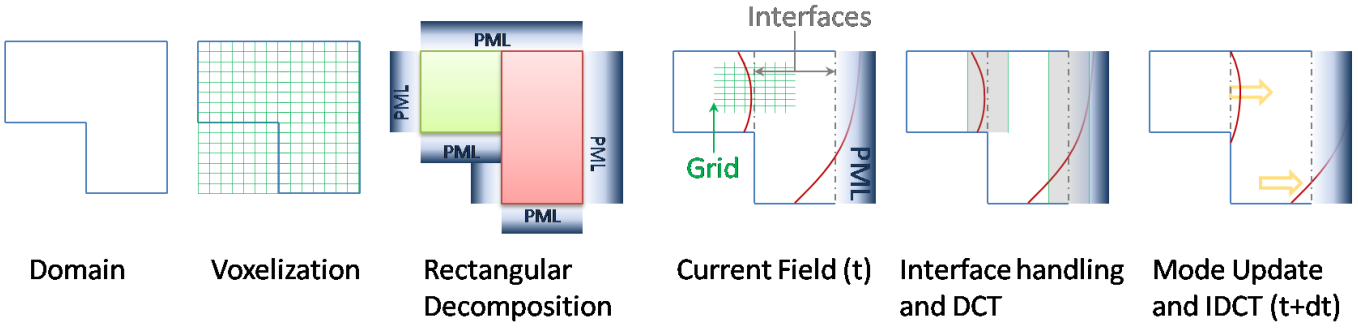

Fig. 1. **Stages of ARD:** In the preprocessing stage, the input domain is voxelized at grid resolution h and adaptively decomposed in rectangles. Artificial Interfaces and PML absorbing layers are created between neighboring partitions and on the scene boundary respectively. During the simulation stage, interface handling is first performed between neighboring partitions and the computed forcing terms transformed to the Cosine spectral basis through DCT. These are then used to update the spectral coefficients (mode update) to propagate waves within each partition. Lastly, the field is transformed back from spectral to spatial domain using IDCT to yield the updated field. Also, Perfectly Matched Layer (PML) partitions might be used to model absorptive surfaces, as shown on the right wall.

memory efficient – ten times more so than FDTD – since it requires fewer cells, as shown in col. 3 of Table 1. This makes it possible to perform simulations on much larger scenes than FDTD without overflowing main memory or GPU memory.

3.3 Computational Pipeline

ARD has two primary stages, *Preprocessing* and *Simulation*. In the preprocessing stage, the input scene is voxelized at the grid resolution h, determined by the maximum simulation frequency $\nu_{\max}$. This is followed by a rectangular decomposition step in which adjacent grid cells generated during voxelization are grouped into rectangles, as illustrated in Figure 1. We call these rectangles *air partitions*. Next, it creates artificial interfaces between adjacent rectangles and PML absorbing layers on the scene boundary. Partitions created for the PML absorbing layer are referred to as *PML partitions*. This one-time computation takes 1-2 mins for most scenes, which is negligible in comparison to the cost of the simulation itself.

During the simulation, the global acoustic field is computed with a time-marching scheme. The computation at each time-step is as follows (see Figure 1):

1) **For all interfaces:** Interface handling to compute force f within each partition
2) **For all air partitions:**
   a) DCT of force f to spectral domain $\tilde{f}$
   b) Mode update for spectral coefficients $\tilde{p}$
   c) IDCT of $\tilde{p}$ to pressure p
   d) Normalize pressure
3) **For all PML partitions:** Update pressure field

During step 1, the coupling between adjacent partitions (air-air and air-PML) is computed to produce forcing values to be applied within partitions. In steps 2 and 3, these forcing values are used to update the pressure fields within the air and PML partitions respectively. While air partitions are updated in the spectral domain, transforming to and from spatial domain using IDCT and DCT, PML partitions employ a finite-difference implementation of a fictitious, highly dissipative wave equation [26] to perform absorption.

Parallelization considerations: Note that only step 1 requires communication between partitions (air or PML). Thus, ARD exhibits a high level of parallelism – all partitions (air and PML) in steps 2 and 3 can clearly be processed in parallel. Also, all interfaces in step 1 could also be processed in parallel. In addition, all the component steps are quite data parallel, as we discuss below. For detailed equations, please refer to the original paper [25].

DCT and IDCT (steps 2a and 2c) can be computed through Fast Fourier Transform(FFT), which is known to be highly data-parallel. Fast, efficient libraries are available for exploiting GPUs to perform this task [13].

Mode update (step 2b) is trivially parallel, since all mode coefficients are updated independently as –

$$\tilde{p}_i^{n+1} = a_i \tilde{p}_i^n - \tilde{p}_i^{n-1} + b_i \tilde{f}^n \tag{3}$$

Here a_i, b_i are mode-specific constants, $\tilde{f}^n$ is the forcing value calculated in step 2a and $\tilde{p}_i^j$ is the mode coefficient value of mode i at time-step j. The total number of modes over all partitions is equal to the number of grid cells (typically in millions) and all these modes can be updated in parallel as above.

Interface Handling and PML (steps 1 and 3) both compute spatial derivatives, which is performed by computing the second derivative D_i at a grid cell (indexed i) by the application of a finite-difference stencil –

$$D_i = \sum_{d=-k}^{k} \beta(d) p_{i+d} \tag{4}$$

Here β is the (constant) finite-difference stencil with fixed width of $2k+1$. For a sixth-order accurate scheme, $k = 3$, corresponding to a stencil width of 7. While each cell needs to read values from neighbors, it updates its own pressure based on the computed value of D_i. Thus, this is a gather operation, which is very amenable to GPUs. Additionally, since every cell only updates its own values, there is no risk of race-conditions while writing and all grid cells on interfaces as well as within PML partitions can be updated in parallel.

4 GPU-BASED ACOUSTIC SOLVER

In previous sections, we discussed computational efficiency and parallelization potential of ARD. In this section we describe our parallel GPU-based acoustic wave equation solver built on top of ARD. We discuss key features of our approach and some of the issues that arise in parallelizing our approach on many-core GPU architectures.

4.1 Key Issues

Two levels of parallelism. ARD technique exhibits two levels of parallelism a) *coarse grained* and b) *fine grained*. Coarse grained parallelism is due to the fact that each of the partitions(air or PML) solves the wave equation independently of each other. Therefore, given enough number of processors, each partition can be solved in parallel at the same time. Fine grained parallelism is achieved because within each partition all the grid cells are independent of each other with regards to solving the wave equation at a particular time-step. For solving the wave equation at the current time-step, a grid cell may use $p, f, \tilde{p}, \tilde{f}$ values of its neighboring cells computed at previous time-step but is completely independent of their $p, f, \tilde{p}, \tilde{f}$ values at the current time-step. Therefore within each partition all the grid cells can run in parallel exhibiting fine grained parallelism.

Our GPU-based acoustic solver exploits both these levels of parallelism. We launch as many kernels in parallel as there are partitions. Each kernel is responsible for solving the wave equation for a particular partition. Within each kernel, each grid cell corresponds to a thread and we create as many threads as the number of grid cells in that rectangle. All these threads are grouped into blocks and grids and scheduled by the runtime environment on the GPU.

Avoid host-device data transfer bottleneck. The host-device data link via PCI express or Infiniband, is a precious resource that has a limited bandwidth. Many prior GPU-based numerical solvers based upon the hybrid CPU-GPU design suffer from data transfer bottleneck as they have to transfer data between host-device at each simulation step.

We have designed our GPU-based solver to ensure that the data-transferred between the CPU-host and GPU-accelerator is minimal. In our case, we avoid the hybrid CPU-GPU approach and instead parallelize the entire ARD technique on the GPU. The only host-device data transfer that is done is for storing the pressure grid p after each simulation step. Since the pressure grid is stored at a much lower resolution for typical auralization applications, typically $1/4^3, 1/8^3, 1/16^3$ of the original size, this cost is negligible. The only limitation is that our simulation is limited to cases where the entire rectangular decomposition needs to fit into GPU memory.

To provide an intuition of host-device data transfer, consider a room of volume $10,000m^3$ for which we solve the wave equation at $\nu_{\max} = 2,000Hz$. We consider a hybrid CPU-GPU system of Raghuvanshi et al. [24] where only the DCT/IDCT stages of the technique are parallelized on GPU. In this case, at each time-step the grid f is transferred from CPU to GPU for DCT, $\tilde{f}$ is returned back by the GPU, $\tilde{p}$ is transferred from CPU to GPU for IDCT and the final pressure p is return by the GPU to the CPU. Since the size of all $p, f, \tilde{p}, \tilde{f}$ is equal to number of grid cells, the total data transfer cost per time-step is 4 x # grid cells x sizeof(*float*) $= 4V\left(\frac{s\nu_{\max}}{c}\right)^3$ x 4 bytes $=$ 4x10000x $\left(\frac{2.6\text{x}2000}{340}\right)^3$ x 4 bytes = 145 MB. As a result, only 2-3X speedup is achieved by this approach on the GPU. In our case, we only need to transfer pressure grid p for storage at the end of time-step and at lower resolution($1/8^3$). Thus our data transfer per time-step = $1/8^3$ x # grid cells x 4 bytes = 3 kB. Data-transfer for such a small size is almost immediate(< 1 msec). Our GPU-based solver achieves a speedup of 6-14X.

Scene	Volume (m^3)	$\nu_{\max}$ Hz	# partitions (air+pml)	# grid cells (in millions)
L-shaped room	13520	1875	424+352	22
Cathedral	13650	1800	7979+14303	21
Walkway	9000	1875	937+882	20
Train station	83640	1350	3824+4945	17
Living room	7392	1875	3228+4518	17
Small room	162	7000	3778+5245	20

TABLE 2
Typical values of scene parameters for all the benchmarks at different values of ν_{max}. Number of partitions are calculated by using our computationally optimal decomposition. Number of pressure values updated at each time-step is equal to the number of grid cells.

Computationally optimal decomposition. Rectangular decomposition proposed by Raghuvanshi et al. [25] uses a greedy heuristic to decompose the voxelized scene into rectangles. Specifically, they place a random seed in the scene and try to find the largest fitting rectangle that can be grown from that location. This is repeated until all the free cells of the scene are exhausted. The cost of DCT and IDCT steps implemented using FFT depends on the number of grid cells in each partition. FFT operations are known to be extremely efficient if the number of grid cells are powers of 2. The proposed heuristic may produce partitions with irregular number of grid cells(not necessarily powers of 2) significantly increasing the cost of the DCT and IDCT operations.

We propose a new approach to perform the rectangular decomposition that takes into account the computational expenditure of FFTs and its efficiency with powers of 2. Specifically, while performing rectangular decomposition, we impose the constraint that the number of grid cells in each partition should be a power of 2. Similar to the original approach, we try to fill the largest possible rectangle that could fit within the remaining air volume of the scene. But instead of directly using it we shrink its size in each dimension to the nearest power of two and declare the remaining cells as free. We repeat this step until all free cells of scene are exhausted. This increases the efficiency of the FFT computations and results in a speedup of 3X in the running time of DCT and IDCT steps on the GPU.

Our approach might produce higher number(2-3X) of rectangular partitions, but since the total number of grid cells in the entire volume of domain remains constant($N = V/h^3$), it does not increase the total FLOPS and thereby running time of the DCT, IDCT and mode update steps. However, since more partitions results in larger interface area, the interface handling cost increases by 25-30%. But since DCT and IDCT are the most time-consuming steps of the ARD technique(Figure 5), the gain achieved by faster DCT and IDCT far outweighs the increased interface handling cost.

4.2 Our GPU approach

Among ARD's two main stages, the pre-processing is performed only once in the beginning and its contribution to the total running time is negligible, we keep this stage on the CPU itself and parallelize the simulation stage on the GPU. We perform the voxelization and rectangular decomposition on the CPU. Once we have the rectangular decompositions, we create the corresponding pressure p, force f, spectral pressure $\tilde{p}$, spectral force $\tilde{f}$ data-structures on the GPU. The simulation stage has 5 main steps(see Figure 1) and each of them is performed in sequential order. In other words, the step $i+1$ only starts after step i is finished. We now discuss the parallelization of all these steps on GPU in detail.

Interface handling. This step is responsible for computing forcing terms f at the artificial interfaces between air-air & air-PML partitions. These forces account for the sound propagation between partitions. As briefly discussed in Section 3.3, this step is quite data parallel – to compute the forcing term at a cell, only values in its spatial neighborhood are needed. The overall procedure consists of iterating over all interfaces, applying the finite difference stencils to compute forcing values and additively accumulating them at the affected cells. Thus, all interfaces could potentially be processed in parallel as long as there are no collisions and no two partitions update the forcing value at the same cell. This can happen at corners, as shown in the figure below.

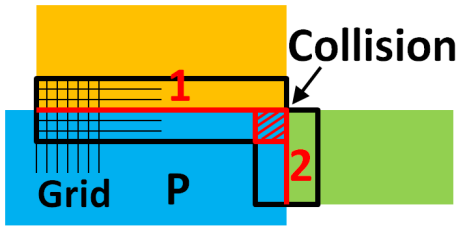

Fig. 2. Interfaces 1 and 2 update forcing values of cells lying in their neighboring partitions. There is a Concurrent Write(CW) hazard in the hatched corner region (labeled "Collision")

Interfaces 1 and 2 both update the forcing values 3-cells deep of their shared partitions. However, for partition P, cells lying in the hatched region (marked "Collision") are updated by *both* interfaces 1 and 2. These corner cases need to be addressed to avoid race conditions and concurrent memory writes. The GPU and its runtime environment places the burden of avoiding concurrent write (CW) hazards on the programmer.

CW hazards can be avoided by running the interface handling step one interface at a time and synchronizing all the threads after each step. But for an arbitrary scene, the number of interfaces can run into thousands, making this approach very inefficient. Another possible approach is to use $AtomicAdd$ instruction supported by many GPUs. $AtomicAdd(m, c)$ adds a value c to the value at shared memory location m in a single instruction in a thread-safe manner. However, the $AtomicAdd$ instruction for floating point values is not supported on most GPUs.

Fortunately, CW hazards can be avoided completely by using a conceptually simple technique – All interfaces are grouped into 3 batches consisting of interfaces with normals in the X, Y and Z directions respectively. Since all partitions are axis-aligned rectangles, every interface has to fall into one of these categories. By processing all interfaces within each batch in parallel and separating batches by a synchronization across all threads, all CW hazards in the corners are avoided completely. Our approach is an order of magnitude faster than the *one kernel per interface* approach and at least as fast as using $AtomicAdd$ but more general and well-supported on all GPUs.

DCT(f) . The DCT step converts the force f from the spatial domain to the spectral domain $\tilde{f}$. DCT's are efficiently computed using Fast Fourier Transforms(FFTs). Typical FFT libraries running on GPU give are an order of magnitude faster than optimized CPU implementations. Since DCT and IDCT steps are among the slowest steps of the technique, parallelization of these steps have a drastic improvement in the running time of the entire ARD technique.

Mode update $\tilde{p}$. The mode update step uses the pressure and force in spectral domain $\tilde{p}, \tilde{f}$ of previous time-step to calculate $\tilde{p}$ at the current time-step. This step consists of linear combinations of $\tilde{p}, \tilde{f}$ terms (see Equation 3) and can be trivially parallelized. We spawn as many threads as the number of modes in the scene and perform the computations of Equation 3 on the GPU.

Normalize pressure p. This step multiplies a constant value to the pressure p and similar to mode update step, this is also trivially parallelizable.

IDCT($\tilde{p}$). This step converts the pressure in spectral domain $\tilde{p}$ back to pressure in spatial domain p. Similar to DCT, the IDCT are also efficiently computed using FFTs.

PML absorption layer. The PML absorbing layer is responsible for sound wave absorption by the surfaces and walls of the 3D environment. It is applied on a 5-10 cell thick partition depending on the desired accuracy [26], [25] and uses a 4th order finite-difference stencil for computation. Based upon the distance of the grid cell from the interface, PML performs different computations for different grid cells. Due to this, there are a lot of inherent conditionals in the algorithm. An efficient implementation of PML depends on minimizing the effect of these conditionals. We achieve this by moving the conditionals out of the kernel and launching separate

kernels for different execution paths of the algorithm.

5 Implementation

The original CPU-based ARD solver uses a serial version of FFTW library [9] for computing DCT and IDCT steps. The CPU code uses two separate threads - one for air partitions and other for PML partitions, and performs both these computations in parallel. For simplicity of comparison with our GPU-based implementation, we measure the sequential performance of the CPU-based solver with only a single thread.

We implemented our GPU-based wave equation solver using NVIDIA's CUDA API with version 3.0 and require minimum compute capability 1.0. Both the original CPU-based ARD code and our current GPU-based ARD code are sufficiently accurate in single precision. The following compiler and optimizations options are used for our GPU code:

nvcc CUDA_v3.0 : Maximize Speed ($/O2$)

Our DCT and IDCT kernels are based upon the FFT library developed by Govindraju et al. [13]. We use CUDA routine *cudaThreadSynchronize()* for synchronizing threads during interface handling and after each step of the simulation stage. The performance of the GPU-based ARD algorithm described in Section 4 can be improved by means of following optimizations. These include:

Batch processing. Interface handling, DCT, IDCT, mode update and pressure normalize kernels form the main components of our GPU-based solver, where each kernel corresponds to a step of the simulation stage. Launching a new kernel for each individual rectangular partition, PML absorbing layer partition and interface can be very inefficient, especially when the number of partitions and interfaces run in thousands. This is typically the case for complex models like the cathedral, train station etc. Each kernel launch has an associated overhead and launching thousands of kernels can have a drastic impact on the overall running time. To avoid this overhead we perform *batch processing*, i.e. group together partitions and interfaces into independent batches and launch a kernel for each batch. Therefore, instead of launching $P + I$ kernels where P is the number of partitions and I is the number of interfaces, we launch as many kernels as there are the number of batches.

For DCT and IDCT kernels, partitions are grouped into batches by using the BATCH FFT scheme of the GPU-FFT library [12]. Mode update and normalize pressure steps have no dependency between different partitions, and can be grouped in a single batch resulting in just one kernel launch. Interface handling, as discussed before, can have data dependency among different interfaces while updating the pressure values. For interface handling, we group the interfaces in separate independent batches with one kernel launch for each batch followed by a call of *cudaThreadSynchronize()*, for synchronizing all the CUDA threads.

Maximizing coalesced memory access. Global memory on the GPU is not cached and the access pattern can have a huge impact on its bandwidth. Global memory accesses are most efficient when memory accesses of threads of a half-warp can be *coalesced* in a single memory access. Our $p, f, \tilde{p}, \tilde{f}$ data-structures and their memory access patterns for the mode update and normalize pressure kernels are organized in a way such that each thread of index i accesses these single precision float data-structures at position i itself. Thus the memory access pattern of a half-warp is perfectly coalesced. DCT and IDCT kernels based upon FFT library [13] use memory coalescing as well. Our PML handling kernel achieves for thread i accesses memory at locations $\alpha + i$ where α is constant. This kind of access results in a coalesced memory access on device with compute capability $>= 1.2$ but not on 1.0 and 1.1. The interface handling step can access p, f from many partitions depending upon the rectangular decomposition and therefore achieving coalesced memory access for this kernel is difficult.

Minimize path divergence . The impact of conditionals on the performance of the GPU kernel can be very severe. The interface handling and the PML absorbing layer steps of the simulation stage have conditionals that are based upon the distance of the grid cells from the interface. In our implementation, we take specific care in minimizing these effect of conditional branching. Instead of launching a single kernel with conditional branching, we launch separate small kernels corresponding to different execution paths of the code.

6 Results

We compare our GPU-based acoustic wave equation solver with the CPU implementation provided by the authors of ARD [25]. We use Nvidia Quadro FX-5800 graphics card with a core clock speed of 650 MHz, graphics memory of 4 GB with 240 CUDA cores. We also profiled our algorithm on other GPU's with 128 and 32 CUDA cores, to evaluate scaling of our algorithm with number of cores. CPU timings are reported for Intel Xeon X5560 with processor speed of 2.8 GHz. Timings are reported by running the simulation over 100 time-steps and take the average.

In Figure 3(a), we compare the performance of the CPU-based solver with our GPU-based solver with varying maximum frequency $\nu_{\max}$ of simulation. As can be seen, GPU-based solver performs better as $\nu_{\max}$ increases. Figure 3(b) shows the speedup achieved by our GPU-based solver over the CPU. For smaller frequencies, the amount of work available is considerably less, resulting in nominal speedup. But as the frequency increases above 1300 Hz, our GPU-based acoustic solver outperforms its CPU counterpart by a factor of 6-13X on different scenes. Rectangular decomposition of simple scenes like L-shaped room gives fewer air partitions(see Table 2 column 4) resulting in fewer batches(Section 5) of larger size. Fewer batches means fewer kernel calls

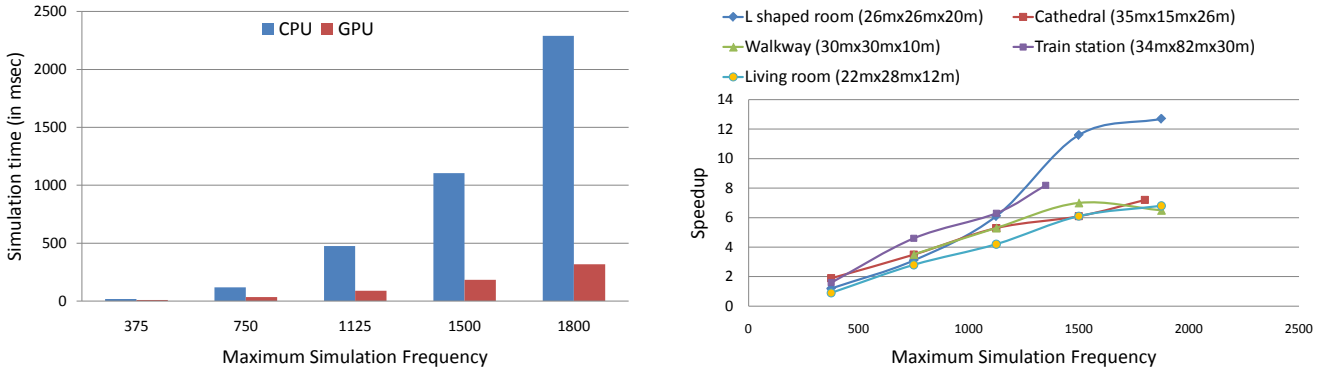

Fig. 3. a) (Left) Performance results(simulation time per time-step) of CPU-based and GPU-based ARD solver with varying maximum frequency $\nu\max$ for the cathedral scene of dimensions 35m x 15m x 26m. b) (Right) Speedup(=CPU time/GPU time) achieved by our GPU-based ARD solver over the CPU-based solver with varying $\nu\max$ for the different test scenes. For $v_{\max} > 1300Hz$, we achieve a speedup of 6-13X.

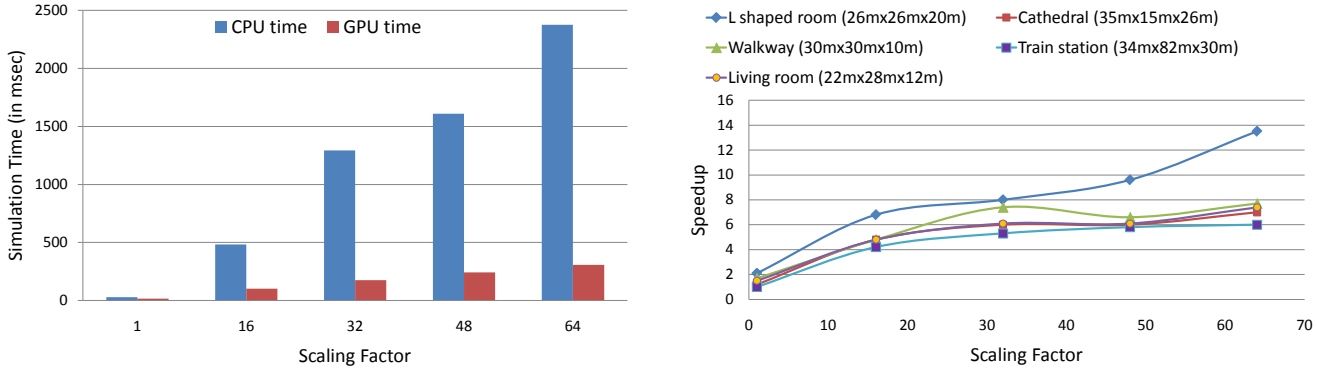

Fig. 4. a) (Left) Simulation time(in msec) per time-step of CPU-based and GPU-based ARD solver with varying scene volume for the walkway scene with fixed $\nu_{\max} = 472Hz$. b) (Right) Speedup(=CPU time/GPU time) achieved by our GPU-based ARD solver over the CPU implementation with varying scene volume for the different test scenes. $v_{\max}$ values used are as follows- L-shaped room(450Hz), Cathedral(412Hz), Walkway(472Hz), Train station(285Hz) and Living room(487Hz). For both figures, we scale the original volume of the test scenes by the *Scaling Factor*. As the scene volume increases, we achieve a higher speedup and for 64 times the original volume, the speedup becomes 6-14X.

reducing the total overhead of kernel launches. Even the DCT and IDCT based on GPU-FFT are much more efficient on large sized inputs. This results in higher speedups for simpler scenes.

We also analyze the performance of our solver with varying scene volume. We take the Cathedral scene and scale its volume uniformly in the range of 1X to 64X. In Figure 4(a), we observe again that as the amount of work increases with increasing scene volume, the performance of GPU-based solver scales better compared to the CPU solver. Speedup achieved by our GPU-based solver for varying scene volume also shows a similar behavior(see Figure 4(b)). Scaling the volume by 16X gives a speedup of 4-8X and as the scaling factor increases to 64X, we achieve a speedup of 6-14X on different scenes.

Figure 5 shows the breakup of the time spent on various steps of the simulation stage. In the original CPU-based ARD solver, the DCT/IDCT steps and the Perfectly Matched Layer(PML) heavily dominate the computation time. But for the GPU-based solver, as can be seen, all the stages of the pipeline are more or less balanced except mode update and normalize pressure, whose costs become negligible compared to other steps. We investigate the speedup of our GPU acoustic solver in detail, by measuring the individual speedup achieved by different stages of the ARD technique on the GPU (see Figure 6). Our DCT & IDCT kernels implemented using FFT library [12], give us a speedup of 7X on the GPU. Mode update stage and normalize pressure both of which are trivially parallel and PML without conditionals, achieve a higher speedup of 14X, 10X and 9X respectively on the GPU. The last stage of ARD, interface handling, involves a lots of uncoalesced memory accesses resulting in a nominal speedup of 2X. But since the contribution of interface handling to the overall running time is far less than DCT, IDCT and PML(see Figure 5), it does not become a bottleneck.

We tested our algorithm on three different GPUs each with different number of CUDA cores : Quadro FX 5800, GeForce 8800GTX and GeForce 9600M GT each with 240, 128 and 32 CUDA cores respectively. We performed scalability analysis of our solver with different number of CUDA cores. Figure 7 shows the performance of our solver on the cathedral and the small room scene with varying $\nu_{\max}$ as the number of CUDA cores increase. As can be seen, our GPU-based solver scales linearly with the number of cores available. Increasing the number of CUDA cores 4 times from 32 to 128 results in a speedup of $3-4$X and from 32 cores to 240 cores gives a speedup of $7-8$X. As the amount of work increases

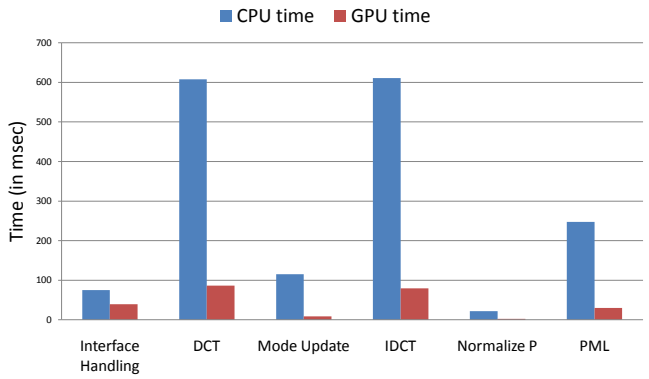

Fig. 5. Simulation stages - Interface handling, DCT, mode update, IDCT, normalize pressure and PML, and the corresponding time spent in the CPU-based and GPU-based ARD solver for the walkway scene (30mx30mx10m) at maximum frequency of 1875 Hz. For our GPU-based solver, almost all the stages of the simulation stage are more or less balanced, except mode update and normalize pressure, whose costs become negligible.

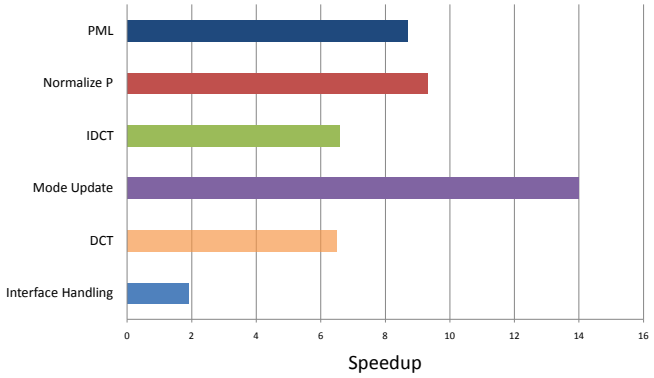

Fig. 6. Speedups achieved by individual stages of the GPU-based ARD over the CPU-based solver for the walkway scene (30mx30mx10m) at maximum frequency of 1875 Hz.

with increasing $\nu_{\max}$, the performance scaling becomes perfect. This shows that our GPU-based ARD solver is compute-bound rather than by memory bandwidth.

We also perform a performance comparison of FDTD solver, CPU-based ARD solver and our GPU-based ARD solver with varying maximum frequency of simulation $\nu_{\max}$(see Figure 8). As can be seen, CPU-based ARD-solver achieves a speedup of 50-75X over the FDTD. Our GPU-based ARD solver achieves a speedup of over 500X over FDTD for the same scene. Since FDTD runs out of memory for $\nu_{\max} > 3750Hz$, we use the timings below $3750Hz$ and the fact that simulation time varies as fourth power of ν_{max}, to calculate the projected timings for FDTD above $3750Hz$

7 Conclusion and Future Work

In this paper, we have presented an efficient GPU-based solver for acoustic wave equation. Our formulation is based on adaptive rectangular decomposition of the acoustic space and we present methods to map all the steps of the algorithm to GPU architectures. We observe more than two orders of magnitude improvement over prior solvers based on FDTD. Moreover, the use of GPUs can accelerate the computation by almost one order of magnitude as compared to the CPU-based ARD solver.

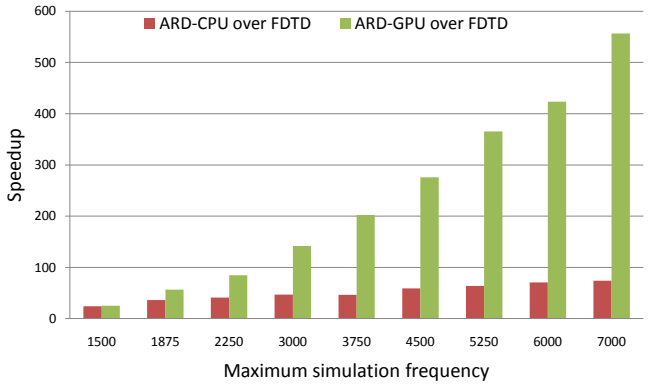

Fig. 8. We plot speedup achieved by CPU-based ARD solver and our GPU-based ARD solver over FDTD with varying ν_{max} for the small room (6.1mx7.8mx3.4m) as our benchmark. We calculate the speedup of ARD solvers over FDTD as (simulation time per time-step for FDTD)/(simulation time per time-step for ARD). FDTD runs out of memory from $\nu_{max} > 3750$ Hz for this scene. We use projected time to calculate FDTD timings above 3750 Hz. Our GPU-based ARD solver achieves a maximum speedup of 550X over FDTD compared to CPU-based ARD which only achieves a maximum speedup of 75X.

Our approach has some *limitations*. The ARD formulation assumes uniform medium and does not model the variations in the temperature. Moreover, we assume that the entire spatial decomposition fits into GPU memory and the approach may not work well over very large acoustic spaces. Moreover, our current implementation is based on single precision arithmetic. Future GPUs are expected to support double precision arithmetic, though it may reduce the speedup.

In terms of future work, we would like to overcome these limitations. We would like to apply our approach to more complex acoustic spaces such as CAD models. It would be very useful to extend our approach to multi-GPU clusters.

References

[1] J. B. Allen and D. A. Berkley. Image method for efficiently simulating small-room acoustics. *J. Acoust. Soc. Am*, 65(4):943–950, 1979. 2

[2] Joshua A. Anderson, Chris D. Lorenz, and A. Travesset. General purpose molecular dynamics simulations fully implemented on graphics processing units. *J. Comput. Phys.*, 227(10):5342–5359, 2008. 3

[3] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, New York, NY, USA, 2009. ACM. 2

[4] D. Botteldooren. Acoustical finite-difference time-domain simulation in a quasi-cartesian grid. *The Journal of the Acoustical Society of America*, 95(5):2313–2319, 1994. 2

[5] D. Botteldooren. Finite-difference time-domain simulation of low-frequency room acoustic problems. *Acoustical Society of America Journal*, 98:3302–3308, December 1995. 2

[6] John P. Boyd. *Chebyshev and Fourier spectral methods*. Dover Publications, 2 revised edition, December 2001. 2

[7] Carlos A. Brebbia. *Boundary Element Methods in Acoustics*. Springer, 1 edition, October 1991. 2

[8] Chris Chapman. *Fundamentals of Seismic Wave Propagation*. Cambridge University Press, August 2004. 2

[9] M. Frigo and S. G. Johnson. The design and implementation of fftw3. *Proc. IEEE*, 93(2):216–231, 2005. 7

[10] Michael Garland, Scott Le Grand, John Nickolls, Joshua Anderson, Jim Hardwick, Scott Morton, Everett Phillips, Yao Zhang, and Vasily Volkov. Parallel computing experiences with cuda. *IEEE Micro*, 28(4):13–27, 2008. 2

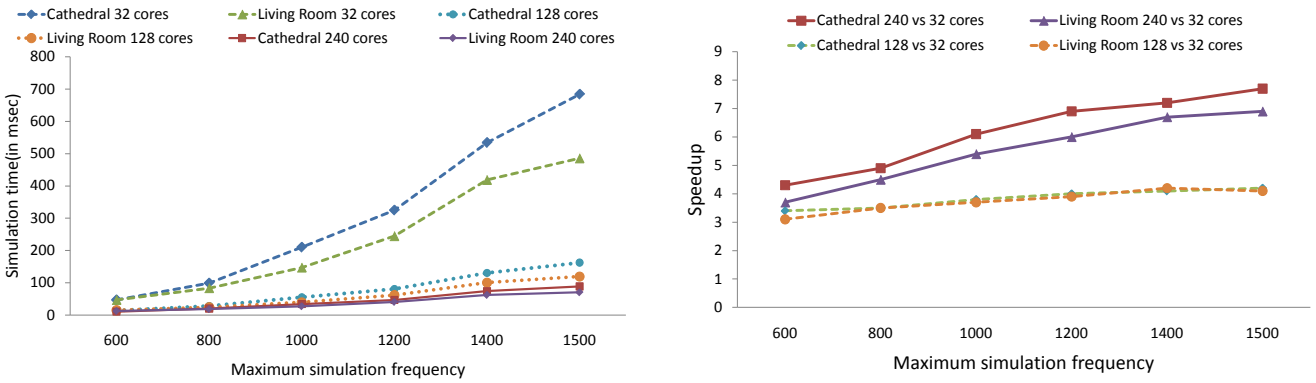

Fig. 7. We use Cathedral (35mx15mx26m) and living room(22mx28mx12m) as our benchmark and run the simulations with varying $\nu_{\max}$ on 3 different GPU's with different number of CUDA cores - GeForce 9600M GT(32 cores), GeForce 8800GTX(128 cores) and Quadro FX 5800(240 cores). a) (Left) Simulation time per time-step of our GPU-based ARD solver. b) (Right) Scalability of our GPU-based ARD solver. We calculate the speedup wrt to the GPU with 32 CUDA cores i.e. Speedup on GPU with X cores = (Simulation time on GPU with X cores)/(Simulation time on GPU with 32 cores). As the number of cores increase from 32 to 128 (4 times) our GPU-based ARD solver becomes 3-4X faster. Similarly from 32 to 240 cores, we get around 7-8X speedup. This scaling is perfect at higher values of $\nu_{\max}$.

[11] Naga K. Govindaraju, Scott Larsen, Jim Gray, and Dinesh Manocha. A memory model for scientific algorithms on graphics processors. In *in Proc. of the ACM/IEEE Conference on Supercomputing (SC06*. ACM Press, 2006. 2

[12] Naga K. Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith, and John Manferdelli. High performance discrete fourier transforms on graphics processors. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press. 2, 7, 8

[13] Naga K. Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith, and John Manferdelli. High performance discrete fourier transforms on graphics processors. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press. 4, 7

[14] K. R. Kelly. *Numerical Modeling of Seismic Wave Propagation (Geophysics Reprints Series, No 13)*. Society Of Exploration Geophysicists, January 1990. 2

[15] A. Klckner, T. Warburton, J. Bridge, and J.S. Hesthaven. Nodal discontinuous galerkin methods on graphics processors. *Journal of Computational Physics*, 228(21):7863 – 7882, 2009. 3

[16] Mendel Kleiner, Bengt-Inge Dalenbäck, and Peter Svensson. Auralization - an overview. *JAES*, 41:861–875, 1993. 2

[17] U.R. Krockstadt. Calculating the acoustical room response by the use of a ray tracing technique. *Journal of Sound Vibration*, 1968. 2

[18] Heinri Kuttruff. *Room Acoustics*. Taylor & Francis, October 2000. 2

[19] K. Liu, Xiao bing Wang, Yang Zhang, and Cheng Liao. Acceleration of time-domain finite element method (td-fem) using graphics processor units (gpu). pages 1 –4, oct. 2006. 3

[20] Qing H. Liu. The pstd algorithm: A time-domain method combining the pseudospectral technique and perfectly matched layers. *The Journal of the Acoustical Society of America*, 101(5):3182, 1997. 2

[21] John Michalakes and Manish Vachharajani. Gpu acceleration of numerical weather prediction. *Parallel Processing Letters*, 18(4):531–548, 2008. 3

[22] Paulius Micikevicius. 3d finite difference computation on gpus using cuda. In *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 79–84, New York, NY, USA, 2009. ACM. 2, 3

[23] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, 2008. 2

[24] Nikunj Raghuvanshi, Brandon Lloyd, Naga K. Govindaraju, and Ming C. Lin. Efficient numerical acoustic simulation on graphics processors using adaptive rectangular decomposition. In *EAA Symposium on Auralization*, June 2009. 3, 5

[25] Nikunj Raghuvanshi, Rahul Narain, and Ming C. Lin. Efficient and accurate sound propagation using adaptive rectangular decomposition. *IEEE Transactions on Visualization and Computer Graphics*, 99(1), 2009. 3, 4, 5, 6, 7

[26] Y. S. Rickard, N. K. Georgieva, and Wei-Ping Huang. Application and optimization of pml abc for the 3-d wave equation in the time domain. *Antennas and Propagation, IEEE Transactions on*, 51(2):286–295, 2003. 3, 4, 6

[27] J. H. Rindel. The use of computer modeling in room acoustics, 2000. 2

[28] S. Sakamoto, T. Yokota, and H. Tachibana. Numerical sound field analysis in halls using the finite difference time domain method. In *RADS 2004*, Awaji, Japan, 2004. 2

[29] Shinichi Sakamoto, Takuma Seimiya, and Hideki Tachibana. Visualization of sound reflection and diffraction using finite difference time domain method. *Acoustical Science and Technology*, 23(1):34–39, 2002. 2, 3

[30] Shinichi Sakamoto, Ayumi Ushiyama, and Hiroshi Nagatomo. Numerical analysis of sound propagation in rooms using the finite difference time domain method. *The Journal of the Acoustical Society of America*, 120(5):3008, 2006. 2

[31] Peter U. Svensson. Modelling acoustic spaces for audio virtual reality. November 2002. 1

[32] Allen Taflove and Susan C. Hagness. *Computational Electrodynamics: The Finite-Difference Time-Domain Method, Third Edition*. Artech House Publishers, 3 edition, June 2005. 2

[33] J.C Thibault and I. Senocak. Cuda implementation of a navier-stokes solver on multi- gpu desktop platforms for incompressible flows. 2009. 3

[34] Lonny L. Thompson. A review of finite-element methods for time-harmonic acoustics. *The Journal of the Acoustical Society of America*, 119(3):1315–1330, 2006. 2

[35] Tolke, J., Krafczyk, and M. Teraflop computing on a desktop pc with gpus for 3d cfd. *International Journal of Computational Fluid Dynamics*, 22(7):443–456, August 2008. 3

[36] Vasily Volkov and James W. Demmel. Benchmarking gpus to tune dense linear algebra. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press. 2

[37] Kane Yee. Numerical solution of initial boundary value problems involving maxwell's equations in isotropic media. *IEEE Transactions on Antennas and Propagation*, 14(3):302–307, May 1966. 2

[38] O. C. Zienkiewicz and R. L. Taylor. *The Finite Element Method Set, Sixth Edition*. Butterworth-Heinemann, 6 edition, January 2006. 2

[39] David W. Zingg. Comparison of high-accuracy finite-difference methods for linear wave propagation. *SIAM J. Sci. Comput.*, 22(2):476–502, 2000. 2