Audio Engineering Society

Convention Express Paper 42

Presented at the 153rd Convention
2022 October

GPU-Accelerated Drum Kit Synthesis Plugin Design

Travis Skare[1]

[1]*CCRMA, Stanford University*

Correspondence should be addressed to Travis Skare (`travissk@ccrma.stanford.edu`)

ABSTRACT

We present a real-time, GPU-accelerated drum set model: the application itself, a description of the synthesis involved, and a discussion of GPGPU development strategies and challenges. Real-time controls enabled by these synthesis methods are a focus. The project is and will remain noncommercial.

1 Introduction

Rich, physically-accurate drum set component models are proposed by e.g. the NESS project (overview[1] and with specific models such as snare[2]); these and other FDTD models are high-quality but may be computationally prohibitive running many drums in real-time as part of a larger audio production. In contrast, the plugin discussed herein utilizes "spare" resources on a GPU and less expensive "building blocks" to synthesize drum set sounds, but may be seen as more of an analysis-resynthesis system with real-time controls different from those expected from a more common sample-based virtual instrument.

The plugin is developed with the JUCE toolkit; UI may be seen in Figure 1.

The resources page for this paper, including sound examples, and code pointers may be found at: https://ccrma.stanford.edu/~travissk/aes2022media/

2 Methods

2.1 Synthesis

Cymbals and shells are implemented using two strategies: modal synthesis and digital waveguides/meshes.

Modal cymbal synthesis runs up to 2,000 modes (dynamic) per cymbal. Implementation is via phasor filters which allow simple, stable per-sample modulation of parameters. Such filters, suggested by Mathews and Smith [3] and for modal reverberators by Abel et al. [4], consist of a recursive complex update equation:

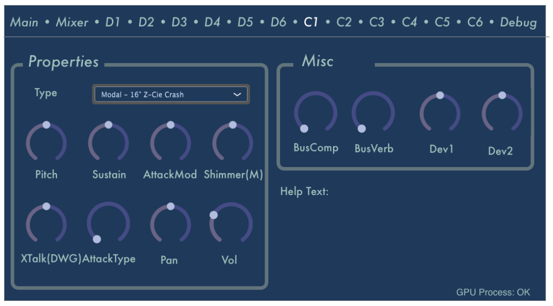
$$y_m(t) = \gamma_m x(t) + e^{(j\omega_m - \alpha_m)} y_m(t-1) \qquad (1)$$

where:

$x()$ is an input or excitation signal,

ω_m is mode frequency for mode index m,

γ_m is a per-mode complex input amplitude gain, and


Type: Modal – 16" Z-Cie Crash

GPU Process: OK

Fig. 1: Plugin's Simple UI, viewing a details page for a cymbal. The user may choose a precomputed modal or waveguide network cymbal model and adjust real-time model and effect parameters. A styling refresh is in progress.

α_m is a per-mode dampening factor.

Mode responses for three or more velocities of 10 cymbals are captured and analyzed offline. At runtime, relevant sets of modes are excited. It is possible to simply map MIDI velocity to a mode set or a mix of multiple sets; a current and more interesting approach feeds velocity to a re-excitable energy envelope to choose a mode set. This was found to be an efficient method to simulate cymbal rolls, as the user may keep a cymbal in a high-energy regime with several lower-velocity strokes. These calculations are performed on the CPU.

Number of modes may be limited to 50 per critical band to reduce cost. Shell-based drums may also be implemented as such, with far fewer modes; three velocities are. In this case, a preprocessing step to "round" modes close to each other.

Novel performance controls are introduced: Artifact-free pitch shifting and time-stretching or even freezing (extending a cymbal wash indefinitely) are computationally "free:" we adjust ω_m or α_m from Equation 1 across all modes, or a subset.

A *shimmer* control is essentially multi-band tremolo and seeks to simulate movement and modulation of cymbal crash sounds. As with the the other effects, the modal filter bank makes this computationally trivial, with one more multiply of the amplitude gains α. In contrast with a traditional tremolo effect operating on the entire signal, we may ramp modulation of α_m frequency and/or phase across bands.

A damping or tilt control adjusts frequency-dependent gain on a per-instrument basis.

A shortcoming of especially linear modal synthesis is the rich, inharmonic attack of a crash symbal. Multi-velocity captures and simple linear analysis-resynthesis provides some improvement, but towards, a separate set of high-velocity modes is optionally faded in with high . This is currently behind the development control in Figure 1.

A second approach utilizes a digital waveguide network consisting of three (shells), to ten or more (cymbals) digital waveguides[5], here acting as rich oscillators or "modal compression," a way of efficiently synthesizing and controlling several harmonics in one block. As with the modal parameters, these are mapped to a sound recording ahead of time.

For harmonically-rich sources such as cymbals, an approximate approach was developed to place these waveguide oscillators: Obtain a set of sound sources for a cymbal at multiple velocity levels. Then, while the remaining signal energy is above some level:

1. Compute the top modes, only considering frequencies below some f_{th} (e.g. 1KHz) as a baseline[1].
2. Place a digital waveguide oscillator at the most prominent mode. Our baseline digital waveguide includes three variations with stronger allpasses for frequency smearing, and the closest one is mapped. There is certainly more rigor that could be introduced into this process.
3. Synthesize and subtract from the residual. We operate on half-critical-band sized granularity.
4. If residual above f_{th} remains, it could be synthesized with modes or an alternative approach.

An approach training a Machine Learning network to this task was explored but did not yield generalizable results; we suspect a more comprehensive input data set compared to the mode sets captured and augmented with synthetic data might fit the task better.

[1] discarding very high modes altogether might even work for cymbals, if upper modes and damping are to be introduced and controlled with a performance control

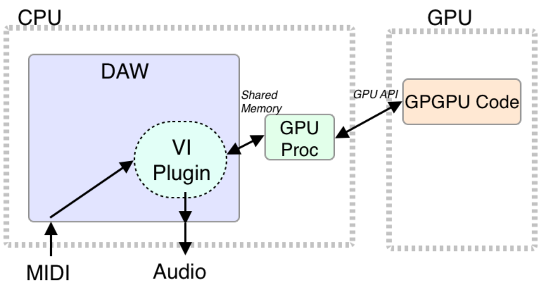


Fig. 2: System architecture: a DAW plugin, on audio callback request, sends control data to a GPU process, which launches and waits for kernels to run on the GPU.

2.2 Mixing and Effects

We may optionally post-process the drums on the GPU. There is a tradeoff here: as we would like to avoid multiple transfers to and from the GPU.

GPU-side insert effects on each drum may be added after the synthesis step, reusing registers and thread-local memory. Two examples: A simple textbook digital compressor may be implemented with only a handful of registers. Convolution reverb was an early GPU-accelerated effect, with academic[6], open-source, and commercial applications.

As an example, simple two-pole lowpass and highpass filters are provided to reduce mud or ring.

Two bus compressors are available to mix into GPU-side. These are “textbook” implementations rather than specific emulations of specific hardware, but have configurable threshold, attack and release.

3 Discussion

3.1 Implementation Considerations

The system comprises two pieces (Figure2). The user interacts with a DAW plugin or standalone app component, which accepts MIDI input and outputs multichannel audio, as most virtual instruments do.

The plugin/app communicates to the second component, a background synthesis process which itself handles communication to the GPU. This is an evolution of the approach from our earlier cymbal-only work [7] and the GPU process may be adapted to serve requests for applications such as network audio for a multichannel installation.

Communication between the two is over shared memory and semaphores, all using low-level OS primitives for compatibility and to avoid adding a dependency on an additional library.

Implementation is via the CUDA GPGPU toolkit and was developed on the Windows platform; the GPU process and plugin code have Linux shared memory support as well.

An optimization to this system is to execute kernels a buffer ahead of time, requesting on a thread outside the audio processing callback threads. This would remove the synchronous wait, but could introduce a buffer of latency to respond to input.

3.2 Mixing and Effects

After each set of GPU threads processes a buffer, the state of each synthesis element (modal filter, digital waveguide, mesh) must be written to persistent global GPU memory. At the beginning of the next block it is read from that memory.

Audio state for each output audio channel is written to a separate area of global memory, and copied back to the CPU host, explicitly or implicitly depending on the GPU API used.

As the data is available in global memory, it is straightforward to add a bus processing step that operates on this data and appends additional audio channels. Because this waits for data, it introduces additional sequential execution time. Figure 3 shows a timeline of dependencies for a single buffer processing request. Note that the synthesis and effects kernels may comprise more than one processing batch, but the GPU scheduler handles this transparently.

4 Results

Please see the sound examples, or the convention demo for practical results. A spectrogram of a synthesized cymbal can be seen in Figure 4.

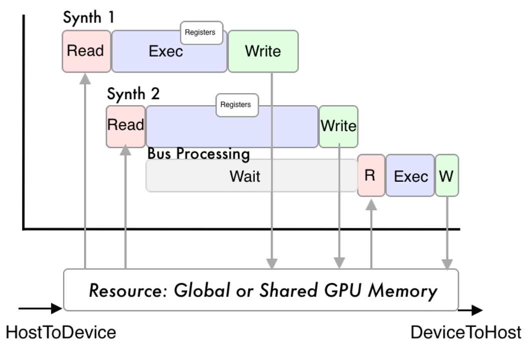


Fig. 3: Processing timeline for two parallel synthesis kernels and one sequential effects bus processing kernel

4.1 Performance

Our test system is consumer-level, containing an Intel i5 3570K CPU and NVIDIA GeForce GTX 1080Ti GPU.

The synthesis process runs at 44.1KHz and during development and demos, DAW or JUCE buffer sizes of either 512 (5.8ms) or 256 were used. 128 seemed qualitatively practical, however in debug mode we noted occasional missed audio callback deadlines. At 256+, 100k modes, 10k waveguides, and dozens of small meshes may be synthesized in parallel with either concurrent or sequential kernels; sequential kernels started to run against real-time considerations.

Synthesis kernel benchmarks were run on a system-on-chip developer kit successfully as a proof of concept,

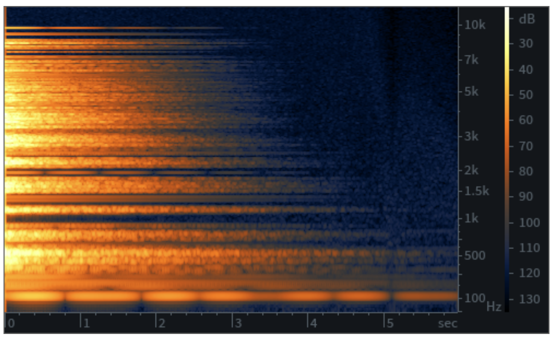


Fig. 4: Spectrogram of modally-synthesized cymbal, 1000 phasor filters.

though the entire app has not been ported.

For specific performance figures: micro-benchmarking and single-instrument benchmarking has been covered in our prior work, but the introduction of a second synthesis kernel raises the need to benchmark the real synthesis kernels.

Table 1 measures mean and maximum kernel execution for the two (modal/DWG) kernels running sequentially and in parallel at buffer sizes of 256 and 1024, which must execute in at 5.6ms and 23.2ms at 44.1KHz respectively. We see they execute with room to spare, though system overhead is not accounted for and we are dependent on the DAW for scheduling.

Approach	Mean	Max
Sequential (N_{samp}=256)	1.34	2.10
Parallel (N_{samp}=256)	1.15	1.76
Sequential (N_{samp}=1024)	5.26	8.21
Parallel (N_{samp}=1024)	4.45	8.49

Table 1: Sequential vs. Streamed synthesis kernels, buffer processing time, milliseconds.

5 Future Work

Most notably, we would wish to benchmark audio GPGPU latency on modern systems with unified CPU/GPU memory. This includes embedded system-on-chip and certain newer consumer PCs/laptops with custom silicon.

For the waveguide synthesizer, 2D Digital Waveguide mesh-based elements were explored but are not used in the current version of the plugin. In terms of GPU acceleration, we found hundreds of rectangular meshes of edge dimension 10-20 may be synthesized in parallel at audio rates, but it was not trivial to grow them in size across GPU execution blocks; FDTD methods and research by the NESS project and followup work cover acceleration strategies for grid methods for investigation.

Finally, the waveguide fitting method mentioned here could use additional rigor, or be replaced with either a closed-form matrix solve, or a Machine Learning-based approach; while initial experiments were unsuccessful here, we believe introducing more real data or revisiting our synthetic data approach would have promise as this should map well to the domain of neural nets.

6 Summary

A real-time drum set synthesis plugin was presented which performs core synthesis on a GPU using two separate synthesis kernels, and may perform optional GPU-side bus post-processing effects. Discussions of development and performance considerations that drove most of the project were included.

References

[1] Bilbao, S., Desvages, C., Ducceschi, M., Hamilton, B., Harrison-Harsley, R., Torin, A., and Webb, C., "Physical modeling, algorithms, and sound synthesis: The NESS project," *Computer Music Journal*, 43(2-3), pp. 15–30, 2019.

[2] Bilbao, S., "Time domain simulation and sound synthesis for the snare drum," *The Journal of the Acoustical Society of America*, 131(1), pp. 914–925, 2012.

[3] Mathews, M. and Smith, J. O., "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.

[4] Abel, J. S., Coffin, S., and Spratt, K. S., "A modal architecture for artificial reverberation," *The Journal of the Acoustical Society of America*, 134(5), pp. 4220–4220, 2013.

[5] Smith, J. O., "Digital Waveguide Modeling of Musical Instruments," 2003.

[6] Cowan, B. and Kapralos, B., "Spatial sound for video games and virtual environments utilizing real-time GPU-based convolution," in *Proceedings of the 2008 Conference on Future Play: Research, Play, Share*, pp. 166–172, 2008.

[7] Skare, T. and Abel, J., "Real-time modal synthesis of crash cymbals with nonlinear approximations, using a gpu," in *Proc. 22nd Int. Conf. Dig. Audio Effects*, 2019.