ON THE LIMITS OF REAL-TIME PHYSICAL MODELLING SYNTHESIS WITH A MODULAR ENVIRONMENT

Craig J. Webb
Acoustics and Audio Group
University of Edinburgh
Edinburgh, UK
craigwebb@physicalaudio.co.uk

*Stefan Bilbao**
Acoustics and Audio Group
University of Edinburgh
Edinburgh, UK
sbilbao@ed.ac.uk

ABSTRACT

One goal of physical modelling synthesis is the creation of new virtual instruments. Modular approaches, whereby a set of basic primitive elements can be connected to form a more complex instrument have a long history in audio synthesis. This paper examines such modular methods using finite difference schemes, within the constraints of real-time audio systems. Focusing on consumer hardware and the application of parallel programming techniques for CPU processors, useable combinations of 1D and 2D objects are demonstrated. These can form the basis for a modular synthesis environment that is implemented in a standard plug-in architecture such as an Audio Unit, and controllable via a MIDI keyboard. Optimisation techniques such as vectorization and multi-threading are examined in order to maximise the performance of these computationally demanding systems.

1. INTRODUCTION

A principle objective of physical modelling synthesis is to emulate existing acoustic or analog electronic instruments. However, another wide-ranging goal is to create entirely new instruments that do not necessarily have existing real-world counterparts—yet which generate sounds of a natural character. If the goal is the latter, then one often-used approach is to provide the user (the instrument designer) with a collection of primitive or canonical objects, as well as a means of connecting them, so as to allow for the generation of relatively complex sound-producing objects. Such a modular approach of course has a long history in audio synthesis, and deeper roots in the world of analog electronic synthesizers. In terms of physical modelling synthesis, modularity underlies various synthesis methodologies including mass-spring networks [1, 2] modal synthesis [3], as well as scattering-based structures [4].

A different approach, also allowing for modular design, involves the use of direct spatiotemporal integrators such as finite difference time domain methods. Such methods have various benefits, such as minimal pre-computation and memory usage, and also provable stability conditions [5]—which are extremely important under the very general design conditions that a musician will almost certainly demand. Such methods can be used to create large-scale models of instruments that can be coupled inside a 3D virtual space [6]. However, the simulation of such complex systems is computationally very intensive, to the extent that even with high performance computing sound output can not be produced near the real-time threshold. Simulation for certain systems, however, is now coming within range of real-time. Simpler 1D and 2D linear objects have manageable levels of computation, and can form the basis of a modular system when using multiple connected objects. Nonlinear behaviour can be introduced through the connection elements, leading to a wide range of possible sonic outputs. The purpose of this study is to demonstrate the audio programming techniques used in order to develop a useable real-time system which runs on consumer hardware (i.e., a laptop/basic desktop machine, with implementation in a standard plug-in architecture such as an Audio Unit [7]).

The remaining sections of this paper are as follows: Section 2 presents model equations for stiff strings and plates, and for the nonlinear connections between these objects, as well as a general description of the numerical update equations in the run-time loop. Section 3 examines the maximum size of the most expensive element, the linear plate, that can be computed in one second at a sample rate of 44.1kHz, and the various optimisations that can be used for CPU computation. Section 4 applies the same testing, but within a working Audio Unit plug-in. Finally, Section 5 details the possibilities and issues involved in designing full modular systems using multiple objects, whilst Section 6 gives concluding remarks. A number of audio examples can be found at this webpage: http://www2.ph.ed.ac.uk/∼cwebb2/Site/Plugins.html

2. INSTRUMENT MODELS AND FINITE DIFFERENCE SCHEMES

A complete modular percussion synthesis environment, based on user-specified interconnections among a set of bars and plates, has been described in detail in [8]. In that case, the environment was implemented in the Matlab language, and performance was a long way from being real-time, in all but the simplest configurations. In this section, the model's basic operation is briefly summarized.

2.1. Components

The current system is composed of a collection of stiff strings/bars and rectangular plates, vibrating, in isolation, under linear conditions. For such an object in isolation, and subject to an excitation force, the dynamics are described by the following equation:

$$\mathcal{L}u + g_e f_e = 0 \tag{1}$$

Here, $u(\mathbf{x}, t)$ represents the transverse displacement of the object, as a function of time t, and at spatial coordinate $\mathbf{x} \in \mathcal{D}$. If the object is a stiff string, the spatial domain is a one-dimensional interval $\mathcal{D} = [0, L]$, for some length L, and for a plate, the domain

* This work was supported by the European Research Council, under grant number StG-2011-279068-NESS

is a two-dimensional rectangular region $\mathcal{D} = [0, L_x] \times [0, L_y]$, for side lengths L_x and L_y. See Figure 1.

In the case of a stiff string, the operator $\mathcal{L} = \mathcal{L}_s$ is defined as

$$\mathcal{L}_s = \rho A \partial_t^2 - T\partial_x^2 + EI\partial_x^4 + 2\rho A\sigma_0\partial_t - 2\rho A\sigma_1\partial_t\partial_x^2 \quad (2)$$

Here, ρ is the material density, in kg/m^3, A is cross-sectional area, in m^2, T is tension, in N, E is Young's modulus, in Pa, I is the moment of inertia, in m^4, and σ_0 and σ_1, both non-negative, are parameters allowing for some control over frequency dependant loss. ∂_t and ∂_x represent partial differentiations with respect to time t and the spatial coordinate x, respectively. The stiff string equation must be complemented by two boundary conditions at each end of the domain $\mathcal{D}$. In the current environment, these may be chosen as clamped, simply supported or free, separately at each end of the string. Note that under low tension, the system describes the behaviour of a stiff bar.

In the case of a plate, the operator $\mathcal{L} = \mathcal{L}_p$ is defined as

$$\mathcal{L}_p = \rho H \partial_t^2 + \frac{EH^3}{12\left(1-\nu^2\right)}\Delta^2 + 2\rho H\sigma_0\partial_t - 2\rho H\sigma_1\partial_t\Delta \quad (3)$$

Here, ρ, E, σ_0 and σ_1 are as before, H is thickness, in m, and ν is Poisson's ratio. Here, Δ is the Laplacian operator in two spatial dimensions. The plate equation must be complemented by two boundary conditions at each edge of the domain $\mathcal{D}$. Here, tensioning effects (to simulate a membrane) have not been included, as computational costs become too large for real-time under most conditions—it is not difficult to introduce an extra term above allowing for such tensioning effects.

In either case, the term $g_e f_e$ represents an excitation. Here, $g_e = g_e(\mathbf{x})$ is a spatial distribution representing the region over which the excitation is applied, and $f_e(t)$ is an externally applied force. For typical striking or plucking gestures, $g_e(\mathbf{x})$ is highly localised (and perhaps idealised to a Dirac delta function). For a strike or pluck, f_e is also usually localised. For example, the function f_e defined by

$$f_e(t) = \frac{f_0}{2}\left(1 - \cos\left(q\pi\frac{t-t_0}{T}\right)\right), \qquad t_0 \le t \le t_0 + T \quad (4)$$

and which is zero otherwise is a good approximation to the force history of a strike, occurring at time $t = t_0$, of duration T seconds, and of maximum force f_0 when $q = 2$, and a pluck when $q = 1$.

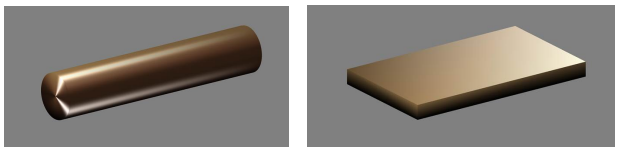

Figure 1: *Basic stiff string or bar (left) and plate (right) elements.*

2.2. Connecting elements

Consider now two components a and b with transverse displacements u_a and u_b, under a single connection:

$$\mathcal{L}_a u_a + g_a f_c = 0 \qquad \mathcal{L}_b u_b - g_b f_c = 0 \quad (5)$$

Here, $\mathcal{L}_a$ and $\mathcal{L}_b$ are linear operators of the types given in (2) and (3), each defined by an appropriate set of material and geometric parameters; the components can be of either stiff string of plate type. Here, g_a and g_b are again distributions, of dimension appropriate to the particular component, selecting a region of interaction for the connection element. $f_c(t)$ is the connection force in N. In particular, it acts in an equal and opposite sense on the two objects (see Figure 2).

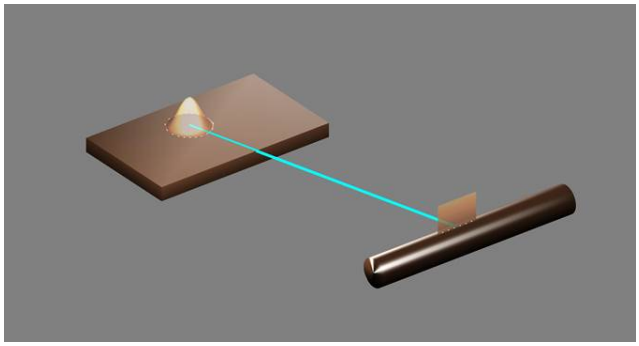

Figure 2: *Connection between stiff string element and plate element.*

Until the form of the connection has been specified, the system is not complete. To this end, define an interaction distance $\eta_c(t)$, averaged over the connection region, by

$$\eta_c = \int_{\mathcal{D}_a} g_a u_a dV_a - \int_{\mathcal{D}_b} g_b u_b dV_b \quad (6)$$

where $\mathcal{D}_a$ and $\mathcal{D}_b$ are the domains of definition of the two components, and where dV_a and dV_b are differential elements of appropriate dimension for the two components. The connection force $f_c(t)$ can be related to the interaction distance $\eta_c(t)$ by

$$f_c = K_1\eta_c + K_3\eta_c^3 + R\frac{d\eta_c}{dt} \quad (7)$$

for connection parameters K_1,K_3 and R, all assumed non-negative. Such a connection may be thought of as a combination of a linear spring, a cubic nonlinear spring, and a linear damper. Many other choices are of course possible, but such a configuration already allows for a wide variety of connection types and, furthermore, in the numerical case, leads to an efficient implementation in which energy-based stability guarantees are available, see [8]. Such numerical stability guarantees are particularly important in the case of arbitrary nonlinear modular networks in the hands of a composer/designer/musician. Once a single excitation and a single connection element have been described, it is not a large step to describe a network composed of a multitude of objects, linked by an arbitrary number of connections, and under an arbitrary number of excitations, as in Figure 3.

2.3. Finite Difference Schemes and State Space Update Form

In a finite difference implementation, each object is represented by a set of values defined over a grid—1D in the case of a stiff string, and 2D in the case of a plate. The complete mechanics of the construction of finite difference schemes for such objects is provided in [8], and is far too involved to be re-presented here, particularly as this paper is concerned mainly with real-time implementation. It is useful nonetheless to briefly describe the vector-matrix update equations for a complete system.

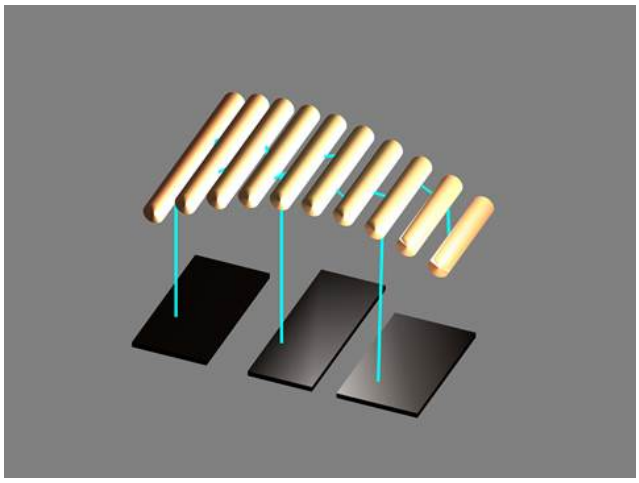

Figure 3: *Interconnected network of plate and stiff string elements.*

Consider an object of either stiff string or plate type in isolation, subject to a single excitation, and for which the dynamics obey (1). Any such object may be approximated by values over a grid—in 1D, for a stiff string, or 2D for a plate (Figure 4). Suppose now that the column vector $\mathbf{u}^n$ represents the concatenation of all values for all system components at time step n; here, time steps are separated by T_s seconds, where the sample rate F_s is defined as $F_s = 1/T_s$. The defining equations are of second order in the time variable t, and in the discrete case, two-step recursions are minimal in terms of memory usage, and are a natural choice.

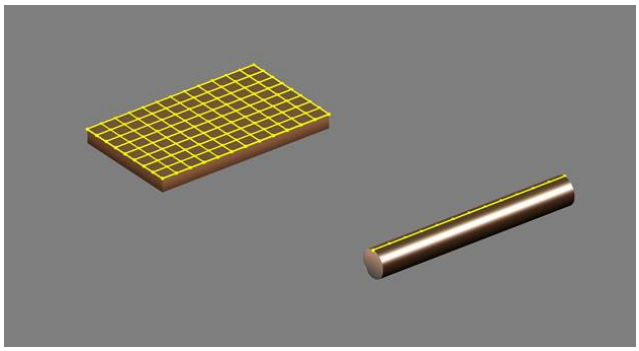

Figure 4: *Grids for plate and stiff string elements.*

The following recursion, operating at an audio sample rate, may be derived from the combination of external excitations, as given in (1) and connections, as per (5), for a system constructed of multiple components and connections, using standard procedures:

$$\mathbf{u}^{n+1} = \mathbf{B}\mathbf{u}^n + \mathbf{C}\mathbf{u}^{n-1} - \mathbf{G}_e\mathbf{f}_e^n - \mathbf{G}_c\mathbf{f}_c^n \tag{8}$$

Here the total state size (that of $\mathbf{u}^n$) is N, the total number of connections is N_c, and the total number of separate excitations is N_e. The constant matrices $\mathbf{B}$ and $\mathbf{C}$ are square and $N \times N$, sparse, and incorporate the effects of boundary conditions in the various components. In particular, they may be derived directly from discretisation of the individual defining operators $\mathcal{L}_s$ and $\mathcal{L}_p$ for the various components.

The update (8), including only the terms involving the matrices $\mathbf{B}$ and $\mathbf{C}$ is a simulation of a set of isolated, unforced components. The column vector $\mathbf{f}_e^n$ is of size $N_e \times 1$, and consists of the external forcing signals, which could in principle be of any form, but in the current system are constrained to be sampled from the pluck/strike waveforms given in (4). The constant matrix $\mathbf{G}_e$ is $N \times N_e$, consisting of columns representing the spatial distributions of the various excitations, sampled over the constituent grids. Finally, $\mathbf{f}_c^n$ is an $N_c \times 1$ vector consisting of the connection forces, and, as in the case of the excitations, $\mathbf{G}_c$ is an $N \times N_c$ constant matrix, the columns of which contain the spatial connection distributions.

In the absence of the connection forces, update (8) is a complete recursion for a set of isolated externally-forced objects. The connection forces $\mathbf{f}_c^n$, however, are related nonlinearly to the yet-to-be-determined state $\mathbf{u}^{n+1}$. A semi-implicit choice of discretisation, as discussed in [8] is useful, in that it leads to numerical stablity conditions, and also to the simple expression:

$$\mathbf{A}^n\mathbf{f}_c^n = \mathbf{b}^n \tag{9}$$

where $\mathbf{A}^n = \mathbf{A}\left(\mathbf{u}^n, \mathbf{u}^{n-1}\right)$ and $\mathbf{b}^n = \mathbf{b}\left(\mathbf{u}^n, \mathbf{u}^{n-1}\right)$. Though nonlinear, it may be solved through an inversion of the $N_c \times N_c$ matrix $\mathbf{A}^n$ (which is positive definite by construction); indeed, if the connection spatial distribution functions are non-overlapping, then $\mathbf{A}^n$ is diagonal, and the solution to (9) may be computed using N_c divisions. Once $\mathbf{f}_c$ has been determined, it may then be used to advance the solution in (8).

3. INITIAL COMPUTATIONAL TESTING

The computational complexity of the individual objects, at an audio rate of 44.1 kHz, can be ranked as follows:

1. Connection: $\approx$ 20 floating-point calculations/time step.
2. Bar / String: $10^2 \sim 10^3$ floating-point calculations/time step.
3. Linear Plate: $10^3 \sim 10^5$ floating-point calculations/time step.

Such operation counts, for the stiff string and plate, follow from the required grid resolution, which is itself dependent on the sample rate. This section details initial testing of the most expensive element, the linear plate, outside of a real-time audio system. As a basic 'benchmark' we can assess the largest size of plate that can be computed within one second of wall clock time, at 44.1kHz. In the absence of any real-time audio overheads, this would be the outer limit of achievable computation; the amount of computation that can be performed in an actual audio plug-in is somewhat less than this, as is shown in Section 4.

As the main objective of this study is to evaluate what is achievable on recent consumer hardware, all testing is performed on an Intel Ivy Bridge Core i7 3770S processor. This has four physical cores, a base clock rate of 3.1GHz, turbo clock rate of 3.9GHz, and a maximum memory bandwidth of 25.6 GB/s. Preliminary testing on a newer Haswell Core i7 showed similar results, with a small improvement of around 5%. The testing performed here is restricted to double precision floating-point—see Section 6 for further discussion on this point. The purpose of this section is to analyse the options for implementing the plate object in C code, and the various optimisation strategies available to the programmer.

Updating the state of each point on the plate requires a weighted sum of thirteen neighbouring points from the previous time step

(See Figure 5), as well as five neighbouring points from two time steps past. The grid also includes a boundary layer, and halo layer of non-updated ghost points. A clamped boundary condition is used for this testing section.

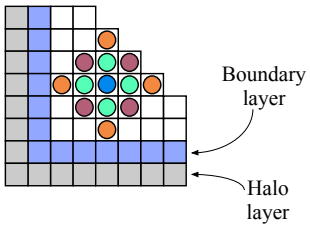

Figure 5: *13 point update stencil, boundary layer and halo.*

In terms of high level design, there are two principal methods for implementing such a scheme: in vector-matrix form, or as an 'unrolled' update equation applied to individual grid points.

3.1. Vector-matrix form

The vector-matrix update form is given by (8), accompanied by the connection force calculation (9). The matrices $\mathbf{B}$, and $\mathbf{C}$ are sparse and (nearly) block Toeplitz. Figure 6 shows the sparsity pattern of the matrix $\mathbf{B}$ corresponding to a single isolated plate. The vectors hold the 2D state data, which is decomposed linearly using either a row or column major approach. Implementing this form in C code is straightforward [9], requiring only a matrix by vector multiplication, and a function for vector addition, to be applied at each time step of the simulation. The CSR (compressed sparse row) format was used as the sparse matrix container.

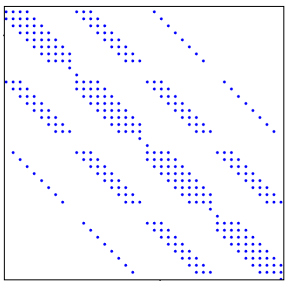

Figure 6: *Sparsity pattern of coefficient matrix* $\mathbf{B}$ *for a single isolated plate.*

3.2. Unrolled update equation

Whilst the vector-matrix form is highly concise (and useful for prototyping in systems such as Matlab), is it clearly somewhat wasteful as it requires storing and reading a large number of constant coefficients. Applying the update equation to individual grid points alleviates this, as factorized scalar coefficients are all that is required. Figure 7 shows a basic implementation of this, inside of the time domain loop. A small optimisation can be achieved by pre-defining the offset constants such as `2*wid`, which are shown here for clarity.

Note that whilst it is possible to use a non-factorized version equivalent to the matrix operations, it is not optimal here (there is

```
// Time loop
for(n=0;n<Nf;n++){

  // Loop over state data in 2D, as we have boundaries
  for(Y=2;Y<(Ny-1);Y++){
    for(X=2;X<(Nx-1);X++){

    // Calc linear index, from row-major decomposition
    cp = Y*width+X;
    // Calculate update equation
    u[cp] = B1*(u1[cp-1]        + u1[cp+1]
              + u1[cp-width]    + u1[cp+width])
          + B2*(u1[cp-2]        + u1[cp+2]
               +u1[cp-2*width]  + u1[cp+2*width])
          + B3*(u1[cp+width-1]  + u1[cp+width+1]
               +u1[cp-width-1]  + u1[cp-width+1])
          + B4* u1[cp]
          + C1*(u2[cp-1]        + u2[cp+1]
              + u2[cp-width]    + u2[cp+width]
          + C2* u2[cp] );
    }
  }

  // Read output
  out[n] = u[out_location];

  // Swap pointers
  dummy_ptr = u2;
  u2 = u1;
  u1 = u;
  u = dummy_ptr;
}
```

Figure 7: Standard C code time loop and state update.

no fused multiply-add instruction in Ivy Bridge). This basic implementation in a single thread is unlikely to make the most of the potential of the processor, even when using compiler optimisation. To do that, further manual optimisation is required, in the form of vector instructions and multi-threading.

3.3. AVX vector instructions

AVX instructions make use of register vector units that are capable of computing arithmetic operations over multiple data simultaneously. For example, rather than computing a single multiplication with two input operands, an AVX instruction will perform a number of multiplications using vector operands, and giving a vector result. On Ivy Bridge and Haswell/Broadwell architectures these are 256 bit registers, and so can operate on four double precision data elements at a time (in the upcoming Skylake architecture these are extended to 512 bit registers). Clearly the ability to perform basic arithmetic operations on multiple data at the same time is well suited to finite difference schemes, where each updated element is independent in a given time step. Figure 8 shows the vector implementation of the update scheme, using nested intrinsic functions to compute the stages of the update.

Note that the loop over the state data is now one-dimensional, and is incremented by the vector size. Reducing down to a single `FOR` loop allows the SIMD vector operations to be performed in a continuous manner over the entire state (save for any additional overs at the end). This does require corrections at the boundaries, but these are at a minimal cost, especially when alternative boundary conditions are applied. Further manual unrolling of the loop (i.e. to a size of eight using two sets of updates applied consecutively) did not provide a further speedup.

```
// Time loop
for(n=0;n<Nf;n++){

  // Loop over state data
  for(cp=start;cp<vecend+1;cp+=vector_size){

    // Load up state data into separate vectors
    u1m2   = _mm256_load_pd(&u1[cp-2]);
    u1m1   = _mm256_load_pd(&u1[cp-1]);
    ...
    ...

    // Perform update equation in stages
    s1 = _mm256_mul_pd(B1v,_mm256_add_pd(_mm256_add_pd(
         u1m1, u1p1),_mm256_add_pd(u1mw,u1pw)));

    s2 = _mm256_mul_pd(B2v,_mm256_add_pd(_mm256_add_pd(
         u1m2, u1p2),_mm256_add_pd(u1m2w,u1p2w)));

    s3 = _mm256_mul_pd(B3v,_mm256_add_pd(_mm256_add_pd(
         u1pwp1,u1pwm1),_mm256_add_pd(u1mwp1,u1mwm1)));

    s4 = _mm256_mul_pd(B4v, u1cp);

    s5 = _mm256_mul_pd(C1v,_mm256_add_pd(_mm256_add_pd(
         u2m1, u2p1),_mm256_add_pd(u2mw, u2pw)));

    s6 = _mm256_mul_pd(C2v, u2cp);

    s7 = _mm256_add_pd(_mm256_add_pd(_mm256_add_pd(s1,
         s2), _mm256_add_pd(s3, s4)), _mm256_add_pd(s5,
         s6));

    // Store result
    _mm256_store_pd(&u[cp], s7);

  }

  // Deal with overs at the end of the state size, and
       correct boundaries that were over-written
  ...

  // Read output
  out[n] = u[out_location];

  // Swap pointers
  dummy_ptr = u2; u2 = u1; u1 = u; u = dummy_ptr;
}
```

Figure 8: AVX vectorization time loop.

3.4. Multi-threading

Whilst vector extensions provide a highly effective optimisation, the system is still only making use of a single core of the processor. Even the consumer-based i7 has four cores available, and so a further optimisation is to explore the use of multiple threads that can parallelize the operation over these cores. Again, the finite difference scheme is well suited for this, as the state can simply be partitioned into a suitable number of parts, with each thread operating over an independent section of data at each time step. This can be combined with AVX instructions as used above. Whilst frameworks such as OpenMP can be used to implement multi-threading using compiler directives, this section considers the manual use of POSIX threads (Pthreads).

There are, however, some design issues that need to be considered. The primary concern is how to issue threads that operate over spatial elements of the grid, but also take into account the updates over time. There are two possible approaches. First, one could create the threads at the beginning of each time step, perform the state update, and then destroy them. This would then be repeated each time step, but has the benefit of not requiring any additional thread barrier synchronisation. However, the overheads involved in the approach mean that it only becomes viable when large-scale arrays are used [10]. At the scale of the plates possible in real-time, this approach does not yield any performance benefits.

In order to hide the latencies in issuing threads, we need to create them not at each time step, but only once over many time steps. In the context of real-time audio, creating them at the start of an audio buffer (i.e. every 256 time steps) works well. Figure 9 shows the detail of the time loop.

```
// Time loop
for(int n=0;n<Nf;n+=buffer_size){

  // Create threads
  for (i=0; i<NUM_THREADS; i++) {
    td[i].n = n;
    td[i].u  = u;
    td[i].u1 = u1;
    td[i].u2 = u2;
    pthread_create(&threads[i], NULL, updateScheme, &td[
        i]);
  }

  // Destroy threads
  for (i=0; i<NUM_THREADS; i++) {
    pthread_join(threads[i], NULL);
  }
}
```

Figure 9: Time loop for multi-threading using POSIX.

Note that the time loop is now incremented by the buffer size, and the current time step integer is passed into the thread kernel in order to correctly read the output into an array. The coefficients are loaded into the `td` struct prior to the time loop. As the threads are now operating over time, a barrier is required inside the kernel to ensure that the threads are synchronised prior to performing the pointer swap. The outline of the kernel function is shown in Figure 12. Initial testing revealed that using two threads only provides a small increase over a single thread, eight threads actually perform worse (despite hyper-threading showing eight logical cores), and four is the optimal configuration here.

3.5. Summary

Testing was performed using a square plate at 44.1kHz, with the size varied such that one second of clock time was required to compute 44,100 time steps. Timing functions from `<sys/time.h>` were used, employing 'wall clock' functions that give accurate measurement even when using multi-threading. All codes were compiled using LLVM, with -O3 optimisation (-O0 was also tested as a comparison). Table 1 shows the results for each implementation.

The first result to note is the difference between the matrix form, which achieved 41 x 41, and the standard C code which achieved 80 x 80, nearly four times more efficient. Clearly the extra data movement is a cost, but also the compiler is more likely to aggressively optimise the unrolled version. However, the standard C code is still using only a fraction is the possible CPU performance. The AVX instructions provide a 2X speedup, clearly well below a linear 4X but there are overheads in loading the data points into the vector registers. From there, applying multiple threads achieves a further 35% increase in performance. Again, this is significantly below a linear increase over the single thread version, partly due to the decrease in clock frequency when running over

Code version (-O3 unless otherwise stated)	**Plate size** (grid points)	**Total state size** (grid points)
CSR matrix	41 x 41	1,681
C with -O0	44 x 44	1,936
C	80 x 80	6,400
C with AVX	112 x 112	12,544
C with AVX & 4 Pthreads	130 x 130	16,900

Table 1: *Maximum plate sizes for one second computation time, running for 44100 time steps.*

multiple cores, and also latencies involved in thread synchronisation at each time step. This is typical of multi-threaded codes, where best performance is achieved when parallelising over much larger arrays [10].

The algorithm is clearly memory bandwidth limited, and so it is useful to assess the memory throughput efficiency of the implementation. Taking one read and one write per grid point (assuming all other reads will be from cache), this can be calculated as 130 x130 grid points x 2 x 44100 time steps x 8 bytes = 11.9 GB/sec. This is half of the theoretical maximum, but there does not seem to be any obvious further avenues for optimisation. Note the importance of applying vectorization and multi-threading. Without them, the basic C code with -O3 only achieves 38% of the maximum performance.

4. TESTING WITHIN A REAL-TIME AUDIO UNIT

Audio Units are Apple's native plug-in architecture for enhancing digital audio hosts such as Logic Pro or Ableton. From a programming perspective, they are executable code within a standard API defined by the Core Audio system [11]. The plug-ins can be created either by using a development framework such as JUCE [12], or by directly sub-classing the Core Audio SDK. The latter approach is used here (the AU v2 API), to minimise any overhead involved with the use of additional structures.

The purpose of this section is to determine the maximum size of plate that can be computed from within an actual working plug-in. The basic unrolled code, as well as the AVX and Pthread codes were all tested, with the size of the plate varied to determine the largest size achievable just prior to audio dropouts. Within the real-time system, audio dropouts are caused by the buffer not being written completely, resulting in 'clicks' at the output stream. Testing was performed using the AU Lab host application, which gives visual warnings when buffer under-runs occur. Drop outs typically start at around 90% of CPU load. A buffer size of 512 samples was used, although the load appeared constant across sizes from 128 to 2048 samples. Table 2 shows the resulting plate sizes.

The multi-threaded and vectorized code still produces the best result, but at a reduced margin over the single thread AVX version. However, compared to the testing in the previous section, it achieves only 50% of the maximum plate size. As previous mentioned, this is largely an issue of data size, where smaller arrays achieve less performance benefits. Considering the single thread AVX code, this achieved 65% of the maximum achieved outside of the real-time system, with a size of 90 x 90 compared to 112 x 112. The non-vectorized code performs at 68%, so in general terms we can say that C code that runs in just less than 0.7 seconds

Code version (-O3 unless otherwise stated)	**Plate size** (grid points)	**Total state size** (grid points)
C	66 x 66	4,356
C with AVX	90 x 90	8,100
C with AVX & 4 Pthreads	92 x 92	8,464

Table 2: *Maximum plate size for an Audio Unit instrument plug-in, running at 44.1kHz.*

will be the maximum computable within the real-time plug-in for the single thread case.

5. IMPLEMENTING A COMPLETE SYSTEM

Having established the maximum performance for a single plate within an Audio Unit plug-in, this section examines the potential for real-time modular systems containing multiple object types. Creating a usable system that is capable of producing a wide sonic range requires the combination of multiple stiff strings, connections and plates. Clearly only one full-scale plate will be possible, but there is scope for a large number of additional 1D objects.

Very stiff strings (or bars) can be tuned a fundamental frequency over a range of around two octaves, from C3 to C5. This results in very small state arrays of around twenty elements each. Low stiffness strings require more state, varying from around 30 elements at C5 to around 250 elements for a low C2. Both can benefit from the use of AVX instructions, which result in 2X performance gains. At these sizes, a large number of such objects can be computed, depending on the size of the plate that is used.

Bars can also be used as resonators, when using a low stiffness value, and here the state size can increase to up to a thousand elements. A plate of dimensions comparable to a plate reverberation device (steel, 1.5m x 1.3m and of thickness 2mm) requires a state size of approximately 7000. Testing showed that this still leaves around 20% of the CPU to compute 1D objects and connections. This is sufficient to run a system such as that shown in Figure 10.

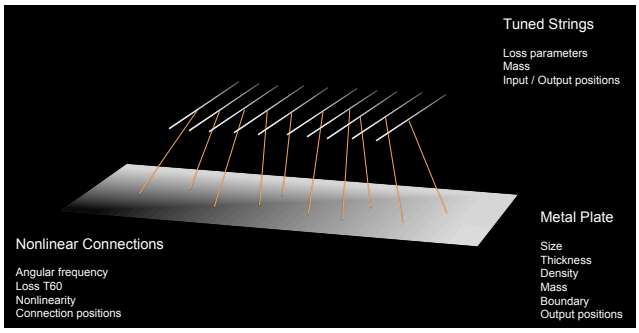

Figure 10: *Tuned string and plate system, with control parameters.*

There are a number of issues involved in actually implementing such a system in a real-time plug-in. The first is the question of parameter updating. Whilst some of the parameters such as the output position and angular frequency of the connections can be varied during real-time computation, others, such as the density of the plate cannot as the underlying grid representation will necessarily be altered. The parameters therefore need to be grouped

into 'state' and 'real-time' controls, and an appropriate system of applying state control changes is required.

The second major issue is how to deal with the large number of parameters that arise. A system consisting of, e.g., 20 stiff string objects, requires 200 parameter settings. This is problematic in terms of user interface design, and requires some level of grouping to be manageable in the first instance. The ability to work with hundreds of parameters may only be possible with a fully-realised visual interface, where individual objects could be manipulated using graphical representations.

There is also the question of the usable parameter space—understanding the combinations and limits of control parameters that lead to usable output. Experimentation in this regard is certainly easier inside a real-time environment. Despite these issues, initial prototype Audio Units have been created, such as systems of pitched bars connected to resonator bars, and tuned strings connected to a large-scale plate. These are playable from a standard MIDI keyboard, and make use of control change signals to vary the real-time parameters. Audio examples can be found at this webpage: http://www2.ph.ed.ac.uk/∼cwebb2/Site/Plugins.html

These prototypes were created using a two-stage process. First, individual C++ objects were created that define a stiff string, a plate, and a connection. These used common interface methods in order to facilitate the second stage, which is to create an instrument model. Here, a number of objects were instantiated using the primitive elements, and a further interface is constructed to allow the instrument to be easily tied into the Audio Unit SDK. Figure 11 shows the class diagram of the string and plate instrument used to create the plug-in.

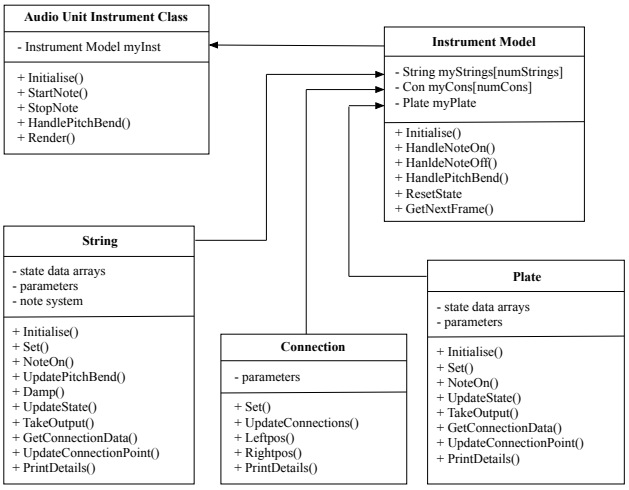

Figure 11: *Class diagram of the string and plate instrument plug-in, showing common interface elements.*

6. CONCLUSIONS AND FUTURE WORK

This paper has demonstrated the use of CPU optimisation techniques to compute coupled 2D and 1D finite difference schemes in real-time, as well as implementation techniques within real-time plug-ins such as Audio Units. Whilst AVX instructions provide a 2X speedup at double precision floating-point, the use of multi-threading is more complex. For the size of plates that can be computed in real-time, the use of multiple threads within a single object does not produce significant performance benefits. A more efficient approach may be to apply threads to separate objects, such as a thread for a plate, and another operating over the 1D objects and connections.

A further avenue for experimentation is the use of single precision floating-point. Although this may result in artefacts in the system due to the nonlinear elements, it would widen the possibilities for computation on consumer hardware. Vector extensions can operate over twice the number of single precision data, which could easily provide performance benefits on the CPU. Also, single precision would allow the use of consumer level GPUs. There have already been some experiments in this regard [13] [14], but with a simpler system consisting of a single 2D wave equation solver. The use of a desktop machine such as Apple's recent Mac Pro has interesting possibilities as it contains two GPUs, both capable of double precision calculations. This would require the use of the OpenCL language, and could also be used to explore multi-core CPU operation.

7. REFERENCES

[1] C. Cadoz, A. Luciani, and J.-L. Florens, "Cordis-anima: A modeling and simulation system for sound and image synthesis," *Computer Music Journal*, vol. 17, no. 1, pp. 19–29, 1993.

[2] J.-L. Florens and C. Cadoz, "The physical model: Modeling and simulating the instrument universe," in *Representations of Musical Signals*, G. DePoli, A. Picialli, and C. Roads, Eds., pp. 227–268. MIT Press, Cambridge, Massachusetts, 1991.

[3] D. Morrison and J.-M. Adrien, "Mosaic: A framework for modal synthesis," *Computer Music Journal*, vol. 17, no. 1, pp. 45–46, 1993.

[4] F. Pedersini, A. Sarti, S. Tubaro, and R. Zattoni, "Towards the automatic synthesis of nonlinear wave digital models for musical acoustics," in *Proceedings of EUSIPCO-98, Ninth European Signal Processing Conference*, Rhodes, Greece, 1998, vol. 4, pp. 2361–2364.

[5] S. Bilbao, *Numerical Sound Synthesis: Finite Difference Schemes and Simulation*, Wiley, 2009.

[6] S. Bilbao, B. Hamilton, A. Torin, C.J. Webb, P. Graham, A. Gray, K. Kavoussanakis, and J. Perry, "Large Scale Physical Modeling Sound Synthesis," in *Proceedings of the Stockholm Music Acoustics Conference*, Stockholm, Sweden, 2013, pp. 593–600.

[7] Apple Incorporated, "Audio Unit Programming Guide," *[Onine document][Cited: 7th June 2015.] https://developer.apple.com/library/mac/documentation/MusicAudio/Conceptual/AudioUnitProgrammingGuide*, July 2014.

[8] S. Bilbao, "A modular percussive synthesis environment," in *Proceedings of the 12th International Conference on Digital Audio Effects*, Como, Italy, 2009.

[9] G. Golub and C. Van Loan, *Matrix computations (3rd ed.)*, Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[10] C. Webb, *Parallel computation techniques for virtual acoustics and physical modelling synthesis*, Ph.D. thesis, University of Edinburgh, 2014.

[11] W. Pirkle, *Designing Software Synthesizer Plug-Ins in C++*, Focal Press, 2015.

[12] M. Robinson, *Getting started with JUCE*, Packt Publishing, 2013.

[13] B. Hsu and M. Sosnick, "Realtime GPU audio: Finite difference-based sound synthesis using graphics processors," *ACM Queue*, vol. 11, no. 4, May 2013.

[14] M. Sosnick amd B. Hsu, "Implementing a finite difference-based real-time sound synthesizer using GPUs," in *Proceedings of the International Conference on New Interfaces for Musical Expression*, Oslo, Norway, 2011.

```
void *updateScheme(void *targ){

  // Calculate state array partition points for thread
  int psize  = 1 + ( (Nx+1)*(Ny+1)-1)/NUM_THREADS;
  ...

  // Load coefficients into vector objects
  __m256d B1v = _mm256_set1_pd(B1);
  ...

  // Loop over buffer size
  for(int n=0;n<buffer_size;n++){

    // Loop over state data
    for (cp = start;cp<vecend+1;cp+=vector_size){

      // Load up state data into separate vectors
      u1m2   = _mm256_load_pd(&u1[cp-2]);
      u1m1   = _mm256_load_pd(&u1[cp-1]);

      ...

      // Perform update equation in stages
      s1 = _mm256_mul_pd(B1v,_mm256_add_pd(_mm256_add_pd
           (u1m1, u1p1),_mm256_add_pd(u1mw,u1pw)));

      s2 = _mm256_mul_pd(B2v,_mm256_add_pd(_mm256_add_pd
           (u1m2, u1p2),_mm256_add_pd(u1m2w,u1p2w)));

      ...

      // Store result
      _mm256_store_pd(&u[cp], s7);

    }

    // Deal with overs
    ...

    // Read output
    if (tid==0) md->out[...] = u[md->out_location];

    // Barrier
    pthread_barrier_wait (&barrier);

    // Swap pointers
    dummy_ptr = u2; u2 = u1; u1 = u; u = dummy_ptr;

  }

}
```

Figure 12: POSIX thread kernel function.