REAL-TIME MODAL SYNTHESIS OF CRASH CYMBALS WITH NONLINEAR APPROXIMATIONS, USING A GPU

Travis Skare

CCRMA
Stanford University
Stanford, CA
travissk@ccrma.stanford.edu

Jonathan Abel

CCRMA
Stanford University
Stanford, CA
abel@ccrma.stanford.edu

ABSTRACT

We apply modal synthesis to create a virtual collection of crash cymbals. Synthesizing each cymbal may require enough modes to stress a modern CPU, so a full drum set would certainly not be tractable in real-time. To work around this, we create a GPU-accelerated modal filterbank, with each individual set piece allocated over two thousand modes. This takes only a fraction of available GPU floating-point throughput.

With CPU resources freed up, we explore methods to model the different instrument response in the linear/harmonic and nonlinear/inharmonic regions that occur as more energy is present in a cymbal: a simple approach, yet one that preserves the parallelism of the problem, uses multisampling, and a more physically-based approach approximates modal coupling.

1. INTRODUCTION

Modal synthesis is an effective way of capturing a variety of physical sounds. Its parameters are intuitive and have a space-efficient representation. Parameters may be computed by solving the equations governing the system, analytically from a series of system measurements or instrument recordings, or by combining or morphing between other instruments' coefficients.

Increasing the number of modes N of our synthesizer is important to broaden the types of sounds we may represent. A cowbell may be synthesized with a few dozen modes, but qualitatively, a cymbal wash benefits from hundreds or even thousands of modes. Compare the whole-sound spectra of a cowbell against that of a cymbal in Figure 1. To see behavior over time, the spectrogram of a Sabian 16" HHX Evolution Crash, a relatively "dark" or "complex" crash, is shown in Figure 2. We note evidence of highly nonlinear effects, such as higher frequencies emerging in the 100ms-300ms range. These cannot be simulated directly with our basic linear modal synthesis method, but we will explore methods that attempt to obtain such sounds by extending our linear synthesis system with approximations.

In addition to having a complex sound with many modes, thin shells exhibit highly nonlinear behavior, with response often divided into three regimes: linear, inharmonic, and chaotic. This makes creating accurate physical models difficult[1, 2, 3], though the sounds and control space are rich.

Ducceschi et al.[4] solve the von Kármán equations that govern the underlying nonlinear plate physics of plates and cymbals in terms of the system's modes. They obtain values for coupling coefficients offline, yet efficiently. In [5] the authors extend this work and apply a simpler, though less inherently stable, Störmer-Verlet method for time integration to obtain synthesis results running fairly quickly–only 8x slower than real-time on a CPU. Nguyen et al. [6] use a similar time stepping method to cymbal synthesis, paying particular attention to specific cymbal geometry variations that is relatively unique in the literature–cymbals are not plates of uniform thickness, but vary from bell to inner bow to edge, and the authors demonstrate this is an important property to model when considering cymbals over gongs.

Our end goal is to run simulation of several cymbals in realtime with enough processing power left over for the rest of a drum set and the other instruments. GPU acceleration is attractive for this application–using the *single instruction, multiple thread* model of execution, modern graphics processors excel at executing one piece of code simultaneously across tens, hundreds, or even thousands of threads, each performing the same operations on different data. This is a more parallel version of vector instructions, and while it is not applicable to every application (handling one input through a series of connected effects plugins, for example), there are some audio applications where it excels. In [4] for example, Ducceschi et al. reduced computation time from 90 minutes per second of audio on a desktop CPU down to 55 seconds per sound. And we again emphasize their work is physically accurate while our work here will aim for simpler approximations for

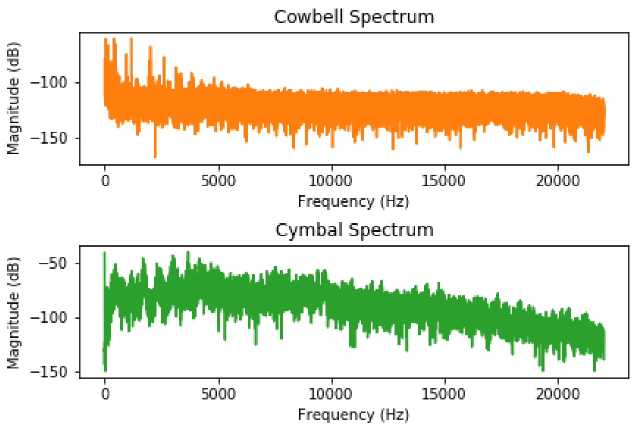

Figure 1: *Top: Cowbell spectrum. Bottom: Sabian HHX Crash Cymbal Spectrum. Note the latter has a significantly higher density of modes, while the former can be better represented by a smaller number of local maxima.*

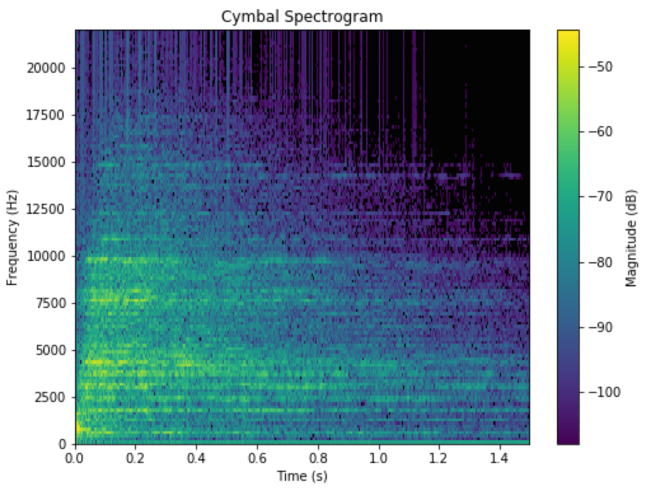

Figure 2: *Sabian HHX Evolution Crash Spectrogram.*

cymbal-like sounds.

The literature has several projects involving GPU-Accelerated additive synthesis or massive parallel filtering:

Savioja et al.[7] give an overview of potential audio tasks that may be accelerated via GPGPU programming at audio rate and reasonable buffer sizes for real-time performance. For example, FFTs running on a GPU were able to be eight times as long as their CPU-implementation counterparts. GPU-accelerated FIR filters were able to be 130x as long as the baseline versions. In [8], the authors synthesized 1.9 million sinusoids in real-time, a 1300x speedup over a serial lookup table computation on one CPU. This was on a GPU that is six generations behind ours and three major GeForce architecture revisions behind the card we are using, and we note that our card is itself a generation behind state of the art.

Trebien et al.[9] use modal synthesis to synthesize physically-accurate collision sounds between objects of different materials – balls rolling down a ramp, for example. They introduce a transform of the system's IIR filters to be able to compute several samples in parallel, turning the operation into linear convolutions that are well-suited for a GPU. More recently, Cirio et al.[10] tackle this problem with specific attention on cymbals and gongs, and in particular attempt to model the chaotic and wave-turbulent effects. This work leverages parallelism between frequency bands as one strategy to parallelize, and as they simulate nonlinear effects in a more physically-accurate fashion than we will, they obtain rich sounds with only 100 high-frequency modes. Simulation costs for one cymbal are still 43x slower than real-time, but that is a 70x speedup over their target algorithm, making the approach tractable for generation audio to graphics offline. Chadwick et al.[11] also apply modal strategies, including simulated coupling, plus introduce far-field acoustic transforms, for such "virtual foley" work. Collisions of thin shells including water bottles and crash cymbals sound realistic. Simulation and precomputation are expensive, but the runtime acoustic transfer map step runs relatively faster, at 16x slower than real-time.

Belloch et al. [12] apply a transform on IIR filters to make them more suitable for the GPU. At 44.1kHz they run over 1,200 256th-order IIR filters simultaneously, with latency of less than a millisecond. Subsequently, Belloch et. al apply massively parallel filtering[13] to Wave Field synthesis on a 96-speaker array[14], running nearly ten thousand fractional-delay room filters with thousands of taps. Several dozen sources were able to be placed into the field.

Bilbao and Webb[15] describe a GPU-accelerated model of a set of timpani, simulating both the drums as well as the space outside. They obtain speedups of 8x to 30x over CPU implementations, and the drum update runs in near-real-time (2 milliseconds per sample), with the bottleneck being a linear drum membrane update.

With advances in machine learning and deep learning fields, some groups are using GPUs to train models that output audio during inference, an interesting approach different from physical modeling. The convolutional WaveRNN by Kalchbrenner, Elsen et al.[16] can run inference (generation) even on mobile CPUs.

The physical modeling applications mentioned above often exploit parallelism across time – different GPU threads are simultaneously working on $x(t_n)$ and $x(t_{n+k})$ for small k. This is often the result of a clever transform, for example moving IIR filters to parallel form. In our application, we'd like to retain the option to modulate system parameters based on a signal we are fed sample-by-sample, and unfortunately cannot apply such transforms. In [17] we demonstrated our modal synthesis filters can be run at audio rates on modern GPUs in a serial fashion–that is, GPU core speed and FPU throughput have advanced sufficiently in recent years such that developers do not need to parallelize across time for certain audio applications (of course, if you can, it still unlocks a great deal of parallelism). Floating-point throughput on modern high-end consumer hardware is enough to run over a million such filters.

Since that work on building blocks and benchmarks, a real-time GPU filterbank was developed, which we present here and then leverage toward cymbal synthesis.

2. BUILDING BLOCKS: VERY HIGH-Q PHASOR FILTERS

We use a modal filterbank for synthesis. This consists of N resonant filters. We make the assumption that all the filters are uniform in construction and vary in parameters, though a GPU could run multiple styles of filters in parallel, either through conditional execution or simultaneous kernel execution.

Modal synthesis may be performed using the developer's favorite resonant filter, or via adding sinusoids. In practice, rapidly changing the coefficients on e.g. Direct-Form II filters may result in audible artifacts. Adding sinusoids is efficient, especially if we're simply performing lookups into a table that's in processor cache, but we have a preference for another approach if computational resources are available.

In [18] Max Mathews and Julius Smith proposed a filter that is very-high-Q, numerically stable, and artifact-free, based on properties of complex multiplication. The parameters map one-to-one with the physical properties of our system which makes it a great choice for modal synthesis and modal reverberators such as in [19]. The recursive update equation we need to implement is:

$$y_m(t) = \gamma_m x(t) + e^{(j\omega_m - \alpha_m)} y_m(t-1) \quad (1)$$

where:
$x()$ is an input or excitation signal.
ω_m is mode m's frequency.
γ_m is a per-mode complex input amplitude gain.

α_m is a per-mode damping factor.

In terms of implementation details, the state we store for each mode is limited to the prior output $y_m(t-1)$, plus having the input parameters available (α_m, γ_m, and ω_m). This fact will be relevant when we walk through the GPU code.

This filter has some nice properties, such as that it preserves phase across restrikes if parameters are updated on zero-crossings. We next present how we may run hundreds of thousands of them in parallel.

3. GPU FILTERBANK IMPLEMENTATION

We present a system that has three major pieces across two processes. A system diagram is in Figure 3.

- The **CPU Client** is a JUCE C++ plugin that accepts MIDI events, translates them into modes, and populates data structures to facilitate communication to the GPU (modal parameters) and from the GPU (audio data to copy to a DAW audio callback buffer). For a set of cymbals, we may copy in a set of modes that are mapped to a particular MIDI note representing that instrument selection. For a tonal instrument application, modes would be scaled to start on the correct fundamental frequency.
- The **GPU Process** contains code to allocate memory on the graphics card and on the host PC. It copies relevant data to the GPU and manages kernel execution.
- The **GPU kernel** is the code that runs on the GPU and represents the core of the synthesis.

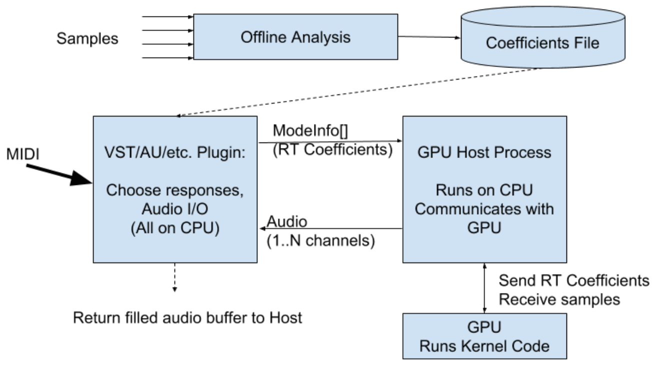
Samples
Offline Analysis
Coefficients File
MIDI
VST/AU/etc. Plugin:
Choose responses, Audio I/O (All on CPU)
ModeInfo[] (RT Coefficients)
Audio (1..N channels)
GPU Host Process
Runs on CPU Communicates with GPU
Send RT Coefficients Receive samples
Return filled audio buffer to Host
GPU Runs Kernel Code

Figure 3: *Overall System Diagram. Top: offline components analyze modes from samples. Bottom: real-time system synthesizes using the proposed parallel architecture.*

For the sake of discussion, we aim to keep the GPU kernel code as simple, short, and approachable as possible. Specifically, it will take as input our modal filter parameters $\omega_N, \gamma_N, \alpha_N$ and input signal $x()$, and simply synthesize the resulting audio output. Later versions may perform parameter interpolation, modal coupling, etc. or post-process audio.

Keeping the GPU process as simple as possible allows much of the complexity to be moved to an application that calls the GPU code, enables easy extension to multiple client applications, enables rapid development, and allows for code reuse across projects as the GPU process will happily run in the background indefinitely while only taking a few MB of RAM.

Communication between host and GPU processes was accomplished using shared memory, and signaling between processes was done using semaphores. The shared memory is simply a chunk of memory mapped into each of the two CPU processes. It is a series of `ModeInfo` structures as defined in Listing 1 followed by a chunk of memory to allow transmitting audio data back to the JUCE plugin so it can in turn be sent back to the DAW or the next plugin in the chain for more processing. This was chosen for simplicity and speed; the code should compile without any external libraries beyond CUDA. Other projects may wish to replace the cross-process communication with, for example, an OSC client and run the host application on a tablet.

```
struct ModeInfo {
  bool enabled;  // is this filter on?
  bool reset;    // should we zero state?

  // filter parameters
  float amp_real;  // Re{alpha}
  float amp_imag;  // Im{alpha}
  float damp;      // gamma
  float freq;      // omega

  // currently-unused optimizations
  bool amp_changed;  // did alpha change?
  bool freq_changed; // did omega change?
};
```

Listing 1: ModeInfo structure used in shared memory block.

Most development and demos were run at 256 samples at 44.1kHz (5.6ms); a buffer of 128 samples was tested successfully.

Our graphics card is an NVIDIA GeForce 1080Ti; it should run on some older cards where core frequency and floating-point throughput are sufficient. It is also expected to run with even better performance on the newer RTX series cards. We have not attempted a port to OpenCL to run on AMD cards, but with a high enough core speed and floating-point throughput it should be straightforward. For those interested in CUDA GPU programming, a variety of books are available, and the NVIDIA developer documentation[20] likely contains all that is required to get started.

3.1. CPU vs GPU limits

In synthetic benchmark settings in [17], where the entire system is concentrated on running these filters, we found a GPU could run over 3.5 million phasor filters at audio rate, or 1.8 million allowing for continuous modulation of filter parameters (requiring additional multiplications). Porting this test to run on an Intel i7 4770k CPU core showed the CPU could run just over 3,000 phasor filters in real-time on one thread, allowing for per-sample parameter modulation. A synthetic benchmark constructed in FAUST using alternate building blocks–second-order resonant filters–could run 4,600 filters on one CPU thread.

In terms of real-world use cases, a “cymbal-verb” modal reverberator plugin was developed with such a phasor filterbank at its core; when running inside inside a DAW, at 128 samples, pressured by audio callbacks, with a few other audio tracks plus one instance of the modal filter effect plugin, overruns happened between 1600 and 1900 modes, with significant variance. A 2012 MacBook Pro, with a laptop Intel i7 3820QM, suffered dropouts when synthesizing 1000-1100 modes. The GPU process running on the desktop alongside the DAW was able to successfully synthesize 100,000 modes, above which we are blocked on a need to optimize the communication between the processes. We note in this case we are utilizing the GPU, while the DAW does not leverage that resource. Our GPU resource closest to bottlenecking in this trial is the 32-bit floating-point units, which debugging tools estimated at 5.5% utilization.

We note our benchmark parts in this system are slightly mismatched: the CPU is a mid-to-high-end 2013 consumer part while the GPU is the flagship model from its series from 2017, and cost over twice as much. We also note that the majority of audio plugins will have a natural affinity for running on a CPU, but highly parallel tasks may take advantage from a GPU especially if it would otherwise sit idle.

3.2. GPU Kernel Code

The code is cross-platform; Linux and Windows used different APIs for shared memory and semaphores, so we simply guard platform-specific code with e.g. `#ifdef WINDOWS`. The GPU kernel code itself is completely platform-agnostic, however there are two important limitations as of this writing: first, JUCE support for VST3 under Linux is under development. Second, CUDA drivers are not available for MacOS 10.14+. We benchmarked with the JUCE plugin wrapper, and ran integration tests with a DAW on Windows.

GPU Kernel code is presented in Listing 3, and is available online on the CCRMA GitLab[1]. It is commented at the block level in the listing; the code walk in section 3.4 communicates a higher-level explanation.

3.3. GPU Programming Considerations

A few notes are provided for readers unfamiliar with GPGPU programming to get up to speed for this application. Anyone with GPGPU experience may wish to skip to the next section.

The code provided here will be executed in parallel by many threads. Specifically, we may have 10 instruments at 2,000 modes per instrument, allocate one thread each, and run 20,000 threads in parallel. In practice, the GPU will run this work in batches[2], but we expect thousands of threads to be active at any time, versus our CPU which has four physical cores. Each of the GPU threads is less general-purpose than a CPU thread. There is currently no out-of-order execution or branch prediction, and arithmetic APIs are less rich, though still very capable for many applications.

Bundles of 32 GPU threads are called a *warp*, execute in lockstep, and have some memory shared between them for fast communication. It is possible to send data across warps, but if we have subtasks that can be executed with 32 or fewer threads (even in multiple steps), it might be worth trying to keep communicating threads inside the same warp.

Memory is allocated on the device (GPU) using a special API call `cudaMalloc` that looks similar to `malloc`. It returns a *device pointer*; GPU code can read and write from this location but CPU code cannot. Sharing data between the CPU and GPU generally involves copying it to and from, though some API calls exist to simplify this or even hide latency. Our host code uses the original `cudaMalloc` for simplicity and greatest compatibility.

GPU programming is often an interesting puzzle of how to best utilize the resources of the device. For example, we have a very small number of extremely fast registers, a few kilobytes per thread of fast memory, and access to slow, but plentiful RAM (11GB on our card). Different cards may execute different numbers of single- and double-precision floating point arithmetic; in particular consumer cards tend not to have strong FP64 performance.

NVIDIA provides documentation for more details, and also provides development, debugging, and profiling tools for Eclipse or Visual Studio to help maximize performance.

3.4. Code Walk

First, we remember the GPU kernel code is launched, executes, and returns to the host, and will be rescheduled again for the next audio buffer where it is launched and returns. It may or may not execute on the same *streaming multiprocessor* as it did the previous iteration. Perhaps some cards may zero memory to prevent data leaks. And for high numbers of modes we can be sure that this assumption can't hold, because we have more threads than resources available. We can't depend on our state to be kept for us.

Our first task, therefore, is to decide how to save and restore the state of the system across executions. Recall that from the description of these filters in Section 2 that the filter's state is limited to the prior complex output, so our state is a mere 64 bits per filter: two 32-bit single-precision floats to make up a complex number. We store these sequentially in global memory,
$\Re_0\Im_0\Re_1\Im_1...\Re_{N-1}\Im_{N-1}$.

The first few lines instruct each thread to determine which mode $m_i \in [0..N]$ it is responsible for computing. `blockIdx` and similar are local variables provided by CUDA so each thread may orient itself in the world in terms of the parallel problem the developer has written.

The next lines load state in from global memory. If we had no prior state this will be garbage, so there is a `bool reset` flag that allows for easily reseting the filter state to zero; useful for this case but also useful if we are for example switching between presets in a plugin.

Note that accessing global memory tends to be slow; in the digital waveguide acceleration portion of [17] we found we would be limited in applications if we had to access main memory to read and write each sample at audio rate speeds–but here we just need to load the state once.

Next, all the input for this mode m_i is available as the ith `ModeInfo` structure. We use this data to perhaps update the inner exponential term for the filter. Versions of the code where γ_m or α_m could be modulated on a per-sample basis would require moving this computation inside the loop.

Next, we loop to compute our audio data sample-by-sample. The complex math is provided by the CUDA library, with the exception of `cexpf` which needed to be written; it's provided in Listing 2, or in the GitLab repository.

```
__device__ __forceinline__
cuFloatComplex custom_cexpf(
    cuFloatComplex z) {
  cuComplex res;
  float t = expf(z.x);
  sincosf(z.y, &res.y, &res.x);
  res.x *= t;
```

[1] https://cm-gitlab.stanford.edu/travissk/dafx19-gpu-kernel

[2] Our request to execute 20,000 threads exceeds the 3,584 CUDA cores supported on our GPU. In such cases, the GPU will schedule warps on streaming multiprocessors as they become available, so our work is actually run in smaller parallel batches rather than *fully* parallel. Furthermore, this code will bottleneck on the availability of floating-point units and threads will compete for that resource as well. However, from the calling code's perspective the kernel executes as one unit of work.

```
  res.y *= t;
  return res;
}
```

Listing 2: cexpf implementation

We sum the newly-computed sample across all threads in a warp and write this to our output buffer; some applications may also wish to sum across warps especially if using hundreds of thousands of modes.

Our computation is done; there's nothing to clean up but we do need to write our new system state back to device memory so it can be read by the next kernel execution. Then, the function returns, our host code stops waiting, it can sum audio data generated by each warp, copy it back to shared memory, and notify the plugin in the DAW process that computation has finished.

Next, we consider how we may use all these high-Q filters.

4. EXTENDING MODAL SYNTHESIS

4.1. Baseline: Implementing Linear Models

Using the GPU filterbank, linear modal synthesis is straightforward.

First, we compute modal frequencies and decay rates offline and on the CPU. An offline analysis script trimmed sounds to several seconds, then computed DFTs of a second or more, and used these to compute modal coefficients and decay rates. This step is not time-sensitive, but did complete faster than real-time. This analysis step may also be performed at runtime, for example importing a user's samples, substituting an approximation algorithm if speed is of the essence.

The modal coefficients are stored in a file on disk. A directory with recordings captured from over a dozen cymbals is loaded into memory of the JUCE plugin on launch. The user may assign different cymbals to different MIDI notes; this is done by copying mode data from the appropriate file into `ModeInfo` structures in shared memory; the data files are small and memory-mapped so this operation is fast.

The plugin's process method is called for each audio buffer. If it detects MIDI notes that are assigned instruments, we feed an excitation signal based on the input velocity into the instruments' input signal shared memory buffer[3].

Then, the GPU processes enough samples to fill our audio buffer. Sample data is copied from the GPU's onboard memory to the RAM inside the GPU process, to shared RAM, which the plugin will read and copy back out to audio buffers.

Qualitatively, this approach works very well for bar percussion, cowbells, etc. where we have clear, exponentially decaying modes. With enough modes allocated, cymbal tails sound somewhat realistic. While the cymbal attack is instantaneous, and clearly lacks the interesting "bloom" as we move from linear to nonlinear regimes, a high density of modes bring about qualitative time-varying behavior from beat frequencies and adjacent frequencies having different decay rates. Noting the attack is limited by a linear modal synthesis model, we explore a couple ways to introduce nonlinear approximations.

[3]as in the GPU code description, we could alternatively just send a note-played signal and leave the GPU to handle articulations

4.2. Multi-sampling

Sample libraries commonly include multiple velocities for a given sound to capture tonal variation when an instrument is played at varying intensities. Taking inspiration from this, we capture several recordings for cymbals struck at different velocities.

The simplest use for the recordings at V velocity levels is to compute all the modes as before for each level, and set our filterbank to accept the union of all modes in all files:
$[m_{0,v_0}..m_{N-1,v_0}; ...; m_{0,v_{V-1}}..m_{N-1,v_{V-1}}]$
Then, as the player strikes the virtual cymbal we excite one set of V modes (or an interpolation). A few variations exist:

- Per-note velocity lookup: The control application sends a signal such as note-on velocity, which we use to pick the mode data closest to that velocity level. This data is used for the every sample of that strike contributing to $x()$.
- Per-sample velocity lookup: Similar to the above, but we continuously vary the mode parameters excited on a sample-by-sample basis: the early part of an excitation contributes from low-velocity V, and the maximum amplitude portion contributes from higher-velocity V captures (thus, the γ in $\gamma_m x(t)$ depends on $x(t)$ rather than being static).
- System-energy lookup: the amount of energy in the system governs which response we use regardless of immediate input level. This may be the most physically-accurate option.

The output of a simulation using the per-sample velocity switching approach is shown in Figure 4; the model in this case is driven by an increasing noise signal which introduces modes sampled from higher regimes over time. Of course, we are still synthesizing using a linear filterbank and not truly modeling interactions between modes at this stage, but this modification results in a more playable instrument for little overhead, so long as we stay with the maximum N allowed by our system.

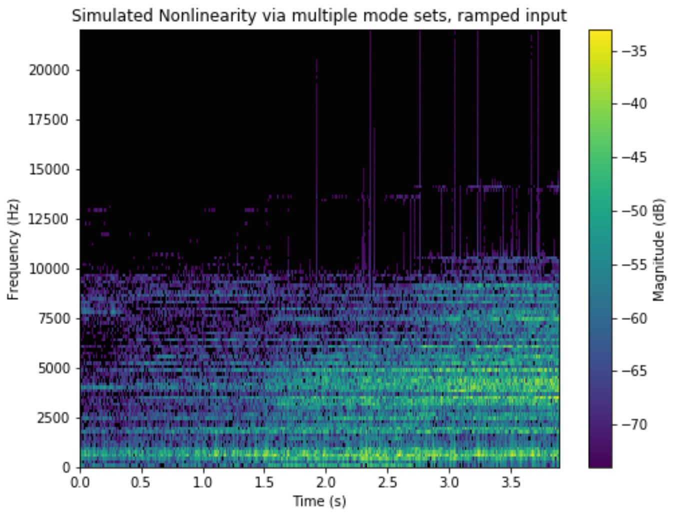

Figure 4: *Response using switched velocity samples and ramped input.*

This is similar to cross-fading samples, with the improvement that re-strikes will re-excite the system versus playing a duplicate sample. However, we immediately recognize it is an inaccurate representation of the physical system: we would have modes excited that do not exist simultaneously in practice. As cymbals vibrate and bend, some modes will come in and out of existence. Experimentally, we saw that depending on the cymbal, roughly 60%-80% of modes were shared between any set of 3-5 files.

However, qualitatively this is still an improvement, and we get more accurate responses playing at low versus high velocities on a drum pad.

It is likely even better results could be obtained by continuously exciting the cymbal, by driving it mechanically or even with continuous strikes with mallets at a certain velocity. This allows for capturing the response to a narrow range of energy; if we take the DFT of a several-seconds long recording of a high-velocity cymbal strike, we will incorporate low-, medium-, and high-energy regimes as the cymbal sound blooms and decays.

4.2.1. *Multi-sampling for Performance Characteristics*

One final note on this approach - while this paper focuses on crash cymbals, a variation of the multisampling-based approach seems to work well for modeling ride cymbals: we capture a few velocity hits for each of bow, bell, and edge strikes, and use those to excite certain modes in one shared modal system. Modifying playing position between the bell and bow allows energy to build up and brings out a sound that is difficult to capture with pure sampling.

4.3. Frequency Shifts

An extension to this method is to capture the frequency shifts by pairing modes across velocity levels that are at nearby frequencies: $\omega_{m,v1}$ and $\omega_{m',v2}$ under some threshold frequency shift. At performance time, we not only choose which modes are excited, but are able to change their frequencies as well.

As an offline computation we have many options as far as clustering and nearest-neighbor algorithms; if we ever needed to perform it real-time, a tracking algorithm such as that included in PARSHL [21] or linear programming [22] are worth considering, if they cope with spectrally dense content.

4.4. Implementing Modal Coupling

Finally, we introduce modal coupling to approximate a phenomena in the nonlinear regimes: identify the set of frequencies that only appear in high-velocity captures, and couple each one of these to one or more modes that is present at lower velocities by $\psi_{m,n}$, representing a coupling from mode m to n. Modes may feed multiple other modes, for example both $\psi_{m,n1}$ and $\psi_{m,n2}$ may exist to establish a one-to-many relationship. At performance time, with some system energy envelope, we establish an energy transfer between modes $\propto k\psi_{m,n}$. This is one area where the high-Q filters we've chosen excel; we simply immediately scale down the previous complex output value y for one mode, and inject that energy into another mode (this can be a new term adjacent to the input response term $\gamma_m x(t)$). When the update is performed at a zero-crossing this preserves phase; qualitatively in terms of sound, waiting for a zero-crossing does not seem critical here.

In practice, the prior sections on multi-sampling and frequency gating may capture some of this behavior. However, this approach has the opportunity to introduce real-time performance control over how much coupling is present in the system – k in the above expression. It may easily be suppressed, exaggerated, made frequency-dependent, etc.

While this evolves our synthesis beyond a simple modal filterbank excitation, it is still related to the linear modal model, and is not a true physical simulation of the chaotic regime of cymbals. Nevertheless it is a computationally efficient approximation that starts to bridge the gap between our simple modal responses and a real-time playable cymbal synthesizer.

GPU programming details have been absent from previous sections, as the choice of modes is up to the plugin running in the DAW process and we can treat the GPU as a black box. It is relevant here, however, as modes directly influence each other and must communicate, and the GPU kernel would need to perform the trading of energy between modes.

The fact that threads run in groups of 32 has some relevance. If we can keep groups of modes that couple in clusters of 32, then programming becomes much easier since threads inside a warp have access to a block of shared memory that may be used for computation. It is not a showstopper if we instead must communicate using global memory, just less optimal.

We tried a few sub-approaches as to how to choose modes: modes could pair with the closest (in frequency) unused higher-regime mode, modes could look for modes close to an octave up, fifths, etc., or we could simply pair modes randomly. Qualitatively, these resulted in similar effects, with more difference when using smaller N.

A sound example for modal coupling is in Figure 5. We drive a virtual cymbal mechanically with constant noise, and k is set fairly high so modes are introduced quickly once crossing a manually-specified energy threshold between regions.

As a comparison with the real cymbal spectrograms, we see modes emerging over time, an improvement over our first step of pure linear modal synthesis, and have model controls over when and how quickly the modes marked as high-energy-regime emerge, which adds an interesting performance dimension. Work remains, however, to automatically model the attack for multiple stick hit velocities; our examples here all involved some manual iteration.

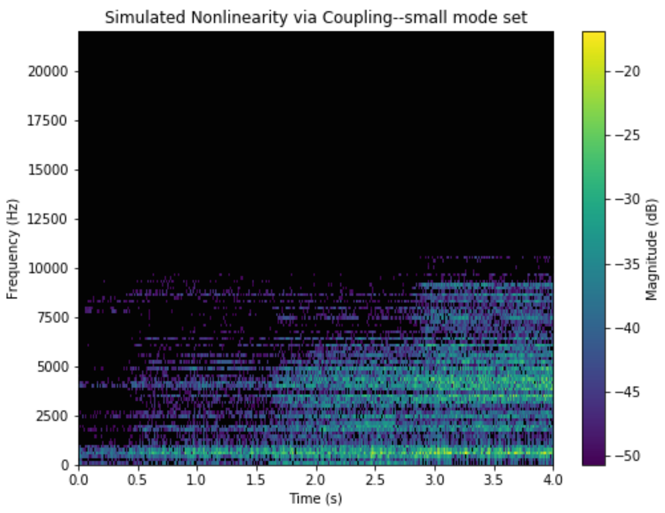

Figure 5: *Response using simulated modal coupling; coupled sources/targets only*

```
__global__ void filterbankKernel(
    float* yprev,         // Previous state
    const ModeInfo* mi,   // Mode frequencies and dampings.
    const float* input,   // Input signal.
    float* output) {      // Buffer we write output samples to.
  // Initialization - each thread figures out its
  // place in the world (i.e., it's the ith of N threads).
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  int which_warp = (int)(i / 32);
  bool is_first_thread_in_warp = (i % 32) == 0;

  // Load prior state from global memory.
  cuComplex y;
  y.x = yprev[2 * i];
  y.y = yprev[2 * i + 1];
  // If prior state should be discarded, zero it.
  if (mi[i].reset) {
      y.x = 0.0f;
      y.y = 0.0f;
  }

  // Introduce variables to make expressions more readable later.
  cuComplex input_amp;
  input_amp.x = mi[i].amp_real;
  input_amp.y = mi[i].amp_real;
  cuComplex input_complex;
  cuComplex exp_term;
  int which_drum = (int)(i / MODES_PER_DRUM);
  const float *input_base = input + (BUFFERSIZE*which_drum);

  // Regenerate the exponential term at the start of each buffer.
  // This is moved inside the loop in case of parameter interpolation,
  // modifying params on zero-crossings, or similar cases.
  cuComplex e_term_tmp;
  e_term_tmp.x = -mi[i].damp;
  e_term_tmp.y = mi[i].freq;
  exp_term = custom_cexpf(e_term_tmp);

  // Main loop - run enough cycles to generate the whole buffer.
  for (int samp = 0; samp < BUFFERSIZE; samp++) {
    y = cuCmulf(exp_term, y);
    // We always compute input for now, but could skip this multiply
    // if we know there's no input.
    input_complex.x = input_base[samp];
    input_complex.y = 0.0f;
    y = cuCaddf(y, cuCmulf(input_complex, input_amp));

    // Merge audio across all threads in the warp (32 threads),
    // and store the sample in our output buffer.
    // This sums in parallel, and completes in log_2(32) = 5 steps.
    // We use the real part of the complex sample as output audio.
    float merge_output = y.x;
    for (int offset = 16; offset > 0; offset /= 2)
        merge_output += __shfl_down_sync(0xffffffff, merge_output, offset);
    if (is_first_thread_in_warp) {
        output[which_warp*BUFFERSIZE + samp] = merge_output;
    }
  }
  // Save state back to shared memory, so we may restore it
  // on the next kernel launch.
  yprev[2 * i] = y.x;
  yprev[2 * i + 1] = y.y;
}
```

Listing 3: *CUDA Kernel code for basic modal filterbank. A code walk is available in Section 3.4.*

5. ACKNOWLEDGMENTS

Thanks to the conference organizers, and to the anonymous reviewers for suggestions, comments, and time.

6. CONCLUSIONS

This work presented a GPU modal filterbank system architecture, with a walkthrough of CUDA code that implements the GPU kernel. This enabled acceleration of our modal synthesis building blocks; we were resource-constrained with a single cymbal on the CPU but can run an entire ensemble of cymbals in real-time on the GPU with room to spare. Next, a few extensions to basic linear modal synthesis were presented toward bringing modal cymbal synthesis closer to real models which are highly nonlinear. These extensions, while approximations still based on a linear synthesis system, improve playing dynamics and allow introduction of new performance controls.

7. REFERENCES

[1] Antoine Chaigne, Cyril Touzé, and Olivier Thomas, "Nonlinear vibrations and chaos in gongs and cymbals," *Acoustical science and technology*, vol. 26, no. 5, pp. 403–409, 2005.

[2] KA Legge and NH Fletcher, "Nonlinearity, chaos, and the sound of shallow gongs," *The Journal of the Acoustical Society of America*, vol. 86, no. 6, pp. 2439–2443, 1989.

[3] Stefan Bilbao, "The changing picture of nonlinearity in musical instruments: Modeling and simulation," in *Proc. Int. Symp. Musical Acoustics*, 2014.

[4] M Ducceschi and Cyril Touzé, "Modal approach for nonlinear vibrations of damped impacted plates: Application to sound synthesis of gongs and cymbals," *Journal of Sound and Vibration*, vol. 344, pp. 313–331, 2015.

[5] Michele Ducceschi and Cyril Touzé, "Simulations of nonlinear plate dynamics: an accurate and efficient modal algorithm," in *18th International Conference on Digital Audio Effects (DAFx-15)*, 2015.

[6] Quoc Bao Nguyen and Cyril Touzé, "Nonlinear vibrations of thin plates with variable thickness: Application to sound synthesis of cymbals," *The Journal of the Acoustical Society of America*, vol. 145, no. 2, pp. 977–988, 2019.

[7] Lauri Savioja, Vesa Välimäki, and Julius O Smith, "Audio signal processing using graphics processing units," *Journal of the Audio Engineering Society*, vol. 59, no. 1/2, pp. 3–19, 2011.

[8] Lauri Savioja, Vesa Valimaki, and Julius O. Smith III, "Real-time additive synthesis with one million sinusoids using a gpu," *128th Audio Engineering Society Convention 2010*, vol. 1, 5 2010.

[9] Fernando Trebien and Manuel Oliveira, "Realistic real-time sound re-synthesis and processing for interactive virtual worlds," *The Visual Computer*, vol. 25, pp. 469–477, 5 2009.

[10] Gabriel Cirio, Ante Qu, George Drettakis, Eitan Grinspun, and Changxi Zheng, "Multi-scale simulation of nonlinear thin-shell sound with wave turbulence," *ACM Transactions on Graphics (TOG)*, vol. 37, no. 4, pp. 110, 2018.

[11] Jeffrey N Chadwick, Steven S An, and Doug L James, "Harmonic shells: a practical nonlinear sound model for near-rigid thin shells.," *ACM Trans. Graph.*, vol. 28, no. 5, pp. 119–1, 2009.

[12] Jose Belloch, Balazs Bank, Lauri Savioja, Alberto Gonzalez, and Vesa Valimaki, "Multi-channel iir filtering of audio signals using a gpu," in *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, 05 2014, pp. 6692–6696.

[13] Belloch Rodríguez, *Performance Improvement of Multichannel Audio by Graphics Processing Units*, Ph.D. thesis, 2014.

[14] Jose A Belloch, Alberto Gonzalez, Enrique S Quintana-Orti, Miguel Ferrer, and Vesa Välimäki, "Gpu-based dynamic wave field synthesis using fractional delay filters and room compensation," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 25, no. 2, pp. 435–447, 2017.

[15] Stefan Bilbao and Craig J Webb, "Physical modeling of timpani drums in 3d on gpgpus," *Journal of the Audio Engineering Society*, vol. 61, no. 10, pp. 737–748, 2013.

[16] Nal Kalchbrenner, Erich Elsen, Karen Simonyan, Seb Noury, Norman Casagrande, Edward Lockhart, Florian Stimberg, Aaron van den Oord, Sander Dieleman, and Koray Kavukcuoglu, "Efficient neural audio synthesis," *arXiv preprint arXiv:1802.08435*, 2018.

[17] Travis Skare and Jonathan Abel, "Gpu-accelerated modal processors and digital waveguides," in *Linux Audio Conference 2019 (LAC-19)*.

[18] Max Mathews and Julius O. Smith III, "Methods for synthesizing very high q parametrically well behaved two pole filters," in *Proceedings of the Stockholm Musical Acoustics Conference (SMAC 2003)(Stockholm), Royal Swedish Academy of Music (August 2003)*, 2003.

[19] Jonathan S. Abel, Sean Coffin, and Kyle Spratt, "A modal architecture for artificial reverberation with application to room acoustics modeling," in *Audio Engineering Society Convention 137*, Oct 2014.

[20] NVIDIA Corporation, "NVIDIA CUDA toolkit documentation," `https://docs.nvidia.com/cuda/`, [Online].

[21] Julius O Smith and Xavier Serra, "Parshl: An analysis/synthesis program for non-harmonic sounds based on a sinusoidal representation," in *Proceedings of the 1987 International Computer Music Conference, ICMC; 1987 Aug 23-26; Champaign/Urbana, Illinois.[Michigan]: Michigan Publishing; 1987. p. 290-7.* International Computer Music Conference, 1987.

[22] Julian Neri and Philippe Depalle, "Fast partial tracking of audio with real-time capability through linear programming," in *21st International Conference on Digital Audio Effects (DAFx-18)*, 2018.