The MMMAudio Computer Music Environment

Sam Pluta
Peabody Institute of the Johns Hopkins University
spluta1@jhu.edu

Ted Moore
Peabody Institute of the Johns Hopkins University
tmoore97@jhu.edu

ABSTRACT

We introduce MMMAudio, a new audio creative coding environment designed to close the gap between instrument building and low-level DSP development while reducing the maintenance burden typical of monolithic, compiled systems. Contemporary computer music languages such as Max, Pure Data, and SuperCollider excel at graph-based instrument design but impose steep barriers when custom DSP is required, pushing users into C/C++ plugin workflows with unfamiliar APIs, build systems, and cross-platform complexities. MMMAudio addresses these issues by centering its programming model on Mojo for high-performance DSP and seamless Python–Mojo interoperability for tooling, AI, and scientific libraries. In MMMAudio, unit generators (UGens) are simple Mojo structs, enabling users to write, test, and distribute new UGens without leaving their code editor or contending with external build pipe-lines. This design simultaneously encourages new DSP creation, leverages Python's mature ecosystem for machine learning and data processing, and exploits Mojo's performance features (e.g., SIMD) for fast, real-time audio processing. We present the system's architecture, programming model, and extension mechanisms.

1. INTRODUCTION

A new software system, called MMMAudio, is in the early stages of development. Its design foregrounds three strengths that we feel set it apart from existing Computer Music Environments (CMEs): 1) It unifies instrument building and DSP authoring under the same workflow and language (Modular's Mojo), letting creators prototype innovative DSP directly within the same environment where they are building instruments. 2) It leverages the existing, well supported programming languages, Mojo and Python, as well as their infrastructure, taking advantage of the many packages in the Python ecosystem. This will allow the dev-team to focus on core functionality and DSP, instead of having to maintain multiple layers of tooling: IDE environments, GUI packages, i/o libraries, etc. 3) It embraces industry-leading and foundation-supported AI and analysis tools, connecting to Python and Mojo's mature ML ecosystems rather than requiring teams of developers create entirely new systems for AI training and inference[1, 2].

MMMAudio achieves this by leveraging Mojo-Python interoperability, creating a Python-based control environment with fast DSP written in Mojo. This allows us to fully engage the robust Python ecosystem for everything from GUI to serial communications, use any IDE (from VS Code to vim) for development, and use industry-leading AI tools like PyTorch and sci-kit learn for AI engagement, all-the-while running custom, fast DSP in Mojo. The current state of MMMAudio can be found at:
https://github.com/spluta/MMMAudio.

2. WHY MOJO

2.1 If it Ain't Broke, Don't Fix It (History)

The leading CMEs all have a similar structure. They split operations between a "slow" but facile scripting language for data manipulation/control structures and a fast, compiled language for audio manipulation and complex math. In his 2002 paper, *Max at Seventeen*[3], Miller Puckette attributes the influence of Max's design to Max Matthews, whose "RTSKED [programming environment] attacked the problem of real-time scheduling of control operations for a polyphonic synthesizer" using "sporadically occurring events that caused state changes in the synthesizer."
"RTSKED modeled the performance as a collection of tasks running in parallel." "This separation of the real-time control problem into separate tasks was a key advance necessary for the computer to act as a musical instrument."

The "Max design," with parallel, "sporadic" operations first controlled audio via MIDI on a completely separate machine, the IRCAM 4X[3]. In 1991, Puckette maintained this split when he added tilde$\sim$ audio processing objects, and the system became Max/MSP. "Max provides a separate data type for audio signals which does not mix smoothly with messages; conversion from audio to messages is problematic, and the scheduler treats control and audio-rate processing separately [3]." Thus, 1991 marked the release of the first Max version that ran a parallelized control system alongside a fixed sample-rate audio graph—all on the same machine.

James McCartney's SuperCollider[4] language originally grew out of a similar split. McCartney had written "Synth-O-Matic which was a non-real-time C-like synthesis programming language" and "a MAX object called Pyrite which contained the interpreter for the language which was extended and used in SuperCollider. Writing SuperCollider involved integrating the language interpreter, garbage collector and function library of Pyrite with the synthesis engine and functions of Synth-O-Matic[5]." While SuperCollider 1 and 2 felt as though the scripting language and the synthesis engine were part of the same basic language, "SuperCollider Server" (often referred to as SC3), the current version in use, more fully separates their functions, go-

ing so far as to make them separate applications: SCLang and SCServer[4].

Like SuperCollider, Max, and Pure Data, MMMAudio follows a control-language/audio-engine split, using Python as the control environment and Mojo for the audio engine. Because Mojo syntax is designed to be Pythonic [6], MMMAudio has a similar "feel" as when using "Max" (control data) and "MSP" (audio$\sim$ processes) in the same Max patch. Jumping back and forth between writing code for signal processing and controlling systems is smooth. Furthermore, Python is widely considered to be one of the easiest programming languages to learn, which we believe will be echoed with Mojo as more users learn it. This lowers the threshold-for-entry and puts writing low-level DSP in Mojo just one step beyond Python programming, making MMMAudio an ideal environment for beginners and experts alike.

2.2 The Problem Mojo is Designed to Solve

> "we see Mojo as being an incredible part of the Python ecosystem. We're not looking to break Python or change it, or, quote, unquote, 'fix it.' We love Python for what it is. Our view is that Python is just not done yet... it just doesn't have those features that you would use to do C-like programming. And so if you say, okay, well, I'm forced out of Python into C for certain use cases, well, then what we're doing is we're saying, 'Okay, well, why is that? Can we just add those features that are missing from Python back up to Mojo?' And then you can have everything that's great about Python, all the things that you're talking about that you love plus not be forced out of it when you do something a little bit more computationally intense, or weird, or hardware-y" [6]

In the quote above, Chris Lattner, the author of Mojo, describes the impetus behind creating the Mojo Programming Language[7]. He describes the issue he is solving as this: AI developers train their models in Python using high-level tools such as PyTorch and TensorFlow. These tools provide industry-leading, easy-to-use frameworks for training deep learning models. However, while these frameworks are great for training models, once models are trained, running inference on them at scale requires a different approach. Python is no longer the right tool because it is a scripting language that is not designed for speed. Lattner points out later in the same interview that those most versed in training AI models aren't usually as skilled (or even interested) in coding in performant languages like C++ or Rust. Thus, a whole second team of developers is needed to translate the trainings from Python to compiled languages that can run the trainings at high speed.

While Lattner is specifically discussing issues around AI inference, he might as well be describing the current state of creative coding for audio, where a similar issue exists. Languages like SuperCollider, Max, and Pure Data are outstanding environments for instrument building and interface design, but they (SuperCollider and Max at least) fall short when it comes to DSP implementation.

2.3 C and C++ Based Systems

According to Puckette, Max and Pure Data (and we include SuperCollider as well) provide "a way of combining pre-designed building blocks into configurations useful for real-time computer music performance. This includes a protocol for scheduling control and audio-rate computations, an approach to modularization and component intercommunication[3]." In these systems, the user constructs control graphs and DSP graphs using pre-existing unit generators. These UGens are pre-compiled and each system has a convenient way of plugging the generators together.

However, an issue arises when the user needs to do something that is not possible with existing UGens. When low-level DSP design is desired, the user has to leave their creative coding environment of choice to work in a completely different environment: one with a different set of tools and a different skill-set necessary for development.

Plugins for CMEs are usually written in C or C++ or some combination thereof, and the learning curve goes beyond just having to learn a new language. In order to make a plugin, a musician-turned-developer has to learn:

1. a whole new language: C or C++.
2. the API for the CME
3. compilation tooling such as CMake, Make, etc.
4. differences between Windows and Unix compilers, etc.
5. usually a whole new code editor and development environment
6. distribution strategies, such as GitHub Actions, if you want users to download a build for their machines

The potholes around CMake are large enough, but the sum total of barriers to plugin development for the average creative coder are profound. Even Cycling74's excellent gen [8] environment, while looking like Max, is different enough from Max to create real barriers to entry.

2.4 Enter MMMAudio and Mojo

MMMAudio avoids these problems. MMMAudio code is written in Mojo, and MMMAudio plugins are written in Mojo. If you know MMMAudio, you know Mojo and if you know Mojo, you can make a plugin. Importantly, the user doesn't need to leave the development environment or even the current file to do so. A UGen in MMMAudio is simply a Mojo struct with a suitable "next" function ready to give the next sample when called upon. Further, the entire source code for the MMMAudio project is available inside the programmer's IDE, so starting a new plugin is as simple as finding a plugin resembling what is needed, copying it, and editing the code for new functionality.

3. EXTENDING MMMAUDIO

3.1 Creating New UGens

All CMEs discussed in this paper share a common design. They all allow instrument designers to construct complex structures out of simpler structures using *composition*. SuperCollider users can create Pseudo-UGens out of UGens

and then construct SynthDefs from UGens and Pseudo-UGens [9]. Max and pd users can construct subpatchers from objects and other subpatchers. And gen patchers can contain other gen patchers.

MMMAudio's approach to composition takes inspiration from the Faust Programming Language's[10], in that environment's DSP library is built out of the DSP library itself. Complex single-sample unit generators are built out of simpler single-sample UGens, all of which are written in Mojo. For example, the code below represents a comb filter written in Faust:

```
fb_comb(maxdel, N, b0, aN) = (+ <: delay(maxdel,
↪  N-1),_) ~ *(-aN) : !,*(b0) : mem;
```

Listing 1: Faust's feedback comb filter.

This function builds a comb filter by placing Faust's delay object (which is declared elsewhere in the code-base) directly in its feedback path. There are three important points: 1) the complex UGen is composed of simpler UGens, all written in the same language. 2) A coder can use this language to create single-sample feedback UGens where other UGens can be placed in the feedback path. 3) This new UGen of UGens can be placed inside of other UGens, or infinitely many, that can also operate at single-sample delay. 4) The result is a compiled UGen, not a patch or SynthDef, that is no different from any other UGen.

In the MMMAudio code below (both more verbose and more readable than the Faust example above), we similarly insert a delay into the feedback loop of the Comb UGen. This kind of feedback loop could be constructed in Max (with tapin/tapout) or SuperCollider (with LocalIn/LocalOut), but in both cases the minimum length of the delay time would be the block size, limiting its use.

```
struct Comb[num_chans: Int = 1, interp: Int =
↪  2](Movable, Copyable):
    var world: World
    var delay: Delay[Self.num_chans, Self.interp]
    var fb: MFloat[Self.num_chans]

    fn __init__(out self, world: World, max_delay:
    ↪  Float64 = 1.0):
        self.world = world
        self.delay = Delay[Self.num_chans,
        ↪  Self.interp](self.world, max_delay)
        self.fb = MFloat[self.num_chans](0.0)

    fn next(mut self, input: MFloat[self.num_chans],
    ↪  delay_time: MFloat[self.num_chans] = 0.0,
    ↪  feedback: MFloat[self.num_chans] = 0.0) ->
    ↪  MFloat[self.num_chans]:
        fb_in = input + self.fb * clip(feedback,
        ↪  -1.0, 1.0)
        var out = self.delay.next(fb_in, delay_time)
        self.fb = out
        return out
```

Listing 2: MMMAudio comb filter in Mojo.

Because MMMAudio and Faust (and Pure Data in certain subpatchers) operate sample-by-sample, filters, analog modeling, sample-accurate granulation, feedback systems, and more, are expressible, where their creation in SuperCollider and Max is stunted due to block-size limitations.

3.2 A Library of Composed UGens

Using MMMAudio, larger UGens can be constructed by composition, just like SuperCollider SynthDefs or Max subpatchers, but in MMMAudio these composed UGens are compiled into their own objects, rather than functioning as a linked-together lists of pre-compiled objects. A clear example of MMMAudio's composition style is its PitchShift UGen, which is comprised of many familiar building blocks: a Recorder object with embedded SIMDBuffer, a Dust (to facilitate triggering Grains with time dispersion), and multiple Grain UGens. Each Grain contains a buffer player called Play. Playback of the Recorder's SIMDBuffer inside the Play UGen is controlled by a Impulse phasor, which also drives the Envelope. A PolyTrigger UGen controls polyphonic voice allocation, growing the list of Grains when more voices are needed.

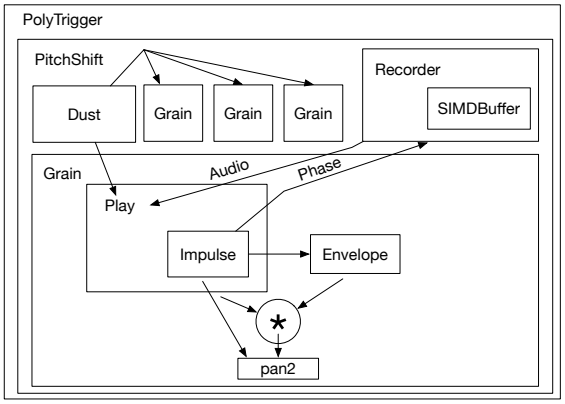

Figure 1: Composition-based construction of the Pitchshift UGen

4. JOINING THE PYTHON ECOSYSTEM

4.1 Mojo-Python Interop

Maybe the biggest advantage of MMMAudio is that it is situated in the Python ecosystem. Mojo, MMMAudio's DSP language, is designed for Python interoperability[11], and Mojo's PythonModuleBuilder allows building MMMAudio graphs directly in Python. When one runs a MMMAudio graph from Python, Mojo's PythonAudioBuilder automatically determines whether the graph has changed since last compiled, and if necessary, compiles a new executable before starting the audio engine. The MMMAudio Graph then runs as a Python external directly in the Python environment where it was compiled, ready to receive control messages from its Python host. If the graph has not changed since its last compilation, it runs instantly without needing to compile.

4.2 Many Many Packages

Being inside the Python ecosystem allows us to take full advantage of the many packages necessary for building a creative coding environment. We use mido[12] for MIDI, hidapi[13] for hidapi bindings, python-osc[14] for Open

Sound Control, matplotlib[15] for plotting, pyautogui[16] for the mouse location, PySide6 (Qt)[17] for gui, etc. We use Python to parse the messages received from these various packages, then send modified messages to MMMAudio's audio engine using the MMMAudio messaging system: a simple key-value system able to send floats, ints, triggers, lists, text (for loading files like audio files and json files), etc, back and forth between Python and Mojo. Beyond control data, we use pyaudio[18] bindings for Portaudio[19] and have access to Numpy[20] and SciPy[21] for data manipulation.

Having these tools available for use was priceless in developing MMMAudio quickly and efficiently. Since these tools are available and are maintained by the Python community, the MMMAudio development team does not have to create and maintain our own bindings (though we do plan to bind portaudio and Mojo in the future). Instead, we look to the open-source community to make these available. Importantly, many of these packages have major foundation/commercial support (Python Foundation, KDE Free Qt Foundation, and Modular), which means they will be open-source and available for years to come.

4.3 Python is the Place for AI and Mojo was Designed for AI

By joining the Python ecosystem we gain access to the most powerful and widely used tools in AI: PyTorch[22] and TensorFlow [23] for deep-learning and scikit-learn[24] for training smaller models, analysis, classification, decomposition, etc. This means that *any neural network that can be run in Python, using industry-leading AI tools, from simple multi-layer perceptrons to transformers, can run in MMMAudio*. Training happens in Python, but inference can be made either on the Python side, or, since Mojo can run Python code internally through its Python interop, directly in the audio-engine. Further, Mojo is designed for GPU use and highly parallelized processes (Modular's MAX inference engine is written in Mojo[25]). This will streamline development of further AI inference and allow us to run neural networks and FFT processes parallelized on the GPU when useful.

4.3.1 Example 1: PyTorch Inside Mojo for Low-level Synthesis Control

Multi-Layer Perceptrons are commonly used in synthesis systems for mapping low-dimensional input vectors to higher dimensional control vectors. These, in turn, manipulate the parameters of complex synthesizers [26]. The following example shows where MMMAudio can excel at this task, giving the user high-dimensional control of low-level parameters.

In MMMAudio, neural network training happens in Python, using standardized AI tools like PyTorch. We can then run inference on our trained network in Python, but because Python code can run directly in Mojo, the ideal approach is to place the PyTorch MLP code directly inside our Mojo synth. In this example, our synth is a modified Freeverb algorithm with added delayed feedback and crossfeedback, sending each channel of its stereo outputs back into both its inputs. For each of the two channels there are 8 lowpass comb filters in parallel followed by four allpass filters in series. These filters expose all of their parameters to the multi-layer perceptron. The MLP controls 70 parameters in total: delay time, feedback, and lowpass frequency for each of the 16 Comb filters, delay time and feedback coefficient for each of the 8 allpass filter pairs, feedback and crossfeedback coefficients for overall feedback of the system, and delay time for each channel of the delay on the feedback. The entire structure is controlled by a 3 dimensional input: 1 slider and 1 xy pad.

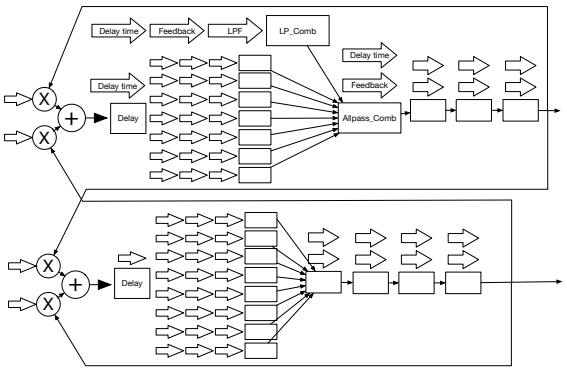

Figure 2: MLP Control of a Single Sample Feedback Network. Each ARROW represents an output from the MLP.

While this structure is technically possible in any CME (or combination of CMEs), our ability to make accessible every parameter of every UGen at single sample feedback, combined with the capacity to embed industry-leading inference engines inside any UGen, makes this approach straight-forward, idiomatic, and accurate in MMMAudio.

4.3.2 Example 2: Audio Descriptors and Python-based Data Processing

As made popular by CataRT[27], a common practice when approaching data science for musical creativity includes analyzing and plotting audio slices for inspection, validation, or performance. Mubu[2] and FluCoMa[1] added machine learning to this approach. Because it is situated in Python, MMMAudio easily replicates and extends these patterns of use by drawing on extensive existing tools such as scikit-learn[24] and matplotlib[15].

The following example uses MMMAudio's Python-based non-real-time slicing and analysis tools as described in Section 5.5. A collection of test audio files (that come with the FluCoMa Toolkit[1]) is sliced using a spectral-flux-based onset detection algorithm. Each slice is analyzed for 14 MFCC coefficients. The non-real-time analysis for each slice takes place in Mojo and is returned to Python as a Numpy[20] array with shape `( Num FFT Frames, 14 MFCCs )` and data type `float64`. Once in Python, the MFCC coefficients of each slice are summarized as the mean of the coefficient over all the FFT frames in the slice. The zeroth coefficient is removed to avoid representations of overall spectral energy (see Listing 3). With each slice now described by a vector of 13 coefficients, umap-learn[28] is used to reduce the dimensionality of the dataset to two dimensions for plotting in matplotlib[15] (see Figure 3). A callback function is used to send the mouse position to a fit KDTree (from scikit-learn[24]), find the slice nearest to the mouse, and send the necessary data for that slice to Mojo for playback.

```
    d = {
        "path": "resources/src.wav",
        "thresh":0.01,
        "min_slice_len":0.1,
        "window_size":1024,
        "hop_size":512,
        "num_coeffs": 14
    }
    y, sr = librosa.load(d["path"], sr=None)
    slices = MBufAnalysis.spectral_flux_onsets(d)
    data = np.ndarray((len(slices)-1, 13))
    for i in range(len(slices)-1):
        start = int(slices[i])
        end = int(slices[i+1])
        d["start_frame"] = start
        d["num_frames"] = end - start
        mfccs = MBufAnalysis.mfcc(d)
        data[i] = mfccs[:, 1:].mean(axis=0)
```

Listing 3: Excerpt of MMMAudio MFCC extraction in Python.

While other CMEs are also able to create this functionality, the strength of MMMAudio is its access to the plethora of data science and machine learning tools in Python. Many more are accessible in Python than what has been ported to existing CMEs. As new machine learning methods and tools become available, they are almost certain to appear in Python, at which point MMMAudio users can immediately experiment with their potential for musical applications.

4.4 IDE Toolchains

It cannot be overstated how important sophisticated IDE toolchains are in modern coding systems. Because MMMAudio uses industry supported languages, we have access to Pylance (or other Python language server) and the Mojo Language Server, which provide essential code intelligence features like autocomplete, error detection, go-to-definition, and symbol search. In addition, both Python and Mojo can be run in Jupyter Notebooks [29], adding an expressive tool for prototyping and live-coding alike.

5. FURTHER SPECIFICS

5.1 Built-in SIMD Processing

Mojo is a language written primarily for GPU-based AI inference. Outside the Fourier Transform and Neural Network inference, vanilla audio DSP has few applications for parallel processing. However, our codebase leverages Mojo's inherent SIMD types, "enabling you to write code that automatically leverages modern CPU vector capabilities while maintaining code clarity and portability[30]." This enables parallel processing in filters, distortions, FFT channels, and algorithmic generators, essentially giving us the second channel of a stereo effect at little or no extra CPU cost.

Data types like SIMDBuffer and Delay store audio in sequential lists of SIMD vectors instead of interleaved arrays. This optimizes reading and writing audio data, since 1 copy or write is needed where 2 would be necessary when using an interleaved buffer. Stereo UGens can output 2D SIMD vectors, and when these vectors get sent down

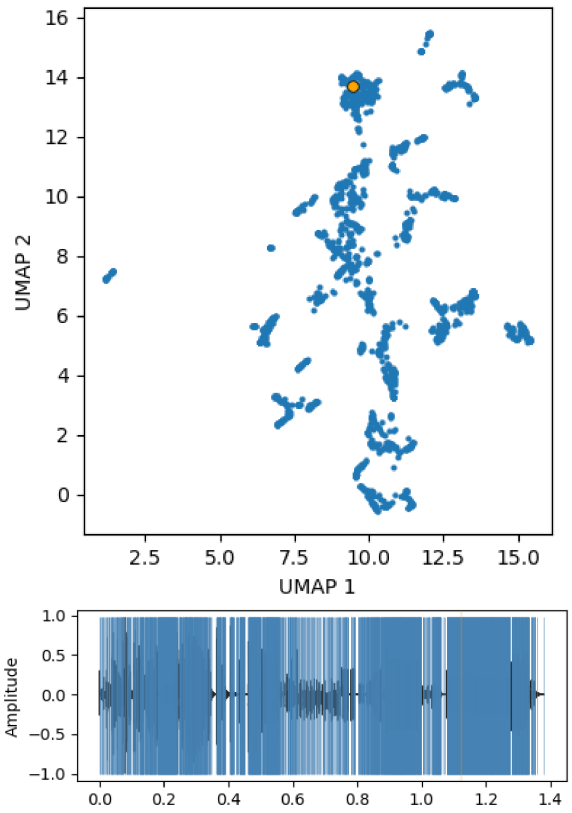

Figure 3: UMAP dimensionality reduction plot of MFCC analyses and corresponding waveform slice visualizer.

the processing chain, MMMAudio UGens written to receive any-dimensional SIMD vectors will process the audio in parallel, maintaining the dimensionality of an input vector as its output.

```
struct OnePole[N: Int = 1](Movable, Copyable):
    var last_samp: MFloat[N]

    fn __init__(out self):
        self.last_samp = MFloat[N](0.0)

    fn next(mut self, input: MFloat[N], coef:
    ↪  MFloat[N]) -> MFloat[N]:
        coef2 = clip(coef, -0.999999, 0.999999)
        var output = (1 - abs(coef2)) * input +
        coef2 * self.last_samp
        self.last_samp = output
        return output
```

Listing 4: a simple one-pole filter, able to process a Float64 or N dimensional SIMD Float64 vector.

This complexity is largely hidden from the user. Because Mojo's Float64 type is a one dimensional SIMD Vector, the user does not need to fully understand SIMD to reap its rewards. They can start learning MMMAudio by thinking in floats. [1]

[1] SIMD vectors do have limitations; in Mojo, they must be sized at a power of 2, and the user should not make SIMD vectors larger than two times the processor's vector size, or the code will be slow[30]. Arm processors like the Apple M1 can process two 64 bit floats in parallel, while Intel chips can process up to eight in parallel[31].

Since stereo is a common audio format, we have chosen SIMD as our default data type. MMMAudio can process larger signals using Mojo's various collections, like List.

5.2 Interpolation

At the core of any audio engine are great sounding oscillators. One of the basic DSP operations MMMAudio provides is a single, robust, wavetable-oscillator struct, `Osc`, with various approaches to interpolation (linear, cubic, quadratic, lagrange, sinc) and oversampling for alias-free high-quality audio synthesis. Any wavetable, from the default base waveforms to commercial 3D morphing wavetables, can be read by `Osc`, and it can interpolate between any number of wavetables, resulting in 3D wavetable lookup. Frequency and phase modulation and phase synchronization are possible as well. This provides a solid foundation for exploration of alias-free timbrel synthesis.

5.3 Oversampling

Additionally, MMMAudio has a built-in Oversampling struct that can be folded into any Unit Generator or synth graph. This struct holds an N-SIMDsample oversampling vector (corresponding to N times oversampling), a function to add SIMD samples to the buffer, and a function to low pass filter the buffer and return a single SIMD vector of output. Mojo's parameter arguments allow the user to bypass the oversampling code completely (at compile-time) or parameterize and inline the oversampling process. This means that any UGen designed as a "standard" 1X process can easily be adapted to run as an oversampled UGen, while maintaining efficiency when the oversampling is bypassed.

5.4 FFT Processing

MMMAudio includes an intuitive and powerful system for implementing custom FFT-based processing and analysis algorithms. The user creates a custom struct (coded in Mojo), with a `next_frame` function to which two lists of `Float64` values are passed: (1) the magnitudes and (2) phases of the most recently processed FFT window. These values can be used to compute an analysis or can be transformed in-place. The output of the custom struct is then passed to an IFFT and eventually returned as a stream of audio samples reflecting the user-implemented spectral processing.

To be used, the custom struct is simply passed to MMMAudio's existing `FFTProcess` struct, which deals with all the necessary plumbing for a Fast Fourier Transform. It buffers the most recently received `window_size` of input audio samples and performs an FFT every `hop_size`. The FFT results are passed to the user-defined struct's `next_frame` function, after which an IFFT can be performed. The audio samples output by this process are buffered similarly to the input so they can stream in-to and out-of `FFTProcess` sample-by-sample like any other UGen. Latency is the necessary `window_size` samples. A user can specify any of the common window shapes (hann, hamming, blackman, rect, sine, kaiser, or triangle) for pre- and/or post-FFT windowing.

Similar to this FFT-based processing system, there is a system for implementing user-defined buffered processes

```mojo
struct FFTLowPass[window_size: Int =
↪  1024](FFTProcessable):
    var lpbin: Int

    fn __init__(out self, world: World):
        self.lpbin = 20

    fn next_frame(mut self, mut magnitudes:
    ↪  List[Float64], mut phases: List[Float64]) ->
    ↪  None:
        for i in range(self.lpbin,(self.window_size
        ↪  // 2) + 1):
            magnitudes[i] = 0.0
```

Listing 5: FFT brick wall low pass filter in Mojo.

that do not require an FFT or that need access to the raw input samples (for example an RMS calculation or the YIN[32] pitch detection algorithm). MMMAudio's `BufferedProcess` struct similarly buffers input and output samples and passes the most recent `window_size` samples to a user's custom struct every `hop_size` samples.

5.5 Audio Analysis

Using the `FFTProcess` and `BufferedProcess` described in Section 5.4, MMMAudio offers, for real-time and non-real-time analyses, many of the common audio descriptors found in other toolkits[1, 2, 33, 34, 35]. These include RMS, pitch (using YIN[32]), spectral centroid, spectral spread, spectral skewness, spectral kurtosis, spectral rolloff, spectral flatness, spectral crest, spectral flux (including a thresholded onset detection), mel-bands, and MFCCs. These analyses have been validated by comparing against two of the common audio analysis toolkits (FluCoMa[1] and Librosa[33]) to ensure that MMMAudio's audio analysis results match.

In order to take full advantage of the many data science and machine learning tools found in Python, it is necessary to have audio-analysis not only on the Mojo side of MMMAudio but also in Python. While Python-based audio analysis packages already exist[33, 34], if audio analyses used in Python are later to be used alongside real-time analyses in Mojo, it is important for both analyses to derive from the *same code functions*. This is because the variety of toolkits contain differences in how certain analyses are computed and variations such as floating-point accuracy can introduce discrepancies that may contaminate data.

MMMAudio takes advantage of Mojo's Python interop[11] to offer non-real-time analyses in Python using the same Mojo code that performs MMMAudio's real-time audio analyses. This ensures that Python-based MMMAudio non-real-time audio analyses will match analyses in the real-time context. To maintain full precision and smoothly integrate with Numpy-based tools, MMMAudio returns the Mojo-based analyses as Numpy data structures with data type `float64`, which can then be used with powerful toolkits such as scikit-learn[24] and SciPy[21].

5.6 Efficiency and Parallelism

A trade-off for MMMAudio's flexible, single-sample, 64-bit design is that it is less performant than SuperCollider on a single CPU. While there is no way to make a fully

legitimate comparison (MMMAudio, Max, and pd are 64 bit internally while SC is 32 bit, etc), on our M2 mac running with a 128 sample block size, MMMAudio was able to play 6000 simultaneous sine wave oscillators vs SuperCollider's 18000. Max was at just under 5000 using mc.cycle and Pure Data around 12000 using the clone object. In all cases, that is a lot of sine waves. MMMAudio's secret weapon is that, as SuperCollider can run multiple servers alongside one sclang instance, by the magic of Python's multiprocessing library, MMMAudio can run multiple audio engines from inside a single Python instance. This allows our M2 to run over 45000 oscillators at one time, all controlled by a single "control-rate" Python instance running in its own process. And getting multiple engines up and running is as simple as creating each as its own variable:

```
many_mmms = []
for i in range(4):
    many_mmms.append(MMMAudio(128,
    ↪  graph_name="ParallelGraphs",
    ↪  package_name="examples"))
    many_mmms[-1].start_process()
    many_mmms[-1].start_audio()
```

Listing 6: creating 4 MMMAudio instances with a `for` loop.

5.7 No Busses

Some CMEs introduce an extra layer of abstraction for splitting, combining, and routing audio to different parts of a program (such as SuperCollider's busses). Because MMMAudio processes audio one sample at a time and because Mojo gives parent structs access to all child member variables as deep into the data structure as desired, MMMAudio has no concept of busses. If data needs to be accessed, it can either be passed into a struct as an argument or accessed directly, not requiring a user to converse with additional data structures. This simplifies reasoning about signal flow and program design, while natively enabling sample-accurate audio routing and mixing over disparate parts of a program.

5.8 Documentation

Through MMMAudio, we aim to foster a community that welcomes many kinds of music making. To support this, we are developing comprehensive documentation and learning resources that are approachable for beginners from a range of backgrounds, while still offering the depth needed to take full advantage of the system. And our library contains detailed and great-sounding examples that demonstrate how different processes function.

Because MMMAudio enables creative decisions at every level—from low-level DSP to instrument design and performance—we expect users to extend it in their own idiosyncratic directions, rather than being funneled into the pre-existing workflows or aesthetics of the authors.

6. CONCLUSIONS

> "Is a specialized computer music language even necessary? In theory at least, I think not."
>
> -James McCartney [4]

We believe that MMMAudio shows an alternative to specialized computer music languages. It leverages Mojo / Python interop to provide a framework for expressive audio coding on par with current CMEs. By joining the Python ecosystem, it is able to take advantage of the vast library of Python packages that provide much of the functionality of the environment.

We came to create MMMAudio because we were exploring the use of neural networks to control highly dimensional synthesis and feedback systems[26]. This led us to build instruments with novel approaches to single sample feedback and DSP. In order to do that, we left our chosen CME to develop new DSP in C++ and Faust. We realized that we needed a system that could accelerate the fusion of machine learning, instrument building, and digital signal processing by making all of these practices happen in one place. MMMAudio is aimed to be this place, encompassing Faust's hard-core DSP world and the Max / SuperCollider / Pure Data instrument-building world, while at the same time allowing direct use of industry leading AI platforms. Using Mojo for both DSP and instrument design and Python for control and machine learning, we bring together these traditionally separate practices under the same development environment, knowledge base, and community.

Acknowledgments

Special thanks to Brad Garton, James McCartney, Miller Puckette, and Chris Lattner.

7. REFERENCES

[1] P. A. Tremblay, O. Green, G. Roma, J. Bradbury, T. Moore, J. Hart, and A. Harker, "The Fluid Corpus Manipulation Toolbox," Jul. 2022. [Online]. Available: https://doi.org/10.5281/zenodo.6834643

[2] N. Schnell, A. Röbel, D. Schwarz, G. Peeters, R. Borghesi *et al.*, "MuBu and friends–assembling tools for content based real-time interactive audio processing in Max/MSP," in *ICMC*, 2009.

[3] M. Puckette, "Max at Seventeen," *Computer Music Journal*, vol. 26, no. 4, pp. 31–43, 2002.

[4] J. McCartney, "Rethinking the Computer Music Language: SuperCollider," *Computer Music Journal*, vol. 26, pp. 61–68, 2002. [Online]. Available: https://api.semanticscholar.org/CorpusID:30952359

[5] ——, "SuperCollider, a New Real Time Synthesis Language," in *International Conference on Mathematics and Computing*, 1996. [Online]. Available: https://api.semanticscholar.org/CorpusID:5976007

[6] L. Fridman and C. Lattner. Chris Lattner: Future of Programming and AI | Lex Fridman Podcast. [Online]. Available: https://www.youtube.com/watch?v=pdJQ8iVTwj8

[7] Modular.com. Mojo: Powerful CPU+GPU Programming. [Online]. Available: https://www.modular.com/mojo

[8] G. Wakefield, "gen~," *unpublished*, 2011.

[9] SuperCollider Help. Writing Unit Generators. [Online]. Available: https://doc.sccode.org/Guides/WritingUGens.html

[10] D. F. Y. Orlarey and S. Letz, "Faust: An efficient functional approach to dsp programming," *unpublished*, 2009.

[11] Modular.com. Python Interoperability. [Online]. Available: https://docs.modular.com/mojo/manual/python/

[12] Mido Developers. mido. [Online]. Available: https://github.com/mido/mido

[13] P. Rusnak. hidapi. [Online]. Available: https://github.com/trezor/cython-hidapi

[14] T. Attwad. python-osc. [Online]. Available: https://github.com/attwad/python-osc

[15] J. D. Hunter, "Matplotlib: A 2D graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[16] A. Sweigart. pyautogui. [Online]. Available: https://github.com/asweigart/pyautogui

[17] Qt Group. PySide6. [Online]. Available: https://doc.qt.io/qtforpython-6/index.html

[18] H. Pham. PyAudio. [Online]. Available: https://people.csail.mit.edu/hubert/pyaudio/

[19] R. Bencina and P. Burk, "PortAudio - an Open Source Cross Platform Audio API," in *International Conference on Mathematics and Computing*, 2001. [Online]. Available: https://api.semanticscholar.org/CorpusID:32143419

[20] C. R. Harris, K. J. Millman, S. J. van der Walt, *et al.*, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2

[21] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, *et al.*, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[22] A. Paszke, S. Gross, F. Massa, *et al.*, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-\deep-learning-library.pdf

[23] M. Abadi, A. Agarwal, P. Barham, *et al.*, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[24] F. Pedregosa, G. Varoquaux, A. Gramfort, *et al.*, "Scikit-learn: Machine learning in Python," *Journal of machine learning research*, vol. 12, no. Oct, pp. 2825–2830, 2011.

[25] Modular.com. Modular MAX. [Online]. Available: https://www.modular.com/max

[26] S. Pluta, "Multi-mapped Neural Networks for Control of High Dimensional Synthesis Systems," in *Proceedings of the 2nd Conference on AI Music Creativity*, 2021. [Online]. Available: https://aimc2021.iem.at/papers/

[27] D. Schwarz, G. Beller, B. Verbrugghe, and S. Britton, "Real-time corpus-based concatenative synthesis with catart," in *9th International Conference on Digital Audio Effects (DAFx)*, 2006, pp. 279–282.

[28] L. McInnes, J. Healy, and J. Melville, "Umap: Uniform manifold approximation and projection for dimension reduction," *arXiv preprint arXiv:1802.03426*, 2018.

[29] B. E. Granger and F. Pérez, "Jupyter: Thinking and Storytelling With Code and Data," *Computing in Science Engineering*, vol. 23, no. 2, pp. 7–14, 2021.

[30] Modular.com. SIMD. [Online]. Available: https://docs.modular.com/mojo/stdlib/builtin/simd/SIMD/

[31] Wikipedia. Single instruction, multiple data. [Online]. Available: https://en.wikipedia.org/wiki/Single_instruction,_multiple_data

[32] A. De Cheveigné and H. Kawahara, "YIN, a fundamental frequency estimator for speech and music," *The Journal of the Acoustical Society of America*, vol. 111, no. 4, pp. 1917–1930, 2002.

[33] B. McFee, C. Raffel, D. Liang, D. P. Ellis *et al.*, "librosa: Audio and music signal analysis in python," in *Proceedings of the 14th python in science conference*, vol. 8, 2015.

[34] D. Bogdanov, N. Wack, E. Gómez, S. Gulati *et al.*, "Essentia: An audio analysis library for music information retrieval," in *Britto A, Gouyon F, Dixon S, editors. 14th Conference of the International Society for Music Information Retrieval (ISMIR); 2013 Nov 4-8; Curitiba, Brazil.[place unknown]: ISMIR; 2013. p. 493-8.* International Society for Music Information Retrieval (ISMIR), 2013.

[35] G. Peeters *et al.*, "A large set of audio features for sound description (similarity and classification) in the CUIDADO project," *CUIDADO Ist Project Report*, vol. 54, no. 0, pp. 1–25, 2004.