Large scale physical modeling synthesis, parallel computing, and musical experimentation

The NESS Project: Large Scale Physical Modeling Synthesis, Parallel Computing, and Musical Experimentation

Stefan Bilbao[(1)], James Perry[(2)], Paul Graham[(3)], Alan Gray[(3)], Kostas Kavoussanakis[(2)], Gordon Delap[(4)], Tom Mudd[(1)], Gadi Sassoon, Trevor Wishart[(5)], Samson Young

(1): Acoustics and Audio Group/Reid School of Music, University of Edinburgh, Edinburgh, UK

(2): EPCC, University of Edinburgh, Edinburgh, UK

(3): Nvidia, Reading, UK

(4): Maynooth University, Maynooth, Ireland

(5): University of Durham, Durham, UK

Accepted author manuscript (author's final version), Computer Music Journal, MIT Press, 20 November 2019.

Abstract

Physical modeling sound synthesis, emulating systems of a complexity approaching and even exceeding that of real-world acoustic musical instruments, is becoming possible, thanks to recent theoretical developments in musical acoustics and algorithm design. Severe practical difficulties remain, both at the level of the raw computational resources required, and at the level of user control. An approach to the first difficulty is through the use of large-scale parallelisation, and results for a variety of physical modeling systems are presented here. Any progress with regard to the second requires, necessarily, the experience and advice of professional musicians. A basic interface to a parallelised large-scale physical modeling synthesis system is presented here,

accompanied by first-hand descriptions of the working methods of five composers, each of whom generated complete multichannel pieces using the system.

Introduction

Physical modeling sound synthesis is, to say the least, a computationally costly undertaking. Throughout the history of computer music, it has often been the case that, during the development of new synthesis tools, there has been an initial phase during which prototypes were built in specialised hardware. Often the aim was to achieve real- or near real-time performance using relatively established synthesis methods. Well-known examples are the Samson box at CCRMA (Samson 1980) and the 4N series at IRCAM, leading ultimately to the IRCAM Music Workstation (Lindemann et al. 1991) (later IRCAM Signal Processing Workstation), which ran the early signal processing version of Max via customised DSP boards in a NeXT workstation (Puckette 1991).

In the case of physical modeling synthesis, particularly for complex systems, computational costs can be many orders of magnitude beyond those of standard abstract synthesis methods, and thus, for the moment, specialised hardware is again necessary. Real-time performance was not the aim of the NESS project, but rather “reasonable"-time performance—in order to get a glimpse of what kind of sound output is possible. Raw acceleration has thus been one of the main goals, and a variety of approaches have been taken, all relying on parallelisation at different scales.

Physical modeling sound synthesis algorithms for many instrument types, from emulations of existing instruments to purely virtual constructions without a counterpart in the real world, were developed under the NESS Project, and are detailed in the companion article (Bilbao et al. 2019). Though most do not operate in real time, they generate sound quickly enough to constitute a point of departure for musical exploration, which was the second, deeper and less defined goal of NESS. New major

issues relating to usability, interfaces and control emerge—the musician is faced with the large task of not merely learning to play a new musical instrument, but more often than not designing it as well. Sound synthesis in parallel hardware has been explored with regard to standard abstract synthesis techniques (Savioja et al. 2010, 2011), as well as in the case of physical modeling applications, particularly on graphics processing units (GPUs) (Zhang et al. 2005; Hsu and Sosnick-Pérez 2013) and using field-programmable gate arrays (Pfeifle and Bader 2015; Motuk et al. 2007). Alongside work on synthesis, another major thrust was towards the development of large-scale room acoustics simulations on GPU—here our approach intersects with that of work in virtual acoustics, divorced from sound synthesis applications—see, e.g., Southern et al. (2010); Mehra et al. (2012).

This article is a detailed account of the practicalities, both technical and musical, involved in bringing physical modeling synthesis into the hands of musicians. For a more technical view of parallel computing in physical modeling synthesis, see Perry et al. (2015).

Physical Modeling Synthesis: Computational Complexity

Computational complexity for a physical modeling synthesis algorithm is of course highly dependent on the particular model. One very crude measure of the complexity follows from the required state size—i.e., the amount of memory required in order to sufficiently represent the dynamics of a given instrument, given some perceptual criterion. Interestingly, it is possible to provide a rather simple lower bound on such memory requirements. To this end, consider any acoustic entity, characterised by its material and geometry—which could be a musical instrument, or a component of one, or perhaps even an enclosing space. Loss is usually low in any such system, and under linear conditions, the system may be characterised by a number of natural frequencies or

modes $N(f_c)$ below a chosen cutoff frequency f_c. Each such mode behaves, individually, as a harmonic oscillator with two degrees of freedom, and thus requires the updating of two real numbers in memory, and thus the minimum memory requirement is $2N(f_c)$ real numbers (normally double-precision floating point). A synthesis algorithm using less than this amount of memory will necessarily be discarding potentially audible dynamics, and one using more is inefficient—often unavoidably so. In synthesis, $f_c = 20000$ Hz is the most usual choice of cutoff and represents a basic perceptual criterion—the upper limit of human hearing. Although the aforementioned bound on memory requirements follows from an analysis in terms of modes, it is actually quite general—any synthesis method (modal, digital waveguide, FD, etc.) must respect this.

Consider first the basic case of a vibrating string, of fundamental frequency f_0 Hz. Here, $N(f_c) = f_c/f_0$, which is on the order of about 10 to 500 for musical strings, which is relatively small. At the other extreme, consider the case of a room of volume V m^3, and where the sound speed is c m$\cdot$ s^{-1}. Now, N is approximately $4Vf_c^3/c^3$. For a medium-sized room ($V = 2000$ m^3), and where the speed of sound $c = 343$ m$\cdot$ s^{-1}, then $N \approxeq 10^9$, which is very large indeed, but reasonable-time simulation is still within the realm of possibility on specialised hardware (such as GPUs). Ranges of values of N, for a cutoff of $f_c = 20000$ Hz are given for a variety of acoustic objects in Figure 1. Notice in particular the large gulf between problem sizes for 1D and 2D musical instrument components, and for 3D modeling; parallelisation strategies for 3D modeling have a different character, due to the problem size.

Operation counts are more dependent on the particulars of the system at hand—in most nontrivial cases, however, the number of arithmetic operations per second, for a system with a state size of $2N$ and with a sample rate of f_s will scale as at least $O(Nf_s)$ (and of course, from sampling considerations, $f_s \geq 2f_c$). The digital waveguide is a notable exception in this regard, requiring $O(1)$ operations per time step—an efficiency

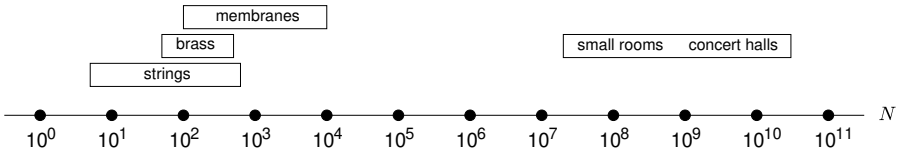

Figure 1. N for various different systems in musical acoustics.

advantage linked specifically to wave propagation over 1D lossless homogeneous media, with the ideal string and acoustic tube as examples. As will be seen, however, actual runtimes are very much dependent on the particular implementation and the possibility of parallelisation. More details about the relevant computational structures appear in the following section. Furthermore, such estimates are very crude—they do not take into account more detailed information about auditory perception, which could be used to further reduce costs.

General Algorithm Structure and Key Operations

The majority of the time-stepping methods employed in the NESS project share many common features across different instrument types. Though it is impossible to describe here the complete workings of all the physical modeling sound synthesis algorithms described here and in the companion article (Bilbao et al. 2019), an attempt is made here to give the user some notion of the technical challenges involved, particularly keeping in mind, from the previous discussion, that in some cases the problem size can be very large. For a more technical presentation of the implementation of such time-stepping methods, see (Bilbao et al. 2014).

Representing State

As a first step towards understanding the implementation of such methods, consider the representations of the state in typical time-stepping algorithms. Depending on the system at hand, the state (at an integer time step n) consists of the values representing the physical variables of the complete system. Such values could represent, for example, the displacements and velocities of a string or membrane, defined over a grid, or pressure and velocity values within an acoustic tube, or a combination of values representing the state of a heterogeneous system made up of a combination of components. See Figure 2 at left. For the purposes of the discussion below (and not necessarily in practice!) it is useful to concatenate the entire state, at a given time step n, as a vector $\mathbf{u}^n$. See Figure 2 at right.

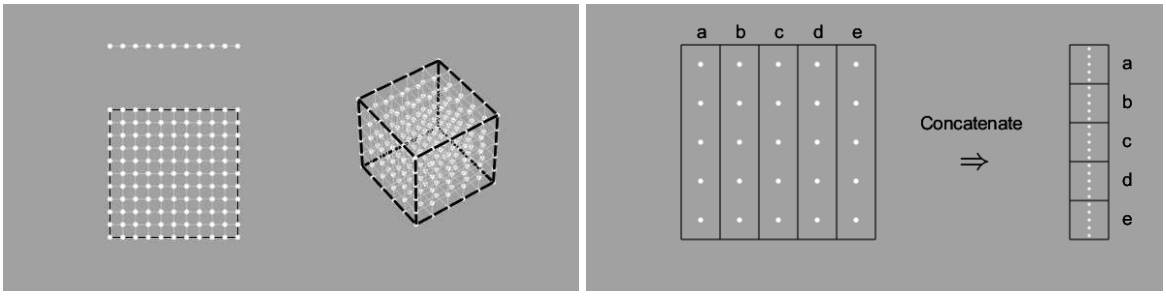

Figure 2. Left: spatial grids in 1D, 2D and 3D. Right: concatenation of a 2D grid into a vector.

Recursions

The main computational work across all the NESS code modules is the update of the state at an audio rate—usually chosen to be 44.1 kHz or 48 kHz, although all algorithms can produce sound output at any specified rate. At time step n, the state vector $\mathbf{u}^{n+1}$ must be computed using previously computed values of the state. In virtually all synthesis code modules, updates are "two-step": $\mathbf{u}^{n+1}$ may be computed using only $\mathbf{u}^n$ and $\mathbf{u}^{n-1}$. To avoid producing an excess of computed data, once $\mathbf{u}^{n+1}$ is computed, $\mathbf{u}^{n-1}$ may then be discarded (or overwritten). Here, the state vector $\mathbf{u}^n$, assembled by

Table 1. Update Types.

Type I	$\mathbf{u}^{n+1} = \mathbf{B}\mathbf{u}^n + \mathbf{C}\mathbf{u}^{n-1}$
Type II	$\mathbf{A}\mathbf{u}^{n+1} = \mathbf{B}\mathbf{u}^n + \mathbf{C}\mathbf{u}^{n-1}$
Type III	$\mathbf{A}\left(\mathbf{u}^n, \mathbf{u}^{n-1}\right)\mathbf{u}^{n+1} = \mathbf{B}\left(\mathbf{u}^n, \mathbf{u}^{n-1}\right)\mathbf{u}^n + \mathbf{C}\left(\mathbf{u}^n, \mathbf{u}^{n-1}\right)\mathbf{u}^{n-1}$
Type IV	$\mathbf{g}\left(\mathbf{u}^{n+1}, \mathbf{u}^n, \mathbf{u}^{n-1}\right) = \mathbf{0}$

concatenation, is assumed to be of size $N \times 1$, where the total state size is $2N$.

Updates take on different forms, depending on the physics of the problem at hand. In Table 1, generic update equations of four different type are shown (labeled I through IV). In this simplified representation, input and output operations are not included.

In the simplest case, the update step is of Type I, and requires only matrix multiplication. This corresponds to the case of explicit finite difference schemes for linear and time-invariant systems. The $N \times N$ matrices $\mathbf{B}$ and $\mathbf{C}$ may be precomputed at the setup stage, and the internal structure of these matrices follows directly from the physics of the system. They are generally very sparse; the sparsity follows from the use of FD approximations, according to which derivatives are approximated using neighbouring values on a grid. This is the ideal case for a parallel implementation. See Figure 3, illustrating typical sparsity patterns, in the case of a matrix $\mathbf{B}$ employing a basic approximation to the Laplacian.

It is sometimes the case that the update corresponds to a linear system solution, as in Type II, for a known constant $N \times N$ matrix $\mathbf{A}$. This corresponds to the case of implicit finite difference schemes for linear and time-invariant systems. $\mathbf{A}$ may be precomputed at the setup stage, but the solution to the Type II case is in general much more problematic than that of Type I. As the size of $\mathbf{A}$ is potentially large, it is inadvisable to precompute the inverse of $\mathbf{A}$—though $\mathbf{A}$ is usually sparse, $\mathbf{A}^{-1}$ will generally not be, and storage requirements can quickly become unmanageable for all but the smallest systems. Depending on the particular structure of $\mathbf{A}$, different approaches to linear solution are available. Some, such as Jacobi iteration (Strikwerda 1989), are easily

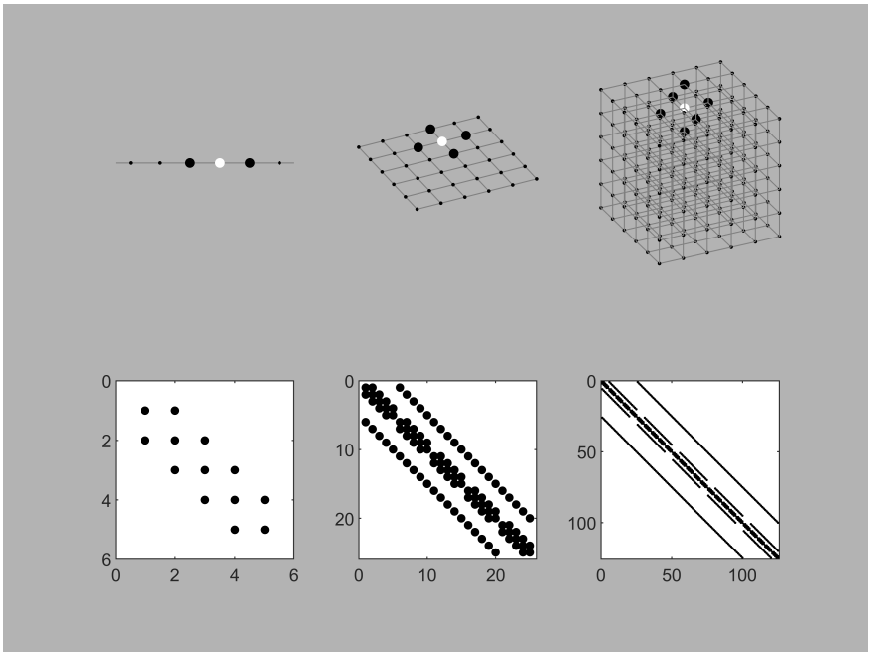

Figure 3. Top: spatial grids in 1D, 2D and 3D, showing the action of an approximation to the Laplacian at a given location (centered at the white point, and drawing on values at neighbouring black points). Bottom: matrix forms of the Laplacian operator. The matrices operate on a state vector arrived at through concatenation, as illustrated in Figure 2.

parallelisable, but require additional conditions on $\mathbf{A}$ (such as diagonal dominance), which are not always satisfied. In the worst case, standard general methods, perhaps targeted towards sparse systems (Saad 2003), may be necessary.

In some cases involving nonlinearities of geometric type (such as those occurring in the case of gong vibration, or string vibration at high amplitudes, updates of Type III are necessary. Now the matrices $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ are dependent on the previously computed state $\mathbf{u}^n$, and thus cannot be precomputed. This leads to extra computational cost during the execution of the runtime loop; usually, though, $\mathbf{A}$ remains sparse, and the linear system solution may be carried out as for systems of Type II.

The most general case is that of an update of the form of Type IV for a known nonlinear vector-valued function $\mathbf{g}$. This arises when nonlinearities of more complex form are present, as in the case of collisions (in, e.g., the string-fretboard interaction, or

for the mallet/membrane interaction). In this case, except for in highly pathological situations, iterative methods (such as, e.g., the Newton-Raphson method (Press et al. 1992)) are usually required, and opportunities for parallelisation are very limited.

In practice, for realistic and complex musical instrument designs, the update is often a mixture of the forms above. More precisely, it is often possible to partition the state $\mathbf{u}^n$ into subvectors, over which updates of different types are performed. Such a partition follows naturally from the physics of the problem under consideration. For example, in the case of the 3D model of the snare drum, updating of the acoustic field is of Type I; that of the membrane of Type II; and that of the snares themselves of Type IV, due to the highly nonlinear collision mechanism.

Inputs and Outputs

Inputs and outputs to the synthesis algorithms, obviously essential for any meaningful musical work, have been not been described above. In general, in physical modeling synthesis, the excitation mechanism driving a given instrument design will be nonlinearly coupled to the instrument dynamics—this is the case, e.g., for all of the standard playing modes (bowing, striking, plucking, blowing). The case of output is simpler—generally, values may be drawn from the appropriate location in the state, as it evolves, in the manner of a pickup or microphone. In the NESS code, simultaneous multiple inputs and outputs are possible; as a result, these synthesis methods are geared towards multichannel composition, and became a large influence on the mode of work of the associated musicians, and the ultimate forms that their compositions took, as described later.

Under greatly simplified conditions, the entire input/output system may be viewed in a form resembling a state space form (Kailath 1980). For example, considering the case of the simple linear form given as Type I above, an extension to incorporate I/O may be

written as

$$\mathbf{u}^{n+1} = \mathbf{B}\mathbf{u}^n + \mathbf{C}\mathbf{u}^{n-1} + \mathbf{J}_i\mathbf{w}^n \qquad \mathbf{y}^n = \mathbf{J}_o\mathbf{u}^n \tag{1}$$

Here, subscripts i and o refer to input and output, respectively. $\mathbf{w}^n$ is an $N_i \times 1$ vector input vector consisting of a set of N_i independent input signals at time step n; $\mathbf{J}_i$ is an $N \times N_i$ matrix, each column of which specifies the grid point or set of grid points to which the input will be added. Similarly, $\mathbf{y}^n$ is an $N_o \times 1$ vector of output signals, and $\mathbf{J}_o$ is an $N_o \times N$ matrix, each row of which selects a particular set of grid points in the physical model from which output will be drawn (with scaling included).

Generally, the computational cost of input and output operations is very small compared to the raw calculation which must be carried out over the entirety of the computational grid. Extensions to the time-varying case are possible as well—input locations can be variable (as in, e.g., the case of bowing or finger-stopping in the guitar).

Synthesis Environments

A variety of distinct environments developed over the course of the NESS project, based around the investigations of different instrument families described in the companion article (Bilbao et al. 2019). Acceleration strategies depend highly on the particular system, and are described subsequently, but all environments have ultimately been made available to musicians through a web interface.

The *zero code*, developed in 2013, was the first attempt at a large-scale modular synthesis network. The basic units are plates, of dimensions and material properties as specified by the user. Connections between objects are point-like, and behave as nonlinear mass/spring/damper systems, where the nonlinearity is of cubic type. This restriction to cubic nonlinearities eases computational requirements considerably, as it is possible to develop stable algorithms of Type III, where the linear system to be solved is

diagonal (thus requiring simple scalar divisions in the run-time loop). Input can be specified through a list of events, each of which corresponds, roughly, to a strike at a given location on a given object at a given time, and of a given force. Each such event is translated, ultimately, into a short force signal fed into the network. Other input types are a bowing gesture, described through a breakpoint function for bow force, velocity and position, and also audio input, if the network is to be used as an effect (emulating, e.g., plate reverberation). Multichannel output may be obtained by reading velocities at a set of locations throughout the network, with the option for the normalisation of individual channels in the case that output amplitudes are widely disparate. These modular environments are described in Bilbao (2009).

The *brass code*, developed in 2013/14 allows for the flexible construction of brass and brass-like instruments, of arbitrary bore profile and valve configurations; control is through mouth pressure and lip stiffness, as well as a set of control signals representing valve positions. Because of its very fast execution time (several times faster than real time) it has been a very popular choice among composers, and extensively explored. Part of the reason for the fast execution time is the relatively small state size (for reasonable sized instruments, and at an audio rate, N is on the order of 100 to 400), but another is that a Type I explicit update is used for the bulk of the calculation. Control is of a different type from that used in the zero code, in that continuous data streams are necessary, rather than a discrete list of events. Breakpoint functions (piecewise linear here, but easily generalised) are used in order to specify such control streams. The complete brass synthesis environment is described in Harrison et al. (2015).

The single- and double-membrane drum code and the gong code were developed between 2013 and 2015. They incorporate many realistic features, including membrane/plate nonlinearity, mallet interactions, snares, and are ultimately embedded in 3D, allowing for spatialised sound output. This was the closest approach to fully

virtual musical instruments, in that it is possible to embed multiple instruments within a potentially large space. The calculation is very large, however, as it relies on updates of Types I, II, III and IV simultaneously; and for very large state sizes, it is relatively slow (taking minutes to calculate several seconds of output), which makes this approach harder to explore musically. As in the case of the zero code, input is specified through strike forces and locations over multiple objects. In contrast, however, full multichannel output may be drawn from the 3D acoustic field simulation. Such virtual percussion instruments are described in Torin et al. (2014); Bilbao and Webb (2013).

The *guitar code* was developed in 2014/15, and was perhaps the most elaborate instrument attempted in terms of control. The user specifies string parameters, such as tension, length, stiffness and T_{60} times for N strings. In addition, the user supplies a common backboard profile, against which strings will collide, as well as the positions of frets (as many as desired). The user must further specify the properties of fingers which will interact with the instrument, and in particular their masses and stiffnesses. From the control point of view, breakpoint functions are used to specify the positions of the fingers along the length of the string, as well as the forces exerted by the player on each string over time. In addition, the user also specifies plucking or striking events along the string. The guitar environment is described in Bilbao and Torin (2015).

Net1 was the last modular code developed, in 2016. It differs from the zero code in that plates are not included, and that bar and string behaviour are both unified into a single model, and thus the model can behave as a string or percussion instrument. In addition, nonlinear connections are of a very different type, allowing for intermittent loss of contact of a type reminiscent of a rattle, and leading to a much larger variety of timbres. I/O is much the same as in the case of the zero code. Due to the form of the nonlinearity, iterative methods of Type IV are required, leading to much slower runtimes than in the case of the zero code.

Acceleration in Parallel Hardware

Acceleration in parallel hardware proved to be a significant challenge, and various different methods were combined in order to achieve good performance. In all cases, prototyping was carried out in Matlab. The next step was to port the code to generic C++, as a starting point for further experimentation with parallelisation techniques. All of the models were ported into a common code framework so that useful generic functionality could be shared among them.

The appropriate strategies for speeding up a given piece of code are dependent not only on the code itself, but also on the target hardware: large scale supercomputers and clusters demand a different approach from desktop computers and smaller servers. The optimised NESS code was targeted primarily at multicore Intel Xeon servers with multiple NVIDIA Tesla GPU cards, since it was used to run the NESS service for the composers, but it was also designed with portability in mind and can run on most Windows, Mac and Linux machines. See Figure 4.

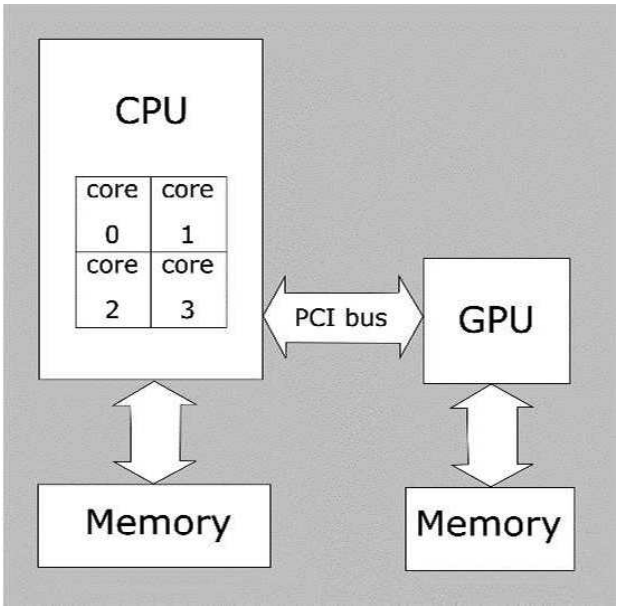

Figure 4. GPU server architecture.

Multicore

One form of parallelism that is almost ubiquitous in computers of all sizes today, and that is conceptually quite simple, is *multicore*. Modern CPUs contain multiple cores, each of which can be running an independent thread of program execution at the same time. These cores share a single memory space, so the program threads can operate on the same data, though care must be taken to ensure synchronisation and that the threads do not interfere with each other.

This type of parallelism raises the question of how best to divide up the program's work between the various threads. Two different approaches were tested for the NESS code modules:

Assigning each distinct component of the simulation to its own thread. For example, when running the net1 code, each string, bar and connection is processed on a separate thread. In the case where the components outnumber the available CPU cores, each thread may have to process multiple components in sequence.

Decomposing the domain of the simulation across multiple threads. For example, in the case of a membrane with 1000 elements, the first 500 elements could be updated on one thread while the last 500 are simultaneously updated on another.

The first approach was adopted as it had two major advantages: firstly, it is very simple to implement and requires minimal communication among the threads, since each thread is working on a mostly self-contained entity; and secondly, since the component update itself is unchanged from the serial case, it works with any algorithm. In contrast, method 2 requires changes within the update code, which adds complexity and is difficult to do efficiently for implicit or non-linear (Type II, III and IV) updates. The main disadvantage of the chosen method is that it is useless for simulations that cannot be broken down into multiple components, but these are generally either

lightweight enough to run fast even on a single core (brass instruments, for example), or are large room simulations that are better run exclusively on GPUs (as discussed later).

The standard for high performance multithreaded code is OpenMP (Dagum and Menon 1998), a directives-based framework that allows work to be easily shared across multiple threads in a shared memory environment. Unfortunately this was found to be unsuitable for the NESS code; OpenMP imposes a certain overhead at each simulation timestep, and this overhead becomes problematic due to the large number of timesteps required when working at audio rate. We created a simple threading framework, which on Windows uses Win32 threads and on Linux and macOS uses the low-level thread libraries (Butenhof 1997) provided by the operating system. This provided full control over the behaviour of the threads, allowing them to be kept in a state where they could respond instantaneously when needed.

In the best-case scenario, the performance of a multicore code scales with the number of CPU cores available. In practice this is rarely achieved due to synchronisation overheads and imbalances between the work assigned to each thread, but due to the minimal inter-thread communication required, the NESS multicore code does approach the best-case scaling in certain cases.

Vectorisation

Modern CPUs also provide opportunities for finer grained parallelism within each CPU core. While a traditional machine instruction (the most basic building block of all computer programs) performs a single calculation, a vector instruction can perform multiple calculations simultaneously. For example, the SSE2 (Raman et al. 2000) vector instructions, present on all 64-bit Intel and AMD CPUs, can perform two double precision or four single precision floating point calculations at once, and the more advanced AVX (Firasta et al. 2008) instructions extend this to four double precision or

eight single precision operations. See Figure 5.

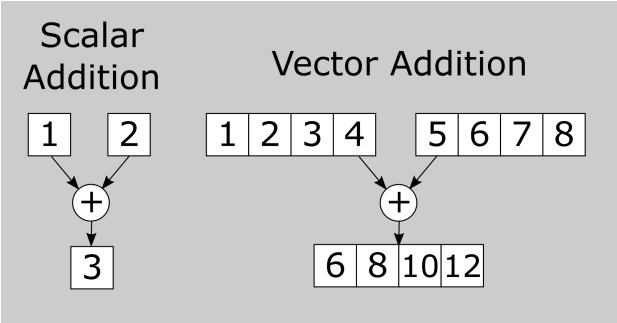

Figure 5. Scalar and vector addition.

This is extremely useful for looping through an array of numbers and applying the same operation to all of them, a pattern which occurs frequently in the NESS simulations (especially Type I updates, but also when computing the matrices in Type III). Modern compilers can sometimes vectorise such code automatically. Vector instructions can also be used to speed up more complex operations, but this takes more manual effort, typically requiring the use of compiler intrinsics (special functions that translate directly into specific machine instructions).

A 4-element vector operation will be at most four times faster than the equivalent scalar operations, but often this upper performance bound cannot be reached in practice due to memory bandwidth or other limitations. For NESS, AVX instructions were used extensively in the main state update of the brass code, giving a 2.5x speed up over the serial C++ code and allowing many instruments to be simulated faster than real time. The triangular solve operation used in the preconditioner of the linear system solve used in the type III updates was vectorised using SSE, again giving an approximate speed up of 2.5x over the original C++ version. (In this case SSE was actually faster than AVX due to technical limitations of the AVX instruction set).

GPU acceleration

Graphics processing units were originally designed for rendering real time graphics for games, but in recent years there has been much interest in exploiting them for other purposes, and GPU vendors provide tools like NVIDIA's CUDA (Nickolls et al. 2008) and the open standard OpenCL to facilitate this. GPUs are essentially massively parallel processors containing hundreds or even thousands of cores, though each individual core is very primitive and slow compared to a traditional CPU core.

Because the power of GPUs comes from their extreme parallelism, they are best suited to algorithms with large numbers of operations that can be performed simultaneously, and much weaker for running sequential code. In addition, they typically have their own memory space and are separated from the rest of the computer system by a relatively slow PCI Express bus; care is required to design code modules in such a way that not too much data needs to be transferred between CPU and GPU.

GPUs are therefore well suited to running the NESS algorithms that perform a relatively simple linear operation (Type I update) across a large domain—either a three-dimensional room or a large two-dimensional plate or membrane. One-dimensional and smaller two-dimensional entities are generally too small to benefit from GPU acceleration, and the more complex updates (Types III and IV) require too many sequential computations on a single thread Bilbao et al. (2013); Perry et al. (2015).

A flexible approach was taken in the NESS framework, allowing all simulation components to take advantage of the GPUs if beneficial, or to remain on the CPU if not. Additionally, GPU versions of most of the input, output and modular interconnection algorithms were provided, allowing these operations to be performed directly on the GPU when required, instead of avoid having to run them on the CPU and then copy the resulting data across the slow PCI bus.

Although the effort involved in porting code to CUDA is relatively high, the performance gains can be significant. For example, updating the acoustic field surrounding the bass drum is roughly six times faster on the GPU than it is on the CPU (Perry et al. 2015).

Algorithm Tuning and Optimisation

In addition to the parallelisation methods already described, some NESS code was sped up significantly using other optimisations. One example was the Newton-Raphson solver used in the guitar code to simulate collisions between strings and other items. In its original form this involves solving a relatively large, sparse linear system, which can be a time-consuming operation. However, in most cases this can be converted to a much smaller dense linear system by rank reduction. This small dense system (containing typically less than 20 unknowns) can then be solved very quickly by direct factorisation.

In addition, most of the NESS code makes extensive use of sparse matrix and vector operations such as multiplications and additions. In the general case these operations can be expensive as they need to cater to every possible type of matrix, but they can often be sped up dramatically by using an algorithm and matrix storage format more suited to the specific matrices involved. For example, the generation of matrices in the Type III algorithm originally used generic operations and compressed sparse row matrix storage. This was replaced by a custom banded matrix format and specialised operations, often combining multiple operations into a single step so that intermediate results did not have to be written to memory and later read back again. The overall effect was to speed up this part of the algorithm by a factor of 10.

A dramatic speed up (around 300 times) was obtained for the bowed string code by replacing a simple linear search of a table with a more sophisticated algorithm that did a binary search on a sorted version of the table and then mapped the results back onto the

original table.

Acceleration Summary

As described above, the various optimisation methods all have their strengths and weaknesses and are better suited to some algorithms than others. Most of the NESS code was optimised using one or two of these methods; however, the bass drum code and gong code were more challenging and required a hybrid approach making use of all four methods. The simulation of the surrounding acoustic field is GPU-accelerated; the plates and membranes each run on their own CPU core; some elements of the plate and membrane code are vectorised with SSE2; and extensive algorithmic changes were implemented to speed up sparse matrix generation. The software framework developed for NESS makes it relatively easy to combine all these disparate methods, and the end result is a 60-80 times speed up over the original Matlab version for typical instruments. For example, in the case of the bass drum, for typical choices of parameters, simply porting the code from Matlab to C++ sped it up by roughly 5.9 times. This was then further improved with speed-ups of 2.4 times for multicore, 2.3 times for CUDA, 1.8 times for algorithmic changes and 1.2 times for vectorisation, leading to an over-all speedup of 70.3 times.

Table 2 shows the methods used to optimise the code for each instrument, and the rough overall speed up of the final optimised C++ code compared with the original Matlab.

User Control

Since most of the synthesis algorithms developed under the NESS project were slower than real time, there were no attempts at a sophisticated graphical interface. Rather, work on the control side was at the lower level of determining usable parameter

Table 2. Acceleration relative to a single-core Matlab implementation.

Code	Multicore	Vectorisation	GPU	Algorithm Tuning	Speedup
Brass		✓			15×
Bowed String	✓			✓	300×
Guitar	✓			✓	21×
Bass Drum	✓	✓	✓	✓	60-80×
Zero Code	✓		✓		25×
Net1	✓			✓	20×

sets which could be approached by a non-expert. From an early stage, it was decided to adopt the score/instrument breakdown of user-supplied input data, which has been standard across various iterations of MUSIC N synthesis environments.

Instrument and Score Files

The NESS synthesis system accepts an instrument file and a score file as input, and also possibly audio input, if the synthesis algorithm is to be used as an effect. In Figure 6 below, a rudimentary instrument/score file pair is shown, in the case of the *zero code* modular plate synthesis network. The instrument file specifies the sample rate, as well as the parameter sets defining a set of thin metal plates, as well as connection elements, linking two plates together (or a plate to itself). For a tensioned rectangular plate, for example, the parameters are the material, thickness, tension, dimensions, as well as specifications of 60 dB decay time at DC and 1 kHz, and an integer value specifying the type of boundary condition (in this case, clamped, simply supported or free). For a connection, the parameters are the coordinates with respect to two different plates, as well as a linear and nonlinear stiffness parameters, and an additional 60 dB decay time for the connection itself (realised as a damper). Outputs are also defined, with reference to coordinates on the plates. The score specifies the total duration of the simulation, and also strike events, as well as a bowing gesture. Audio input is also an option.

Instrument File

```
# zcversion 0
# set 44100Hz sampling rate
samplerate 44100

# Define two steel plates.  Arguments:
# <name> <material> <thickness> <tension> <X> <Y> <T60@0Hz> <T60@1kHz> <boundarytype>
plate plat1 steel 0.001 0.0 0.4 0.7 11.0 6.0 4
plate plat2 gold 0.001 0.0 0.3 0.6 10.0 6.0 4

# Define a connection from one point on the plate to another on this or a different plate.  Arguments:
# <component> <component> <X1> <Y1> <X2> <Y2> <linearstiffness> <nonlinearstiffness> <T60>
connection plat1 plat2 0.7 0.4 0.3 0.7 10000.0 10000000.0 1000000.0
connection plat1 plat1 0.2 0.3 0.5 0.5 10000.0 10000000.0 1000000.0

# Define two outputs from the two plates.  Arguments:
# <component> <X> <Y> <pan>
output plat1 0.6 0.6 -1.0
output plat2 0.3 0.7 1.0
```

Score File

```
highpass off # no high-pass filter
duration 1.0 # one second simulation

# Define two strikes.  Arguments:
# <starttime> <component> <X> <Y> <duration> <amplitude>
strike 0.0 plat1 0.4 0.7 0.002 400000.0
strike 0.0 plat2 0.3 0.6 0.001 100000.0

# Define a bowing action.  Arguments:
# <starttime> <component> <X> <Y> <duration> <forceamp> <velocityamp> <friction> <ramptime>
bow 0.3 plat1 0.3 0.9 4.0 2.3 2.8 1.1 0.02

# Define an audio input.  Arguments:
# <file> <starttime> <component> <X> <Y> <gain>
audio drumming.wav 0.1 plat1 0.2 0.4 1.0
```

Figure 6. Instrument and score files for the zero code *modular plate environment.*

Web-based Interface

A simple web-based user interface was created to allow the composers to access the NESS service through a standard web browser. It consists of a form allowing the user to upload an instrument file, a score file and an optional audio input file and then launch the NESS code on the GPU server. A demo mode is supported for most code modules; this runs a lower sample rate simulation to give the user an impression of how their submission will sound, but generally completes much faster than a full quality run. The synthesis environment to run is selected via a comment in the instrument file. After the job completes, the user is able to download the generated audio outputs (individual

mono channels as well as a stereo mix file) as .wav files.

Musical Experimentation

An integral part of the NESS project was collaborative work with electroacoustic composers, which commenced approximately 18 months into the project. All work was carried out in a 16 channel space, during intensive workshops with the NESS team. In most cases, the musicians worked directly through the web-based interface to the GPU server. As described in the previous sections, the interface itself is quite raw—there were thus many opportunities for musicians to develop their own approaches to instrument design, writing sufficiently interesting and complex scores, and handling multichannel output. Approximately 10 complete works resulted, whose numbers of channels ranged from 8 to 32. In this section, a variety of composers provide their own insights into the experience of working with the NESS system. They are arranged approximately chronologically, and cover the period from 2013 to 2019.

Gordon Delap: *Ashes to Ashes* (2013)

In *Ashes to Ashes*, instruments were assigned certain physical properties of uranium. Perhaps understandably, this choice was received with some degree of skepticism, although the assignment of parameters relating to radioactive elements was intended principally as a conceptual starting-point. More fundamentally, the work was conceived of as a reflection on life and death. It was also envisaged that the instruments would display behaviours of real-world objects while exhibiting surreal qualities, although this aspect usually had more to do with non-traditional topologies than material specifications.

A vital concern was the investigation of compositional applications of the technology. It seemed appropriate, then, to set constraints:

Modification of audio output via post-processing techniques was not permitted

Sound materials generated via external means could not be used, except where such sounds were processed by being fed through one of the NESS project's models

These constraints presented challenges. Firstly, the generation of audio took an extremely long time in the early stages of the NESS project. Overnight processing for instruments with large dimensions was not unusual, while fine-tuning of instruments was often difficult. Secondly, the composition of electroacoustic music typically allows for the deployment of a vast and powerful armory of digital signal processing techniques. Such techniques have been developed and refined over decades of experimentation and investigation. However, these methods could only have been accessed at the expense of overshadowing unmanipulated output from the models.

From the outset, there was an awareness of the types of raw sound output that might be expected. For instance, idiophones and membranophones were readily obtainable. Blown brass-type instrument were also under development at the time. *Ashes to Ashes* was especially influenced by the ritual music of Tibet. It seemed that many aspects of this soundworld lay within the capabilities of the physical models under development, although the composition is not reducible to this concern.

Learning how to play the brass instrument models was difficult. Embouchure parameters called for precision. Much to the credit of the model's accuracy, when this principle was violated, the outcomes sounded every bit as deficient as those produced in real life by unskilled brass players.

Trevor Wishart: *Dithyramb-Kepler 63c* (2014)

The work *The Secret Resonance of Things* attempts to derive music from scientific data of various kinds (the spectra of supernova, the onset of turbulent flow, and so on).

Dithyramb-Kepler 63c, the 3rd movement, uses the NESS physical modelling software to conjure up a musical celebration on an earth-like planet using alien instruments and an “unknown" musical style. I used semi-random sets of values as input to the NESS models of both brass and plates, searching, initially haphazardly, for sonically interesting outputs. I wanted a rich sonic palette of brass instrument articulations and various different percussives (wood-like, clunk-metal-like, marimba-like etc) to create my alien ensemble. Having found sonorities and articulations that I liked, I explored small adjustments to the parameter values to uncover sets of closely related sounds (which might conceivably have come from the same instrument, and player). Using the Composers’ Desktop Project tools to adjust pitch, layer and otherwise expand the sounds, I was able to compose lines of complex and plausible instrumental music. The streams from these various “players" were spatialised in 8-channel sound-surround to create the illusion of an encircling ensemble. Such spatialisation particularly animates the rhythmic percussive sequences in the piece. As often in my compositional history, I developed the software interface (to allow brass-profiles to be directly drawn, rather than entered numerically) only after I had completed the piece!

Gadi Sassoon: *Multiverse* (2016–2018)

My investigation of the NESS algorithms (Guitar, Net1 and Brass) revolves around their application for music production and their integration with analogue synthesis and acoustic instruments. Through the NESS physical models I was able to explore a space that weaves in and out of realism and abstraction, seeking to develop a textural bridge between synthesisers and orchestral instruments in pursuit of a cinematic sound that blurs the edges between synthetic and organic.

To that end I have explored the parameter space’s fringe areas of operation: with some experimentation the models allow for the rendering of instruments designed with impossible physics (e.g. gigantic brass instruments blown with very hot air, needle-thin

fingers with no loss gently rattling against a string's harmonic on a fret, or lattices of bound masses infinitely vibrating). The non-linearities of the systems provide exceptional detail and help maintain quasi-mechanical semblance even in otherworldly simulations. The ability to sample some models in more than one location makes for compelling multichannel output, allowing gestures ranging from short bursts to complex immersive soundscapes.

Interacting with NESS has pushed me to develop new sonic vocabulary: the unconstrained nature of a research-oriented system presented unique challenges, requiring creative solutions which in turn prompted previously unimagined musical ideas. The result of our collaboration is my Multiverse album, comprising pieces (such as Collision Suite and Black Hole Fanfare) entirely made from code output, and others (like Life On A Tidally-Locked Planet and Order And Chaos) which are a combination of NESS output, modular synthesisers and live strings.All the pieces have both stereo and 8-channel mixes: they have been performed at IRCAM, CCRMA, NIME 2018 and Music Tech Festival 2018; a Multiverse Immersive installation will be unveiled at Sonar 2019 using interactive sculptures, head tracking headphones and the Traverse platform.

Samson Young: *Possible Music #1* (2018)

I visited Edinburgh to work with NESS twice for this research, briefly in the summer of 2017 and then again for a 2.5 weeks in November of the same year. My collaboration with the NESS project resulted in a new commission at the Guggenheim Museum in New York city, titled *Possible Music #1* (feat. NESS and Shane Aspegren), which was on view at the museum from May to October of 2018, and has since become a part of Guggenheim's collection.

Using the NESS software, I created a series of musical compositions that are sometimes enriched with human vocals. These arrangements are activated according to

a precise schedule that references the use of bugle calls in the military to signal orders and events. Using old and new sounds to collapse temporal and spatial divides, my aim was to interrogate our search for truth, and music's unsung role in shaping the birth and progress of civilization.

For this work, I first created a few dozen brass instruments with the NESS software. I designed a Pure Data patch to help me generate score files. In this PD patch I am able to draw various musical parameters in a graphical interface, which allowed me to work visually and intuitively with the code. This process resulted in a large database of short mono samples, each ranging from 15 to 30 seconds long. In a sequencer, these short samples are combined into longer compositions that are each 1 to 1.5 minutes in length. Each composition is consisted of 6 to 10 channels of sound. Finally these compositions are spatialized in a 10.1 channels speaker array setup, through a custom-built MAX/MSP patch.

Tom Mudd: ***Brass Cultures*** **and** ***Three Algorithms for Hans Reichel*** **(2018–2019)**

I have been using the NESS brass model since 2015, and the guitar model since late 2017. The material has been used for performances at Café Oto in London in 2017 and 2019, and at Saint Cecilia's Hall in Edinburgh in 2018.

The simple text-based score and instrument format used in the NESS interface makes it relatively easy to create algorithms that generate both score and instrument files. Multiple scores and instruments can be created by the same algorithm, allowing different instruments to be played in a coordinated fashion.

Each piece for the *Brass Cultures* project was created with a Python script that generates scores and instruments. Score parameters were generated according to a

variety of rules: creating lip frequency sequences from pre-defined sets, choosing breath pressures in certain ranges, with defined envelopes, creating different rules for difference sections of each piece, and different rules for the timings of each section, and so on.

As part of the work *Three Algorithms for Hans Reichel* with the guitar model, a converter was implemented in JavaScript that takes plain text guitar tab as input and creates NESS-formatted score files as shown in Fig. 7. For physical modeling, tablature is a more appropriate score mechanism than other input options such as MIDI, as it denotes the specific finger placements explicitly rather than allowing the performer to decide how to assign individual notes to the guitar strings. Guitar tab displays a row for each string and uses numbers to denote which fret to place a finger on (the finger number being unspecified). Duration is indicated using hyphens; for example, the following 4/4 measure of tab for one string shows the first, fourth, and fifth notes being held for twice as long as the second and third notes: | 5—5-4-5—0— |.

While the tablature converter can be used to render slightly mechanical versions of existing pieces, it can also be used as a rapid way to create simple structures and patterns for use with unconventional string instruments that may or may not yield obviously pitched results.

Concluding Remarks and Perspectives

The NESS project was intended as an opportunity to step back from the constraints of real time performance, in order to fully come to grips with the entire physical modeling “production chain," from basic algorithm design to implementation and musical use. Indeed, given the computational demands of this type of physical modeling synthesis, and the unknown territory of algorithm parallelisation, it seemed appropriate to use this “offline" research model—which was inspired, perhaps subliminally, by the classic working methods at Bell Labs in the 1950s and 1960s.

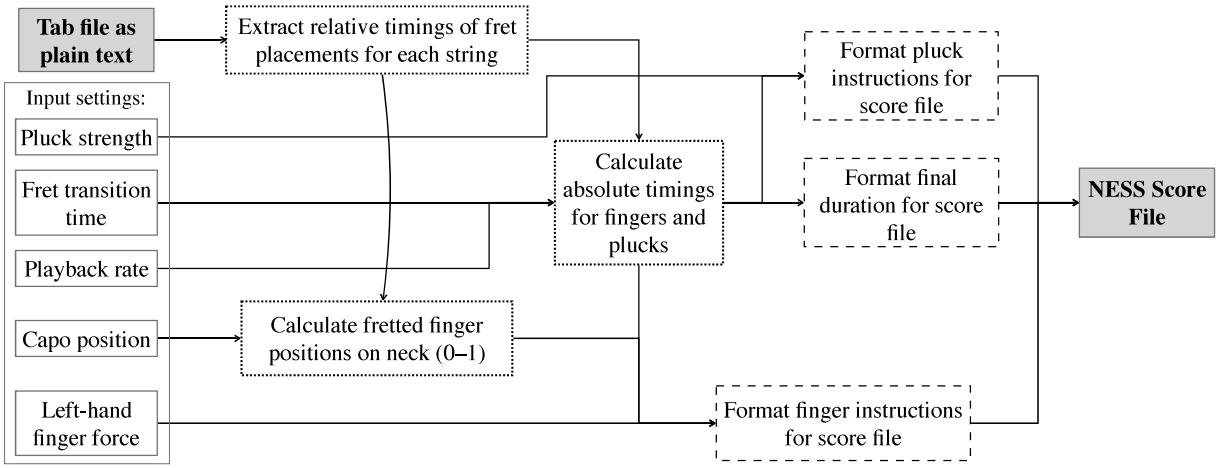

Figure 7. Diagrammatic structure of a tablature converter for the guitar code.

But the real time challenge remains—and real time performance is what the majority of musicians ultimately want. A small subset of the algorithms presented here can operate faster than real time, and in particular the brass synthesis environment. In other cases, it is fairly clear that real time performance will not be possible for the foreseeable future—examples are 2D instruments such as gongs and drums, where the strong nonlinearity forces a choice of algorithm design which is not easily parallelisable, and full 3D room acoustics simulation, which, though algorithmically simpler, is a problem of a vast scale. Modular designs are inherently scalable, and are a good choice for real-time operation. The first modular physical modeling synthesis system, *Derailer*, was released as a plugin by Physical Audio in 2018 (Webb 2019), and more advanced models are in various stages of porting from the NESS code modules.

Beyond the question of raw computation time, instrument design and control remain difficult problems, which is not unexpected, given the history of musical instrument design and performance—building an instrument, and learning to play it has always required an inordinate amount of patience and labour. A relatively low-level mode of design and control (instrument and score files) was the point of departure in

joint work with musicians—intended to allow freedom of exploration of the resulting soundspace, and, more importantly, not to over-constrain it. Any instrument should probably be hard to build and learn to play if it is ever to produce musically-interesting sound—but hard within reason.

At this point, though, where creative concerns become central, the engineer's role must recede into the background—and this was exactly our approach. Each musician we have worked with has approached the instrument design and control issue in a distinct way, and in some cases has built higher-level working tools in order to intuitively generate instruments and scores. Ultimately, it is these individual approaches to design and control that give each completed piece of work its distinctive sound—and are perhaps a further step towards the mature use of physical modeling sound synthesis.

Acknowledgment

This work was supported by the European Research Council, under grants 2011-StG-279068-NESS and 2016-PoC-737574-WRAM.

References

Bilbao, S. 2009. "A Modular Percussion Synthesis Environment." In *Proceedings of the 12th International Digital Audio Effects Conference*. Como, Italy, pp. 321–328.

Bilbao, S., et al. 2019. "The NESS Project: Physical Modeling, Algorithms and Sound Synthesis."

Bilbao, S., et al. 2013. "Large Scale Physical Modeling Synthesis." In *Proceedings of the Stockholm Musical Acoustics Conference*. Stockholm, Sweden, pp. 593–600.

Bilbao, S., and A. Torin. 2015. "Numerical Modeling and Sound Synthesis for Articulated String/Fretboard Interactions." *Journal of the Audio Engineering Society* 63(5):336–347.

Bilbao, S., et al. 2014. "Modular Physical Modeling Synthesis Environments on GPU." In *Proceedings of the International Computer Music Conference*. Athens, Greece, pp. 1396–1403.

Bilbao, S., and C. J. Webb. 2013. "Physical modeling of timpani drums in 3D on GPGPUs." *Journal of the Audio Engineering Society* 61(10):737–748.

Butenhof, D. 1997. *Programming with POSIX threads*. Boston, Massachusetts: Addison-Wesley Longman Publishing Company.

Dagum, L., and R. Menon. 1998. "OpenMP: an industry standard API for shared-memory programming." *IEEE Computational Science and Engineering* 5(1):46–55.

Firasta, N., et al. 2008. "Intel® AVX : New Frontiers in Performance Improvements and Energy Efficiency." *Intel White Paper* Available online https://software.intel.com/en-us/articles/intel-avx-new-frontiers-in-performance-improvements-and-energy-efficiency.

Harrison, R. L., et al. 2015. "An Environment for Physical Modeling of Articulated Brass Instruments." *Computer Music Journal* 29(4):80–95.

Hsu, B., and M. Sosnick-Pérez. 2013. "Realtime GPU Audio." *ACM Queue* 11(4):40:40–40:55.

Kailath, T. 1980. *Linear Systems*. Englewood Cliffs, New Jersey: Prentice Hall.

Lindemann, E., et al. 1991. "The Architecture of the IRCAM Musical Workstation." *Computer Music Journal* 15(3):41–49.

Mehra, R., et al. 2012. "An efficient GPU-based time domain solver for the acoustic wave equation." *Applied Acoustics* 73(2):83–94.

Motuk, E., et al. 2007. "Design Methodology for Real-Time FPGA-Based Sound Synthesis." *IEEE Transactions on Signal Processing* 55(12):5833–5845.

Nickolls, J., et al. 2008. "Scalable parallel programming with CUDA." *ACM Queue* 6(2):40–53.

Perry, J., S. Bilbao, and A. Torin. 2015. "Hierarchical Parallelism in a Physical Modelling Synthesis Code." In *Proceedings of the International Conference on Parallel Computing*. Edinburgh, UK.

Pfeifle, F., and R. Bader. 2015. "Real-Time Finite-Difference Method Physical Modeling of Musical Instruments Using Field-Programmable Gate Array Hardware." *Journal of the Audio Engineering Society* 63(12):1001–1016.

Press, W., et al. 1992. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge, UK: Cambridge University Press.

Puckette, M. 1991. "Combining Event and Signal Processing in the MAX Graphical Programming Environment." *Computer Music Journal* 15(3):68–77.

Raman, S., V. Pentkovski, and J. Keshava. 2000. "Implementing Streaming SIMD Extensions on the Pentium III Processor." *IEEE Micro* 20(4):47–57.

Saad, Y. 2003. *Iterative Methods for Sparse Linear Systems*. Philadelphia: SIAM.

Samson, P. 1980. "A General-purpose Digital Synthesizer." *Journal of the Audio Engineering Society* 28(3):106–113.

Savioja, L., V. Valimaki, and J. O. S. III. 2010. "Real-time additive synthesis with one million sinusoids using a GPU." In *Proceedings of the 128th AES Convention, paper 7962*. London, UK.

Savioja, L., V. Valimaki, and J. Smith. 2011. "Audio Signal Processing Using Graphics Processing Units." *Journal of the Audio Engineering Society* 59(1/2):3–19.

Southern, A., et al. 2010. "Finite Difference Room Acoustic Modelling on a General Purpose Graphics Processing Unit." In *Audio Engineering Society Convention, paper 8028*.

Strikwerda, J. 1989. *Finite Difference Schemes and Partial Differential Equations*. Pacific Grove, California: Wadsworth and Brooks/Cole Advanced Books and Software.

Torin, A., B. Hamilton, and S. Bilbao. 2014. "An energy conserving finite difference scheme for the simulation of collisions in snare drums." In *Proceedings of the 17th International Conference on Digital Audio Effects*. Erlangen, Germany, pp. 145–152.

Webb, C. 2019. "Physical Audio: www.physicalaudio.co.uk."

Zhang, Q., L. Ye, and Z. Pan. 2005. "Physically-Based Sound Synthesis on GPUs." In *Lecture Notes in Computer Science*, volume 3711/2005. Berlin/Heidelberg, Germany: Springer, pp. 328–333.