Shader-based Physical Modelling for the Design of Massive Digital Musical Instruments

Victor Zappi
Department of Advanced Robotics
Istituto Italiano di Tecnologia
Genoa, Italy
victor.zappi@gmail.com

Andrew Allen
Google, Inc.
Mountain View, CA
bitllama@google.com

Sidney Fels
Department of Electric and Computer Engineering
University of British Columbia
Vancouver, BC
ssfels@ece.ubc.ca

ABSTRACT

Physical modelling is a sophisticated synthesis technique, often used in the design of Digital Musical Instruments (DMIs). Some of the most precise physical simulations of sound propagation are based on Finite-Difference Time-Domain (FDTD) methods, which are stable, highly parameterizable but characterized by an extremely heavy computational load. This drawback hinders the spread of FDTD from the domain of off-line simulations to the one of DMIs. With this paper, we present a novel approach to real-time physical modelling synthesis, which implements a 2D FDTD solver as a shader program running on the GPU directly within the graphics pipeline. The result is a system capable of running fully interactive, massively sized simulation domains, suitable for novel DMI design. With the help of diagrams and code snippets, we provide the implementation details of a first interactive application, a drum head simulator whose source code is available online. Finally, we evaluate the proposed system, showing how this new approach can work as a valuable alternative to classic GPGPU modelling.

Author Keywords

Physical modelling synthesis, GPU, OpenGL, DMI design

ACM Classification

• Computing methodologies∼Massively parallel algorithms • Computing methodologies∼Real-time simulation • Applied computing∼Sound and music computing

1. INTRODUCTION

Physical modelling has the potential to synthesize a wide range of musical sounds to be used in Digital Musical Instruments (DMIs). The high number of parameters often involved in physical simulations makes possible to achieve fine audio control [14] and attracts DMI designers interested in exploring novel interactive metaphors [11].

The work course of sound propagation in 2D is the Finite-Discreet Time-Domain (FDTD) method, which can be used to model different phenomena at the basis of sound and music. Examples of FDTD-based physical modelling include the simulation of drums [10], wind instruments [1] as well as voice [12].

However, physical simulations and FDTD are notorious for being computationally extremely heavy. To simulate 2D sound wave propagation using an FDTD scheme on a grid of only 40x40 points, a system must be capable of running at least 64000 floating point operations per sample. Running at audio rate in real-time, this translates in almost 3 Giga FLOPS, a big challenge for the technology underlying an average DMI.

The use of Graphics Processing Units (GPUs), as opposed to Central Processing Units (CPUs), is becoming a common solution to boost the performances of non-branching, parallelizable physical modelling synthesis, like the case of FDTD. *General Purpose* computing on GPUs (GPGPU) frameworks facilitate the development of parallel programs that make use of the GPU's streaming multiprocessors for applications that do not target graphics rendering. However, due to the complexity of the GPU's architecture and to the intrinsic challenges of parallel programming paradigms, it is still not an easy goal to leverage off the full computational power of a graphics card when using GPGPU.

In this paper, we present the details of a different approach to GPU physical modelling synthesis, working within the standard graphics pipeline. Based on a novel OpenGL implementation, our system combines vertex and fragment shaders to allow for seamless full usage of the GPU's streaming multiprocessors and texture memory. The result is an extremely fast alternative to GPGPU implementations of massive simulations, that allows for sample-based control of synthesis parameters and can render a useful real-time graphical representation of the state of the system.

The contribution of this paper is threefold. First, we show how FDTD solvers map to the graphics rendering pipeline, to implement highly optimized real-time physical modelling synthesis. Second, we include code snippets as well as a link to the source code of a complete C++ and OpenGL Shading Language (GLSL) example of a drum head simulator, for the NIME community to replicate it an extend it. Finally, we compare our system with two other implementations, one running on the CPU, the other based on GPGPU [10], and discuss the advantages of the proposed approach from the perspective of novel DMI design.

2. RELATED WORK

As the desire for more accurate and expressive sound synthesis grows, so do the computational requirements. To accelerate computation, many signal processing algorithms have been adapted to the GPU. Several examples of Fast Fourier Transform implementations based on GPGPU or shader languages can be found in the literature [4, 3]; this led to the development of a standard CUDA library as well

NIME'17, May 15-19, 2017, Aalborg University Copenhagen, Denmark.

as a native OpenGL implementation[1]. Both a million-voice real-time additive synthesizer [9] and a multi-object modal synthesis engine [14] were demonstrated using GPU hardware that is now several years-old.

As already introduced, GPUs are leveraged for their ability to efficiently execute and synchronize many lightweight cores for parallelized operation. FDTD solvers, while computationally expensive, often easily parallelize, making GPUs an obvious choice for processing. Room acoustics simulation can use FDTD equations, thus, GPUs have been used quite often in this domain to measure reflections and compute band-limited room impulse responses [8]. Pre-computed full-band room impulse responses were later studied [5]. Additionally, many FDTD-based numerical models of various wind and percussive musical instruments have been explored [2], including a GPU-based timpani simulation that accurately models sound synthesis from the drum membrane [13].

Though the domain of audio-rate FDTD simulation is largely offline, there are many recent examples demonstrating real-time performance. Musical instrument simulation is a natural choice for such applications. Researchers demonstrated a variety of possibilities including a percussion instrument using modal synthesis with multi-touch input [6], a 2D drum membrane simulation [10], and 2D virtual wind instrument sandbox simulation [1]. In the area of room acoustics, real-time 3D band-limited small room simulations have been studied [7].

3. SYSTEM DESIGN

In this section, we describe the implementation details of our GPU drum head simulator. This practical example is used to show how to set up an OpenGL-based physical modelling audio synthesis application. First, the acoustic model underlying the simulation is presented; then, the actual main steps of its GPU implementation are illustrated, with the support of diagrams and code snippets. The full and commented source code of the application can be found at `http://toomuchidle.com/opengl-fdtd/`.

3.1 Acoustic Model

The behavior of our drum head is modelled according to the following discretization of the standard 2D acoustic wave equation:

$$p^{n+1} = \frac{2p^n + (\mu-1)p^{n-1} + \rho(p_L+p_R+p_U+p_D - 4p^n)}{\mu+1} \quad (1)$$

with:

$$p_{L,R,U,D} = \begin{cases} p^n\gamma & \textit{if boundary} \\ p^n_{l,r,u,d} & \textit{else} \end{cases} \quad (2)$$

The equations above describe a quite common FDTD scheme, where p denotes the acoustic pressure value in the current grid point and the indices $n+1$, n and $n-1$ refer to the time step, respectively next, current and previous; p_l, p_r, p_u, p_d represent the pressure value sampled on the four immediate neighbors of the current point, on the right, on the left, on top and below it respectively. Equation 2 describes how boundary conditions are enforced using these neighbor values; the term γ in the boundary case allows to transition between two conditions: fully clamped edge ($\gamma = 0$) and free edge ($\gamma = 1$). The properties of the simulated material are defined by the absorption coefficient μ ($0 < \mu < 1$) and by the term ρ, which is defined as $\rho = \left(c\frac{\Delta t}{\Delta s}\right)^2$; c is the speed of sound in the medium (i.e., the simulated material), Δt is the inverse of the simulation sample rate (44100 Hz) while Δs is the size of each single point on the grid. To assure stability, the Courant-Friedrichs-Lewy condition must be satisfied ($\rho \leq 0.5$).

Our drum head is composed of a grid of points surrounded by a boundary layer. When excited by an impulse (i.e., a virtual strike on the drum head), the pressure values at each point start to oscillate and can be sampled and treated as an audio stream. Different sounds can be obtained by modifying the material's properties (μ and ρ), and by means of changing the number of grid points, the shape and the type of the enclosing boundary layer (from clamped to free). Furthermore, it is possible to change the bandwidth of the excitation impulse, to simulate different kinds of strikes, e.g., sticks, mallets.

3.2 GPU Implementation

The GPU implementation of the solver relies on the usage of a texture to store pressure propagation on our simulated drum head, with one texture fragment representing a single grid point. The actual pressure calculation algorithm (Equation 1 and Equation 2) is defined in a fragment shader, that determines the behavior of each grid point and runs in parallel on the drawn texture fragments. The number of concurrent threads depends on the specifications of the used GPU. An OpenGL Frame Buffer Object (FBO) is employed to make the shader render to the texture instead of filling the render buffer (and render on screen).

The FDTD scheme described in Section 3.1 is an explicit solver, since the next pressure value depends only on the current and the previous state of the system. In particular, to compute the next value of a pressure point, the solver needs to know the current and the previous value of the same point on the grid, as well as the current values of the four neighbor points. This scheme must be preserved in the GPU implementation and in the operations defined in the fragment shader.

3.2.1 Fragment Read/Write Access

Since version 4, OpenGL supports read/write fragment operations on each fragment of a texture, a feature that can be appropriated to design a highly optimized parallel solver. In particular, when the RGBA channels of a fragment are drawn, OpenGL 4 allows values sampled from other portions of the texture to be used to carry out the calculation of the newly assigned color (as opposed to using only the current fragment values). Thanks to this feature, instead of using separate textures to store pressure values for each time step, we can employ a single texture divided in portions, using the R channel to save pressure. Since the solver needs the current and the previous pressure values, the texture is divided in two parts only, one to store values from time step n, the other to store values from time step $n-1$. At each simulation cycle, only the fragments of the $n-1$ portion are updated (rendered) and filled with the $n+1$ values, computed by the fragment shader; then, time step indices are swapped, i.e., the updated portion changes from $n-1$ to n and, vice versa, portion n becomes $n-1$. In this way we work in a thread-safe context, since we always retrieve neighbor pressure values for time step n ($p_{L,R,U,D}$ in Equation 1) and the current point's pressure values n and $n-1$ ($p^{n,n-1}$ in Equation 1) from portions of the texture that are not being modified by concurrent threads. Eventually, to assure thread synchronization, a memory barrier call is performed at the end of every simulation cycle (see the code snippet in Section 3.2.4).

This structure makes the system very lightweight. The single texture is bound only once during the initialization

[1]GLFFT website: https://github.com/Themaister/GLFFT

of the simulation, while the indices swapping is handled by means of alternatively enabling two sets of *draw arrays* within an OpenGL Vertex Buffer Object (VBO) (more details in Section 3.2.3 and Section 3.2.4). In contrast, using two different textures would require alternate binding, an operation that would remarkably slow down the whole rendering process.

3.2.2 Full Texture Layout

As described in Section 3.2.1, the used texture is divided into two portions to accommodate the pressure values from time steps n and $n-1$. These two portions, which from now on will be referred to as $Tex0$ and $Tex1$, are equally sized (each portion contains as many pixels as the number of grid points in our simulation domain) and lie side by side within the full texture. At each simulation cycle, $Tex0$ and $Tex1$ alternatively contain the current pressure values in the domain and the previous ones.

The full texture layout (Figure 1) also includes a third and a fourth portion ($Tex2$ and $Tex3$), placed on top of $Tex0$ and $Tex1$. $Tex2$ is much like an audio buffer inside the texture; it is one pixel high, as wide as $Tex0$ and $Tex1$ placed side by side, and it serves to store the pressure values computed on the grid point we want to sample. At each simulation cycle, a single pressure sample is lined up in $Tex2$, which is then read by the CPU once the buffer is full, by means of *getPixels*, a standard OpenGL Pixel Buffer Object (PBO) operation. The details on how the sampling happens in the fragment shader are illustrated in Section 3.2.5.

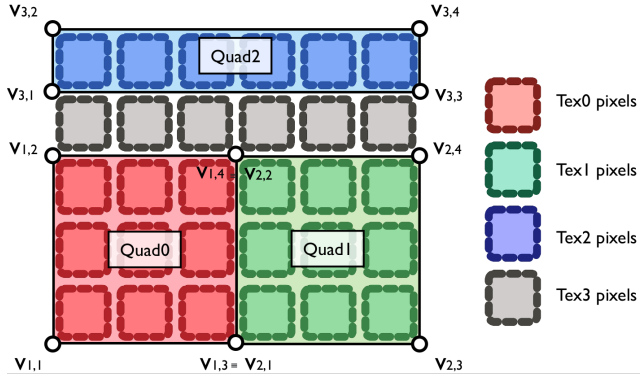

Figure 1: This diagram shows the full texture layout and its mapping on the vertices.

The fourth and final portion ($Tex3$) has same size as $Tex2$ and is placed between it and $Tex0/Tex1$. This portion acts as an empty separator between the audio buffer and the actual simulation domain, so that the fragments placed on the top edge of $Tex0$ and $Tex1$ do not access the audio samples stored in the buffer when accessing p_U. The channels of this texture portion contain all zeros and they are never updated throughout the simulation.

$Tex3$ is the only extra portion needed to prevent the simulation from being corrupted by "dirty" samples. Whenever a fragment tries to fetch a sample outside the texture (e.g, p_D from the bottom edge fragments of $Tex0$ and $Tex1$), a black color value is retrieved (i.e., no pressure).

3.2.3 Vertices and Texture Mapping

Once the layout of the full texture has been designed, it is necessary to define the polygons on whose faces OpenGL will place the different portions of the texture, to read (texture mapping) and update (render-to-texture) the pixels. In this way, invoking a draw call on a specific polygon will result into the proper read/write update of $Tex0$, $Tex1$ (simulation step) or $Tex2$ (audio sampling).

We use three different flat rectangular surfaces ($Quad0$, $Quad1$ and $Quad2$) as polygons, each described by four coplanar 3D vertices ($z = 0$). These three Quads cover almost the whole OpenGL workspace and their proportions are the same as the ones of the texture portions $Tex0$, $Tex1$ and $Tex2$: $Quad0$ and $Quad1$ are equally sized and placed side by side, while $Quad2$ is a single line placed on top of them, but separated by an empty row. This displacement, combined with the usage of the FBO, determines a physical overlapping of the polygons with the texture portions (Figure 1), so that $Tex0$ and $Tex1$ are respectively updated every time $Quad0$ and $Quad1$ are drawn, while a draw call on $Quad2$ triggers the update of $Tex2$. There is no Quad associated with $Tex3$ (thus the empty line left between the Quads), since, as mentioned before, this portion is never updated.

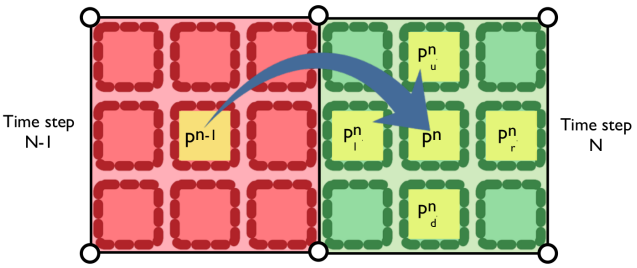

Figure 2: Fragment access pattern. The fragments that the shader needs to sample to compute the next pressure value are highlighted in yellow.

The position in the workspace of each vertex is stored into a VBO, together with a series of attributes. Attributes are the standard way to perform texture mapping; by means of coupling each vertex with some texture coordinates, the shader program can locally access (read) the texture[2] when drawing all the fragments on the polygons' faces. In our case, the fragment shader has to access six different texture coordinates: p^n, p^{n-1} and the four neighbor values $p_{L,R,U,D}$. As depicted in Figure 2, to read the value p^n needed to compute p^{n+1}, a fragment has to access the portion of the texture that is on the opposite side of the Quad where it resides (from time step $n-1$ to n). For this reason, each vertex is coupled in the VBO with the texture coordinates of the respective point in the opposite texture portion. The same coordinates are then repeated for each of the four direct neighbors by simply applying a one-pixel shift in the selected direction. A final set of coordinates could be added to the VBO to access p^{n-1} as this value at this time is stored in the texture portion overlapped with the Quad that the vertex belongs to. However, to avoid an extra texture sampling operation (repeated for every fragment in every time step), we can store the previous pressure value in the G channel of the texture. By doing so, a fragment can retrieve both p^n (channel R) and p^{n-1} (channel G) with a single color sampling.

Useful additional information about the grid points can be stored in the other unused channels of the texture, i.e., B and A. We use B to distinguish boundary points ($B = 0$) from regular drum head points ($B = 1$), while A is used to identify excitation points ($A = 1$). These settings are passed to the texture during initialization and they can be modified at run-time adding an extra PBO within the source code. The possibility to define different subsections within the domain (also dynamically), each with its own shape and

[2]Since we are using an FBO, the texture accessed by the shader is the same that the shader updates. This happens in a transparent way and does not require specific attribute setups.

excitation points, is particularly handy to turn this application into an actual DMI, as discussed in Section 4.2.

3.2.4 Simulation Cycle

The simulation cycle is composed of two main steps: the advancement of the simulation across the whole grid (simulation step) and the storage in the buffer of the new audio sample from a chosen grid point (audio step).

As introduced in the previous sections, in each cycle the simulation step alternatively updates $Tex0$ or $Tex1$. This generates two different step sub-cases: one where $Quad0$ is drawn and the other where $Quad1$ is drawn. This alternation also affects the audio step, since the buffer has to contain audio samples alternatively picked from one of the two texture portions. To handle this, four subsequent states have been defined, two per each step: $state0$, in which $Quad0$ is drawn (simulation step), $state1$, in which the audio buffer samples $Tex1$ (audio step), then $state2$, during which $Quad1$ is drawn (simulation step) and finally $state3$, when the audio sample is grabbed from $Tex0$ (audio step). The order of the four states enforces audio sampling on the previous pressure values instead of on the most recent ones. This is to avoid *race conditions* between simulation and audio step threads, for the $n+1$ time step update might not be complete yet when the audio steps are called, as is relevant especially in large domains.

A single shader program implements these four behaviors, using the current state to switch between them; its content will be described in the next section. The simulation cycle resides inside of a loop automatically called by an ALSA audio callback when a new buffer is required. Its C/C++ implementation is displayed in the following code snippet:

```
// pass next excitation value
glUniform1f(excitation_loc, excitation);

// simulation step (state0 or state2)
state = quad*2;
glUniform1i(state_loc, state);
glDrawArrays(GL_TRIANGLE_STRIP,
  vertices[quad][0], vertices[quad][1]);

// audio step (state1 or state3)
glUniform2fv(wrCoord_loc, 1, wrCoord);
glUniform1i(state_loc, state+1);
glDrawArrays(GL_TRIANGLE_STRIP, vertices[2][0],
  vertices[2][1]);

// prepare next cycle
quad = 1-quad;
wrCoord[1] = int(wrCoord[1]+1)%4;
if(wrCoord[1]==0)
  wrCoord[0]+=fragWidth;

// re-sync all parallel GPU threads
glMemoryBarrier(GL_TEXTURE_FETCH_BARRIER_BIT);
```

The *vertices* array contains the indices of the vertices that belong to each Quad, reflecting their order within the VBO. These values are passed to the draw call to choose which Quad to render, according to the current state (both $state1$ and $state3$ render $Quad2$, i.e., the audio Quad).

The additional steps pass the new excitation value (floating point variable *excitation*) to the shader and update the state variables for the the next cycles. The array *wrCoord* serves as a write pointer to access the audio buffer in the next available location; its first element contains the x coordinate of the next free fragment in $Tex2$. Since up to four samples can be stored in each fragment (in the RGBA channels), the second element contains the index of the next available channel. When the audio buffer is full, all its pixels (residing in $Tex2$) are read in one block by the CPU and the write pointer is reset. This is also an appropriate place where a second shader program can be called, which simply renders to an OpenGL window the fragments of the latest state of the system to provide visual feedback. Doing so, the visual update is considerably slower than the audio rate; a more frequent screen render (e.g., 44100 Hz) would undermine the performance of the system, without adding any beneficial visual effect.

The excitation value, the write pointer as well as the current state are passed to the shader program as OpenGL *uniforms*. In a similar fashion, other control parameters can be sent to control the shader program in real-time.

3.2.5 Shaders

The drum head simulation is based on two shader programs, one implementing the actual FDTD solver, the other acting as a screen renderer. Each program is composed of a vertex and a fragment shader written in GLSL.

In both cases, the vertex shader applies the positions passed by the draw call to the drawn vertices in the simulation cycle. It also reads the attributes associated with these positions and forwards them to the fragment shader it is combined with. The fragment shader of the screen render is a simple color-mapper that fills the fragment with a different color according to its type (channels B and A) and to the stored pressure (channel R), to visualize the current state of the system.

The solver fragment shader is composed of three main sections. The first one is composed of a main *if-satement* structure that checks the current state (uniform variable) and either triggers the FDTD solver ($state0$ or $state2$) or the audio sampling ($state1$ or $state3$). The following code snippet shows the GLSL implementation of the FDTD solver:

```
// p_n and p_n-1
vec4 frag_c  = texture(txture, tex_c);
vec4 p       = frag_c.rrrr;
float p_prev = frag_c.g;

// neighbours (pl_n, pr_n, pu_n, pd_n)
vec4 p_neigh;
vec4 b_neigh;

// left
vec4 frag_l = texture(txture, tex_l);
p_neigh.r   = frag_l.r;
b_neigh.r   = frag_l.b;

// repeat for right, up and down
//...

// parallel computation of pL,R,U,D (eq. 2)
vec4 pLRUD = p_neigh*beta_neigh +
  p*(1-beta_neigh)*gamma;

// assemble equation 1
float p_next = 2*p.r + (mu-1) * p_prev;
p_next += (pLRUD.r+pLRUD.g+
  pLRUD.b+pLRUD.a-4*p.r)*rho;
p_next /= mu+1;

// excitation
int is_excitation = int(frag_c.a);
p_next += excitationInput * is_excitation;

// pack and return
return vec4(p_next, p.r, frag_c.b, frag_c.a);
```

This code is the most computationally intense part of the algorithm and is repeated for every fragment (grid point) in each simulation cycle, which is called 44100 times per second. The implementation does not include any conditional statements (no branching), which would heavily slow down the computation. The neighbor pressures $p_{L,R,U,D}$ (Equation 2) are calculated with vector arithmetic, which can run in parallel on some GPUs. The variables tex_c and tex_l are two of the five texture coordinate attributes passed by the vertex shader.

The audio sampling shader algorithm is displayed in the following code snippet, again in its GLSL implementation:

```
// copy previous samples within fragment
vec4 retColor = texture(txture, tex_c);

//is this next available fragment?
float diffx = tex_c.r-wrCoord[0];
if( (diffx<fragWidth) && (diffx>=0) ) {
  // sample chosen grid point
  int rdState = 1-(state/2);
  vec2 audioCoord = listenerFragCoord[rdState];
  vec4 audioFrag = texture(txture, audioCoord);

  // silence boundaries
  float audio = audioFrag.r * audioFrag.b;

  // put in the correct channel
  retColor[int(wrAudio[1])] = audio;
}
return retColor;
```

This code fills the audio buffer. It is called only on the fragments belonging to $Tex2$, a very small texture portion, making this section not critical. The array *listenerFragCoord* is the uniform that contains the texture coordinates of the grid point we want to sample, on both $Tex0$ and $Tex1$. The fragments' width is fixed throughout the simulation.

4. EVALUATION

We assessed the performances of the presented application on three different machines. Their specifications are quite heterogeneous and provide us with a good test case. The first machine (M1) is a powerful system, with an 8-core Intel i7-3770K processor (3.5 GHz) and equipped with an Nvidia GTX Titan X GPU (second last Nvidia GPU generation at the time of writing). The second machine (M2) has an extremely powerful CPU, an 8-core Intel i7-4790K (4 Ghz), and mounts a relatively recent Nvidia Quadro K5200 (2014). The last machine (M3) is the least performative of the three, its CPU is an Intel Core 2 Duo (3 GHz) and it is equipped with a cheap Nvidia GeGorce GT 640 (2012). All the machines run Ubuntu with a generic kernel and were connected to an M-Audio M-Track Eight, a semi-professional audio interface suitable for live music performances.

A first metric to evaluate the overall drum head application on the three machines consists of estimating the maximum number of grid points (i.e., domain size) the solver can handle in real-time, without producing buffer *underruns*. We selected four relatively small buffer sizes to be tested (256, 128, 64 and 32 samples), to preserve the responsiveness of the application. Per each buffer size, we gradually enlarged the domain, until underruns were detected. Every buffer/domain configuration was tested four times; during each run, the membrane was repeatedly excited, to simulate the computational load of typical DMI interaction. Finally, we repeated the whole protocol this time using a sequential implementation of the same solver, written in C++ and completely running on the CPU.

The second evaluation presented in this work specifically targets the GPU solver and its implementation. Following the example of Sosnick and Hsu [10], we measured the time the solver takes to synthesize 5 seconds of audio, during which the membrane is excited five times at regular intervals. Since the aim was to isolate the performance of the solver, only the actual GPU computation and the CPU/GPU data exchange processes were timed. Two buffer sizes and three domain sizes were chosen for our time test, to check how the performances scale varying the two parameters.

4.1 Results

Table 1 reports the biggest domain sizes we managed to simulate without incurring in underruns on the three machines, in each configuration. Table entries from architectures (i.e., GPUs and CPUs) mounted on the same machine share the same background color. As expected, the reported values reflect the specifications of each architecture, with the GPU implementation always showing better performances. In particular, the difference between CPU and GPU implementation on the same machine is always in the range of two orders of magnitude. It is interesting to note that some preliminary tests (not included in this work) suggested that the specifics of the CPU do influence the performances of the GPU implementation.

Some table entries are missing for M3, since underruns were consistently detected on both the GPU and CPU implementation when the buffer was set to 32 samples, regardless of the domain size. In the case of the GPU implementation, the same happened also with a buffer of 64 samples.

Table 1: Maximum Domain Sizes.

Buffer size (samples) / *Architecture*	256	128	64	32
GTX Titan X	420x420	416x416	402x402	150x150
Intel i7-3770K	48x48	48x48	48x48	46x46
Quadro K5200	282x282	274x274	260x260	194x194
Intel i7-4790K	76x76	74x74	74x74	58x58
GeForce GT640	118x118	114x114	-	-
Intel Core 2 Duo	18x18	18x18	18x18	-

Table 2 contains the results of the time tests, averaged over ten runs, and shows how the efficiency of the solver gets higher by incrementing the domain size. The background color of the entries is used to highlight different buffer size configurations on the same GPU.

On all the three machines, increasing the buffer considerably sped up the computation; in particular, the boosts reported for M1 and M2 are almost identical, across all the domain sizes. In line with the results showed in Table 1, only M1 managed to simulate in real-time (<5000 ms) the domain composed of 320x320 points (fourth column).

Table 2: Execution Times.

Domain size (points) / *GPU - Buffer size*	20x20	80x80	320x320
GTX Titan X - 128 samples	1054 ms	1239 ms	2896 ms
GTX Titan X - 512 samples	843 ms	959 ms	1574 ms
Quadro K5200 - 128 samples	1295 ms	1615 ms	6186 ms
Quadro K5200 - 512 samples	1003 ms	1318 ms	5818 ms
GeForce GT640 - 128 samples	2257 ms	2991 ms	38699 ms
GeForce GT640 - 512 samples	1126 ms	2666 ms	31605 ms

4.2 Discussion

Extremely big domains can be simulated in real-time using the proposed approach, especially when running on modern GPUs. While previous work in literature showed how a small domain (e.g., 21x21 points) is enough to synthesize a wide range of percussive sounds [10], the possibility to interact with a much higher number of grid points allows for the design of more sophisticated instruments. For example, the

domain can be split in sub-sections, each characterized by different shapes and material parameters, that can also be manipulated in real-time. Furthermore, the direct visualization of the propagating waves through these big domains provides a better understanding of the underlying physical phenomena at different scales, and fosters their creative exploration. The result is a *Hyper Drumhead*, an instrument that can extend physical simulation beyond the boundaries of real physics.

It is somewhat not surprising that modern hardware is capable of massively sized simulations, even when using small buffer sizes. In contrast, the performances achieved by M3 are quite remarkable. Thanks to the proposed approach, it is possible to turn an old machine, with a very slow processor and a cheap GPU, into a responsive instrument running heavy-duty physical modelling synthesis, on domains spanning across more than ten thousand grid points.

The results of the time tests can be used to compare our shader-based approach with the FDTD solver presented by Sosnick and Hsu in [10]. In their work, the authors describe a GPGPU implementation of a drum head algorithm that is analogous to the one underlying our application. Furthermore, the specifications of our GeForce GT640 (mounted on M3) fall in between the ones of two of the three GPUs tested in [10]: the GTX285 (GT1) and the 8800 GT (GT2). This provides us with a starting frame of reference, even if GT1 and GT2 are coupled with more powerful CPUs.

When running with a 512 sample buffer and on a domain of 20x20 on M3, our shader-based system reported execution times that are only slightly higher than the ones of the corresponding GPGPU configuration running on GT1, but quite lower when compared to results from GT2. This is a good result, especially considering the difference between the compared architectures. Since efficiency exponentially grows by enlarging the domain, the comparison should continue using a higher number of grid points. However, the reference GPGPU approach allows domains as big as 21x21 points, since the implementation can only use a limited portion of the texture memory.

5. CONCLUSIONS

In this work, we presented a novel approach to real-time physical modelling synthesis, based on the usage of OpenGL vertex and fragment shaders running on the GPU. We showed how with this method we can use the graphics pipeline to conveniently implement fast and scalable parallel physical models, and in particular the ones based on FDTD solvers.

Since FDTD schemes can be employed to design quite sophisticated DMIs, we shared the details and the source code of an example musical application, which simulates the membrane of a drum head. The performances of the application were evaluated on different GPUs and CPUs, both in terms of spatial and temporal scalability. The proposed shader-based approach proved ideal for the design of massively sized, responsive musical instruments, that simulate wave propagation and can also include a visual representation of the current state of the system.

Although the implementation of FDTD solvers as shader programs well matches the structure of GPUs' multi light-weight cores, we do not suggest it should replace GPGPU programming. GPGPU can still be used to heavily parallelize this kind of computation, possibly achieving similar results. Furthermore, other physical modelling examples that do not quite fit the limitations imposed by the graphics pipeline would find a better implementation with GPGPU architectures, like for example finite element methods.

The main purpose of this paper is to provide NIME and GPU enthusiasts with a new alternative to choose from when working in the challenging domain of real-time physical modelling synthesis.

6. ACKNOWLEDGMENTS

This project is supported by a Marie Curie International Outgoing Fellowship within the 7th European Community Framework Programme and by the Natural Sciences and Engineering Research Council (NSERC) of Canada. We would like to thank Dr. Nikunj Raghuvanshi and Arvind Vasudevan for their precious help, and Nvidia Corporation for donating the graphics card used in this work.

7. REFERENCES

[1] A. Allen and N. Raghuvanshi. Aerophones in flatland: Interactive wave simulation of wind instruments. *ACM Transactions on Graphics*, 34(4):134, 2015.

[2] S. Bilbao and J. Chick. Finite difference time domain simulation for the brass instrument bore. *The Journal of the Acoustical Society of America*, 134(5):3860–3871, 2013.

[3] K. Moreland and E. Angel. The fft on a gpu. In *Proc. of the ACM SIGGRAPH/EUROGRAPHICS conf. on Graphics hardware*, pages 112–119. Eurographics Association, 2003.

[4] Y. Ogata, T. Endo, N. Maruyama, and S. Matsuoka. An efficient, model-based cpu-gpu heterogeneous fft library. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–10. IEEE, 2008.

[5] N. Raghuvanshi and J. Snyder. Parametric wave field coding for precomputed sound propagation. *ACM Transactions on Graphics*, 33(4):38, 2014.

[6] Z. Ren, R. Mehra, J. Coposky, and M. C. Lin. Tabletop ensemble: touch-enabled virtual percussion instruments. In *Proc of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 7–14. ACM, 2012.

[7] L. Savioja. Real-time 3d finite-difference time-domain simulation of low-and mid-frequency room acoustics. In *13th Int. Conf on Digital Audio Effects*, volume 1, page 75, 2010.

[8] L. Savioja, D. Manocha, and M. Lin. Use of gpus in room acoustic modeling and auralization. In *Proc. Int. Symposium on Room Acoustics*, page 3, 2010.

[9] L. Savioja, V. Välimäki, and J. O. Smith III. Real-time additive synthesis with one million sinusoids using a gpu. In *Audio Engineering Society Convention 128*. Audio Engineering Society, 2010.

[10] M. Sosnick and W. Hsu. Efficient finite difference-based sound synthesis using gpus. In *Proc of the SMC Conference*, pages 42–44, 2010.

[11] M. H. Sosnick and W. T. Hsu. Implementing a finite difference-based real-time sound synthesizer using gpus. In *NIME*, pages 264–267, 2011.

[12] M. Speed, D. T. Murphy, and D. M. Howard. Characteristics of two-dimensional finite difference techniques for vocal tract analysis and voice synthesis. In *INTERSPEECH*, pages 768–771, 2009.

[13] C. J. Webb. *Parallel computation techniques for virtual acoustics and physical modelling synthesis.* PhD thesis, The University of Edinburgh, Old College, South Bridge, Edinburgh, 2014.

[14] Q. Zhang, L. Ye, and Z. Pan. Physically-based sound synthesis on gpus. In *International Conference on Entertainment Computing*, pages 328–333. Springer, 2005.