SYNTHAX: A Fast Modular Synthesizer in JAX

Manuel Cherep* and Nikhil Singh*
Massachusetts Institute of Technology (MIT)
{mcherep,nsingh1}@mit.edu
*Equal contribution.

Abstract

Modern audio production relies heavily on realtime audio synthesis. However, accelerating audio synthesis far beyond realtime speeds has a significant role to play in advancing intelligent audio production techniques. Fast synthesis methods have been used to generate useful datasets, implement audio matching procedures for automatic sound design, and infer synthesis parameters for real-world sounds. In this paper, we present SYNTHAX, a fast virtual modular synthesizer written in JAX. At its peak, SYNTHAX generates audio over 80,000 times faster than realtime, and significantly faster than the state-of-the-art in accelerated sound synthesis. We present SYNTHAX as an open source[1] and easily extensible API to stimulate and support applications of fast sound synthesis at scale.

1 Introduction

Realtime sound synthesis is a cornerstone of modern audio production. It affords producers the ability to tweak sounds and hear them change; a loop of perception and action that results in diverse auditory creations to support music, film, and other media. Modern audio technologies increasingly employ techniques that benefit from *automatically* tweaking synthesizers, such as optimization and machine learning. In these scenarios, the ability to rapidly tweak sounds and compute with them at scale offers a vast space of opportunities for designing and developing powerful new audio technologies. As such, fast sound synthesis can be an essential tool. We define faster-than-realtime as generating more than one second of audio per second of processing time. In particular, we deal with cases where the processing is a lot faster than this (i.e. $>$1000x).

In this paper, we introduce SYNTHAX, a fast modular synthesizer written using the JAX [1] framework for accelerated and differentiable computing. By offering synthesis at speeds that peak at over 80,000$\times$ realtime, SYNTHAX provides a high-performance, flexible virtual modular synthesizer in the form of an expanding and easily extensible open source Python library. Additionally, we implement an API based on *torchsynth* [2], a recent high-performing synthesizer written in PyTorch, to allow for an easy substitution for end-users. Our results in this paper show considerable speedups over *torchsynth*, ranging up to just under 9$\times$ depending on the hardware configuration and batch size.

[1] https://github.com/PapayaResearch/synthax

2 Related Work

2.1 Programmatic Synthesis

One important element of SYNTHAX is allowing programmatic control of a synthesizer. Indeed, many software synthesizers are ultimately written to be controllable by other software, such as VST plugins by DAW automation. However, not many synthesizers are designed to be fully specifiable and controllable in code written by end-users. Some well-known options include *Surge XT*[2] and *torchsynth* [2]. The former is written as a plugin that offers an API, and the latter is written as a library for non-realtime synthesis. We implement ours following the latter example, which means that there is not a direct application of our method to realtime synthesis. However, since JAX [1] compiles code to XLA, it is likely possible to implement SYNTHAX in a realtime synthesizer plugin to have it bridge these two different approaches to programmatic synthesis.

2.2 *torchsynth*

Developed for audio synthesis, *torchsynth* [2] serves as a modular synthesizer that is capable of generating audio on a single GPU at $\geq$16200$\times$ faster than realtime. It consists of a variety of audio and control modules. The default synthesizer in *torchsynth* is *Voice*, which the authors used to generate a dataset containing a billion audio clips. As we detail later, we base our API and implementation on *torchsynth* as it provides an existing and familiar reference point. We also compare to *torchsynth* in our experiments studying the performance of SYNTHAX.

[2] https://surge-synthesizer.github.io/

3 System Design

The design of the API is inspired by the inherent modularity of hardware synthesizers. SYNTHAX leverages the power of JAX [1] to build on *torchsynth* [2], which is a state-of-the-art high-throughput synthesizer implemented in PyTorch to take advantage of its accelerated computational routines. Maintaining a similar API makes the transition for end-users seamless without any major rewriting or learning curve.

```python
import jax
from synthax.config import SynthConfig
from synthax.synth import ParametricSynth

# Generate PRNG key
config = SynthConfig(
    batch_size=16,
    sample_rate=44100,
    buffer_size_seconds=4.0
)
# Instantiate synthesizer
synth = ParametricSynth(
    config=config,
    sine=1,
    square_saw=1,
    fm_sine=1,
    fm_square_saw=0
)
# Initialize and run
key = jax.random.PRNGKey(42)
params = synth.init(key)
audio = jax.jit(synth.apply)(params)
```

Listing 1: Code snippet for generating audio with a *ParametricSynth*. This synthesizer supports a user-configured architecture, in contrast to the *Voice* synthesizer which encodes a fixed topology design (78 parameters). This allows control of the degrees of freedom available to manipulate the sound synthesis.

Each module serves a different function but can be connected together to create a synthesizer. SYNTHAX modules mimic their counterparts in analog and digital synthesizers, consisting of amplifiers, envelopes, filters, keyboards, low-frequency oscillators (LFOs), mixers, and voltage-controlled oscillators (VCOs). The output from these modules can represent audio signals or control voltages, depending on the module's intended function. Audio modules, such as VCOs, produce audio signals. Control modules, such as LFOs, produce "control voltages" that modulate the parameters of other modules. The keyboard outputs parameters that are used as input for other modules. All modules follow the Flax [3] module system known as Linen to organize the modules into independent components. Figure 1 shows the structure of the API, where a synthesizer consists of modules and a configuration.

In our implementation, we aim for allowing users flexibility in how they specify synthesizers. Modules with parameters can be initialized in a few different ways. If only initial values are given, they are expected to be in human-readable (i.e. unnormalized, e.g. frequency in Hz) range within the default ranges of the parameters. Alternatively, the modules also accept range objects, which specify only a range within which parameter values are initialized uniformly randomly. Finally, users can also provide the initial values and ranges together as an object. In all cases, the parameters store the values in the (normalized) interval [0, 1].

In addition to the differences between SYNTHAX and *torchsynth* that arise from JAX features such as easy and flexible vectorization, parallelization, and just-in-time (JIT) compilation, we introduce these additional features: a filter module, currently containing a simple low-pass filter that can be shaped by control modules; a parametric definition of a synthesizer to easily explore different synthesizer topologies; functions to write and load a synthesizer including its hyperparameters and parameters, in the human- and machine-readable YAML format. This also means that synth specifications can, in principle, be directly composed in YAML and loaded to a synth with a matching parameter architecture.

We adhere to JAX's explicit randomness handling in our design. JAX uses a pseudo-random number generator (PRNG), an algorithm that produces sequences of numbers that approximate true randomness given an initial key (i.e. value). Therefore, users need to provide such random keys to their synthesizers. Though this adds an extra consideration, it also ensures better reproducibility. Listing 1 shows how to define a configuration, instantiate a parametric synthesizer and, finally, synthesize audio.

JAX supports a wide variety of hardware and leverages powerful function transformations such as just-in-time compilation (JIT), auto-vectorization, and hardware parallelism. We can vectorize (jax.vmap) and parallelize (jax.pmap) in a single line of code. It also conforms to the *Single-Program, Multiple-Data* (SPMD) model, which means that the same computation for different input data runs in parallel on multiple devices. In order to maximize performance and throughput when using JAX, SYNTHAX renders audio in batches.

Extending SYNTHAX can be done seamlessly due to its modularity, since the API is designed to easily integrate other synthesizers or modules. SYNTHAX joins the JAX ecosystem and can be easily integrated with other well-known libraries such as Optax [4], evosax [5], EvoJAX [6], and QDax [7].

4 Results

4.1 Performance Evaluation

First, we characterized the speed and memory performance of SynthAX. We used *torchsynth* [2] as a strong baseline to

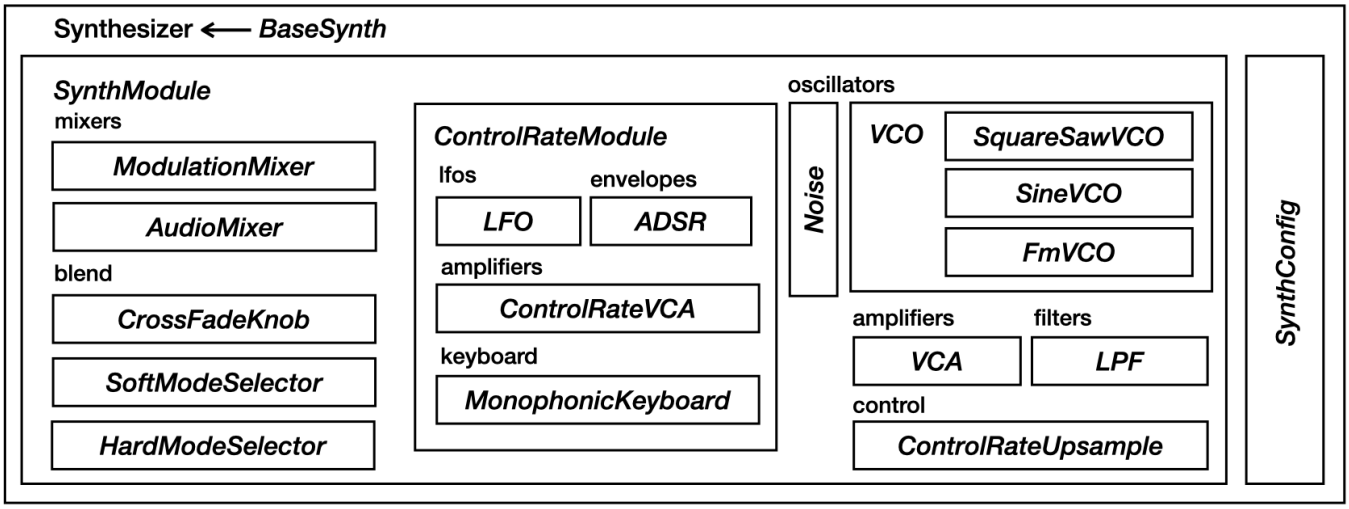

Figure 1: Structure of the API. We separate the synthesis modules into Python modules which group related elements. These modules are shown in lower-case letters above the relevant classes. The class inheritance structure, which mirrors *torchsynth* [2], is indicated by the *TitleCase* names. Inner boxes are subclasses of the larger boxes they are embedded in.

compare against, since it is the *de facto* state-of-the-art fast synthesizer and can take advantage of similar hardware acceleration capabilities (e.g. GPUs). For both synthesis libraries, we use the *Voice* synthesizer with 78 parameters. In our setup, we computed the time needed to synthesize 100 batches of sounds at different batch sizes (powers of 2 from 2 to 1024). We randomized the synthesis parameters for each batch. As *torchsynth* does, we also report the speed as compared with realtime synthesis. We calculated this as

$$\frac{\text{Num. Batches} \times \text{Batch Size} \times \text{Sound Duration}}{t}$$

where t denotes the time taken for one loop of 100 batches. Finally, we also report memory usage in GB after each 100-batch loop.

We report averages over 10 100-batch loops for all three quantities (time, speed $\times$ realtime, and memory), to account for variance. Additionally, we computed a full set of results for a GPU and a CPU, although we expect GPUs to be the primary usage platform. To account for the effect of JAX's [1] JIT compilation, we produced one batch of sounds (for both SYNTHAX and the *torchsynth* baseline) at the very beginning, outside the evaluation loop. This is so that we measure the typical performance, as the JIT compilation only needs to occur once.

These results are given in Figure 2. We do not show error bars as the results are generally stable, resulting in very small variance. Overall, we see that SYNTHAX substantially outperforms *torchsynth* on time-based metrics for both CPU and GPU. At peak performance within this evaluation, SYNTHAX shows more than 80,000$\times$ realtime synthesis speed. SYNTHAX shows a comparable memory utilization profile to *torchsynth*, especially lower at higher batch sizes on GPU and CPU. We disabled JAX's memory preallocation for our experiments to measure the real memory footprint.

For direct comparison, Figure 3 plots the speedup over *torchsynth*. This is computed as the ratio of time taken to synthesize 100 batches, computed per 100-batch loop, and then averaged across the 10 runs. We provide min/max error bars to show the full range. This figure shows that the speedups range from just over 2$\times$ (some batch sizes on CPU and very large batches on GPU) to almost 9$\times$ at the peak speedup level (batch of 32 sounds on GPU).

4.2 *torchsynth* Replication

We replicated the examples from *torchsynth* [2] for reproducibility. These include instantiating ADSR envelopes both randomly and with set parameters, VCOs, LFOs, VCAs, mixers, and their synthesizer architecture *Voice*. Figure 4 shows some of the resulting spectrograms considering different VCOs and setups in *torchsynth* and corresponding match in SYNTHAX.

5 Applications

5.1 Audio Representations

An advantage of synthesized sounds is that they also contain the associated synthesis parameters. In self-supervised representation learning problems, datasets that result from synthesis can be used to formulate parameter prediction problems. For instance, pitch recognition is a prominent auditory processing problem for which synthesized datasets hold significant promise. Recent work on audio representation learning has

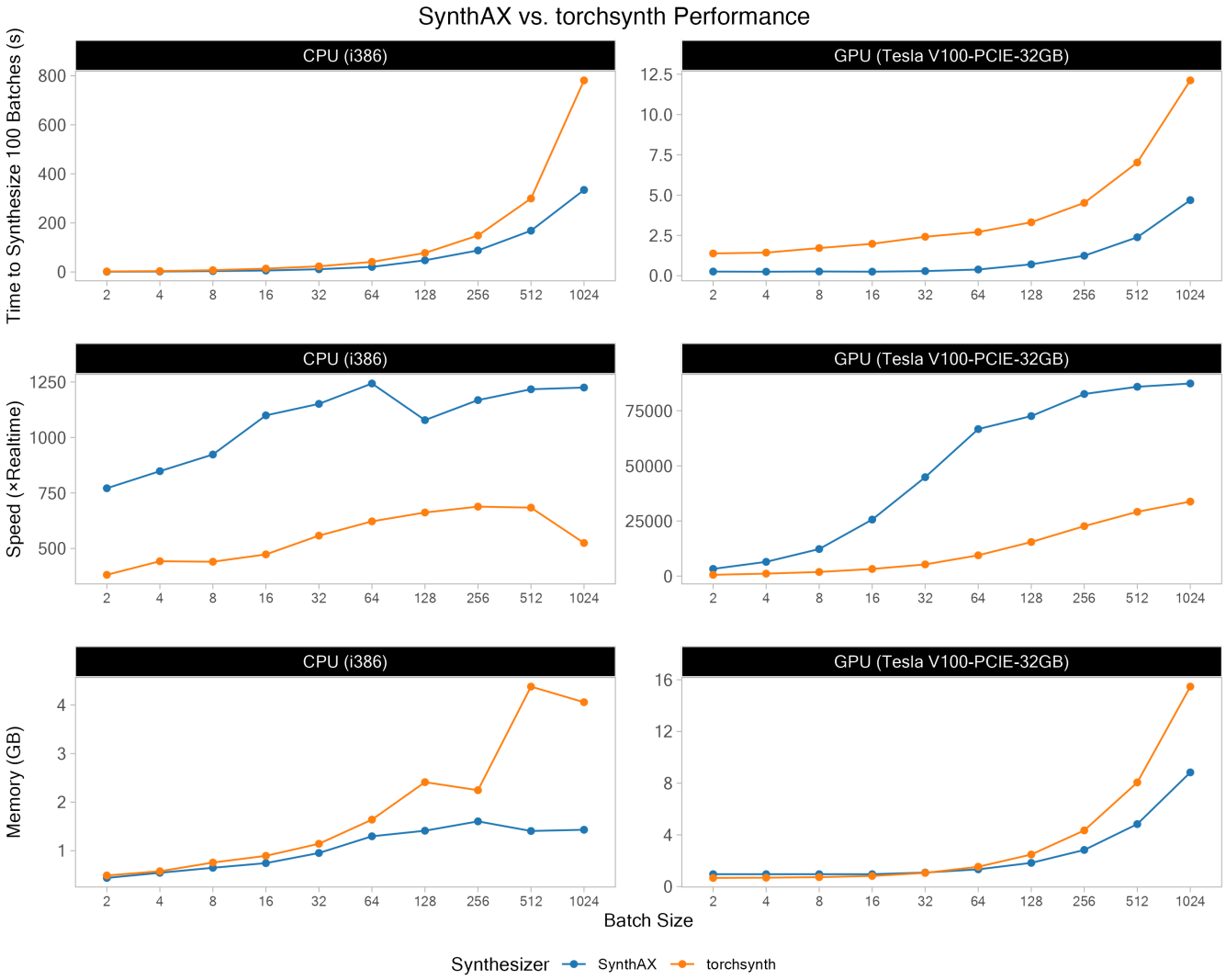

Figure 2: Results from performance evaluation, compared with *torchsynth*, on **(Left)** a 2017 iMac with an Intel Core i7-7700K CPU @ 4.20GHz, and **(Right)** an NVIDIA Tesla V100 GPU. Values shown are averaged over 10 runs. We use the *Voice* synthesizer in both SYNTHAX and *torchsynth*, randomizing parameters each batch. **(Top)** Time to synthesize 100 batches of sound at different batch sizes (given in seconds). **(Middle)** Time reinterpreted as speed × realtime, i.e. seconds of sound generated per second of computation time (see §4.1 for details). **(Bottom)** Memory utilization in GB. Overall, SYNTHAX shows significantly faster performance while retaining a similar memory utilization profile.

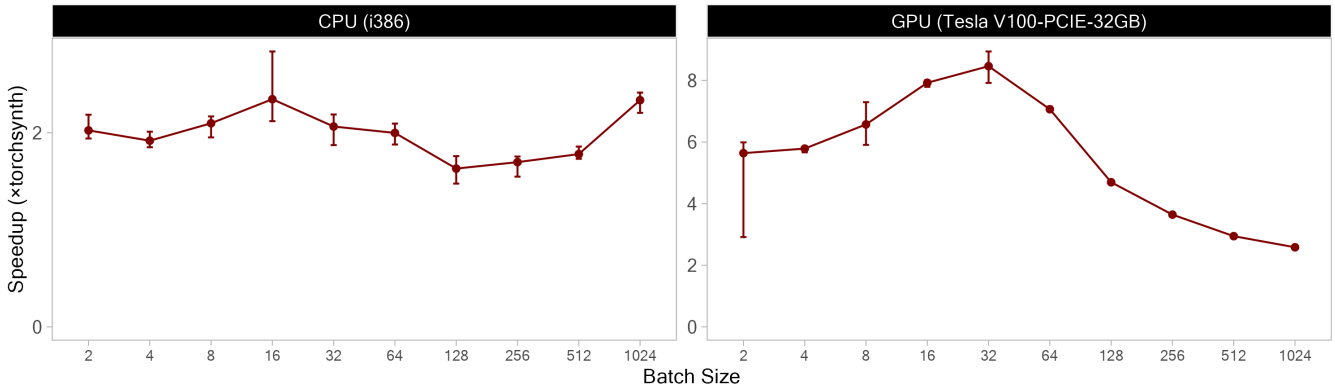

Figure 3: A direct comparison showing speedups relative to *torchsynth* [2] per batch size, again for 100-batch total times averaged across 10 runs. Error bars here show min/max results. Overall, SYNTHAX is more than double the speed in all cases, and peaks at almost 9× the speed of the already accelerated *torchsynth* implementation. As previously, these results are on the *Voice* synthesizer, a 78-parameter synthesizer, where parameters are randomized for each batch.

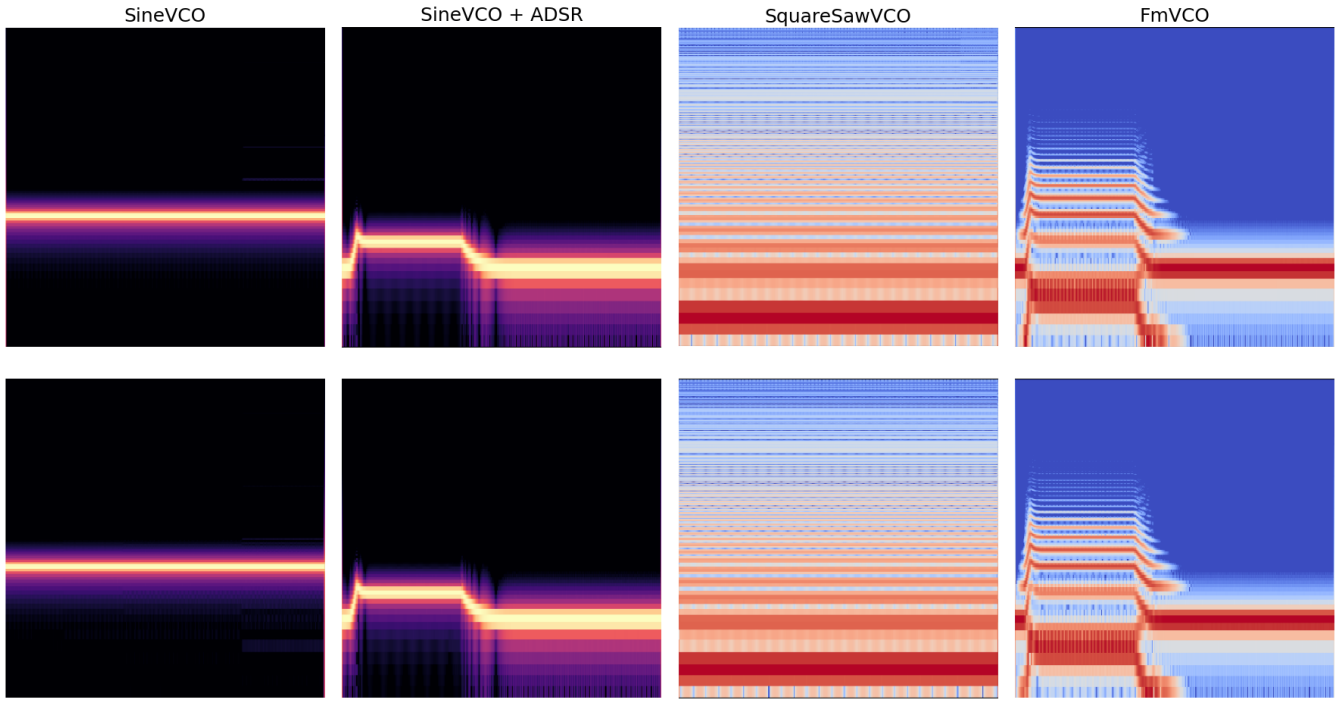

Figure 4: Spectrograms for the examples in *torchsynth* (**Top**) and the replication in SYNTHAX (**Bottom**). From left to right, we show a simple sine wave, a sine wave with an ADSR envelope modulating the frequency, a square wave, and an ADSR envelope-modulated FM patch. The results show clear replication of the output spectrotemporal features.

employed the Surge XT pitch dataset [2] to evaluate representations on such a task [8, 9]. Many other such prediction problems could be formulated for both training and evaluation, as they expose ground truth information as labels. A synthesizer can generate a large variety of sounds that vary in timbre while holding pitch constant, or conversely which vary in pitch but hold timbre constant for a task such as instrument recognition.

5.2 The Synthesizer Programming Problem

One particular area where SYNTHAX can be useful is in the synthesizer programming problem [10], and specifically the task of parameter inference [11]. A canonical formulation of this asks an algorithm to program a synthesizer to match a given sound. The difficulties of manually programming complex synthesizers are well-established [12], and as such a variety of

techniques [13, 14, 15, 16, 17, 18, 19, 11] and even software libraries [20] have been developed to approach this through the lens of automatic matching. Typically, algorithms used are those common to other search and optimization problems, such as genetic algorithms and even gradient-based optimizers. Given a sound, these algorithms seek to minimize some measure (often perceptually-motivated) of the "distance" between the target sound and a synthesized candidate by tweaking the synthesis parameters. SYNTHAX can accelerate such applications by speeding up the synthesis, often the most costly step in these problems. Additionally, SYNTHAX can be combined with other parts of the pipeline written in JAX [1] (such as evosax [5]) to provide a broader speedup for synthesizer programming by matching target sounds.

6 Conclusion

In this paper, we presented SYNTHAX, a fast modular synthesizer implemented in JAX. We showed that SYNTHAX generates sounds orders of magnitude faster than realtime, and significantly faster than existing solutions to accelerated sound synthesis. We discussed the possible applications of this synthesizer in research and production problems involving intelligent sound processing and synthesis. In the future, we intend to expand it with more modules and a user interface. By open sourcing this library, we invite contributions towards a high-performance, robust, and well-documented synthesizer that we hope will eventually parallel commercial software synthesizers in the range of possible sounds producible, while retaining the performance benefits which we observe in our experiments on this initial implementation.

Acknowledgments

The authors acknowledge the MIT SuperCloud [21] and Lincoln Laboratory Supercomputing Center for providing resources that have contributed to the research results reported within this paper. This research was conducted with the partial support of a US-Spain Fulbright grant.

References

[1] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, et al. Jax: composable transformations of python+ numpy programs. 2018.

[2] Joseph Turian, Jordie Shier, George Tzanetakis, Kirk McNally, and Max Henry. One billion audio sounds from gpu-enabled modular synthesis. In *2021 24th International Conference on Digital Audio Effects (DAFx)*, pages 222–229. IEEE, 2021.

[3] Jonathan Heek, Anselm Levskaya, Avital Oliver, Marvin Ritter, Bertrand Rondepierre, Andreas Steiner, and Marc van Zee. Flax: A neural network library and ecosystem for JAX, 2023.

[4] Igor Babuschkin, Kate Baumli, Alison Bell, Surya Bhupatiraju, Jake Bruce, Peter Buchlovsky, David Budden, Trevor Cai, Aidan Clark, Ivo Danihelka, Antoine Dedieu, Claudio Fantacci, Jonathan Godwin, Chris Jones, Ross Hemsley, Tom Hennigan, Matteo Hessel, Shaobo Hou, Steven Kapturowski, Thomas Keck, Iurii Kemaev, Michael King, Markus Kunesch, Lena Martens, Hamza Merzic, Vladimir Mikulik, Tamara Norman, George Papamakarios, John Quan, Roman Ring, Francisco Ruiz, Alvaro Sanchez, Rosalia Schneider, Eren Sezener, Stephen Spencer, Srivatsan Srinivasan, Wojciech Stokowiec, Luyu Wang, Guangyao Zhou, and Fabio Viola. The DeepMind JAX Ecosystem, 2020.

[5] Robert Tjarko Lange. evosax: Jax-based evolution strategies. *arXiv preprint arXiv:2212.04180*, 2022.

[6] Yujin Tang, Yingtao Tian, and David Ha. Evojax: Hardware-accelerated neuroevolution. *arXiv preprint arXiv:2202.05008*, 2022.

[7] Bryan Lim, Maxime Allard, Luca Grillotti, and Antoine Cully. Accelerated quality-diversity for robotics through massive parallelism. *arXiv preprint arXiv:2202.01258*, 2022.

[8] Daisuke Niizumi, Daiki Takeuchi, Yasunori Ohishi, Noboru Harada, and Kunio Kashino. Byol for audio: Exploring pre-trained general-purpose audio representations. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 31:137–151, 2022.

[9] Daisuke Niizumi, Daiki Takeuchi, Yasunori Ohishi, Noboru Harada, and Kunio Kashino. Masked modeling duo: Learning representations by encouraging both networks to model the input. In *ICASSP 2023-2023 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1–5. IEEE, 2023.

[10] Jordie Shier. *The synthesizer programming problem: improving the usability of sound synthesizers*. PhD thesis, 2021.

[11] Ricardo Antonio García. *Automatic generation of sound synthesis techniques*. PhD thesis, Massachusetts Institute of Technology, 2001.

[12] Allan Seago, Simon Holland, and Paul Mulholland. A critical analysis of synthesizer user interfaces for timbre. 2004.

[13] Naotake Masuda and Daisuke Saito. Synthesizer sound matching with differentiable dsp. In *ISMIR*, pages 428–434, 2021.

[14] Masato Hagiwara, Maddie Cusimano, and Jen-Yu Liu. Modeling animal vocalizations through synthesizers. *arXiv preprint arXiv:2210.10857*, 2022.

[15] Harri Renney, Benedict Gaster, and Thomas J Mitchell. Survival of the synthesis—gpu accelerating evolutionary sound matching. *Concurrency and Computation: Practice and Experience*, 34(10):e6824, 2022.

[16] Naotake Masuda and Daisuke Saito. Quality-diversity for synthesizer sound matching. *Journal of Information Processing*, 31:220–228, 2023.

[17] Philippe Esling, Naotake Masuda, and Axel Chemla-Romeu-Santos. Flowsynth: simplifying complex audio generation through explorable latent spaces with normalizing flows. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*, pages 5273–5275, 2021.

[18] Sebastian Löbbers, Louise Thorpe, and György Fazekas. Sketchsynth: Cross-modal control of sound synthesis. In *International Conference on Computational Intelligence in Music, Sound, Art and Design (Part of EvoStar)*, pages 164–179. Springer, 2023.

[19] Hugo Scurto, Bavo Van Kerrebroeck, Baptiste Caramiaux, and Frédéric Bevilacqua. Designing deep reinforcement learning for human parameter exploration. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 28(1):1–35, 2021.

[20] Jordie Shier, George Tzanetakis, and Kirk McNally. Spiegelib: An automatic synthesizer programming library. In *Audio Engineering Society Convention 148*. Audio Engineering Society, 2020.

[21] Albert Reuther, Jeremy Kepner, Chansup Byun, Siddharth Samsi, William Arcand, David Bestor, Bill Bergeron, Vijay Gadepally, Michael Houle, Matthew Hubbell, Michael Jones, Anna Klein, Lauren Milechin, Julia Mullen, Andrew Prout, Antonio Rosa, Charles Yee, and Peter Michaleas. Interactive supercomputing on 40,000 cores for machine learning and data analysis. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2018.