Mathematical and Computer Modelling 57 (2013) 1822–1831

Contents lists available at SciVerse ScienceDirect

Mathematical and Computer Modelling

journal homepage: www.elsevier.com/locate/mcm

Parallelization of the finite-difference time-domain method for room acoustics modelling based on CUDA

José J. López [a,*], Diego Carnicero [a], Néstor Ferrando [b], José Escolano [c]

[a] Instituto de Telecomunicaciones y Aplicaciones Multimedia, Universidad Politécnica de Valencia, Camino de Vera s/n, 46022 Valencia, Spain

[b] Instituto de Instrumentación para Imagen Molecular, Universidad Politécnica de Valencia, Camino de Vera s/n, 46022 Valencia, Spain

[c] Telecommunication Eng. Department, University of Jaén, 23700 Linares, Spain

ARTICLE INFO

Article history:
Received 10 October 2011
Received in revised form 28 November 2011
Accepted 30 November 2011

Keywords:
Acoustics modelling
FDTD
Parallel computing
Graphics processing unit (GPU)
CUDA

ABSTRACT

The parallelization of the finite-difference time-domain (FDTD) method for room acoustic simulation using graphic processing units (GPUs) has been subject of study even prior to the introduction of GPGPU (general-purpose computing on GPUs) environments such as the compute unified device architecture (CUDA) from Nvidia. A mature architecture nowadays, CUDA offers enough flexibility and processing power to obtain important performance gains with naively ported serial CPU codes. However, careful implementation of the algorithm and appropriate usage of the different subsystems a GPU offers can lead to even further performance improvements. In this paper, we present a detailed study between different approaches to the parallelization of the FDTD method applied to room acoustics modelling, and we describe several optimization guidelines to improve the computation speed when using single precision and double precision floating point model data, nearly doubling the performance obtained by previously published implementations.

1. Introduction

In recent years, general-purpose computing on graphics processing units (GPGPU) has experienced a fast evolution, allowing inherently parallel algorithms to experience important performance gains. For an affordable cost, a wide variety of applications have benefited from this evolution, from fluid dynamics simulation [1] to video compression using the H.264/AVC codec [2]. The flagship of GPGPU has been the compute unified device architecture (CUDA). CUDA offers the programmer a variety of resources from the GPU and an easy way of porting serial codes to parallel codes without much effort. However, although naively ported codes can perform much faster than their serial counterparts, careful tuning and usage of the GPU hardware can increase the speed by one or two orders of magnitude compared to the serial code. Several techniques can be employed to improve algorithmic performance on the GPU, as will be discussed in the next section, two of them being the usage of *shared memory* and the usage of *texture memory*.

As a widely used method for numeric calculation, the FDTD method has also been the subject of study applied to electromagnetism simulation [3] and light-scattering simulations [4]. Before CUDA, investigations of the FDTD method made use of the OpenGL API, obtaining promising results [5,6] despite the complexity of translating model data to graphics data in order to be processed by the GPU. It has not been until recently, with CUDA, that investigation has been carried on room acoustics modelling. In [7], an introduction to the implementation on GPUs is presented, [8] shows a high performance implementation for a two-dimensional (2D) mesh, and [9] offers an overview on FDTD schemes for real-time three-dimensional (3D) room auralization. Further investigation on 3D room simulation with boundary losses is presented in [10],

* Corresponding author. Tel.: +34 963879714.
E-mail address: jjlopez@dcom.upv.es (J.J. López).

0895-7177/$ – see front matter © 2011 Elsevier Ltd. All rights reserved.
doi:10.1016/j.mcm.2011.11.075

and [11] discusses the performance of different implementation approaches when working with double precision model data.

This paper presents a comparison between six representative approaches to FDTD parallelization applied to room acoustics modelling, divided into two main categories, with special attention to the usage of *texture memory* and *shared memory* in order to improve data throughput. First, a brief introduction to CUDA is provided. Second, an explanation of each implementation is provided, with a more detailed description of the implementation using texture memory only, and the advantages and disadvantages of every approach are discussed. Finally, three experiments are carried out to test the performance and correctness of the implementations developed: a quantitative performance comparison for a standard problem to test the speed difference between implementations, a comparison with double precision model data to an existing parallel implementation, and a proof of accurate calculation to show the expected effects of diffraction and reflection.

2. The finite-difference time-domain method for acoustics

The finite-difference time-domain (FDTD) method is arguably the most popular numerical method for the solution of problems in electromagnetism and acoustics. Although the FDTD method has existed for over 40 years, its popularity continues to grow as computing costs keep on declining. The method consists in dividing a volume (the room interior in the case of room acoustics) into small cubic cells and applying the wave propagation equation of pressure to each cell. The sizes of the cells determine the maximum frequency to be simulated. The difference equation computes the next time step of pressure as a function of the pressure in the neighbor cells in the two previous time steps in the form

$$p^{n+1}_{i,j,k} = 1/3(p^n_{i+1,j,k} + p^n_{i-1,j,k} + p^n_{i,j+1,k} + p^n_{i,j-1,k} + p^n_{i,j,k+1} + p^n_{i,j,k-1}) - p^{n-1}_{i,j,k}, \quad (1)$$

where p is the sound pressure, (i, j, k) are the coordinates of the cells in the mesh, and n is the time step in the recursion. More details about the FDTD method can be found in [12].

The next section briefly explains CUDA and its limitations, emphasizing the aspects related to performance that are needed to understand several design decisions, and the results and conclusions obtained by the carried experiments.

3. CUDA: compute unified device architecture

In recent years, GPUs have evolved into an affordable, yet powerful architecture for the computation of massively parallel algorithms. CUDA has been at the cutting edge of this evolution and has matured to become a versatile general-purpose parallel computing platform, developed by Nvidia, available for the latest generations of GPUs. From the perspective of CUDA, a GPU is abstracted as a device that contains a set of memories and an array of independent streaming multiprocessors (SMs), with each SM being able to execute a large number of threads simultaneously. A typical CUDA function, referred to as a *kernel*, is performed in parallel by a large number of threads, all of which execute the same instructions.

The threads, organized in blocks, and further clustered in a flat hierarchical structure called a *grid*, are sent to the cores for execution. Once in the core, the threads execute concurrently, but grouped in packs of 32 threads called *warps*. Each warp executes independently of the others, whilst threads that form a warp must execute one common instruction. Inside each core, on-chip *shared memory* allows fast communication between the threads of a block. In addition, the GPU has a *global memory* that enables persistent data storage along application lifetime and a (slower) global scope communication between different thread blocks. Each core has fast read access to both a *texture memory cache*, optimized for 2D spatial locality, and a *constant memory cache*. Finally, the CUDA environment provides a set of synchronization functions that help in coordinating the parallel execution of threads, guaranteeing local and global synchronization between threads when accessing shared and global memory.

To maximize the performance in this environment, the program code should exercise moderate access to the GPU global memory and, when doing so, the access pattern should be optimized in order to achieve *memory coalescing*. This consists in reducing the number of memory read operations to global memory for a group of threads by accessing data in the same memory segment. Each new version of the architecture removes several restrictions in the access pattern and improves the bandwidth. The Fermi architecture [13], for instance, adds a two-level cache for global memory accesses. Although these caches help in reducing access latency and global memory bandwidth requirements compared to the Tesla architecture [14], they are small, and *memory coalescing* still must be taken into account. In addition, the code should try to maximize the number of active threads in each core as well as to minimize the use of flow control instructions, as these can easily make threads inside a warp follow different execution paths, resulting in serial execution for threads in the warp. For further information on CUDA, refer to [15,16].

Taking into account these considerations about the hardware architecture of the GPU and its execution model, the next section describes two strategies to process 3D rooms and presents six different implementations derived from these strategies in order to test which is the fastest.

4. 3D FDTD implementations

The CUDA threading model offers great flexibility when working with 2D data thanks to its hierarchical nature, as stated in the previous section. Unfortunately, it imposes certain limitations with 3D data since thread blocks can be three dimensional but grids can only be two dimensional. Therefore, it is necessary to explore different possibilities on how to work with 3D data efficiently, applied in this paper to the simulation of room acoustics through the FDTD modelling technique. Recent works have explored these possibilities, concluding that the *tiling method* (see Section 4.1) is the one that provides the best performance [11], although not in much detail. This paper aims to offer an exhaustive study on the optimum processing strategies for 3D enclosures, studying the performance provided by existing methods and comparing it to new proposed methods with different data mapping, execution, and memory access strategies.

Taking advantage of the features the CUDA threading model offers and the different memory subsystems an Nvidia GPU possesses, two main techniques to process the 3D data are tested, one of them being the tiling method described in Section 4.1 and the other the *slicing method* described in Section 4.2. For this, six kernels with different approaches to memory access during execution have been developed divided into these two main categories.

The implementations are built on top of a set of basic considerations, so that the speed difference between them is only caused by the difference in data mapping, execution, and data access approaches. In general terms, we use (i) the constant cache to store data that do not change during the simulation; (ii) global memory to store the model and perform global thread synchronization, using optimized access patterns; (iii) the texture cache to access the neighboring nodes in the FDTD iteration; and (iv) shared memory to access neighboring nodes and to store temporal data. Some further guidelines followed are as follows.

- A thread block size of 256 and a limit of 16 and 20 registers per thread when using GPUs with the Tesla architecture and the Fermi architecture, respectively, to maximize SM occupancy.
- Block dimensions that are a multiple of the size of a *half-warp* to allow for coalesced memory access.
- Minimization of memory transfers between host and device.
- Use of register space where possible to prevent any unnecessary accesses to global memory.

The next two sections present an overview of the mentioned tiling and slicing methods and a description of the implementations that will be tested in Section 5.

4.1. Tiling method

Several parallel implementations of the FDTD method have relied on this form of data mapping [10,11] to process 3D rooms, choosing it over sliced methods (see Section 4.2). It consists in flattening the volume into a tiled plane in which each tile corresponds to a 2D slice. Considering the volume extends along the x, y and z dimensions, and with z the slowest varying one in linear memory, the volume should be sliced along z to achieve *memory coalescing* (see Section 3).

Fig. 1 describes how tiles can be laid out in a 1D or 2D fashion. This method maximizes parallel execution by assigning a thread to every data point and keeping the streaming multiprocessors busy, but at a cost. Since thread blocks are executed in a random order, it is not possible to take advantage of spatial or temporal data locality, because every thread needs to read all six adjacent nodes from global memory, worsening the inherent memory bottleneck to the FDTD algorithm [17]. Additionally, as the 3D volume is now mapped in a 2D plane, it is necessary to perform a translation between 2D and 3D data indices so that each thread knows which point in space it needs to read from and write to. Let (i, j, k) denote an arbitrary point in the 3D volume, (S_i, S_j, S_k) the dimensions of the 3D volume, (c, r) an arbitrary point in the 2D plane, and (S_c, S_r) the dimensions of the 2D plane. When using 2D tiling, the index translation equations are as follows:

$$i = c \bmod S_i, \qquad j = r \bmod S_j, \qquad k = S_r \left\lfloor \frac{c}{S_i} \right\rfloor + \left\lfloor \frac{r}{S_j} \right\rfloor .$$

For 1D tiling, lying the tiles along the y dimension, the equations are simplified to

$$i = c, \qquad j = r \bmod S_j, \qquad k = \left\lfloor \frac{r}{S_j} \right\rfloor .$$

The floor and modulo operators are computationally very intensive [18], and while the recent Fermi architecture can cope with the computational cost, the fact that these operations need to be performed by every single thread impose an important performance penalty in the early Tesla architecture.

Two versions of the tiling method have been implemented using the 1D approach as fewer computations are required for index translation. The first version uses direct accesses to global memory to access every single neighboring node for every thread (from now on referred to as *Tiled Global*), totalling six reads from global memory. The second implementation stores the model data first in shared memory (from now on referred to as *Tiled Shared*), with every inner thread of a thread block only having to read from global memory for the nodes above and below. For threads in the outer edges of thread blocks, an additional memory read is performed to access the node value that falls outside the thread block and that has not been stored in shared memory.

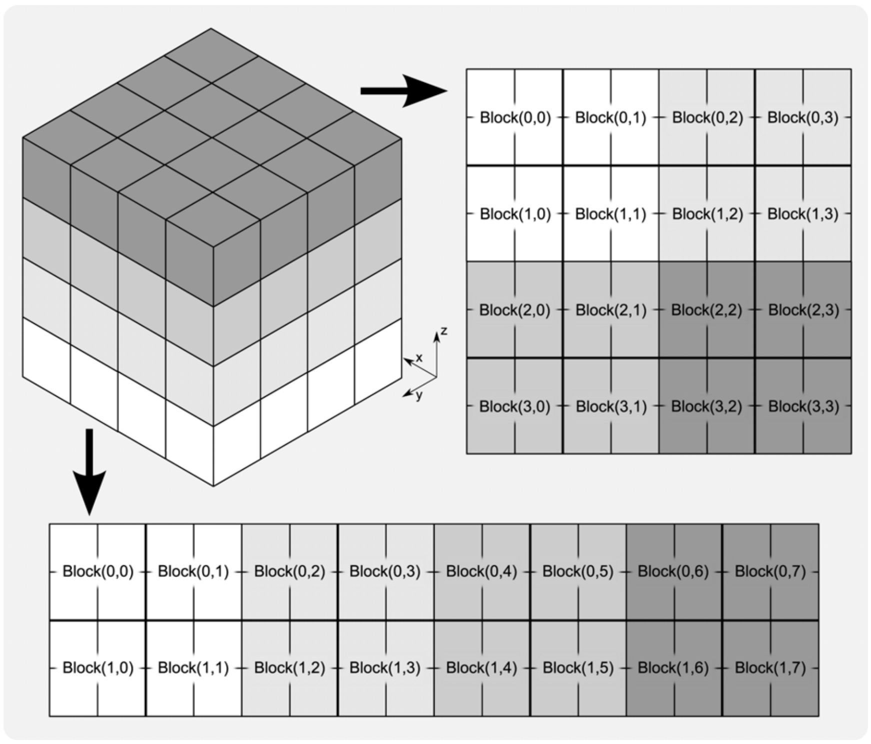


Fig. 1. 3D data tiling in CUDA: (a) 1D tiling, (b) 2D tiling.

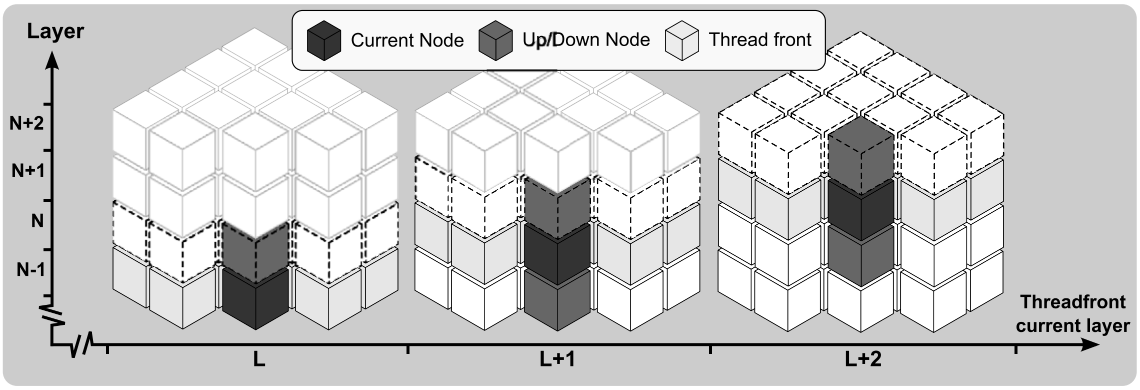
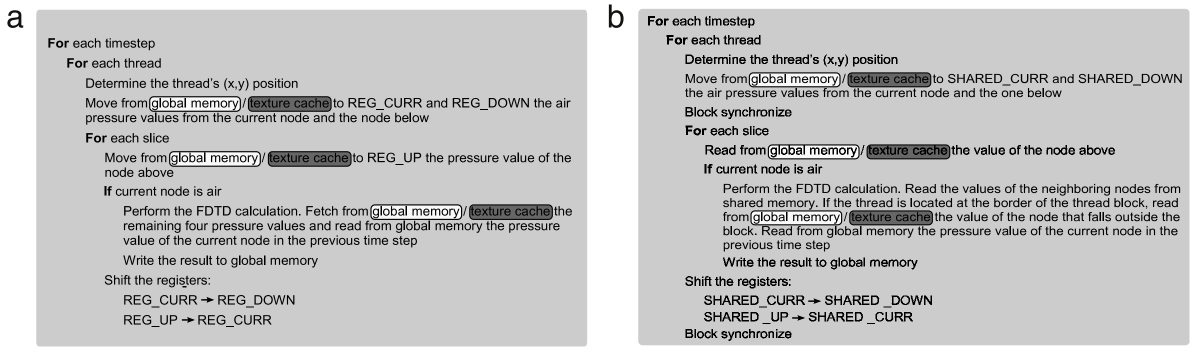


Fig. 2. Data reuse in the slicing method. Nodes to be processed are greyed out whilst dashed nodes represent the layer immediately above the layer the *threadfront* is processing.

4.2. Slicing method

Taking into account the threading hierarchy limitations, a straightforward way of processing a volume is using 2D thread blocks and slicing the volume along its *z dimension* [19] (as the data are stored in global memory in a row major order, this allows for *memory coalescing*). This way a *threadfront* sweeps the volume from bottom to top every time step, with every single thread in the *threadfront* processing a given column of nodes. Fig. 2 depicts how the serial computation of the slices allows for data reuse. Focusing on a single thread, for every slice processed the thread only needs to read the four adjacent nodes in the current slice plus the one above, since the pressure values of the current node and the one below have already been read in the two previous time steps. When the *threadfront* advances one slice up, a simple shift of the registers is performed and the process is repeated.

The following sections describe the proposed sliced implementations. First, the reference or *naive* implementation is introduced, in which no special optimizations are used; second, an implementation that makes use of exclusively texture memory to access neighboring data; and lastly, two implementations that make use of shared memory to store and


a

```
For each timestep
  For each thread
    Determine the thread's (x,y) position
    Move from global memory / texture cache to REG_CURR and REG_DOWN the air
    pressure values from the current node and the node below
    For each slice
      Move from global memory / texture cache to REG_UP the pressure value of the
      node above
      If current node is air
        Perform the FDTD calculation. Fetch from global memory / texture cache the
        remaining four pressure values and read from global memory the pressure
        value of the current node in the previous time step
        Write the result to global memory
      Shift the registers:
        REG_CURR → REG_DOWN
        REG_UP → REG_CURR
```

b

```
For each timestep
  For each thread
    Determine the thread's (x,y) position
    Move from global memory / texture cache to SHARED_CURR and SHARED_DOWN
    the air pressure values from the current node and the one below
    Block synchronize
    For each slice
      Read from global memory / texture cache the value of the node above
      If current node is air
        Perform the FDTD calculation. Read the values of the neighboring nodes from
        shared memory. If the thread is located at the border of the thread block, read
        from global memory / texture cache the value of the node that falls outside the
        block. Read from global memory the pressure value of the current node in the
        previous time step
        Write the result to global memory
    Shift the registers:
      SHARED_CURR → SHARED _DOWN
      SHARED _UP → SHARED _CURR
    Block synchronize
```

Fig. 3. Pseudocode representations of the different implementations. (a) *Naive* and *Pure Texture Fetching*: the memory accesses exclusively performed by the *Naive* implementation are highlighted in white, and in darker colour the accesses performed by the *Pure Texture Fetching* implementation. (b) *Shared Global* and *Shared Texture Fetching*: the memory accesses exclusively performed by the *Shared Global* implementation are highlighted in white, whilst the accesses performed by the *Shared Texture Fetching* implementation are highlighted in darker colour.

share reusable information with two different approaches to access data in the outer edge of thread blocks. Pseudocode representations are provided in addition to compare the instruction flows of each kernel and the memory access strategies.

4.2.1. Naive implementation

The naive or global memory-only implementation serves as a reference to test performance gains over the sliced method achieved by the other sliced approaches described in Sections 4.2.2 and 4.2.3. This implementation uses the core set of optimizations described in Section 4 and the register shifting approach described in Section 4.2; the remaining data needed are fetched directly from global memory. The pseudocode representation can be seen in Fig. 3.

4.2.2. Efficient implementation using texture memory

As stated in [15], the texture memory space resides in device memory and is cached in a two-level [18] cache called the texture cache. This cache space is optimized for 2D spatial locality, providing best performance to threads of the same warp that read texture memory addresses that are close together in two dimensions, so, even without the access patterns that global or constant memory reads must respect to get good performance, high bandwidth can be achieved. Two additional benefits are its ability to broadcast packed data to separate variables in a single operation, and dedicated units that perform the addressing calculations outside the kernel.

Texture memory usage is based on the use of texture references. Texture references are structures that point to certain spaces in global memory and define which part of that space is fetched and in which way.

Addressing limitations. The main limitation of using texture memory is the size of texture references, as the maximum allowed width and height for a 2D texture reference bound to linear memory is $2^{16} * 2^{15}$ in Tesla GPUs and $2^{16} * 2^{16}$ in Fermi GPUs. For the applications the texture memory is intended for, these limits impose no real limitations; however, 3D data structures require some form of mapping in order to accommodate the additional dimension in a 2D space.

Proposed solution. In order to process the entire volume using texture memory, the volume is divided into subsections, from now on subvolumes, creating a stack. The dimension of each subvolume is determined by the maximum allowed width and height for a 2D texture reference, as mentioned in the previous section. As the first and/or last layer of nodes of each subvolume needs to access the preceding and following layers, the subvolume size is set so that it can fit the last layer of the preceding subvolume, if any, and the first layer of the following, if any.

Let wx be the width of the volume and the fastest varying dimension in linear memory, wy the depth, and wz the height. The texture references are created by mapping the 3D subvolume to a 2D surface whose width equals wx and whose height depends on the location of the subvolume in the stack. Three different cases arise, as can be seen in Fig. 4: the subvolume is the first in the stack, the subvolume is in the middle, the subvolume is located on top. When located the first in the stack, the first layer of nodes does not need to access the preceding layer of nodes, so the height of the texture reference equals $wy * (wz_{\text{subvolume}} + 1)$, $wz_{\text{subvolume}}$ being the height of the subvolume. In the second case, the first and last layers of nodes in the subvolume access the last and the first layer of the preceding and following subvolumes, the height of the texture reference being $wy * (wz_{\text{subvolume}} + 2)$. In the third case, the subvolume height is the remainder of the division of the volume in equally sized subvolumes of height $wz_{\text{subvolume}}$, and the height of the texture reference equals $wy * (wz_{\text{remaining}} + 1)$.

This strategy has two main benefits: it bypasses the texture reference size limitation, allowing for the simulation of big rooms, and serves as a way of partitioning the data set in order to be processed in multi-GPU environments. After the division of the volume into subvolumes, the subvolumes are then processed serially: the CPU code is in charge of texture reference creation, whilst the GPU is in charge of processing every subvolume.

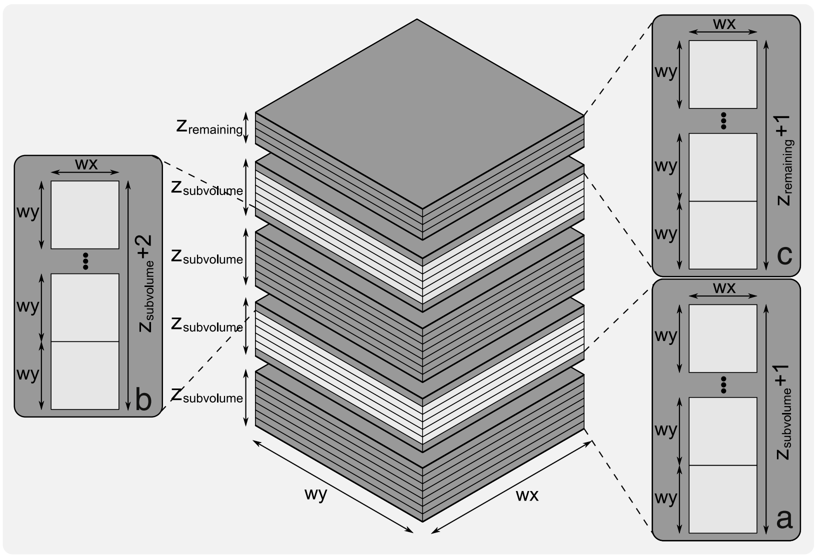


Fig. 4. Volume subdivision. Three different cases. (a) The texture reference includes the following layer of nodes, (b) the texture reference includes the preceding and following layers of nodes of the subvolume, and (c) the subvolume includes only the preceding layer of nodes.

4.2.3. Shared memory implementations

The shared memory implementations make use of shared memory to store reusable data and to prevent any unnecessary access to global memory. In this case, air pressure values from previous, current, and next slices are stored in shared memory, allowing the reads performed within the same slice to access a much faster memory space than global memory. Since shared memory is a per-thread block memory space, threads at the outer edges of thread blocks need to access values that are not stored in shared memory. Two approaches have been implemented to address this issue: one of them reads directly from global memory (from now on referred to as the *Shared Global Memory* implementation) whilst the other fetches the data from texture cache (from now on referred to as *Shared Texture Fetching*). We chose not to allocate additional space in shared memory to store the values that fall outside thread blocks as they are read only once by a single thread, and therefore cannot be reused. Fig. 3 shows the pseudocode representations of these two approaches.

4.3. Other improvements

Besides the implementation guidelines described in Section 4, several additional improvements can be added to the kernel codes, further increasing the computation speed. This section describes a way of reducing memory usage, the optimum thread block dimensions, and the proposed strategy to process arbitrarily shaped rooms.

4.3.1. In-place approach

According to Eq. (1), the computation of pressures at time step $n+1$ requires information from time steps n and $n-1$. As global thread synchronization is achieved by launching different kernel instances, it is possible to use an *in-place* approach. It consists in storing model data just for time steps n and $n - 1$, with each $n + 1$ time step substituting the data related to $n - 1$, thus allowing the simulation of bigger enclosures as less space is used to store model data.

4.3.2. Rectangular thread block dimensions

Thread blocks are typically flat and square in shape. This helps when working with arbitrarily shaped enclosures as it keeps the amount of padding data introduced to a minimum. However, as will be discussed in Section 5.2.1, a square thread block provokes branching when executing the *warps* that form the blocks, and flattening the thread blocks eliminates this issue. Several shapes have been tested, and the results are shown in Section 5.2.1 together with the performance comparison of the different implementations.

4.3.3. Processing of arbitrarily shaped rooms

The ultimate aim of a room acoustic simulator is the ability to process arbitrarily shaped rooms. Since the CUDA threading model allows thread block grids only to be rectangular, it is necessary to introduce padding data, surrounding the room with a rectangular enclosure in order to process it. The approach consists of two main points: storage of boundary data in the same data structure as air pressure data, and computation of the FDTD only when useful data are present.

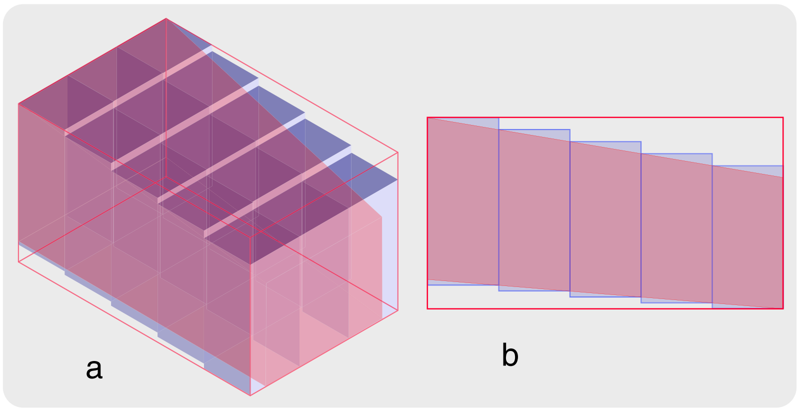


Fig. 5. Schematic view of a room decomposition in blocks the same size as thread blocks during preprocessing. (a) 3D view of the room inside the rectangular enclosure and the array of 3D blocks representing the useful model data zones, (b) 2D projection of the room inside the enclosure.

The storage of boundary data together with air pressure data eliminates the need for an extra data structure with the same size as the ones that store model data. The proposed method is to differentiate air nodes, boundary nodes, and padding nodes using values near the upper limit of the floating point (single or double precision) representable numbers. As the simulations are stable, and sound sources need to emit in a safe pressure margin in order not to saturate, it is safe to use this values.

The introduction of padding has two obvious downsides: it increases the size of the model data structure and slows down the simulation of the enclosures as valuable resources are used to process useless data outside the room. The proposed solution is to process the model data before the first kernel execution in order to detect which parts are useless and which parts are needed for the simulation, identifying from and to which height each thread block needs to be processed. This procedure produces two matrices, each with as many elements as thread blocks composing the thread block grid. The first matrix contains the height from which each thread block needs to start computing the FDTD, and the second matrix contains the height at which it needs to stop. Fig. 5 shows a schematic view of the results produced by the process in a 3D room resembling an auditorium.

5. Performance comparison

This section presents several tests to compare the performance provided by each implementation, the speed gains obtained by applying the improvement described in Section 4.3.2, the speed-ups and relative speed-ups compared to a recent parallel implementation of the FDTD algorithm for room acoustic modelling using double precision model data, and a demonstration of arbitrarily shaped room processing using the method described in Section 4.3.3. Commodity hardware has been chosen for the simulations and tests to show that even with these kinds of component the performance differences between implementations are significant and that processing clusters based on consumer-grade GPUs are a viable option.

5.1. Test setup

The simulations are carried using two different Nvidia GPU cards: a GeForce GTX 260 built with the Tesla architecture and a GeForce GTX 480 built the more recent Fermi architecture [13]. The GeForce GTX 260 is equipped with 1.7 GB of global memory, a memory interface width of 448 bits, 192 CUDA cores, and a memory bandwidth of 111.9 GB/s. The GeForce GTX480 is equipped with 1.5 GB of global memory, a memory interface width of 384 bits, 480 CUDA cores, and a memory bandwidth of 177.4 GB/s. The CPU used is an Intel Core i5 at 3.20 GHz with 4 GB of PC3-10700 DDR3 SDRAM.

5.2. Experiment 1

The first experiment consists of a single precision floating point performance comparison between the different implementations of the approaches introduced in the previous section using both GPU cards. To test the benefits of rectangular thread blocks two cases are tested: a block size of 16×16 and a block size of 32×8. Additionally, the results include the number of registers used by each implementation and the instruction count of the simulations. For the simulation, 800 time steps were computed in a grid of dimensions $512 \times 512 \times 512$, which equals 134 million nodes, and a Gaussian hard source [20].

5.2.1. Results

Table 1 shows the throughput comparison for a block size of 16×16, whilst Table 2 shows a comparison for a block size of 32×8, both of them for single precision model data.

Table 1
Algorithm throughput in Mvoxels/s for 16 × 16 thread block dimensions.

Implementation	GTX 260 (Tesla architecture)			GTX 480 (Fermi architecture)		
	Throughput (Mvoxels/s)	Registers used	Instructions	Throughput (Mvoxels/s)	Registers used	Instructions
Tiled shared	1526	10	20.5e9	4470	11	22.3e9
Tiled global	1784	9	17e9	4622	11	18.6e9
Naive	1712	13	7.7e9	3315	13	10.5e9
Shared texture fetching	2252	15	15.5e9	5189	19	17.6e9
Shared global	2441	13	14.4e9	5268	17	14.9e9
Pure texture fetching	3770	16	8.5e9	4920	18	11.8e9

Table 2
Algorithm throughput in Mvoxels/s for 32 × 8 thread block dimension.

Implementation	GTX 260 (Tesla architecture)			GTX 480 (Fermi architecture)		
	Throughput (Mvoxels/s)	Registers used	Instructions	Throughput (Mvoxels/s)	Registers used	Instructions
Tiled shared	1528	10	20.4e9	5666	11	22e9
Tiled global	1786	9	17e9	5700	11	18.5e9
Naive	2464	13	7.7e9	5122	13	10.5e9
Shared texture fetching	2272	15	15.4e9	5483	19	17.3e9
Shared global	2463	13	13.8e9	7279	17	14.9e9
Pure texture fetching	3808	16	8.5e9	7201	18	11.8e9

For the Tesla architecture, a strong relationship between the instruction count and the performance can be observed. Tiled approaches prove to be inefficient and instruction limited, as at least a division and a modulo operation are needed to obtain the global position of each thread. Instruction per cycle (IPC) counts of 1.08 and 1.03 were obtained for the approaches using shared memory and direct global memory access, respectively. For bigger rooms, 2D tiling may be needed in order to process the entire volume, worsening the performance even further.

Sliced approaches that make use of shared memory are, too, instruction limited, as demonstrated by IPC counts of 1.10 and 1.20 for the *Shared Texture Fetching* and the *Shared Global* implementations, respectively. This is due to the branching and synchronization overhead in warp execution when accessing cached data. Conditional statements and synchronization barriers are needed, thus resulting in serial thread execution within a warp.

In this architecture, the best performance is achieved by the *Pure Texture Fetching* implementation by a 54% margin over the *naive* implementation. This high throughput is due to the flat threadfront in conjunction with the 2D locality optimization of texture memory, thus allowing for temporal and spatial locality.

Thanks to its increased computational power, the Fermi architecture provides an important performance increase over the Tesla architecture, as can be observed despite the high instruction count, doubling the speed of the tiled shared memory approach and significantly increasing the performance of the sliced approaches that makes use of shared memory.

Thread block dimensions of 32 × 8 are the optimum choice, as can be seen in Table 2, concluding that a flattened thread block reduces the branching produced in the previous case. The sliced approach is seen to be the fastest, and, within it, the *Pure Texture Fetching Implementation* and the *Shared Global* implementations provide respectively a 26% and a 27% performance increase over the fastest tiled approach implementation.

The high throughput provided by the *Shared Global* implementation is thanks to the higher computational power of Fermi devices, with instruction execution being essentially free in this case, in contrast to global memory accesses, that, although cached, still keep a high latency for cache misses. This computational power effectively hides the synchronization barriers and conditional statements needed to work with shared memory, and brings about the benefits from data sharing from within the same slice.

Better results can be expected for the *Pure Texture Fetching* implementation in Fermi devices with a higher memory bandwidth, and for the implementations using shared memory in devices with a higher CUDA SM count.

While additional testing was performed for 64 × 4 and 128 × 2 block dimensions, only small performance variations were obtained over 32 × 8 for all the approaches. These dimensions additionally provide a good balance between speed and memory efficiency when working with arbitrarily sized rooms, where padding of the model data is often necessary to accommodate the thread block grid.

5.3. Experiment 2

In this experiment, to assess the computation speed using double precision data, a comparison is performed between the results presented in [10] for the basic scheme. The simulations compute one second of output at 44.1 kHz with varying grid sizes, and, in this case, only the GTX 480 GPU is used. It is important to note the difference in theoretical peak double precision performance between the GPUs used in the tests. While the Tesla C2050 used in [10] have a peak double precision performance of 515 Gflops, the GTX 480 provides 168 Gflops, three times less computation power.

Table 3
Algorithm throughput in Mvoxels/s and times for a 16 million point grid size, speed-up and relative speed-up to the sequential implementation and the parallel implementation using the Fermi architecture respectively presented in [10].

Implementation	Throughput (Mvoxels/s)	Registers used	Time (s)	Speed-up	Relative speed-up
Tiled shared	3322	16	222.7	123	1.62
Tiled global	2834	20	261	105	1.38
Naive	2334	20	317	87.02	1.14
Shared texture fetching	3644	20	203	135.9	1.78
Shared global memory	4016	20	184.23	148	1.95
Pure texture fetching	2101	20	352	78.37	1.02

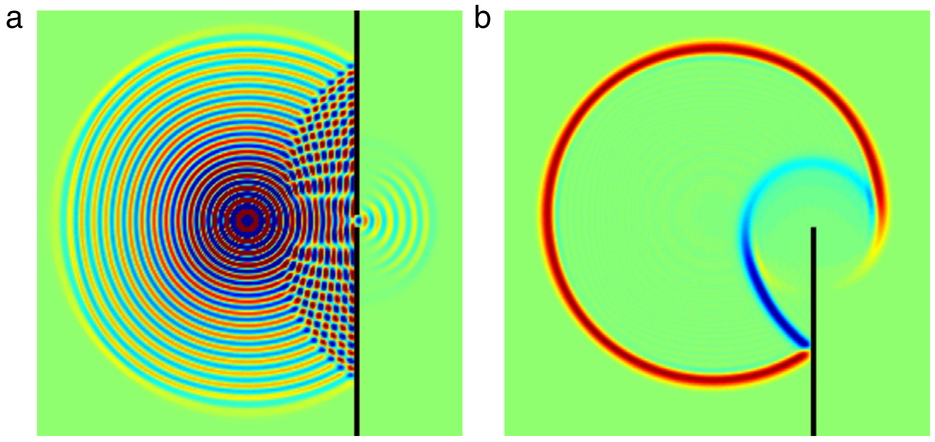


Fig. 6. Wave diffraction effects. (a) Sound wave propagating through a small opening, (b) edge diffraction in a noise barrier.

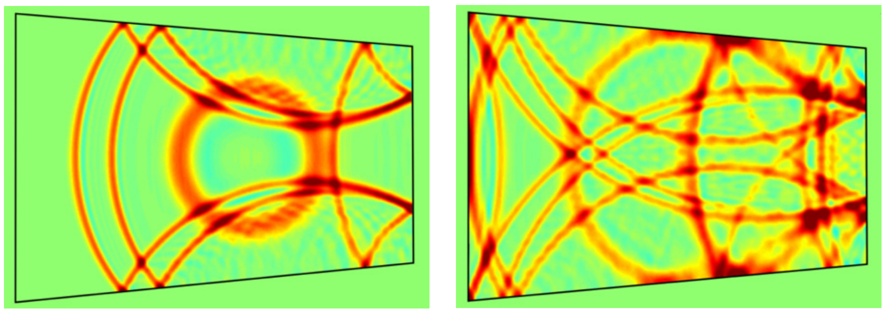


Fig. 7. Sound wave propagation in a 3D enclosure. Horizontal slice of the enclosure at two different time steps.

5.3.1. Results

Table 3 show the throughput comparison for the different implementations computing double precision model data and the speed-ups and relative speed-ups to the results obtained in [10] for a 16 million point grid size.

The GeForce GTX480 GPU, despite its poor double precision computation power compared to the Tesla C2050 GPU, manages to significantly increase the throughput, thanks to the optimized strategy to access model data and with the help of its higher memory bandwidth. Additional testing has been performed for smaller grid sizes, the same tested in [10], with consistent relative speed-ups except for the Pure Texture Fetching implementation, which presents a value lower than 1 already at a 4 million grid size.

5.4. Experiment 3

This last experiment consists of a set of tests that validate the results and the expected behaviour of the simulated sound waves, that is, wall reflection and diffraction around edges and small openings. Fig. 6 shows the diffraction effect that appears at the edge of a sound barrier when a broadband point source is used and the diffraction that appears when a sound wave propagates through a small opening. Finally, Fig. 7 shows a sound wave propagating in the 3D enclosure made of hard walls described in Section 4.3.3 at two different time steps.

6. Conclusions and future work

Room acoustics modelling and auralization have been greatly benefited from the development of new many-core architectures in recent years. GPUs stand out as an affordable example, providing great processing power at a low cost. CUDA offers a simple way of taking advantage of the general-purpose processing capabilities of the GPU; however, special care must be taken in order to make a complete and optimized use of all the subsystems in the device. In this paper we have performed an exhaustive study of how different processing strategies behave in respect to the calculation speed of the FDTD algorithm applied to room acoustics simulation, providing at the same time several implementation guidelines to improve the performance of similar algorithms.

For this, we identified two main processing strategies, and from them developed six different implementations covering the usage of all the memory subsystems the GPU offers. It has been demonstrated that the sliced processing method is the fastest in both hardware architectures tested. It has been proved that the *Pure Texture Fetching* implementation provides the best results when using the Tesla architecture, thanks to the dedicated addressing units and data locality features of this memory. As for the Fermi architecture, the *Shared Global* implementation offers the best performance processing of single or double precision floating point model data. Moreover, speed-ups of two orders of magnitude were obtained over optimized serial implementations of the algorithm and almost a two-fold performance increase over recently published parallel implementations.

Additional testing with a GPU card built with the latest version of the Fermi architecture showed relevant (if not erratic) variations in performance, most likely due to the new superscalar features are added to the SMs; therefore future work should explore the new optimization possibilities these new architectural improvements can offer.

Acknowledgements

We thank the anonymous reviewers for their valuable comments and suggestions. We would also like to thank Nvidia for its support through its Academic Partnership Program. This work has been supported by the Ministry of Education and Science under the project TEC2009-14414-C03-01.

References

[1] M. Bernaschi, M. Fatica, S. Melchionna, S. Succi, E. Kaxiras, A flexible high performance Lattice Boltzmann GPU code for the simulations of fluid flows in complex geometries, Concurrency and Computation: Practice and Experience 22 (2009) 1–14. doi:10.1002/cpe.1466.
[2] Wei-Nien Chen, Hsueh-Ming Hang, H.264/AVC motion estimation implementation on compute unified device architecture (CUDA), in: IEEE International Conference on Multimedia and Expo, ICME, 2008, pp. 697–700. doi:10.1109/ICME.2008.4607530.
[3] S. Adams, J. Payne, R. Boppana, Finite difference time domain (FDTD) simulations using graphics processors, in: Proceedings of the Department of Defense High Performance Computing Modernization Program Users Group Conference, IEEE Comp. Soc., Washington, DC, USA, 2007, pp. 334–338.
[4] A. Balevic, L. Rockstroh, A. Tausendfreund, S. Patzelt, G. Goch, S. Simon, Accelerating simulations of light scattering based on a finite-difference time-domain method with general purpose GPUs, in: Proc. of the 11th IEEE Int. Conf. on Comp. Science and Eng., IEEE Comp. Soc., Washington, DC, USA, 2008, pp. 327–334.
[5] S.E. Krakiwsky, L.E. Turner, M.M. Okoniewski, Acceleration of finite-difference time-domain (FDTD) using graphics processor units (GPU), in: IEEE MTT-S International Microwave Symposium Digest, vol. 2, 2004, pp. 1033–1036. ISBN: 0149-645X. doi:10.1109/MWSYM.2004.1339160.
[6] S. Adams, J. Payne, R. Boppana, Finite difference time domain (FDTD) simulations using graphics processors, in: Proceedings of the Department of Defense High Performance Computing Modernization Program Users Group Conference, IEEE Computer Society, 2007, pp. 334–338.
[7] L. Savioja, D. Manocha, M. Lin, Use of GPUs in room acoustic modeling and auralization, in: Proc. Int. Symp. on Room Acoustics, ISRA, Melbourne, 2010.
[8] A. Southern, D. Murphy, G. Campos, P. Dias, Finite difference room acoustic modelling on a general purpose graphics processing unit, in: 128th Audio Eng. Soc. Convention, London, UK, May 2010.
[9] L. Savioja, Real-time 3D finite-difference time-domain simulations of low and mid-frequency room acoustics, in: 13th Int. Conf on Digital Audio Effects, September 2010.
[10] C. Webb, S. Bilbao, Computing room acoustics with CUDA-3D FDTD schemes with boundary losses and viscosity, in: IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP, May 2011, pp. 317–320. doi:10.1109/ICASSP.2011.5946404.
[11] C. Webb, S. Bilbao, Virtual room acoustics: a comparison of techniques for computing 3D-FDTD schemes using CUDA, in: 130th Audio Engineering Society Convention. London, UK, May 2011.
[12] D. Botteldooren, Finite-difference time-domain simulation in a quasi-Cartesian grid, J. Acoust. Soc. Am. 95 (5) (1994) 2313–2319.
[13] N. Leischner, V. Osipov, P. Sanders, Fermi Architecture White Paper, 2009.
[14] E. Lindholm, J. Nickolls, S. Oberman, J. Montrym, NVIDIA Tesla: a unified graphics and computing architecture, IEEE Micro 28 (2) (2008) 39–55.
[15] Nvidia Corp, CUDA C Programming Guide v3.2. http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf.
[16] Nvidia Corp, CUDA C Best Practices Guide v3.2. http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Best_Practices_Guide.pdf.
[17] S. Ryoo, C.I. Rodrigues, S.S. Baghsorkhi, S.S. Stone, D.B. Kirk, W.W. Hwu, Optimization principles and application performance evaluation of a multithreaded GPU using CUDA, in: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP, 2008, pp. 73–82.
[18] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, A. Moshovos, Demystifying GPU microarchitecture through microbenchmarking, in: IEEE International Symposium on Performance Analysis of Systems Software, ISPASS, March 2010, pp. 235–246.
[19] P. Micikevicius, 3D finite-difference computation on GPUs using CUDA, in: GPGPU-2: Proceedings of the 2nd Workshop on General Purpose Processing on Graphics Processing Units, Washington, DC, USA, 2009, pp. 79–84.
[20] J. Angus, A. Caunce, A GPGPU approach to improved acoustic finite difference time domain calculations, in: 128th Audio Engineering Society Convention, London, UK, May 2010.